Elektrotehnički fakultet

Univerzitet u Beogradu

Doktorska disertacija

MEHANIZAM INJEKTIRANJA U KEŠ MEMORIJU KOD MULTIPROCESORSKIH SISTEMA SA DELJENOM MEMORIJOM

Aleksandar Milenković

Jun 1999.

Apstrakt

Problem velikih kašnjenja u pristupu memoriji predstavlja glavnu prepreku postizanju veće performanse kod multiprocesorskih sistema sa deljenom memorijom. Štaviše, ovaj problem dobija na značaju ako se imaju u vidu tehnološki trendovi koji predviđaju da će se raskorak u brzini procesora i memorije povećavati u budućnosti. U takvim uslovima, tehnike za prikrivanje kašnjenja u pristupu memoriji postaju ključne u povećanju stepena iskorišćenja procesora u multiprocesorskim sistemima.

U osnovi svih tehnika za prikrivanje kašnjenja je pokušaj da se pristup memoriji preklopi sa korisnim izračunavanjima i/ili drugim pristupima memoriji. Najpoznatije tehnike za prikrivanje kašnjenja su relaksirani modeli memorijske konzistencije, dohvatanje podataka unapred (*prefetching*), prosleđivanje podataka budućim korisnicima (*forwarding*) i višekonteksna obrada (*multithreading*). Poslednjih godina veliki broj istraživanja posvećen je softverski kontrolisanim tehnikama *prefetching* i *forwarding*. Dokazana je visoka efikasnost ovih tehnika u CC-NUMA multiprocesorima. Međutim, utvrđeno je da tehnika *prefetching* ne pokazuje očekivanu efikasnost u SMP multiprocesorima sa zajedničkom magistralom. Sa druge strane, tehnika *forwarding* zbog kompleksnosti implementacije nije analizirana u SMP sistemima sa zajedničkom magistralom.

U ovoj tezi, polazeći od dobrih osobina postojećih tehnika *prefetching* i *forwarding* i osobina zajedničke magistrale, predložena je nova tehnika za prikrivanje kašnjenja u pristupu memoriji kod SMP sistema sa zajedničkom magistralom, nazvana injektiranje u keš memoriju (*cache injection*). Kod tehnike injektiranja procesor korisnik podataka instrukcijom `OpenWindow` inicijalizuje specijalnu tabelu injektiranja koja je deo keš kontrolera, u skladu sa očekivanim potrebama. Tokom *snooping* faze ciklusa čitanja ili pisanja na magistrali, keš kontroler proverava sadržaj tabele injektiranja; ukoliko se adresa keš bloka tekuće transakcije na magistrali nalazi u tabeli injektiranja, procesor prihvata taj blok u svoju keš memoriju. U situacijama kada postoji više procesora korisnika podataka, odnosno deljenje podataka tipa 1-Proizvođač-N-Potrošača, jedan procesor inicira dohvatanje keš bloka, dok ostali procesori prihvataju keš blok u svoju keš memoriju zahvaljujući mehanizmu injektiranja. Kada deljenje podataka odgovara uzorku 1-Proizvođač-1-Potrošač, procesor proizvođač podataka, po završetku obrade keš bloka, inicira ciklus upisa u memoriju instrukcijom `Update`. Tokom tog ciklusa procesori koji imaju inicijalizovane tabele injektiranja prihvataju keš blok u svoju keš memoriju. Ovaj pristup može se koristiti i kada postoji deljenje podataka tipa 1-Proizvođač-N-Potrošača. Efikasnost mehanizma injektiranja proizilazi iz eliminisanja promašaja u keš memoriji usled čitanja i smanjivanja saobraćaja na zajedničkoj magistrali. Dodatna hardverska kompleksnost je minimalna i podrazumeva podršku instrukcijama `OpenWindow`, `CloseWindow` (invalidacija odgovarajućeg ulaza u tabeli injektiranja) i `Update` i implementaciju tabele injektiranja u keš kontroleru.

U cilju verifikacije predložene tehnike koristi se simulaciona analiza bazirana na realnom izvršavanju paralelnih aplikacija. Kao radno opterećenje koriste se originalno razvijene paralelne aplikacije i aplikacije iz skupa SPLASH-2. Za svaku aplikaciju posmatra se polazna verzija programa i jedna ili više verzija koje uključuju podršku mehanizmu injektiranja. Instrukcije za podršku injektiranju umetnute su u kôd ručno, na bazi statičke analize kôda i

deljenja podataka. Simulator memorijskog podsistema detaljno opisuje ponašanje keš kontrolera koji sadrži implementirane sve napredne tehnike svojstvene modernim procesorima i zajedničke magistrale koja podržava razdvojene transakcije čitanja. Eksperimenti su ponovljeni za različite parametre memorijskog podsistema kako bi se utvrdila osetljivost mehanizma injektiranja. Dobijeni rezultati pokazuju da je tehnika injektiranja efikasna u prikrivanju kašnjenja u pristupu memoriji. Tako, za neke aplikacije brzina izvršavanja se povećava i do 10 puta. U opštem slučaju, efikasnost mehanizma injektiranja raste sa rastom broja procesora u sistemu, a takođe i sa porastom kapaciteta keš memorije i kašnjenja u pristupu memoriji.

Ključne reči: multiprocesori sa deljenom memorijom (*shared-memory multiprocessors*), tehnike za prikrivanje kašnjenja (*memory tolerating techniques*), dohvatanje podataka unapred (*data prefetching*), prosleđivanje podataka (*data forwarding*), injektiranje u keš memoriju (*cache injection*).

Zahvalnica

Ovo je lista ljudi kojima posebno želim da zahvalim na nesebičnoj pomoći i podršci koji su mi pružili tokom izrade ove teze.

8. Profesorima koji su tokom konferencije INFO-FEST'97 podelili svoje vreme sa mnom i svojim sugestijama pomogli mi da bolje sagledam problem prikrivanja kašnjenja u pristupu memoriji: Per Stenstrom, Ali Hurson, Mateo Valero, Antonio Gonzales, Antonio Prete.

7. Firmi Steffaco koja je svojom donacijom Katedri uvećala raspoloživo procesorsko vreme neophodno za dugotrajne simulacije.

6. Miljanu Vuletiću, Zoranu Dimitrijeviću i Davoru Magdiću koji su mi "suživot" sa operativnim sistemom Linux učinili prijatnijim.

5. Bivšim i sadašnjim članovima istraživačke grupe SMP/DSM za vreme koje su posvetili ovoj tezi tokom redovnih sastanaka grupe: mr Jelici Protić, mr Milanu Jovanoviću, Darku Marinovu, Davoru Magdiću, Igoru Ikodinoviću i Zoranu Dimitrijeviću.

4. Davoru Magdiću koji je razvio programski alat Limes i podelio ga sa istraživačima širom sveta. Takođe, zahvaljujem mu na savetima tokom mukotrpnog procesa razvoja simulatora i proširivanja alata Limes.

3. Doc. dr Milu Tomaševiću za mnogobrojne savete i pomoć u definisanju uslova simulacione analize.

2. Doc. dr Igoru Tartalji koji je poslednju godinu dana strpljivo pratio moj rad i svojim savetima doprineo prevazilaženju mnogih problema i nedoumica. Njegovi predlozi su u velikoj meri oblikovali pristup simulacionoj analizi.

1. Prof. dr Veljku Milutinoviću, mentoru, koji je podelio svoje ideje o injektiranju u keš memoriju sa mnom. Zahvalan sam mu na nesebičnoj pomoći, strpljenju i podršci koju mi je pružio tokom izrade ove teze. Takođe, zahvalan sam mu na pomoći u stvaranju svih pretpostavki za izradu jedne ovakve teze u našem okruženju, počev od obezbeđivanja računara, nabavke časopisa i ostvarivanja kontakta sa drugim istraživačima u svetu.

Sadržaj

Poglavlje 1 Uvod....1
1.1 Keširanje u multiprocesorskim sistemima....3
1.2 Tehnike za prikrivanje kašnjenja u pristupu memoriji....5
1.3 Predmet teze....7
1.4 Metod rada....8
1.5 Osnovni rezultati....8
1.6 Organizacija i sadržaj teze....9
Poglavlje 2 Postojeće tehnike za prikrivanje kašnjenja u pristupu memoriji....10
2.1 Dohvatanje podataka unapred....10
2.1.1 Hardverski inicirano dohvatanje podataka unapred....11
2.1.2 Softverski inicirano dohvatanje podataka unapred....12
2.2 Prosleđivanje podataka budućim korisnicima....14
2.3 Pregled istraživanja od interesa za tezu....17
2.3.1 Algoritam za selektivno dohvatanje podataka unapred kod jednoprocesorskih sistema....18
2.3.2 Algoritam za selektivno dohvatanje podataka unapred kod multiprocesorskih sistema....23
2.3.3 Dohvatanje podataka unapred u aplikacijama sa rekurzivnim dinamičkim strukturama podataka....26
2.3.4 Potencijal dohvatanja podataka unapred kod multiprocesora sa zajedničkom magistralom....29
2.3.5 Algoritam za prosleđivanje podataka kod skalabilnih multiprocesora sa deljenom memorijom....30
2.3.6 Algoritam za vraćanje modifikovanih deljenih podataka u glavnu memoriju....31
2.3.7 Ubrzanje kritičnih sekcija primenom dohvatanja podataka unapred i prosleđivanja podataka....34
2.3.8 Efikasnost komunikacionih primitiva iniciranih od strane procesora proizvođača podataka....36
2.3.9 Eksplicitna komunikacija kod multiprocesora sa deljenom memorijom....37
Poglavlje 3 Mehanizam injektiranja....39
3.1 Motivacija....39
3.2 Mehanizam injektiranja....41

3.2.1 Predložene instrukcije 42
3.2.2 Primena predloženih instrukcija 44
3.3 Primer primene injektiranja na prave deljene podatke 46
3.3.1 Polazni primer (Base) 46
3.3.2 Primena dohvatanja podataka unapred (Pref) 48
3.3.3 Primena prosleđivanja podataka (Forw) 49
3.3.4 Kombinovana primena dohvatanja unapred i prosleđivanja podataka (Pref+Forw) 50
3.3.5 Primena injektiranja (Inject) 51
3.3.6 Uporedni prikaz efikasnosti razmatranih tehnika 54
3.4 Injektiranje i sinhronizacija paralelnih programa 54
3.4.1 Sinhronizacija paralelnih programa 55
3.4.1.1 Sinhronizacione primitive *Lock* i *Unlock* 55
3.4.1.2 Sinhronizaciona primitiva *Barrier* 57
3.4.2 Primena injektiranja u implementaciji sinhronizacionih operacija *lock* i *unlock* 59
3.4.2.1 Test-and-exch *lock* 59
3.4.2.2 LL-SC *lock* 62
3.4.3 Primena injektiranja u implementaciji globalne sinhronizacione primitive *barrier* 64
3.5 Podrška mehanizmu injektiranja u programskom prevodiocu 65
3.6 Hardverska podrška mehanizmu injektiranja 66
Poglavlje 4 Eksperimentalna metodologija 68
4.1 Programski alat LIMES 68
4.1.1 Formiranje simulatora i tok simulacije 70
4.1.2 Komunikacija simulatora memorijskog podsistema sa jezgrom 72
4.1.3 Jedan primer organizacije simulatora memorijskog podsistema 73
4.2 Organizacija memorijskog podsistema 74
4.2.1 MESI protokol 74
4.2.2 Magistrala sa podrškom razdvojenim transakcijama 77
4.2.3 Organizacija keš kontrolera 78
4.2.3.1 Jedinica PCC 79
4.2.3.2 Jedinica BCU & SCC 82
4.3 Simulatori multiprocesorskog sistema 91
4.3.1 PRAM-MESI 91
4.3.2 MESI-SPLIT 92

4.4 Aplikacije i primena mehanizma injektiranja 93
4.4.1 Modifikacija polaznih programa 93
4.4.2 LTEST & BTEST 94
4.4.3 PC 95
4.4.4 MM 95
4.4.5 Jacobi 96
4.4.6 Radix 97
4.4.7 LU 99
4.4.8 FFT 102
4.4.9 Ocean 105
Poglavlje 5 Rezultati 107
5.1 Mehanizam injektiranja kod sinhronizacionih primitiva 107
5.1.1 Lock&Unlock 108
5.1.2 Barrier 110
5.2 Mehanizam injektiranja kod paralelnih aplikacija 111
5.2.1 PC 113
5.2.1.1 PC: PRAM-MESI 113
5.2.1.2 PC: MESI-SPLIT 114
5.2.2 MM 116
5.2.2.1 MM: PRAM-MESI 116
5.2.2.2 MM: MESI-SPLIT 117
5.2.3 Jacobi 119
5.2.3.1 Jacobi: PRAM-MESI 120
5.2.3.2 Jacobi: MESI-SPLIT 121
5.2.4 Radix 122
5.2.4.1 Radix: PRAM-MESI 123
5.2.4.2 Radix: MESI-SPLIT 124
5.2.5 LU 126
5.2.5.1 LU: PRAM-MESI 126
5.2.5.2 LU: MESI-SPLIT 127
5.2.6 FFT 130
5.2.6.1 FFT: PRAM-MESI 130
5.2.6.2 FFT: MESI-SPLIT 131
5.2.7 Ocean 134

5.2.7.1 Ocean: PRAM-MESI 134
5.2.7.2 Ocean: MESI-SPLIT 135
5.3 Rezime 137
Poglavlje 6 Zaključak 140
6.1 Rezime teze 140
6.2 Osnovni rezultati 142
6.3 Pravci mogućih budućih istraživanja 143
Literatura 144

Spisak slika

Sl. 1-1. Organizacija memorijskog podsistema multiprocesora sa centralizovanom memorijom i magistralom kao sprežnom mrežom. 2

Sl. 1-2. Organizacija memorijskog podsistema kod multiprocesora sa distribuiranom memorijom. 2

Sl. 1-3. Iskorišćenje procesora u funkciji kašnjenja u pristupu memoriji. 5

Sl. 1-4. Ilustracija poboljšanja performanse korišćenjem softverski kontrolisanog dohvatanja podataka unapred. 6

Sl. 1-5. Ilustracija poboljšanja performanse korišćenjem softverski kontrolisanog prosleđivanja podataka. 6

Sl. 2-1. Softverski inicirano dohvatanje podataka unapred. 13

Sl. 2-2. Ekskluzivno dohvatanje podataka unapred. 13

Sl. 2-3. Preuranjeno dohvatanje podataka unapred #1. 14

Sl. 2-4. Preuranjeno dohvatanje podataka unapred #2. 14

Sl. 2-5. Primena prosleđivanja podataka u eliminisanju promašaja u keš memoriji usled pravog deljenja podataka. 15

Sl. 2-6. Primena prosleđivanja podataka u eliminisanju promašaja u keš memoriji usled prvog pristupa podatku. 15

Sl. 2-7. Uticaj prosleđivanja podataka na promašaje usled konflikta u keš memoriji 16

Sl. 2-8. Prosleđivanje sa invalidacijom lokalne kopije podatka kao rešenje za komunikaciju između procesora proizvođača. 16

Sl. 2-9. Segment koda korišćen za ilustraciju algoritma za selektivno dohvatanje podataka unapred. 18

Sl. 2-10. Ilustracija promašaja u keš memoriji tokom izvršavanja test primera. 18

Sl. 2-11. Kapacitet podataka kojima se pristupa u toku izvršavanja test primera. 21

Sl. 2-12. *Prefetch* predikati za relevantne pristupe memoriji. 21

Sl. 2-13. Tehnike transformisanja petlji (*loop splitting*). 22

Sl. 2-14. Primena softverske protočnosti (*software-pipelining*). 22

Sl. 2-15. Ilustracija predloženog algoritma za dohvatanje podataka unapred ([Mowry94]) 23

Sl. 2-16. Uticaj deljenja podataka na promašaje u keš memoriji. 24

Sl. 2-17. Primer sa eksplicitnom sinhronizacionom primitivom unutar petlje. 25

Sl. 2-18. Uticaj ekskluzivnog dohvatanja podataka unapred na performanse. 26

Sl. 2-19. Obrada listi kada ne postoji i kada postoji vremenska lokalnost. 27

Sl. 2-20. Ilustracija *pointer-chasing* problema.....28

Sl. 2-21. Ilustracija algoritma *greedy prefetching*.....29

Sl. 2-22. Grafovi toka sa `store` instrukcijama koje upisuju u isti keš blok. 32

Sl. 2-23. Primeri primene algoritma za umetanje instrukcija za ažuriranje memorije *home* čvora.....33

Sl. 2-24. Optimizacije kritičnih sekcija dohvatanjem podataka unapred i prosleđivanjem podataka. 36

Sl. 2-25. Primeri primene optimizacija kritičnih sekcija. 36

Sl. 2-26. Primitive za eksplicitnu komunikaciju kod multiprocesora sa deljenom memorijom. 38

Sl. 3-1. Organizacija tabele injektiranja.....43

Sl. 3-2. Predložene instrukcije za inicijalizaciju i invalidaciju ulaza tabele injektiranja.....43

Sl. 3-3. Predložene instrukcije za ažuriranje memorije nakon poslednjeg upisa. 44

Sl. 3-4. Primer koji demonstrira pravo deljenje podataka.....44

Sl. 3-5. Polazni primer nakon primene injektiranja u keš.....46

Sl. 3-6. Paralelni program koji ilustruje pravo deljenje podataka.....47

Sl. 3-7. Paralelni program nakon primene algoritma za selektivno umetanje `prefetch` instrukcija.....48

Sl. 3-8. Prestruktuiranje kôda sa ciljem da se izbegne zaustavljanje procesora tokom invalidacije.....49

Sl. 3-9. Paralelni program nakon umetanja instrukcija za prosleđivanje.....49

Sl. 3-10. Paralelni program bez redundantnih prosleđivanja. 50

Sl. 3-11. Kombinovanje prosleđivanja podataka sa dohvatanjem unapred. 51

Sl. 3-12. Paralelni program modifikovan da podrži injektiranje tokom ciklusa čitanja. 52

Sl. 3-13. Paralelni program modifikovan da podrži injektiranje tokom ciklusa upisa.....53

Sl. 3-14. Paralelni program nakon kombinovanja tehnika injektiranja i ekskluzivnog dohvatanja unapred. 53

Sl. 3-15. Uporedni prikaz performanse različitih verzija paralelnog programa.....54

Sl. 3-16. Implementacija *lock* i *unlock* primitiva korišćenjem atomske `exch` instrukcije.....56

Sl. 3-17. Implementacija *Lock* i *unlock* primitiva korišćenjem `test-and-exch` pristupa. .57

Sl. 3-18. Implementacija *lock* i *unlock* primitiva korišćenjem `ll` i `sc` instrukcija.....57

Sl. 3-19. Makroi za rad sa barijerama.....58

Sl. 3-20. Jedna implementacija barijere.....58

Sl. 3-21. Pseudokôd kritične sekcije.....59

Sl. 3-22. Izvršavanje polazne kritične sekcije u slučaju *test&exch* implementacije *lock* primitive.....60

Sl. 3-23. Kritična sekcija sa podrškom mehanizmu injektiranja. 60

Sl. 3-24. Izvršavanje kritične sekcije sa injektiranjem u slučaju *test&exch* implementacije *lock* primitive. 61

Sl. 3-25. Izvršavanje kritične sekcije sa injektiranjem u slučaju *test&exch* implementacije *lock* primitive sa modifikovanom `exch` instrukcijom. 61

Sl. 3-26. Saobraćaj na magistrali tokom izvršavanja kritičnih sekcija sa *test&exch* implementacijom *lock* primitive. 62

Sl. 3-27. Izvršavanje polazne kritične sekcije u slučaju `ll-sc` implementacije *lock* primitive. 62

Sl. 3-30. Saobraćaj na magistrali tokom izvršavanja različitih verzija test primera baziranih na `ll-sc` implementaciji *lock* primitive. 64

Sl. 3-31. Modifikovana implementacija barijere koja podržava mehanizam injektiranja. 65

Sl. 3-32. Deo programa koji obezbeđuje podršku mehanizmu injektiranja za barijere. 65

Sl. 4-1. Organizacija programskog alata Limes. 69

Sl. 4-2. Formiranje simulatora u programskom paketu Limes: proces prevođenja i povezivanja. 71

Sl. 4-3. Moduli simulatora memorijskog podsistema multiprocesora sa zajedničkom magistralom....74

Sl. 4-4. Dijagrami stanja/prelaza za MESI protokol. 77

Sl. 4-5. Struktura keš kontrolera. 79

Sl. 4-6. Inicijalno stanje PCC kontrolne jedinice i zahtevi koji dolaze od procesora. 79

Sl. 4-7. Dijagram toka Read zahteva jedinice PCC. 80

Sl. 4-8. Dijagram toka Write zahteva jedinice PCC. 81

Sl. 4-9. Dijagram toka RdC ciklusa na magistrali: faza postavljanja zahteva (*request phase*).84

Sl. 4-10. Dijagram toka RdC ciklusa na magistrali: faza prihvatanja odgovora (*response phase*). 85

Sl. 4-11. Dijagram toka InvC ciklusa na magistrali. 87

Sl. 4-12. Dijagram toka SWbC ciklusa na magistrali. 88

Sl. 4-13. Dijagram toka RWbC ciklusa na magistrali. 89

Sl. 4-14. Dijagram toka *snooping* ciklusa na magistrali. 90

Sl. 4-15. Dijagram toka provere da li postoje uslovi za injektiranje. 91

Sl. 4-16. Sinhronizaciono jezgro LTEST. 94

Sl. 4-17. Sinhronizaciono jezgro BTEST. 95

Sl. 4-18. Množenje matrica. 96

Sl. 4-19. Jacobi: dekompozicija i deljenje podataka. 96

Sl. 4-20. Ilustracija kôda paralelne aplikacije RADIX uključujući instrukcije za podršku mehanizmu injektiranja. 99

Sl. 4-21. 2D raštrkana dekompozicija kod aplikacije LU. 100

Sl. 4-23. FFT: Algoritam i ilustracija faze transponovanja matrice podataka. 103

Sl. 4-25. Ocean: Ilustracija deljenja podataka u fazi rešavanja parcijalnih diferencijalnih jednačina. 106

Sl. 5-1. Relevantni parametri simuliranog memorijskog podsistema. 108

Sl. 5-2. LTEST: LAT i NET; (C=62, D=300). 109

Sl. 5-3. LTEST: LAT i NET; (C=8, D=300). 109

Sl. 5-4. BTEST: LAT i NET; (Tmin=N=40, Tmax=X=40, I=100). 111

Sl. 5-5. Relevantni parametri memorijskog podsistema opisanog PRAM-MESI simulatorom. 112

Sl. 5-6. Relevantni parametri realnog memorijskog podsistema opisanog MESI-SPLIT simulatorom. 113

Sl. 5-7. PC: MR i BT; P=8. 114

Sl. 5-8. PC: MR i BT; P=16. 114

Sl. 5-9. PC: NET i NRST; P=8. 115

Sl. 5-10. PC: NET i NRST; P=16. 115

Sl. 5-11. PC: SU. 116

Sl. 5-12. MM: MR i BT; P=8. 117

Sl. 5-13. MM: MR i BT; P=16. 117

Sl. 5-14. MM: NET i NRST; P=8. 118

Sl. 5-15. MM: NET i NRST; P=16. 119

Sl. 5-16. MM: SU. 119

Sl. 5-17. Jacobi: MR i BT; P=8. 120

Sl. 5-18. Jacobi: MR i BT; P=16. 120

Sl. 5-19. Jacobi: NET i NRST; P=8. 121

Sl. 5-20. Jacobi: NET i NRST; P=16. 122

Sl. 5-21. Jacobi: SU. 122

Sl. 5-22. Radix: MR i BT; P=8. 123

Sl. 5-23. Radix: MR i BT; P=16. 123

Sl. 5-24. Radix: NET i NRST; P=8. 124

Sl. 5-25. Radix: NET i NRST; P=16. 125

Sl. 5-26. Radix: SU. 126

Sl. 5-27. LU: MR i BT; P=8. 127

Sl. 5-28. LU: MR i BT; P=16. 127

Sl. 5-29. LU: NET i NRST; P=8. 128

Sl. 5-30. LU: NET i NRST; P=16. 128

Sl. 5-31. Saobraćaj na magistrali za različite verzije aplikacije LU; CacheSize=128KB, P=16. 129

Sl. 5-32. LU: SU. 130

Sl. 5-33. FFT: MR i BT; P=8. 131

Sl. 5-35. FFT: NET i NRST; P=8. 132

Sl. 5-36. FFT: NET i NRST; P=16. 133

Sl. 5-37. FFT: SU. 134

Sl. 5-38. Ocean: MR i BT; P=8. 134

Sl. 5-39. Ocean: MR i BT; P=16. 135

Sl. 5-40. Ocean: NET i NRST: P=8. 135

Sl. 5-41. Ocean: NET i NRST; P=16. 136

Sl. 5-42. Ocean: SU. 137

Sl. 5-43. Poboljšanje performanse ostvareno mehanizmom injektiranja za aplikacije PC, MM i Jacobi. 138

Sl. 5-44. Poboljšanje performanse ostvareno mehanizmom injektiranja za aplikacije Radix, LU, FFT i Ocean. 139

Poglavlje 1

Uvod

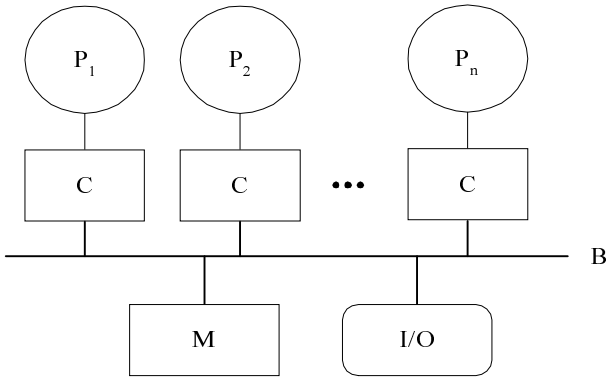
Izuzetan rast performanse mikroprocesora po stopi od preko 60% godišnje obeležio je proteklu deceniju u razvoju računarske tehnike. Međutim, ni takav napredak ni izbliza nije rešio problem stalnog rasta potreba korisnika. Naime, razvoj računarske tehnike uticao je istovremeno i na rast stvarnih i "proizvedenih" potreba korisnika. Pri tom, zahtevi koji se postavljaju pred moderne mikroprocesore ne retko premašaju raspoložive performanse, tako da je postizanje visoke performanse i dalje najveći izazov arhitektama modernih računarskih sistema. Jedan od najefikasnijih načina za postizanje veće performanse jeste povezivanje više standardnih mikroprocesora u jedan multiprocesorski sistem. U idealnom slučaju vreme koje je potrebno za rešavanje nekog problema na multiprocesorskom sistemu sa N procesora trebalo bi da bude N puta kraće od vremena koje je potrebno jednoprocesorskom sistemu.

U praksi su najrašireniji multiprocesorski sistemi bazirani na relativno malom broju komercijalnih mikroprocesora (<100). Podrška multiprocesiranju koja se može sresti kod svih modernih mikroprocesora (Pentium, PentiumPro, PentiumII, PowerPC, R10000, ...) dovela je do pojave velikog broja komercijalnih multiprocesorskih sistema (SPARCCenter, SGI Challenge, Cray T3D, Convex Exemplar, Intel Paragon, IBM SP-2) [Patte*96]. Postoji nekoliko osnovnih razloga koji nas navode na zaključak da će značaj ovih sistema rasti u neposrednoj budućnosti. Prvo, povezivanje više mikroprocesora u jedan sistem predstavlja logičan način za postizanje veće procesne snage. Drugo, ovakav trend razvoja jednoprocesorskih sistema verovatno se neće nastaviti beskonačno. Po nekim predviđanjima, uskoro će trend rasta performanse mikroprocesora pasti na stopu od 30% godišnje. Treće, načinjen je značajan progres u oblasti sistemske podrške za multiprocesorske sisteme (operativni sistemi, kompajleri, biblioteke,...), što stvara preduslove za dalji razvoj ove oblasti. Pored toga, činjenica da će kroz 10 godina arhitekte mikroprocesora na raspolaganju imati milijardu tranzistora pokrenula je mnoge rasprave o tome kako iskoristiti toliki broj tranzistora; mnogi istraživači smatraju da je "multiprocesor na čipu" pristup koji najviše obećava [Hammo*97].

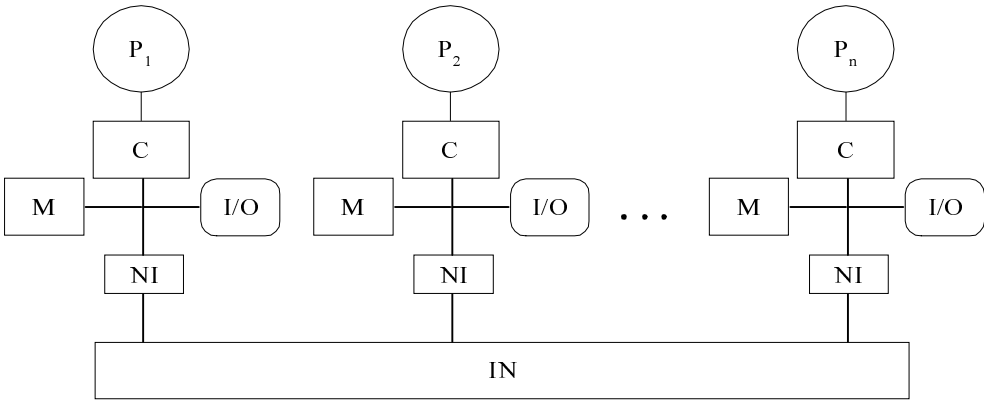
Prema organizaciji memorijskog podsistema, postojeće multiprocesorske sisteme možemo svrstati u dve osnovne grupe [Patte*96]. Prva grupa obuhvata sisteme sa centralizovanom deljenom memorijom (*centralized shared memory architectures*). Kod ovih sistema više procesora preko interkonekcione mreže, najčešće magistrale, pristupa zajedničkoj, centralizovanoj memoriji (Sl. 1-1). Kako je vreme pristupa memoriji jednako za sve

procesore, ovi sistemi se često nazivaju UMA (*Uniform Memory Access*) ili SMP (*Symmetric Multiprocessors*). Trenutno, najpopularniji su SMP sistemi sa zajedničkom magistralom kao interkonekcionom mrežom (*bus-based SMPs*). Međutim, korišćenje magistrale kao interkonekcione mreže čini ove sisteme neskalabilnim, tako da je u praksi broj procesora ograničen na najviše 32.

Da bi se efikasno podržali memorijski zahtevi koje postavljaju arhitekture sa većim brojem procesora, memorija mora biti distribuirana. Svaki čvor poseduje procesor, lokalnu memoriju, neke I/O uređaje i sprežni deo prema interkonekcionoj mreži velike propusne moći preko koje je povezan sa ostalim čvorovima (Sl. 1-2). Ovakav pristup omogućuje pravljenje skalabilnih sistema; pristup lokalnoj memoriji je brz, dok se komunikacija između procesora odvija preko interkonekcione mreže i traje duže. Tipično, I/O uređaji su takođe distribuirani po čvorovima, a u okviru jednog čvora može biti i više procesora (4-8). Ovakvi sistemi se nazivaju *distributed memory systems* ili NUMA (*Non-Uniform Memory Access*), zbog nejednake brzine pristupa memoriji.



Sl. 1-1. Organizacija memorijskog podsistema multiprocesora sa centralizovanom memorijom i magistralom kao sprežnom mrežom.
Opis: P_i - procesor *i*, C (*Cache*) – keš memorija, M (*Memory*) - memorija, I/O (*Input*/*Output*) – ulazno/izlazni podsistem, B (*Bus*) - magistrala.



Sl. 1-2. Organizacija memorijskog podsistema kod multiprocesora sa distribuiranom memorijom.
Opis: P_i - procesor *i*, C (*Cache*) – keš memorija, M (*Memory*) - memorija, NI (*Network Interface*) – komunikacioni kontroler, I/O (*Input*/*Output*) – ulazno/izlazni podsistem, IN (*Interconnection Network*) - interkonekciona mreža.

U zavisnosti od mehanizma komunikacije među procesorima, multiprocesorski sistemi se mogu klasifikovati u dve osnovne grupe: (a) bazirane na deljenoj memoriji (*shared memory*

multiprocessors) i (b) bazirane na prosleđivanju poruka (*message passing*). Kod arhitektura sa deljenom memorijom, memorija koja može biti fizički distribuirana, logički pripada istom adresnom prostoru. To znači da se ista fizička adresa kod različitih procesora odnosi se na jednu te istu lokaciju u memoriji. Komunikacija između procesora se odvija implicitno upisom i čitanjem u deljeni adresni prostor. Kod arhitektura baziranih na prosleđivanju poruka, programer vidi skup nezavisnih adresnih prostora, a procesori komuniciraju među sobom slanjem eksplicitnih poruka. To znači da se ista fizička adresa kod različitih procesora odnosi na različite lokacije u memoriji. Prednost arhitektura sa prosleđivanjem poruka je jednostavnost hardvera u poređenju sa sistemima koji uključuju hardversku podršku održavanju koherencije deljenih podataka. Međutim, težina programiranja i niske performanse, uslovljene podrškom sistemskog softvera komunikacionom mehanizmu, u velikoj meri ograničavaju efikasnost ovog pristupa. Sa druge strane, lakoća programiranja, brzina komunikacije i mogućnost korišćenja hardverski kontrolisanog keširanja deljenih podataka predstavljaju neke od osnovnih prednosti koje arhitekture sa deljenom memorijom čine vrlo popularnim.

Dominantan pravac u razvoju paralelnih arhitektura čine multiprocesori sa manje od 100 procesora, globalnim deljenim adresnim prostorom i podrškom održavanju keš koherencije, poznati kao keš-koherentni multiprocesori sa deljenom memorijom (*Cache-Coherent Shared Memory Multiprocessors*). Sistemi sa distribuiranom memorijom i skalabilnom interkonekcionom mrežom su poznati još pod imenom CC-NUMA, dok su sistemi bazirani na centralizovanoj memoriji i magistrali kao interkonekcionoj mreži poznati pod nazivom *Bus-based SMPs*.

U opštem slučaju, kod svih keš-koherentnih multiprocesora osnovni problem predstavljaju velika kašnjenja u pristupu memoriji, koja u velikoj meri ograničavaju performanse. Osnovni razlozi za velika kašnjenja u pristupu memoriji su: (a) tehnološki raskorak u brzini procesora i memorije (dok se brzina procesora povećava za preko 60% godišnje, brzina pristupa memoriji raste po stopi nešto većoj od 5% godišnje), (b) kontencija na interkonekcionoj mreži, (c) saobraćaj uzrokovan protokolima za održavanje keš koherencije i (d) veća fizička rastojanja između procesora i memorije. Imajući u vidu ove razloge, tehnike za prevazilaženje problema velikih kašnjenja u pristupu memoriji postaju ključne u postizanju visoke performanse i podizanju stepena iskorišćenja procesora u multiprocesorskom sistemu. U opštem slučaju, tehnike za prevazilaženje problema velikih kašnjenja u pristupu memoriji se mogu podeliti u dve velike grupe: tehnike koje eliminišu kašnjenje (*reducing latency techniques*) i tehnike koje prikrivaju velika kašnjenja (*tolerating latency techniques*). Tehnike za eliminisanje velikih kašnjenja su keširanje i različite optimizacije sa ciljem efikasnijeg korišćenja keš memorija. Tehnike za prikrivanje kašnjenja su dohvatanje podataka unapred (*prefetching*), prosleđivanje podataka (*forwarding*), više-konteksni procesori (*multithreading*), odloženi upisi (*buffering*), itd.

1.1 Keširanje u multiprocesorskim sistemima

Keš memorije predstavljaju ključni element u prevazilaženju kašnjenja u pristupu memoriji. Pristup keš memoriji je mnogo brži od pristupa glavnoj memoriji budući da keš memorija sadrži samo mali broj lokacija koje su obično smeštene na samom procesorskom čipu. Zapravo, trenutni trend je postojanje više nivoa hijerarhije keš memorije: prvi nivo uključuje vrlo mali broj lokacija sa vremenom pristupa koje često odgovara procesorskom taktu. Svaki sledeći nivo sadrži veći broj lokacija sa manjom brzinom pristupa. Verovatnoća da će se

podaci naći u keš memoriji zavisi ne samo od organizacije i veličine keš memorije, već i od lokalnosti adresnih tragova za posmatranu aplikaciju. Postoji vremenska lokalnost (*temporal locality*) koja podrazumeva sklonost da će se pristupiti podatku kojem je već pristupano i prostorna lokalnost koja predstavlja sklonost da će se pristupiti podatku koji se nalazi pored podataka kojima je već pristupano.

Keširanje deljenih podataka u privatnim keš memorijama sa tehnikom odloženog ažuriranja (*write-back*) otvara problem koherencije. Naime, u slučaju da više procesora čita isti podatak, on će se naći u keš memorijama različitih procesora. Problem koherencije podataka se javlja u trenutku kada jedan od procesora izvrši upis u blok koji je deljen; keš memorije ostalih procesora imaju neažurnu kopiju posmatranog bloka, što može dovesti do nekorektnog izvršavanja paralelnih programa. Da bi se rešio problem koherencije, potrebno je izvršiti neku akciju kojom bi se sprečilo da ostali procesori koriste neažurnu kopiju podatka koji se već nalazi u njihovim keš memorijama. U zavisnosti od toga ko inicira takve akcije, razlikuju se hardverske [Tomaš*93] i softverske šeme [Tarta*96] za održavanje keš koherencije. Danas gotovo svi moderni mikroprocesori uključuju hardversku podršku za održavanje keš koherencije.

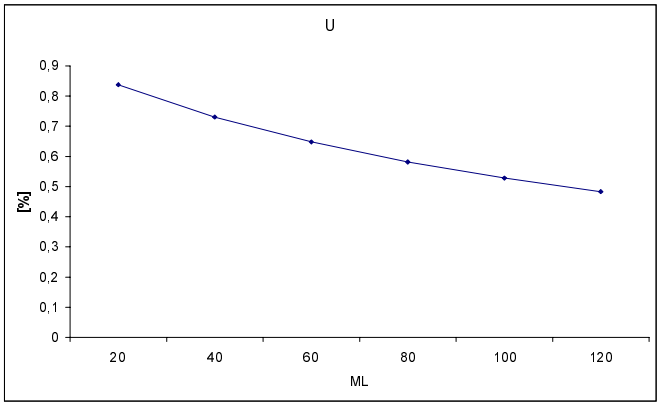
Postoje dva osnovna pristupa održavanju koherencije keš memorija. Prvi pristup podrazumeva da procesor pre upisa nekog podataka obezbedi ekskluzivno vlasništvo tog bloka, invalidujući kopije u keš memorijama ostalih procesora. Protokoli bazirani na ovom pristupu nazivaju se invalidacioni protokoli (*write invalidate*). Alternativni pristup počiva na ažuriranju kopija podatka u keš memorijama drugih procesora, nakon svakog upisa deljenog podatka. Ovakvi protokoli, bazirani na ažuriranju deljenih podataka, nazivaju se *write-update* protokoli.

Kod protokola sa ažuriranjem, procesori uvek poseduju ažurnu verziju podatka, čime se smanjuje broj promašaja u keš memoriji usled pristupa deljenim podacima. Kod protokola sa invalidacijom, sve ostale kopije se invaliduju, tako da je jedina validna kopija u keš memoriji procesora koji je poslednji upisao posmatrani podatak. Međutim, osnovni problem kod protokola sa ažuriranjem je što se svaki upis iznosi na interkonekcionu mrežu i prosleđuje memoriji i/ili keš memorijama procesora koji dele posmatrani podatak. U slučaju više uzastopnih upisa od strane jednog procesora, nepotrebno se opterećuje interkonekciona mreža. Pored toga, budući da keš linije sadrže više reči, a da se ažuriranje vrši na nivou pojedinačnih reči, iskorišćenje interkonkcione mreže (koristan saobraćaj/ukupan saobraćaj) je manje u odnosu na protokole bazirane na invalidaciji. Pored ova dva osnovna pravca predloženi su i brojni hibridni pristupi koji kombinuju ova dva pristupa, ali izvršene analize nisu pokazale neku značajniju korist koja bi opravdala dodatnu kompleksnost [Lenos*92], tako da većina modernih mikroprocesora ima implementirane protokole bazirane na invalidaciji.

Prema načinu implementacije svi protokoli za održavanje keš koherencije podeljeni su u dve velike grupe: protokole bazirane na osluškivanju (*snoopy*) i protokole bazirane na direktorijumu (*directory*). Kod multiprocesora sa zajedničkom magistralom i relativno malim brojem procesora (<32), koherencija keš memorija se održava osluškivanjem (*snooping*). Svaki keš kontroler nadgleda magistralu i ako je potrebno vrši ažuriranje lokalne kopije, u slučaju protokola baziranih na ažuriranju, odnosno invalidaciju lokalne kopije, u slučaju protokola baziranih na invalidaciji. Kod multiprocesora sa većim brojem čvorova, topologija interkonekcione mreže je takva da ne dozvoljava praćenje svakog pristupa memoriji ili je ovakav mehanizam vrlo skup. Stoga se koriste protokoli bazirani na direktorijumima. Direktorijum je deo memorije u kome se čuvaju podaci o svakom bloku deljene memorije.

Međutim, i pored keširanja, problem velikih kašnjenja je i dalje značajan i u velikoj meri ograničava ukupnu procesnu snagu. Ovo tvrđenje ilustrovano je jednim jednostavnim

primerom koji sledi. Posmatra se jedan procesor multiprocesorskog sistema sa idealnom interkonekcionom mrežom. Uvedene su sledeće pretpostavke. Procesor se blokira samo usled promašaja prilikom čitanja. Procesor izvršava 25% `load` instrukcija; 70% `load` instrukcija se odnosi na deljene podatke, a preostalih 30% na privatne podatke; procenat promašaja u keš memoriji je 2% za privatne podatke, odnosno 5% za deljene podatke. Pretpostavlja se da je cena promašaja za privatne podatke uvek fiksna i iznosi 20 procesorskih ciklusa. Na Sl. 1-3 prikazan je uticaj kašnjenja u pristupu deljenim podacima u memoriji na iskorišćenost procesora. Dobijeni rezultati pokazuju da će, i pored keširanja, procesor 50% vremena izvršavanja biti blokiran usled blokirajućih promašaja kada je kašnjenje u pristupu memoriji 100 procesorskih ciklusa. Ovi rezultati ukazuju na izuzetan značaj tehnika za prikrivanje kašnjenja u pristupu memoriji.



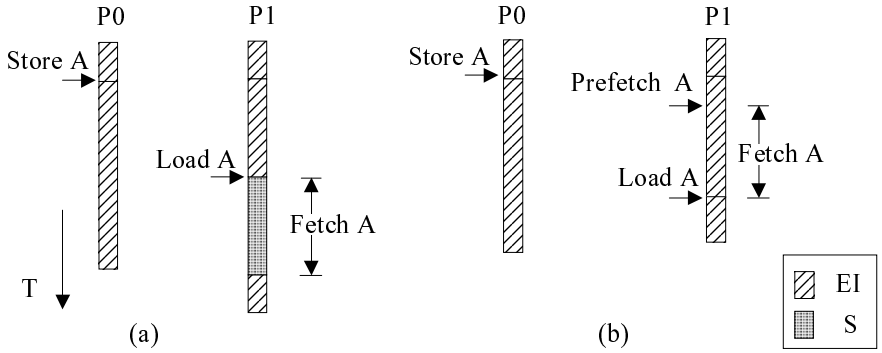
Sl. 1-3. Iskorišćenje procesora u funkciji kašnjenja u pristupu memoriji.
Opis: U (*Utilization*) – iskorišćenje procesora; ML (*Memory Latency*) – kašnjenje u pristupu memoriji predstavljeno brojem procesorskih ciklusa.

1.2 Tehnike za prikrivanje kašnjenja u pristupu memoriji

U osnovi svih tehnika za prikrivanje kašnjenja u pristupu memoriji je pokušaj da se pristup memoriji preklopi sa korisnim izračunavanjem i/ili drugim pristupima. Prikrivanje kašnjenja u pristupu memoriji postiže se relaksiranim modelima memorijske konzistencije (*relaxed memory consistency models*), dohvatanjem podataka unapred (*prefetching*), prosleđivanjem podataka (*forwarding*), kao i više-konteksnom obradom (*multithreading*). U poslednjih nekoliko godina veliki broj istraživanja je posvećen softverski kontrolisanim tehnikama dohvatanja podataka unapred i prosleđivanja podataka budućim korisnicima ([Mowry94], [Mowry*97], [Shafi*97], [Tulls*95], [Tranc*96], [Koufa*96], [Skepp*95]).

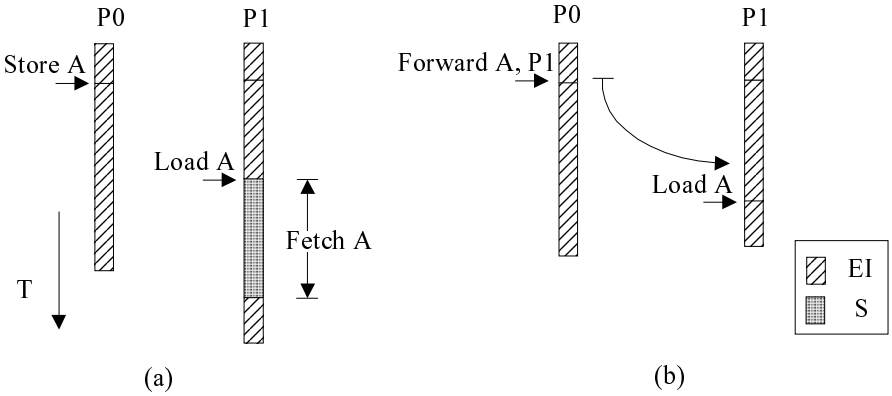
Kod softverski kontrolisanog dohvatanja podataka unapred, procesor izvršava `prefetch` instrukciju koja inicira dohvatanje bloka podataka koji se ne nalazi u lokalnoj keš memoriji, a za koji se predviđa da će biti potreban. Pri tom, procesor nastavlja sa izvršavanjem programske niti. Ukoliko je zahtev iniciran dovoljno pre trenutka stvarnog korišćenja, traženi blok će se naći u keš memoriji posmatranog procesora u trenutku pravog korišćenja, čime se izbegava blokiranje procesora usled promašaja u keš memoriji. Dohvatanje podataka unapred se koristi kako kod jednoprocesorskih sistema, tako i kod multiprocesorskih sistema. Jednostavan primer na Sl. 1-4 ilustruje kako dohvatanje podataka unapred doprinosi poboljšanju performanse. U polaznom slučaju, procesor P1 se blokira prilikom čitanja podatka `A` koji se nalazi u keš memoriji procesora P0. Ukoliko procesor P1 inicira dohvatanje

podatka A dovoljno pre trenutka stvarnog korišćenja podataka, tako da podatak u vreme izvršavanje instrukcije `Load` bude u keš memoriji procesora P1, izbegava se blokiranje procesora usled promašaja u keš memoriji.



Sl. 1-4. Ilustracija poboljšanja performanse korišćenjem softverski kontrolisanog dohvatanja podataka unapred.
Opis: (a) primer bez dohvatanja podataka unapred; (b) primer sa dohvatanjem podataka unapred. EI (*Executing Instructions*) – izvršavanje instrukcija, S (*Stalled Waiting for Data*) – procesor je blokiran usled blokirajućeg promašaja u keš memoriji, T (*Time*) - vreme.

Softverski kontrolisano prosleđivanje podataka podrazumeva da procesor proizvođač podataka (*producer*), nakon završetka obrade podataka, pored ažuriranja lokalne kopije podatka u svojoj keš memoriji, posebnom instrukcijom inicira slanje kopija podatka u keš memorije procesora koji su identifikovani kao budući potrošači podataka (*consumers*). Na taj način, potencijalni korisnici podataka će naći podatke u svojoj keš memoriji, čime se izbegava zaustavljanje procesora usled promašaja u keš memoriji. Na Sl. 1-5 ilustrovano je kako softverski kontrolisano prosleđivanje podataka utiče na vreme izvršavanja programske niti.



Sl. 1-5. Ilustracija poboljšanja performanse korišćenjem softverski kontrolisanog prosleđivanja podataka.
Opis: (a) primer bez prosleđivanja podataka; (b) primer sa prosleđivanjem podataka. EI – izvršavanje instrukcija, S – procesor je blokiran usled blokirajućeg promašaja u keš memoriji, T (*Time*) - vreme.

Poredeći dohvatanje podataka unapred i prosleđivanje podataka budućim korisnicima, treba napomenuti da dohvatanje podataka unapred omogućuje eliminisanje svih tipova promašaja u keš memoriji (*cold*, *capacity*, *coherence*), dok prosleđivanje podataka može da eliminiše samo promašaje nastale usled akcija za održavanje keš koherencije (*coherence*). Dohvatanje podataka unapred nije primenljivo ukoliko adresa podatka nije poznata dovoljno pre trenutka

stvarnog korišćenja, odnosno može pogoršati performanse u slučaju da je dohvatanje unapred inicirano pre završetka obrade podatka od strane procesora proizvođača podatka. Za deljene podatke koji se modifikuju (*write shared data*), prosleđivanje podataka je efikasnije, jer se podaci prosleđuju nakon završetka obrade. Međutim, podrška prevodioca je mnogo složenija, posebno identifikacija procesora budućih korisnika podataka. Takođe, postoji problem implementacije eksplicitnog prosleđivanja na magistrali.

1.3 Predmet teze

Istraživanja efikasnosti dohvatanja podataka unapred u CC-NUMA multiprocesorskim sistemima pokazala su da ova tehnika može eliminisati između 50% i 80% polaznog vremena čekanja usled pristupa memoriji. Međutim, pokazalo se da ova tehnika nije naročito efikasna u eliminisanju promašaja u keš memoriji prilikom pristupa pravim deljenim podacima (*true shared data*). Pored toga, istraživanje prikazano u radu [Tulls*95] pokazalo je da ova tehnika ne daje očekivane rezultate u multiprocesorskim sistemima koji su bazirani na zajedničkoj magistrali. Glavni razlozi za ovakve rezultate su: (a) dodatni promašaji u keš memoriji usled konflikta između tekućeg i budućeg radnog opterećenja, (b) nemogućnost da se redukuje saobraćaj usled deljenja podataka, (c) slabo prikrivanje kašnjenja u pristupu deljenim podacima i (d) povećavanje saobraćaja na magistrali.

Sa druge strane, efikasnost prosleđivanja podataka je do sada ispitivana samo na CC-NUMA arhitekturama, dok njena efikasnost u multiprocesorima sa zajedničkom magistralom nije ispitana. Međutim, kompleksnost zahtevane podrške prevodioca i implementacije prosleđivanja na magistrali u velikoj meri ograničavaju primenu ove tehnike kod SMP sistema sa zajedničkom magistralom.

U ovoj disertaciji predlaže se nova tehnika za prikrivanje kašnjenja u pristupu memoriji kod SMP sistema sa zajedničkom magistralom, nazvana injektiranje u keš memoriju (*cache injection*). Koristeći osobine zajedničke magistrale, predložena tehnika eliminiše neke od nedostataka postojećih tehnika kao što su: negativan uticaj na deljene podatke, kontencija na magistrali, kompleksnost podrške prevodioca i cena koja se plaća usled umetanja novih instrukcija. Međutim, predložena tehnika ne isključuje primenu postojećih i ima za cilj podizanje sveukupne efikasnosti tehnika za prikrivanje kašnjenja. Treba napomenuti da primena ove tehnike nije ograničena samo na SMP sisteme sa zajedničkom magistralom, i da se uz odgovarajuće izmene može implementirati i u drugim sistemima, npr. CC-NUMA sistemima sa interkonekcionom mrežom tipa prsten (*ring*) ili rešetka (*mesh*).

Kod tehnike injektiranja procesor korisnik podataka instrukcijom `OpenWindow` inicijalizuje specijalnu tabelu injektiranja koja je deo keš kontrolera, u skladu sa očekivanim potrebama. Tokom *snooping* faze ciklusa čitanja ili pisanja na magistrali, keš kontroler proverava sadržaj tabele injektiranja; ukoliko se adresa keš bloka tekuće transakcije na magistrali nalazi u tabeli injektiranja, procesor prihvata taj blok u svoj keš. Postoje dva osnovna scenarija kada dolazi do injektiranja: tokom ciklusa čitanja i tokom ciklusa upisa na magistrali.

Injektiranje tokom ciklusa čitanja: kada više procesora čita iste podatke (*read only data*), onda mehanizam injektiranja podrazumeva da oni inicijalizuju svoje tabele injektiranja pre trenutka stvarnog korišćenja podataka. Kada prvi od procesora čita podatak, imaće promašaj u keš memoriji i iniciraće dohvatanje deljenog keš bloka. Tokom ciklusa čitanja, svi ostali procesori koji imaju validan ulaz u tabeli injektiranja sa adresom tog bloka, prihvataju podatak u svoju

keš memoriju. Tako, tokom jednog ciklusa na magistrali svi procesori prihvataju blok u svoju keš memoriju, a samo jedan od njih će biti blokiran, tj. vidi promašaj u keš memoriji.

Injektiranje tokom ciklusa upisa: kada se deljeni podaci modifikuju (*write shared data*), mehanizam injektiranja podrazumeva akcije na strani procesora proizvođača podataka i na strani procesora potrošača podataka. Kao u prethodnom slučaju, procesori potrošači podataka inicijalizuju svoje tabele injektiranja pre trenutka stvarnog korišćenja podataka. Sa druge strane, procesor proizvođač podataka po završetku obrade keš bloka inicira ciklus upisa u memoriju instrukcijom `Update`. Svi procesori koji imaju validan ulaz u svojim tabelama injektiranja prihvataju blok u svoju keš memoriju.

Efikasnost mehanizma injektiranja proizilazi iz eliminisanja promašaja usled čitanja i smanjivanja saobraćaja na zajedničkoj magistrali. Dodatna hardverska kompleksnost je minimalna i podrazumeva podršku instrukcijama `OpenWindow`, `CloseWindow` (invalidacija odgovarajućeg ulaza u tabeli injektiranja) i `Update` i implementaciju tabele injektiranja u keš kontroleru. Pored toga, preliminarne analize ukazuju da bi u nekim slučajevima složenost podrške prevodioca umetanju dodatnih instrukcija bila manja u odnosu na postojeće tehnike.

1.4 Metod rada

U cilju verifikacije teze koristi se simulaciona analiza. Simulaciona analiza se bazira na korišćenju alata za simulaciju multiprocesorskih sistema Limes [Magdic97]. Kao radno opterećenje koristi se niz originalno razvijenih paralelnih test programa i realne paralelne aplikacije iz skupa SPLASH-2 koje su razvijene na Univerzitetu Stanford [WooO*95]. Posmatraju se polazni programi i programi modifikovani tako da podrže injektiranje. Instrukcije za podršku injektiranju se umeću ručno na osnovu statičke analize kôda paralelnih aplikacija.

Preliminarna procena efikasnosti tehnike injektiranja proverava se korišćenjem PRAM-MESI simulatora idealnog memorijskog podsistema koji podržava MESI invalidacioni protokol za održavanje koherencije keš memorije. Kao mera efikasnosti koristi se ukupni procenat promašaja u keš memoriji i saobraćaj na magistrali. Efikasnost mehanizma injektiranja proverava se korišćenjem MESI-SPLIT simulatora realnog multiprocesorskog sistema sa zajedničkom magistralom koja podržava razdvojene transakcije (*split-transactions bus*). Za održavanje koherencije koristi se MESI *write-back* invalidacioni protokol. Kao mera performanse koristi se ukupno vreme izvršavanja (*Execution Time*) i ukupno vreme blokiranja tokom čitanja (*Read Stall Time*).

1.5 Osnovni rezultati

Primarni rezultat ove teze je predlog nove tehnike za prikrivanje kašnjenja u pristupu memoriji kod multiprocesora sa zajedničkom magistralom. Predložena tehnika je evaluirana korišćenjem simulacione analize bazirane na realnom izvršavanju paralelnih aplikacija. Takođe, data je kvalitativna analiza primene tehnike injektiranja na primeru originalnih test programa.

Sekundarni rezultati ove teze su sledeći:

(a) detaljan pregled relevantnih istraživanja iz oblasti softverski kontrolisanih tehnika za prikrivanje kašnjenja u pristupu memoriji kod multiprocesora sa deljenom memorijom;
(b) implementacija PRAM-MESI simulatora idealnog memorijskog podsistema multiprocesora sa deljenom memorijom, za brzu verifikaciju predloženih ideja;
(c) implementacija MESI-SPLIT simulatora realnog memorijskog podsistema SMP sistema sa zajedničkom magistralom koja podržava razdvojene transakcije čitanja;
(d) razvoj originalnih test paralelnih programa LTEST, BTEST, PC, MM i Jacobi koji demonstriraju tipična radna opterećenja paralelnih računara;
(e) instrumentacija koda paralelnih test programa i aplikacija iz skupa SPLASH-2.

1.6 Organizacija i sadržaj teze

U poglavlju 2 dat je pregled postojećih tehnika za prikrivanje kašnjenja u pristupu memoriji. Najpre je objašnjena suština i način primene softverski kontrolisanih tehnika za dohvatanje podataka unapred i prosleđivanje podataka budućim korisnicima. Nakon toga, dat je pregled najznačajnijih istraživanja ovih tehnika.

U poglavlju 3 opisani su osnovni motivi za uvođenje nove tehnike kroz analizu osobina postojećih tehnika i rezultata do sada sprovedenih istraživanja. Nakon toga, objašnjena je suština mehanizma injektiranja i ilustrovana primena ovog mehanizma na jednostavnim test programima od interesa. Na kraju, dat je kratak osvrt na moguću podršku prevodioca mehanizmu injektiranja, a takođe i kompleksnost hardverske implementacije potrebnih resursa.

U poglavlju 4 opisana je eksperimentalna metodologija korišćena u verifikaciji predloženog mehanizma injektiranja. Najpre je opisana organizacija i funkcionisanje programskog alata za simulaciju multiprocesorskih sistema Limes. Zatim, opisana je organizacija i struktura keš kontrolera memorijskog podsistema multiprocesora sa zajedničkom magistralom. Na kraju, dat je kratak osvrt na aplikacije koje se koriste kao radno opterećenje, a takođe i postupak umetanja instrukcija za podršku mehanizmu injektiranja.

U poglavlju 5 dat je prikaz rezultata sprovedene simulacione analize, posebno za sve razmatrane paralelne test programe i SPLASH-2 aplikacije. Za svaku aplikaciju dat je kratak opis uključujući i umetanje instrukcija za podršku injektiranju; nakon toga dati su rezultati simulacione analize bazirane na PRAM-MESI i MESI-SPLIT simulatorima.

Na kraju, u poglavlju 6 dat je zaključak koji uključuje kratak pregled teze i osvrt na buduća istraživanja.

Poglavlje 2

Postojeće tehnike za prikrivanje kašnjenja u pristupu memoriji

U ovom poglavlju opisano je dohvatanje podataka unapred i prosleđivanje podataka budućim korisnicima u odeljcima 2.1 i 2.2, redom. Pregled relevantnih istraživanja u oblasti softverski kontrolisanog dohvatanja podataka unapred i prosleđivanja podataka dat je u odeljku 2.3.

2.1 Dohvatanje podataka unapred

Primena tehnike *prefetching* podrazumeva dohvatanje podataka za koje se očekuje da će biti korišćeni, tako da budu bliže posmatranom procesoru u hijerarhiji memorijskog podsistema pre trenutka stvarnog korišćenja. U idealnom slučaju podaci će se naći u keš memoriji procesora neposredno pre trenutka stvarnog korišćenja, čime se izbegava blokiranje procesora usled promašaja u keš memoriji. Dohvatanje podataka unapred može se primeniti kako kod jednoprocesorskih, tako i kod multiprocesorskih sistema.

U multiprocesorskim sistemima tehnike za dohvatanje podataka unapred se mogu klasifikovati, prema tome da li su podaci nakon dohvatanja vidljivi protokolima za održavanje koherencije ili ne, na nevezane (*non—binding prefetching*) i vezane (*binding prefetching*). Kod vezanog dohvatanja podataka unapred odredišta podataka su obično registri opšte namene ili poseban bafer, pa tako postoji opasnost da dohvaćena kopija podatka postane neažurna ukoliko neki drugi procesor promeni podatak u intervalu između pristizanja podatka i stvarnog korišćenja. Kako je očuvanje korektnosti izvršavanja programa od primarnog interesa, vezano dohvatanje podataka unapred ima ograničenu primenu. Danas se dominantno koristi nevezano dohvatanje podataka unapred, kod koga je korektnost obezbeđena time što podaci ostaju vidljivi protokolima za održavanje koherencije, tj. podaci se smeštaju u keš memoriju. U ovoj tezi se podrazumeva nevezano dohvatanje podataka unapred.

Dva najvažnija pitanja vezana za dohvatanje podataka unapred su: (a) koje podatke treba dohvatiti unapred i (b) kada treba inicirati dohvatanje. Treba napomenuti da je dohvatanje podataka unapred moguće samo ukoliko se adresa podatka može odrediti pravovremeno; u suprotnom slučaju, ukoliko se adresa podatka može izračunati tek neposredno pre trenutka

korišćenja, dohvatanje podataka unapred nema smisla. Kao mera uspešnosti primene tehnike dohvatanja podataka unapred uzima se pokrivenost (*coverage*), koja pokazuje koji procenat originalnih promašaja u keš memoriji može biti eliminisan, odnosno pretvoren u pogotke, primenom tehnike dohvatanja podataka unapred. Međutim, pokrivenost nije jedina mera uspešnosti dohvatanja podataka unapred; vrlo je važno izbeći nepotrebno dohvatanje podataka unapred (*unnecessary prefetching*), koje se javlja u slučaju dohvatanja podataka koji nikad neće biti korišćeni ili podataka koji se već nalaze u kešu. Pored toga, efikasnost ove tehnike u velikoj meri zavisi i od trenutka iniciranja dohvatanja podataka unapred. Tako, ukoliko se dohvatanje podataka inicira prerano, postoji opasnost da posmatrani keš blok bude invalidovan pre trenutka stvarnog korišćenja, ako neki drugi procesor upiše u posmatrani keš blok ili da bude izbačen iz keš memorije usled politike zamene. Sa druge strane, ukoliko se dohvatanje podataka inicira prekasno, onda procesor dolazi do tačke stvarnog korišćenja pre nego što podaci pristignu u keš memoriju. U tom slučaju, tehnika *prefetching* samo delimično prikriva kašnjenje u pristupu memoriji.

Dohvatanje podataka unapred može biti hardverski ili softverski kontrolisano (*hardware-* i *software-controlled prefetching*), prema kriterijumu odgovornosti za iniciranje dohvatanja podataka unapred. Kod hardverskog dohvatanja podataka unapred postoji poseban hardverski resurs koji je odgovoran da na osnovu analize adresnih tragova tokom izvršavanja programa predvidi buduće pristupe memoriji i inicira dohvatanje podataka, dovoljno pre trenutka stvarnog korišćenja. Kod softverskog dohvatanja podataka unapred odgovornost za donošenje odluka o tome šta i kada dohvatiti ima prevodilac, odnosno programer (manje poželjna varijanta). Prevodilac na osnovu statičke analize koda umeće posebne `prefetch` instrukcije koje iniciraju dohvatanje podataka bez zaustavljanja izvršavanja tekuće programske niti.

U odeljku 2.1.1 data je definicija i pregled glavnih istraživanja u oblasti hardverski kontrolisanog dohvatanja podataka unapred, mada to nije od direktnog interesa za tezu. U odeljku 2.1.2 date su teorijske osnove softverski kontrolisanog dohvatanja podataka unapred.

2.1.1 Hardverski inicirano dohvatanje podataka unapred

Hardversko dohvatanje podataka unapred podrazumeva postojanje posebnog resursa koji prikuplja informacije o pristupima memoriji tokom izvršavanja programa. Na osnovu prikupljenih informacija i očekivanja da se detektovani uzorci obraćanja memoriji ponavljaju, hardver određuje koje podatke treba dohvatiti unapred, a takođe i trenutak kada treba inicirati zahtev. Pored zahteva za velikim faktorom pokrivanja, minimiziranjem broja nepotrebnih dohvatanja i pravovremenim iniciranjem zahteva, hardversko dohvatanje podataka unapred mora biti prihvatljivo u pogledu kompleksnosti, odnosno zahtevane površine na čipu i ne sme uticati na povećavanje osnovnog ciklusa takta procesora, tj. ne sme biti na kritičnoj stazi za podatke.

Najjednostavniji vid hardverskog dohvatanja podataka unapred predstavlja korišćenje velikih keš blokova, čime se eksploatiše postojanje prostorne lokalnosti. Efikasnost ovakvog pristupa zavisi od toga koliko vremena protekne od pristupa prvoj reči u keš bloku do pristupa ostalim rečima u bloku. Ukoliko je to vreme kratko, tj. manje od vremena potrebnog za učitavanje celog bloka u keš memoriju, onda je efikasnost ovakvog pristupa mala jer je zahtev za dohvatanje podataka iniciran prekasno. Pored toga, u multiprocesorskim sistemima povećavanje dužine keš bloka negativno utiče na povećavanje broja promašaja u keš memoriji usled lažnog deljenja podataka (*false sharing*). Jedna moguća varijanta hardverskog dohvatanja podataka unapred jeste dohvatanje susednog bloka (OBL - *One Block Ahead*), koja podrazumeva da se pristupom bloku B_i inicira dohvatanje bloka B_{i+1}. Postoji nekoliko

varijanti ovakvog pristupa koje se među sobom razlikuju po trenutku iniciranja dohvatanja narednog bloka: u slučaju promašaja prilikom pristupa bloku B_i, u slučaju bilo kakvog pristupa bloku B_i, itd. Proširenjem ovog pristupa može se inicirati dohvatanje nekoliko narednih blokova, npr. B_{i+1}, B_{i+2}, B_{i+3}, [Dahlg*95].

Sledeći kvalitativni korak je pokušaj detektovanja situacija kada se susedni pristupi ne odnose na susedne memorijske lokacije. Predložena je tabela istorije koja čuva adresu prethodnog pristupa za posmatranu instrukciju; selekcija odgovarajućeg ulaza tabele se vrši na osnovu vrednosti programskog brojača. Na osnovu adrese tekućeg pristupa i adrese prethodnog pristupa iz tabele istorije, izračunava se pomeraj između susednih pristupa i na osnovu toga inicira dohvatanje podatka sa adrese koja se dobija sabiranjem tekuće adrese i izračunatog pomeraja [FuPat91][FuPat*92]. Ovakav pristup je dobar u slučaju konstantnog pomeraja između uzastopnih pristupa; međutim, ukoliko je pristup okarakterisan promenljivim pomerajem, dolazi do dohvatanja nepotrebnih podataka i nepotrebnog saobraćaja na interkonekcionoj mreži. Proširivanjem tabele istorije tako da pored adrese prethodnog pristupa čuva i detektovani pomeraj može se uspešno realizovati dohvatanje podataka unapred i sa promenljivim pomerajem [BaerC*91].

Do sada su izložene tehnike koje su imale prvenstveno za cilj da poboljšaju analizu, odnosno tehnike za detektovanje podataka koje treba dohvatiti unapred. Međutim, problem pravovremenog iniciranja dohvatanja unapred je izuzetno važan. Jedan mogući pristup je uvođenje dodatnog programskog brojača (LAPC - *Lookahead Program Counter*) koji ide ispred tekućeg za onoliko instrukcija koliko je potrebno da bi se prikrilo kašnjenje u pristupu memoriji. Tabeli istorije se pristupa na osnovu vrednosti LAPC.

Zajedničko za sve hardverske tehnike jeste da je njihova efikasnost u velikoj meri ograničena visokom cenom hardverske implementacije sofisticiranih tehnika za analizu adresnih tragova tokom izvršavanja programa. Stoga, ni jedna od predloženih hardverskih tehnika do sada nije podržana u modernim mikroprocesorima i multiprocesorima, sa izuzetkom najjednostavnije OBL tehnike. Imajući u vidu kompleksnost i nefleksibilnost hardverskog dohvatanja podataka unapred i činjenicu da moderni mikroprocesori već sadrže instrukcije za podršku softverskom dohvatanju podataka unapred, može se očekivati dalji rast značaja softverskih tehnika.

2.1.2 Softverski inicirano dohvatanje podataka unapred

Softverski kontrolisano dohvatanje podataka unapred podrazumeva da je skup instrukcija procesora proširen posebnom `prefetch` instrukcijom. Ova instrukcija specificira adresu keš bloka koji se želi dohvatiti unapred. Izvršavanjem ove instrukcije inicira se ciklus čitanja keš bloka, ukoliko se on već ne nalazi u lokalnoj keš memoriji. Pri tom, ne blokira se izvršavanje programske niti, već se nastavlja sa izvršavanjem sledeće instrukcije. Ukoliko se specificirani keš blok već nalazi u keš memoriji, `prefetch` instrukcija je bez dejstva. Pored toga, kod većine komercijalnih implementacija, `prefetch` instrukcija koja izaziva *page fault* se ignoriše. Na Sl. 2-1 je ilustrovana primena `prefetch` instrukcije. Posmatraju se dve programske niti koje se izvršavaju na procesorima P0 i P1 (Sl. 2-1a). Instrukcija `rd x` označena sa (*) ne nalazi podatak u keš memoriji, pa inicira ciklus dohvatanja, a procesor se blokira čekajući na završetak dohvatanja keš bloka. Blokiranje procesora P1 se izbegava ukoliko se polazna sekvenca modifikuje tako da se dovoljno pre trenutka izvršavanja instrukcije označene sa (*) inicira dohvatanje podataka unapred `prefetch` (`pf`) instrukcijom (Sl. 2-1b).

Sl. 2-1. Softverski inicirano dohvatanje podataka unapred.

U multiprocesorskim sistemima sa invalidacionim protokolima za održavanje keš koherencije od interesa je primena ekskluzivnog dohvatanja podataka unapred. Ukoliko se očekuje da će procesor čitati posmatrani blok, inicira se obično dohvatanje podataka unapred; dohvaćeni blok je u stanju S (*Shared*), što znači da se kopija posmatranog bloka može nalaziti u keš memorijama drugih procesora u istom stanju. Ukoliko se očekuje da procesor upisuje u posmatrani keš blok, čak i ukoliko je prvi pristup čitanje, treba obezbediti da taj procesor bude ekskluzivni vlasnik posmatranog bloka, odnosno da invaliduje sve ostale kopije. Stoga, od interesa je uvođenje posebne `prefetch-ex` instrukcije koja je ista kao i obična `prefetch` instrukcija, s tim da je procesor nakon dohvatanja bloka njegov ekskluzivni vlasnik, tj. kopije u keš memorijama ostalih procesora su invalidovane. Primena ove instrukcije može dovesti do dvojake koristi u pogledu poboljšanja performanse. Prvo, ukoliko se podatak prvo čita, a potom modifikuje, primenom ekskluzivnog dohvatanja podataka unapred može se eliminisati kašnjenje zbog invalidacije ostalih kopija prilikom upisa; ovo može rezultovati ili ne skraćivanjem vremena izvršavanja, u zavisnosti od protokola memorijske konzistencije. Druga korist od primene ekskluzivnog dohvatanja je redukcija saobraćaja na interkonekcionoj mreži. Naime, u slučaju da se podatak prvo čita, na interkonekcionoj mreži se inicira transakcija čitanja, a nakon dohvatanja keš blok se proglašava deljenim (*Shared*); nakon upisa inicira se transakcija kojom procesor postaje ekskluzivni vlasnik keš bloka. Umesto toga, ekskluzivno dohvatanje podataka unapred inicira samo jednu transakciju. Na Sl. 2-2a prikazan je primer sa običnom `prefetch` instrukcijom koja dovlači podatak unapred, tako da instrukcija označena sa (*) vidi pogodak u keš memoriji. Međutim, sledeća instrukcija `wr x` inicira transakciju invalidacije. Ukoliko se umesto `prefetch` instrukcije koristi `prefetch-ex` (`pf-ex`) instrukcija, procesor P1 je ekskluzivni vlasnik keš bloka u trenutku upisa.

Sl. 2-2. Ekskluzivno dohvatanje podataka unapred.

Odgovornost za umetanje `prefetch` i `prefetch-ex` instrukcija ima programer ili prevodilac. Najatraktivniji pristup je primena algoritama u prevodiocu koji vrše analizu koji će podaci biti potrebni i umeću odgovarajuću instrukciju za dohvatanje podataka unapred, tako da podatak bude u keš memoriji u trenutku stvarnog korišćenja.

Međutim, dohvatanje podataka unapred može da dovede i do degradiranja performanse. Pre svega, umetanjem dodatnih instrukcija povećava se dužina koda, a samim tim i vreme izvršavanja programa. Takođe, blok koji se dohvata unapred može izbaciti keš blok koji je potreban u daljem izvršavanju. Na kraju, usled nesavršenosti algoritma za predviđanje budućih potreba, moguće je da se inicira dohvatanje nepotrebnih podataka, što rezultuje nepotrebnim saobraćajem na interkonekcionoj mreži. Pored navedenih problema, moguće su situacije koje dovode do degradiranja performanse, a odnose se na vreme iniciranja zahteva za dohvatanjem podataka unapred. U tekstu koji sledi prikazane su takve situacije.

Prvi slučaj odnosi se na prerano inicirano dohvatanje podataka unapred (*too early issued prefetching*) i javlja se kada potencijalni budući korisnik inicira dohvatanje podataka unapred pre nego što je proizvođač završio obradu podataka. Ova situacija je ilustrovana na Sl. 2-3. U polaznoj sekvenci procesor P0 dva puta uzastopno modifikuje *x*, a potom procesor P1 čita taj podatak (Sl. 2-3a). U sekvenci sa Sl. 2-3b procesor P1 inicira dohvatanje unapred podatka *x* koji se nalazi u keš memoriji procesora P0; međutim, nakon toga procesor P0 ponovo modifikuje podatak *x* i tako invaliduje kopiju u keš memoriji procesora P1. Rezultat ovakvog scenarija su nepotrebne operacije na interkonekcionoj mreži i nepotrebno izvršena instrukcija dohvatanja unapred. Pored toga, prerano dohvaćeni podatak može biti izbačen iz keš memorije usled politike zamene. Takva situacija je ilustrovana na Sl. 2-4. Podaci *x* i *y* se smeštaju u isti blok u keš memoriji, tako da će unapred dohvaćeni podatak *x* biti izbačen iz keš memorije, pre trenutka stvarnog korišćenja. Drugi slučaj odnosi se na zakasnelo dohvatanje podataka unapred (*too late issued prefetch*); u tom slučaju, pravo korišćenje podataka se dešava pre završetka započetog dohvatanja unapred, tako da se kašnjenje u pristupu memoriji samo delimično prikriva.

Sl. 2-3. Preuranjeno dohvatanje podataka unapred #1.

Sl. 2-4. Preuranjeno dohvatanje podataka unapred #2.

2.2 Prosleđivanje podataka budućim korisnicima

Prosleđivanje podataka budućim korisnicima (*data forwarding*) je softverska tehnika za prikrivanje komunikacionih kašnjenja u multiprocesorskim sistemima, bazirana na preklapanju pristupa memoriji sa izračunavanjem i/ili drugim pristupima memoriji. Kod prosleđivanja podataka, proizvođač podataka prosleđuje deljene podatke bliže procesorima budućim korisnicima podataka. U idealnom slučaju, procesori korisnici će pronaći podatke u svojoj privatnoj keš memoriji, čime se izbegava blokiranje procesora usled promašaja u keš memoriji.

Interesantno je prodiskutovati prosleđivanje podataka u odnosu na protokole za održavanje koherencije keš memorije bazirane na ažuriranju. Kod protokola sa ažuriranjem svaki upis u deljeni keš blok se prosleđuje svim procesorima koji u tom trenutku poseduju kopiju tog keš bloka. Na taj način, svaki upis u deljeni keš blok inicira transakciju na interkonekcionoj mreži, čime se znatno povećava saobraćaj. Ovakav pristup je posebno neefikasan kada jedan procesor više puta uzastopno upisuje u isti keš blok. Takođe, ažuriraju se samo procesori koji trenutno dele keš blok, a ne i oni koji bi eventualno u budućnosti mogli biti korisnici tog keš bloka. Kod tehnike *forwarding* prosleđivanje se inicira posebnom instrukcijom umetnutom od strane prevodioca ili programera, što omogućava fleksibilnost u smislu izbora koje podatke i kada prosleđivati. Takođe, prevodilac (ili programer) određuje buduće korisnike podataka, pa

se na taj način može inicirati ažuriranje procesora koji u tom trenutku ne poseduju kopiju posmatranog keš bloka; pored toga, moguće je invalidovati kopiju keš bloka procesora koji inicira prosleđivanje.

U ovom paragrafu opisana je jedna moguća implementacija asemblerske instrukcije `wr-forw` (*write and forward*) koja podržava prosleđivanje podataka. Prevodilac ubacuje ovakvu instrukciju umesto obične `wr` instrukcije ukoliko je to poslednji upis u keš blok. Predložena instrukcija specificira adresu keš bloka koji se prosleđuje; pored toga, instrukcijom se specificiraju procesori budući korisnici podataka. Ukoliko `wr-forw` instrukcija nađe specificirani keš blok u keš memoriji u stanju modifikovan (*Modified*), inicira se transakcija kojom se keš blok prosleđuje procesorima budućim korisnicima podataka, a blok postaje deljen (*Shared*). U multiprocesorima koji koriste protokole za održavanje koherencije keš memorije bazirane na direktorijumima, keš blok se prosleđuje čvoru koji sadrži direktorijum za taj keš blok (*home*), a odatle specificiranim procesorima za koje se veruje da će biti budući korisnici podataka. Ukoliko instrukcija `wr-forw` nađe keš blok u stanju S (*Shared*), ili se desi promašaj u keš memoriji, onda se prosleđuje samo modifikovana reč. U slučaju promašaja, odgovorni direktorijum će inicirati slanje celog bloka kako procesorima budućim korisnicima, tako i procesoru proizvođaču podataka.

Sledeći primeri prikazuju primene prosleđivanja podataka. Zbog jednostavnosti pretpostavljeno je da veličina keš bloka odgovara jednoj reči. Na Sl. 2-5 prikazan je primer koji ilustruje primenu prosleđivanja sa ciljem eliminisanja promašaja usled pravog deljenja podataka (*coherence misses*). Sekvenca koda sa Sl. 2-5a ilustruje pravo deljenje podataka. Instrukcija označena sa (*) neće pronaći podatak u svojoj keš memoriji, pa se procesor blokira, čekajući da se završi ciklus čitanja podataka. Ukoliko se poslednja instrukcija upisa `wr` zameni instrukcijom `wr-forw x, 1` (Sl. 2-5b), procesor P0, nakon upisa, inicira prosleđivanje podatka *x* u keš procesora P1. U idealnom slučaju procesor P1 će naći podatak u svom kešu, pa se tako izbegava blokiranje procesora usled promašaja u keš memoriji.

```
P0          P1                  P0                  P1
            rd x                                    rd x
wr x                            wr x
wr x                            wr-forw x, 1
            rd x *                                  rd x
        (a)                             (b)
```

Sl. 2-5. Primena prosleđivanja podataka u eliminisanju promašaja u keš memoriji usled pravog deljenja podataka.

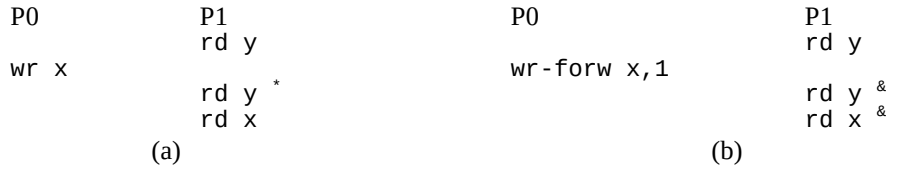
Na Sl. 2-6 prikazan je primer koji ilustruje primenu prosleđivanja u eliminisanju promašaja usled prvog pristupa podatku (*cold misses*). Na Sl. 2-6a prikazana je programska sekvenca koja ilustruje takvu situaciju. Instrukcija označena (*) neće pronaći podatak u svom kešu, pa će procesor biti blokiran. Analogno prethodnom primeru, zamena instrukcije `wr` instrukcijom `wr-forw x, 1` (Sl. 2-6b) omogućuje pogodak u keš memoriji.

```
P0          P1                  P0                  P1
wr x                            wr-forw x, 1
            rd x *                                  rd x
        (a)                             (b)
```

Sl. 2-6. Primena prosleđivanja podataka u eliminisanju promašaja u keš memoriji usled prvog pristupa podatku.

Primena prosleđivanja podataka može imati i negativne posledice. Pre svega, veliki problem predstavlja tačno uočavanje procesora budućih korisnika podataka; u slučaju neregularnog deljenja to je praktično nemoguće. Međutim, problemi mogu nastati čak i ako je podatak

ispravno prosleđen budućem korisniku. Tako, na primer, podaci mogu stići nakon trenutka kada su stvarno bili potrebni. U tom slučaju primena prosleđivanja prikriva deo komunikacionog kašnjenja, ali je problem što je procesor potrošač pre trenutka pristizanja podataka inicirao transakciju čitanja. Sledeći problem može nastati ako se prosleđeni podaci izbace iz keš memorije pre trenutka stvarnog korišćenja. Ovaj scenario je posebno realan u slučaju da su proizvodnja i potrošnja podataka vremenski dosta razdvojeni; u tom slučaju, odloženo prosleđivanje može biti od koristi. Međutim, ovakav scenario ne uvećava broj promašaja u keš memoriji u odnosu na slučaj bez primene prosleđivanja; gubitak se ogleda, pre svega, u nepotrebnoj transakciji na interkonekcionoj mreži i povećavanju broja instrukcija, ukoliko se umeće dodatna `forw` instrukcija i/ili dodatne instrukcije usled reorganizacije koda koja je neophodna da bi se podržalo prosleđivanje. Primena prosleđivanja može dovesti i do negativnog uticaja na ukupan broj promašaja u keš memoriji usled konflikta između blokova (*conflict misses*). Primer koji ilustruje takvu situaciju prikazan je na Sl. 2-7. Pretpostavlja se da podaci *x* i *y* pripadaju keš blokovima koji prave konflikt u keš memoriji. U polaznom primeru instrukcija `rd y` označena (*) nalazi podatak u svom kešu, za razliku od instrukcije `rd x`. Međutim, primena prosleđivanja dovodi do toga da obe instrukcije označene sa (&) vide promašaj u keš memoriji (Sl. 2-7b).



Sl. 2-7. Uticaj prosleđivanja podataka na promašaje usled konflikta u keš memoriji .

U do sada razmatranim instrukcijama `wr-forw` i `forw` pretpostavljeno je da podatak ostaje deljen u keš memoriji procesora proizvođača podataka. Međutim, u slučaju da procesor proizvođač neće koristiti taj keš blok mogu se uvesti dodatne instrukcije `wr-forw-inv` i `forw-inv` kojim bi proizvođač inicirao invalidaciju svoje kopije bloka koji prosleđuje. Ovo posebno može biti od koristi ako je tip deljenja takav da se blok prosleđuje procesoru koji je, takođe, proizvođač podataka (ekskluzivni vlasnik). U primeru na Sl. 2-8a prikazana je programska sekvenca za procesore P0 i P1 koja ilustruje navedeni scenario. Procesor P0 umesto instrukcije `wr x` koristi `wr-forw x`, pa procesor P1 nalazi podatak u svom kešu ali u stanju deljen (*Shared*). Stoga, on mora da inicira transakciju kojom invaliduje kopiju podatka u keš memoriji procesora P0 i postaje ekskluzivni vlasnik podatka. Međutim, ukoliko se koristi instrukcija koja pored prosleđivanja vrši i invalidaciju podatka, procesor P0 postaje ekskluzivni vlasnik podatka odmah nakon prihvatanja keš bloka (Sl. 2-8b); na ovaj način se smanjuje saobraćaj na interkonekcionoj mreži.

Sl. 2-8. Prosleđivanje sa invalidacijom lokalne kopije podatka kao rešenje za komunikaciju između procesora proizvođača.

Ranije je napomenuto da je u slučaju neregularnih komunikacija nemoguće unapred odrediti procesore buduće korisnike podataka. U takvim situacijama, od interesa može biti instrukcija `Update` kojom se inicira ažuriranje memorije čvora domaćina za taj blok u CC-NUMA sistemima, a u zavisnosti od organizacije magistrale i kod multiprocesora sa zajedničkom magistralom.

2.3 Pregled istraživanja od interesa za tezu

Velika fleksibilnost i efikasnost softverskih tehnika za prikrivanje kašnjenja u pristupu memoriji kod multiprocesorskih sistema sa deljenom memorijom i pojava prvih mikroprocesora koji skupom instrukcija podržavaju ove tehnike rezultuju velikim brojem istraživanja u ovoj oblasti. U ovom odeljku dat je pregled najznačajnijih istraživanja.

U odeljku 2.3.1 prikazano je jezgro algoritma programskog prevodioca za selektivno dohvatanje podataka unapred kod jednoprocesorskih sistema [Mowry94]. Ovo istraživanje se delom nastavlja na ranija istraživanja softverski iniciranog dohvatanja podataka unapred [Call*91], [Klaib*91]. Međutim, za razliku od prethodnih istraživanja algoritam za selektivno dohvatanje podataka je opštiji; takođe, predloženi algoritam je po prvi put implementiran u programskom prevodiocu. Modifikacijom polaznog algoritma za jednoprocesorske sisteme dobija se algoritam koji se koristi kod programskih prevodioca za multiprocesorske sisteme koji je prikazan u odeljku 2.3.2 [Mowry94]. Prikazani algoritmi prvenstveno redukuju broj promašaja u keš memoriji kod aplikacija koje su bazirane na nizovskim strukturama podataka. U odeljku 2.3.3 prikazano je istraživanje koje ima za cilj da predloži algoritme programskog prevodioca za podršku dohvatanju podataka unapred kod jednoprocesorkih i multiprocesorkih sistema za aplikacije koje su bazirane na rekurzivnim strukturama podataka [Mowry*97]. Kod svih navedenih istraživanja analiza efikasnosti je sprovedena na primeru CC-NUMA arhitektura. U odeljku 2.3.4 prikazana su istraživanja u kojima se analizira efikasnost "idealnog" algoritma za dohvatanje podataka unapred kod multiprocesora sa zajedničkom magistralom ([Tulls*93], [Tulls*95]).

Tehnika prosleđivanja podataka prvi put je implementirana kod DASH multiprocesora koji je početkom 90-tih razvijen na Univerzitetu Stanford [Lenos*92]. Prosleđivanje podataka je podržano instrukcijom `deliver` koja može u bilo kom trenutku inicirati slanje nekog keš bloka ka procesorima koji su specificirani posebnim bit vektorom. Međutim, u ovom istraživanju nije posebno analizirana efikasnost tehnike prosleđivanja podataka, niti je razmatrana podrška programskog prevodioca umetanju odgovarajućih instrukcija za prosleđivanje. U istraživanju koje je prikazano u odeljku 2.3.5 predloženo je jezgro algoritma programskog prevodioca za umetanje instrukcija prosleđivanja kod paralelnih programa baziranih na *doall* petljama ([Pouls*94], [Koufa*96]). U CC-NUMA arhitekturama, od interesa može biti ažuriranje glavne memorije ukoliko se očekuje da će podatak biti korišćen od nekog drugog procesora. U odeljku 2.3.6 prikazan je algoritam programskog prevodioca koji umeće instrukcije za ažuriranje glavne memorije nakon poslednjeg upisa u keš blok.

Veliki broj istraživanja razmatra efekat koji se dobija kombinovanjem softverskih tehnika *prefetching* i *forwarding*, pretežno u CC-NUMA multiprocesorima [Byrd*99]. Tako, u odeljku 2.3.7 prikazano je istraživanje u kome se analizira ubrzavanje izvršavanja kritičnih regiona kombinovanjem dohvatanja unapred i prosleđivanja podataka. U odeljku 2.3.8 prikazano je istraživanje sprovedeno na Univerzitetu Rice u kome se analizira efikasnost dohvatanja unapred i prosleđivanja podataka, odvojeno i u kombinaciji. U odeljku 2.3.9 prikazano je istraživanje u kome je predložen skup primitiva koji bi podržao dinamičko prilagođavanje tipa komunikacije u skladu sa karakteristikama aplikacije.

2.3.1 Algoritam za selektivno dohvatanje podataka unapred kod jednoprocesorskih sistema

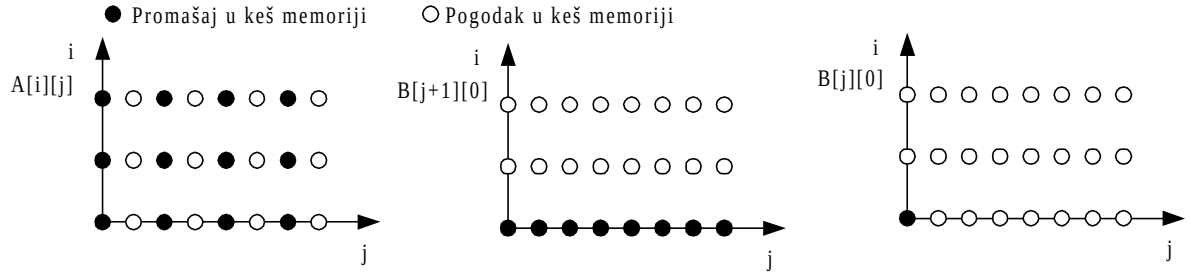
U doktorskoj tezi [Mowry94] predloženo je jezgro algoritma, implementiranog u programskom prevodiocu, za dohvatanje podataka unapred u aplikacijama baziranim na nizovskim strukturama podataka. Predloženi algoritam analizira promašaje u keš memoriji tokom izvršavanja programskih petlji. Jedan pristup dohvatanju podataka unapred podrazumeva da se svakoj memorijskoj referenci pridruži odgovarajuća `prefetch` instrukcija. Međutim, kako se veliki deo podataka kojim se pristupa već nalazi u lokalnoj keš memoriji, ovakav pristup bi doveo do pogoršanja performanse usled izvršavanja nepotrebnih `prefetch` instrukcija. Stoga, od interesa je algoritam za selektivno dohvatanje podataka unapred koji ima za cilj da eliminiše nepotrebno dohvatanje unapred, odnosno da detektuje samo one memorijske reference koje rezultuju promašajem u keš memoriji. U ovom odeljku razmatra se algoritam za selektivno dohvatanje podataka unapred kod jednoprocesorskih sistema.

Algoritam se sastoji od tri osnovna koraka: (a) analiza lokalnosti, sa ciljem da se za svaku instrukciju koja pristupa memoriji odrede pristupi koji će verovatno rezultovati promašajem u keš memoriji, (b) reorganizacija koda, sa ciljem da se izoluju pristupi za koje se očekuje da uzrokuju promašaj u keš memoriji i tako eliminiše dodavanje nepotrebnih uslovnih naredbi u tela petlji, i (c) umetanje `prefetch` instrukcija u kôd, tako da podaci budu prisutni u keš memoriji pre trenutka stvarnog korišćenja.

Algoritam je ilustrovan na primeru koji je dat na Sl. 2-9. Uvedene su sledeće pretpostavke: veličina keš memorije je 8KB, veličina keš bloka je 4 reči, elementi nizova su duple reči, a kašnjenje u pristupu memoriji iznosi 100 procesorskih ciklusa. Na Sl. 2-10 pokazani su promašaji u keš memoriji tokom izvršavanja test primera; svaki čvor na slici odgovara jednoj iteraciji, pri čemu *x* osa odgovara petlji sa indeksom *j*, a *y* osa petlji sa indeksom *i*.

```
for(i=0; i<3; i++)
       for(j=0; j<100; j++)
              A[i][j] = B[i][j] + B[j+1][0]
```

Sl. 2-9. Segment koda korišćen za ilustraciju algoritma za selektivno dohvatanje podataka unapred.



Sl. 2-10. Ilustracija promašaja u keš memoriji tokom izvršavanja test primera.

Analiza lokalnosti

Analiza lokalnosti koristi pojmove višestruko korišćenje (*reuse*) i lokalnost (*locality*). *Reuse* se dešava uvek kada se posmatranom podatku pristupa više puta. Višestruko korišćenje podataka je vrlo važno za razumevanje ponašanja keš memorije, jer podatak može biti u keš memoriji samo ako se tom keš bloku pristupalo pre posmatrane instrukcije. Međutim,

višestruko korišćenje podataka ne rezultuje obavezno pogotkom u keš memoriji, jer keš blok može biti izbačen iz keš memorije u intervalu između dva pristupa. Ukoliko se višestruko korišćeni podatak nađe u keš memoriji, kaže se da posmatrani pristup ima lokalnost. Dakle, za keš memoriju čija veličina odgovara veličini adresnog prostora, svi višestruko korišćeni podaci ispoljavaju osobinu lokalnosti; međutim, u opštem slučaju kada je kapacitet keš memorije ograničen, višestruko korišćeni podaci ne ispoljavaju obavezno osobinu lokalnosti. Analiza lokalnosti ima za cilj da identifikuje memorijske pristupe koji će rezultovati promašajem u keš memoriji. Prvi korak u analizi lokalnosti je detekcija višestrukih korišćenja podataka unutar ugnježdenih petlji (*reuse analysis*). Drugi korak je određivanje višestrukih korišćenja koja će rezultovati pogotkom u posmatranom kešu (*localized iteration space*).

Detekcija višestruko korišćenih podataka ima za cilj da otkrije memorijske pristupe elementima nizova koji se odnose na isti keš blok. Postoje tri vrste višestrukog korišćenja: vremensko (*temporal reuse*), prostorno (*spatial reuse*) i grupno (*group reuse*). U posmatranom primeru memorijska referenca B[j][0] ispoljava vremensko višestruko korišćenje u spoljašnjoj petlji (sa indeksom *i*) jer spoljašnje iteracije pristupaju istim lokacijama. Memorijska referenca A[i][j] ispoljava prostorno višestruko korišćenje unutar petlje sa indeksom *j*, jer se istom keš bloku pristupa unutar dve susedne iteracije. Na kraju, različiti pristupi B[j][0] i B[j+1][0] ispoljavaju grupno višestruko korišćenje jer pristupaju istim podacima. Tačno određivanje podataka sa višestrukim korišćenjem unutar petlji može biti vrlo skupo, to jest zahteva složenu obradu. Umesto toga, koristi se približni postupak baziran na odgovarajućem matematičkom modelu. Ugnježdene petlje se predstavljaju *n*-dimenzionalnim prostorom iteracija, pri čemu spoljašnja petlja predstavlja prvu dimenziju prostora, a poslednja ugnježdena petlja poslednju dimenziju prostora.

Određivanje vremenskog višestrukog korišćenja se svodi na određivanje pristupa istoj lokaciji od strane različitih iteracija. Taj problem se može preslikati na problem određivanja indeksa petlji za koje su indeksi nizova posmatrane memorijske reference jednaki. Indeksi memorijskih referenci se predstavljaju funkcijom koja prevodi *n*-dimenzionalni prostor petlji (*n* predstavlja dubinu ugnježdavanja) u *d*-dimenzionalni prostor indeksa niza (*d* predstavlja broj dimenzija niza). Tako, u posmatranom primeru memorijske reference se predstavljaju na sledeći način: B[j][0] kao $B(\begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\cdot\begin{bmatrix}i\\ j\end{bmatrix})$, A[i][j] kao $A(\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\cdot\begin{bmatrix}i\\ j\end{bmatrix})$, a B[j+1][0] kao $B(\begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\cdot\begin{bmatrix}i\\ j\end{bmatrix}+\begin{bmatrix}1\\ 0\end{bmatrix})$. Dve različite iteracije (i_1, j_1) i (i_2, j_2) pristupaju istom podatku u slučaju memorijske reference B[j+1][0] ako je ispunjen sledeći uslov:

$$\begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\cdot\begin{bmatrix}i_1\\ j_1\end{bmatrix}+\begin{bmatrix}1\\ 0\end{bmatrix}=\begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\cdot\begin{bmatrix}i_2\\ j_2\end{bmatrix}+\begin{bmatrix}1\\ 0\end{bmatrix}, \text{ ili } \begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\cdot\begin{bmatrix}i_1-i_2\\ j_1-j_2\end{bmatrix}=\begin{bmatrix}0\\ 0\end{bmatrix}.$$

Ova jednačina je tačna uvek kada je $j_1=j_2$, bez obzira na vrednost i_1 i i_2. Prema tome vremensko višestruko korišćenje se dešava na nivou spoljašnje petlje.

Sličan postupak se primenjuje za određivanje prostornog višestrukog korišćenja, s tim da sada ne moraju biti jednaki svi indeksi posmatranog niza. Tako, ukoliko se analizira višestruko korišćenje u slučaju niza A[i][j], dobija se da je odgovarajuća matrica $H=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$; međutim kada se analizira prostorno višestruko korišćenje, a elementi matrice su smešteni u memoriji

po vrstama, odgovarajuća matrica za prostornu analizu je $H_s=\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}$; dobijena matrica ukazuje da se prostorno višestruko korišćenje ispoljava u unutrašnjoj petlji (sa indeksnom *j*).

Što se tiče grupnog višestrukog korišćenja, utvrđeno je da se može iskoristiti samo ako se reference uniformno generišu, kao u posmatranom primeru u slučaju referenci B[j][0] i B[j+1][0]. Mada su uniformno generisane reference kandidati za grupno višestruko korišćenje, moguće je da nikad ne pristupaju istim podacima, kao u primeru referenci C[2i][j] i C[2i+1][j]. Stoga, potrebno je uraditi dodatnu proveru koja bi detektovala takve slučajeve. Dodatna provera koristi matricu transformacije da bi se odredilo da li se reference odnose na iste podatke ili ne. Grupno višestruko korišćenje se može javiti u dve varijante: različite reference pristupaju istom podatku (*group-temporal reuse*) i različite reference pristupaju istom keš bloku ali nikad istom podatku (*group-spatial reuse*).

U posmatranom primeru A[i][j] ima prostorno višestruko korišćenje po unutrašnjoj petlji, B[j][0] i B[j+1][0] dele grupno višestruko korišćenje, a takođe imaju vremensko višestruko korišćenje u odnosu na spoljašnju petlju.

Nakon određivanja višestrukog korišćenja, sledeći korak je analiza sa ciljem da se odredi da li višestruko korišćenje rezultuje lokalnošću. Postojanje lokalnosti zavisi od broja iteracija, odnosno količine podataka koji se dohvataju u keš memoriju, i parametara keš memorije kao što su veličina, asocijativnost i algoritam zamene. U posmatranom primeru B[j][0] ispoljava vremensko višestruko korišćenje u odnosu na spoljašnju petlju. Ukoliko je količina podataka kojima se pristupa u unutrašnjoj petlji velika u odnosu na veličinu keš memorije (na primer, gornja granica za *j* je 10000), podaci će sa velikom verovatnoćom biti izbačeni iz keš memorije pre sledeće spoljašnje iteracije. Naravno, nije uvek moguće tačno odrediti da li podaci ostaju u keš memoriji ili ne, pre svega usled simboličkih granica petlji i osobina keš memorije. Umesto pokušaja da se tačno odredi koje višestruko korišćenje rezultuje lokalnošću, uvodi se pojam lokalizovanog prostora iteracije (*localized iteration space*). Lokalizovani prostor iteracija predstavlja skup unutrašnjih petlji kod kojih je količina podataka kojima se pristupa u jednoj iteraciji manja od veličine keš memorije. Ukoliko se broj iteracija ne može odrediti u vreme prevođenja programa, pretpostavlja se mali broj iteracija sa ciljem eliminacije nepotrebnih dohvatanja podataka unapred. Višestruko korišćenje rezultuje osobinom lokalnosti ako leži unutar lokalizovanog prostora iteracija.

Tako, u posmatranom primeru granice petlji su poznate, pa se lako može odrediti kapacitet podataka kojima se pristupa u svakoj od petlji (Sl. 2-11). Niz A[i][j] ima prostorno višestruko korišćenje duž unutrašnje petlje i ne poseduje višestruko korišćenje duž spoljašnje petlje. Tokom jedne iteracije u keš memoriju se dovlači jedan keš blok (16B); ukupno tokom svih iteracija u unutrašnjoj petlji dovlači se 800B, jer jedan keš blok sadrži dva susedna elementa niza. Kako nema višestrukog korišćenja po spoljašnjoj petlji, ukupno se dovlači 800B puta 3 (broj iteracija u petlji sa indeksom *i*), tj. 2400B. B[j+1][0] ne ispoljava osobinu višestrukog korišćenja po unutrašnjoj petlji, a ima vremensko višestruko korišćenje po spoljašnjoj petlji. Tokom izvršavanja unutrašnje petlje pristupi se 100x16B=1600B u prvom prolazu; ovim podacima se ponovo pristupa tokom narednih spoljašnjih iteracija. Kako B[j][0] ispoljava grupnu lokalnost sa B[j+1][0], i kako je B[j+1][0] referenca koja se prvo inicira, B[j][0] ne doprinosi dovlačenju novih podataka u keš memoriju. Ukupno, tokom jedne unutrašnje iteracije dovlači se 32B, a unutar jedne spoljašnje iteracije pristupi se 2400B, pa je i u ovom slučaju pristup lokalizovan jer je veličina keš memorije 8KB. Stoga su obe petlje unutar lokalizovanog prostora iteracija.

Ref.	j petlja (100 iteracija)			i petlja (3 iteracije)		
	Tip lokalnosti	Podaci kojima se pristupa		Tip lokalnosti	Podaci kojima se pristupa	
		Jedna it. [B]	Sve it. [B]		Jedna it. [B]	Sve it. [B]
A[i][j]	Prostorna	16	800	-	800	2400
B[j+1][0]	-	16	1600	Vremenska	1600	1600
B[j][0]	Grupna sa B[j+1][0]	0	0	Grupna sa B[j+1][0]	0	0
Ukupno		32	2400		2400	4000

Sl. 2-11. Kapacitet podataka kojima se pristupa u toku izvršavanja test primera.

Lokalnost se dobija presekom između višestrukog korišćenja i lokalizovanog prostora iteracija. Nakon određivanja lokalnosti, sledeći korak je određivanje *prefetch* predikata. Ukoliko neki pristup ima vremensku lokalnost (*temporal locality*) unutar petlje, onda će samo u prvom prolazu uzrokovati promašaj u keš memoriji. Ukoliko pristup ima prostornu lokalnost onda će samo prvi pristup keš bloku uzrokovati promašaj u keš memoriji. Ukoliko ne postoji bilo kakva lokalnost onda će svaki pristup rezultovati promašajem u keš memoriji. Na osnovu toga, mogu se uočiti sledeći *prefetch* predikati: (a) ukoliko referenca ima vremensku lokalnost, predikat je [i=0] (pretpostavlja se, ne umanjujući opštost, da petlja počinje od indeksa 0), (b) u slučaju prostorne lokalnosti, predikat je [(i mod l)=0], pri čemu je l broj elemenata niza unutar jednog keš bloka, (c) ukoliko referenca nema lokalnost onda je predikat [*true*], i (d) ukoliko referenca ima grupnu lokalnost i nije vodeća referenca u pristupu tom nizu predikat je [*false*]. Predikat unutar ugnježdenih petlji sa više tipova lokalnosti dobija se kombinacijom svih predikata po svakom od postojećih tipova lokalnosti. Tako, u posmatranom primeru pristup nizu A[i][j] nema lokalnost u odnosu na petlju sa indeksom i, što odgovara predikatu [*true*], i poseduje prostornu lokalnost duž petlje sa indeksom j, što odgovara predikatu [(j mod 2)=0]; stoga, celokupni *prefetch* predikat se dobija kao

$$[true] \wedge [(j \bmod 2)=0] \Rightarrow [(j \bmod 2)=0].$$

Odgovarajući *prefetch* predikati za ostale reference u posmatranom primeru prikazani su na Sl. 2-12.

Reference	Lokalnost	*Prefetch* predikat
A[i][j]	$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} - \\ prostorna \end{bmatrix}$	(j mod 2) = 0
B[j+1][0]	$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} vremenska \\ - \end{bmatrix}$	i = 0
B[j][0]	Grupna sa B[j+1][0]	*False*

Sl. 2-12. *Prefetch* predikati za relevantne pristupe memoriji.

Reorganizacija koda

Jednostavan način za implementaciju umetanja `prefetch` instrukcija je korišćenje *if* uslovnih naredbi sa *prefetch* predikatom kao uslovom. Međutim, ovakav postupak može dovesti do povećavanja tela petlji, čime se u velikoj meri mogu poništiti očekivana poboljšanja usled dohvatanja podataka unapred. Ovaj problem se može rešiti dekompozicijom petlji u različite sekcije u zavisnosti od predikata; ovaj postupak je poznat pod nazivom razdvajanje petlji (*loop splitting*). U opštem slučaju, predikat [i=0] zahteva da prva iteracija bude izdvojena iz petlje (*peeling*); predikat [(i mod l)=0] zahteva odmotavanje petlje sa faktorom l (*unrolling*). Odmotavanje petlji se uglavnom koristi kada je vrednost l mala; u suprotnom, odmotavanje može uzrokovati ekspanziju kôda unutar petlji, što opet može

dovesti do negativnih posledica. U takvim situacijama može se koristiti transformacija petlji poznata pod nazivom *strip-mining*. Takođe, u nekim situacijama mogu se koristiti i uslovne naredbe, ukoliko transformacije petlje dovode do ekspanzije kôda. Sl. 2-13 prikazuje primere koji ilustruju pomenute transformacije.

Polazna petlja:
```
for(i=0; i<n; i++) {
  if (i==0) f(i);
  g(i);
}
```

Posle *peeling* transformacije:
```
f(0);
g(0);
for(i=1; i<n; i++) {
  g(i);
}
```

(a) *Peeling* transformacija

Polazna petlja:
```
for(i=0; i<n; i++) {
  if ((i mod 4)==0)     f(i);
  g(i);
}
```

Posle *unrolling* transformacije:
```
for(i=0; i<n; i+=4) {
  f(i);
  g(i);
  g(i+1);
  g(i+2);
  g(i+3);
}
```

(b) *Unrolling* transformacija

Polazna petlja:
```
for(i=0; i<n; i++) {
  if ((i mod 64)==0)
        f(i);
  g(i);
}
```

Posle *strip-mining* transformacije:
```
for(j=0; j<n; i+=64) {
  f(j);
  for(i=j; i<j+64; i++)
        g(i);
}
```

(c) S*trip-mining* transformacija

Sl. 2-13. Tehnike transformisanja petlji (*loop splitting*).

Nakon što je utvrđeno koje podatke treba dohvatati unapred i nakon što su izolovane iteracije u kojima treba inicirati dohvatanje podataka unapred, poslednji korak je umetanje `prefetch` instrukcija na odgovarajuća mesta u kodu, tako da podaci budu prisutni u keš memoriji neposredno pre trenutka stvarnog korišćenja. Softverska protočnost (*software pipelining*) je tehnika koja obezbeđuje preklapanje izvršavanja `prefetch` instrukcija za buduće iteracije sa izvršavanjem tekućih iteracija. Dohvatanje podataka treba da bude inicirano dovoljno pre trenutka stvarnog korišćenja da bi se prikrilo celokupno kašnjenje u pristupu memoriji; sa druge strane, dohvatanje podataka unapred se ne sme inicirati suviše rano, jer podaci mogu biti izbačeni iz keš memorije pre trenutka stvarnog korišćenja. U predloženom algoritmu *prefetch* se inicira l/s iteracija unapred, pri čemu je l očekivano kašnjenje u pristupu memoriji uključujući kontenciju na interkonekcionoj mreži, a s najkraće vreme izvršavanja tela petlje. Generička šema koja pokazuje kako se softverska protočnost koristi za iniciranje dohvatanja podataka unapred prikazana je na Sl. 2-14. Pretpostavlja se da je za prikrivanje kašnjenja u pristupu memoriji potrebno 5 iteracija.

Polazna petlja:
```
for(i=0; i<100; i++) {
        A[i][0] = 0;
}
```

Posle primene softverske protočnosti:
```
for(i=0; i<5; i++)                // prolog
        prefetch(&A[i][0]);
for(i=0; i<95; i++){              // glavno telo
        prefetch(&A[i+5][0]);
        A[i][0] = 0;
}
for(i=95; i<100; i++)             // epilog
        A[i][0]=0;
```

Sl. 2-14. Primena softverske protočnosti (*software-pipelining*).

Na kraju, prikazana je primena opisanih tehnika na polazni primer. Prvo, primenjuju se *loop-splitting* transformacije na bazi definisanih *prefetch* predikata. Predikat [i=0] koji je posledica vremenske lokalnosti reference B[j+1][0] zahteva izdvajanje petlje sa indeksom i=0. Dohvatanje unapred elemenata niza se prema tome radi samo u izdvojenom delu petlje. Predikat [(j mod *2*)=0] zahteva odmotavanje petlje j sa faktorom 2 – i to u izdvojenom delu

petlje za *i*=0 i u glavnoj petlji. Tako, za referencu A[i][j] dohvatanje podataka se inicira za svaki drugi element. Na osnovu usvojenih pretpostavki i dužine tela petlje, prevodilac izračunava da su potrebne 3 iteracije za prikrivanje kašnjenja u pristupu memoriji. Transformacija na bazi softverske protočnosti rezultuje formiranju prologa, glavnog tela i epiloga i u slučaju izdvojenog dela i glavnog dela (Sl. 2-15).

Opisani algoritam za umetanje `prefetch` instrukcija je implementiran u SUIF (*Stanford University Intermediate Form*) prevodiocu koji uključuje mnoge standardne optimizacije i generisanje koda. Analiza efikasnosti algoritma za selektivno dohvatanje podataka unapred u jednoprocesorskim sistemima bazirana je na aplikacijama iz skupa SPEC i jednoprocesorskim implementacijama paralelnih aplikacija iz skupa SPLASH i NAS Parallel Benchmark. Ubrzanje usled primene selektivnog dohvatanja podataka unapred kreće se u opsegu od 5% do 100% u zavisnosti od aplikacije. Redukcija vremena blokiranja procesora usled pristupa memoriji se kreće u opsegu od 33% do 90%. Dobijena poboljšanja pokazuju da dohvatanje podataka unapred može značajno doprineti poboljšanju performanse kod jednoprocesorskih sistema. Modifikacije prikazanog algoritma sa ciljem da se podrži dohvatanje unapred kod multiprocesorskih sistema prikazane su u narednom odeljku.

```
prefetch(&A[0][0]);
for(j=0; j<6; j+=2){
        prefetch(&B[j+1][0]);
        prefetch(&B[j+2][0]);
        prefetch(&A[0][j+1]);
}
for(j=0; j<94; j+=2){
        prefetch(&B[j+7][0]);
        prefetch(&B[j+8][0]);
        prefetch(&A[0][j+7]);
        A[0][j]=B[j][0]+B[j+1][0];
        A[0][j+1]=B[j+1][0]+B[j+2][0];
}
for(j=94; j<100; j+=2){
        A[0][j]=B[j][0]+B[j+1][0];
        A[0][j+1]=B[j+1][0]+B[j+2][0];
}
for(i=1; i<3; i++){
        prefetch(&A[i][0]);
        for(j=0; j<6; j+=2)
                prefetch(&A[i][j+1]);
        for(j=0; j<94; j+=2){
                prefetch(&A[i][j+7]);
                A[i][j]=B[j][0]+B[j+1][0];
                A[i][j+1]=B[j+1][0]+B[j+2][0];
        }
        for(j=94; j<100; j+=2){
                A[i][j]=B[j][0]+B[j+1][0];
                A[i][j+1]=B[j+1][0]+B[j+2][0];
        }
}
```

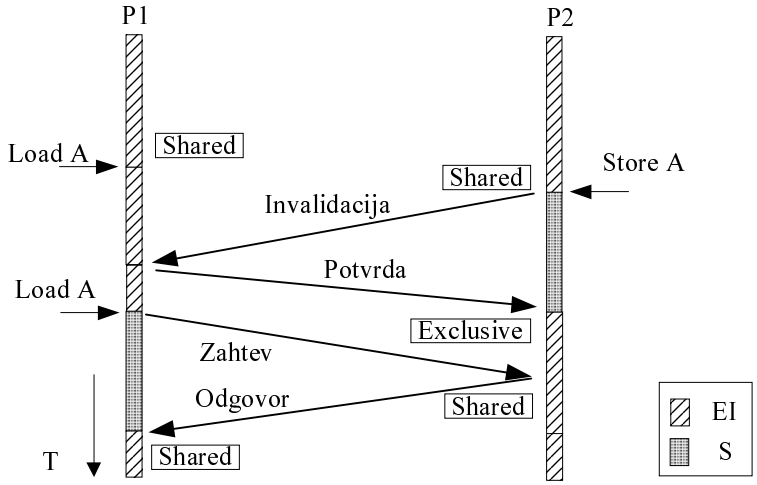
Sl. 2-15. Ilustracija predloženog algoritma za dohvatanje podataka unapred ([Mowry94]).

2.3.2 Algoritam za selektivno dohvatanje podataka unapred kod multiprocesorskih sistema

U multiprocesorskim sistemima moguće je koristiti potpuno identičan algoritam za selektivno dohvatanje podataka unapred koji je opisan u prethodnom odeljku. Korektnost izvršavanja programa je garantovana primenom nevezanog dohvatanja podataka unapred. Međutim, primena predloženog algoritma u prevodiocima za multiprocesore može imati negativnog

uticaja na njegovu efikasnost. Stoga, predložene su modifikacije polaznog algoritma sa ciljem da se on prilagodi uslovima multiprocesorske obrade.

Komunikacija između procesora može značajno uticati na procenat promašaja u keš memoriji. Na Sl. 2-16 dat je jedan takav primer. Procesori P1 i P2 pristupaju istoj lokaciji A, i na početku oba procesora imaju kopije podatka u svojim lokalnim keš memorijama u stanju S (*Shared*). Procesor P1 pristupa lokaciji A dva puta. Pretpostavlja se da u intervalu između ta dva pristupa, procesor P1 ne pristupa nekom drugom podatku koji bi se mapirao u isti keš blok. U slučaju jednoprocesorske obrade drugi pristup bi pod ovim uslovima sigurno rezultovao pogotkom u keš memoriji. Međutim, u posmatranom primeru procesor P2 menja podatak A, što uzrokuje da kopija u keš memoriji procesora P1 bude invalidovana, tako da procesor P1, tokom drugog pristupa, inicira dohvatanje nove kopije podatka. Ovaj tip promašaja, uzrokovan protokolima za održavanje koherencije keš memorije, treba uzeti u obzir pri analizi algoritma za selektivno dohvatanje podataka unapred u multiprocesorskim sistemima.



Sl. 2-16. Uticaj deljenja podataka na promašaje u keš memoriji.
Opis: EI – procesor izvršava instrukcije, S – procesor je blokiran usled promašaja u keš memoriji, T – vreme, Shared – podatak se nalazi u stanju deljen, Exclusive – procesor je ekskluzivni vlasnik podatka.

Algoritam za selektivno dohvatanje podataka unapred kod jednoprocesorskih sistema polazi od toga da lokalnost postoji uvek kada se desi višestruko korišćenje keš bloka, pod uslovom da taj blok nije izbačen iz keš memorije u intervalu između dva korišćenja. Posmatrani keš blok neće biti izbačen iz keš memorije ukoliko je količina podataka kojima se pristupa u intervalu između dva pristupa posmatranom bloku manja od kapaciteta keš memorije. Međutim, u multiprocesorskim sistemima, keš blok će biti invalidovan ako je neki drugi procesor modifikovao taj keš blok. U okviru analize lokalnosti izbacivanje podataka iz keš memorije je modelovano kroz lokalizovani prostor iteracije. Da bi se predvideli promašaji u keš memoriji usled protokola za održavanje keš koherencije, koncept lokalizovanog prostora iteracije je proširen tako da uključi ne samo podatke kojim se pristupa u petlji, već i podatke koji će biti modifikovani od strane drugih procesora.

Tačno predviđanje ovakvih promašaja je veoma teško za programskog prevodioca, jer zahteva tačno poznavanje komunikacije između procesora i njihovih pristupa memoriji, naročito u slučaju eksplicitno paralelizovanih programa kod kojih se informacije o komunikaciji između procesora "kriju" u glavi programera. U tom slučaju od interesa može biti korišćenje

sinhronizacionih primitiva (*lock, unlock, barrier*). Naime, između upisa podatka jednog procesora i čitanja od strane drugog procesora treba da postoji sinhronizaciona primitiva; programski prevodilac može koristiti ove primitive za iniciranje komunikacije između procesora. U idealnom slučaju, poznavanje podataka štićenih sinhronizacionom primitivom pomaže prevodiocu da inicira odgovarajuće akcije, čime se mogu eliminisati promašaji u keš memoriji ([Tranc*96]). Kako se ovakve informacije opet uglavnom "kriju" u glavi programera, u radu [Mowry94] je usvojen konzervativan pristup po kome se pretpostavlja da sve deljene promenljive mogu biti modifikovane na svakoj sinhronizacionoj primitivi. Tako, u multiprocesorskim sistemima važi sledeće: petlja nije lokalizovana ukoliko je (a) količina podataka kojim se pristupa unutar petlje veća od kapaciteta keša, ili (b) postoji eksplicitna sinhronizaciona primitiva.

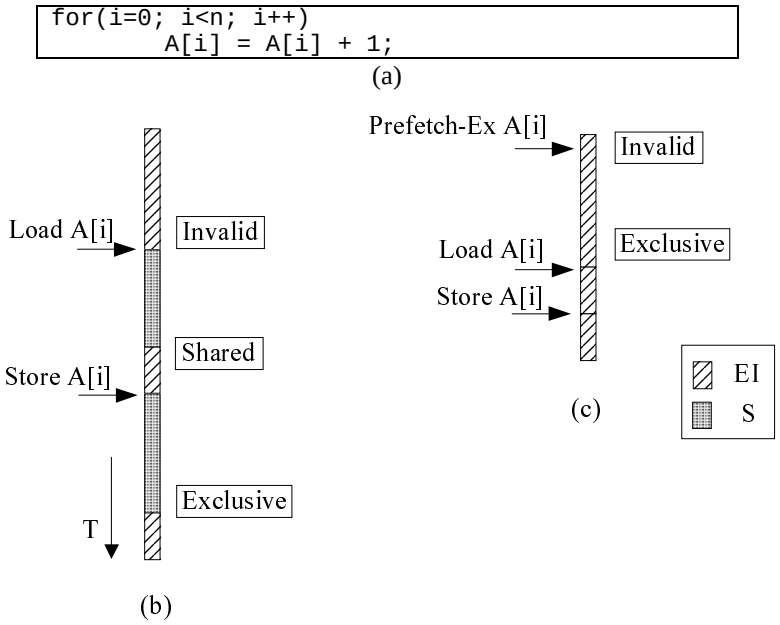
U primeru prikazanom na Sl. 2-17 svaki procesor modifikuje jednu vrstu matrice na osnovu vrednosti `myVal` koja se izračunava na osnovu svih elemenata matrice; izračunavanje se ponavlja nekoliko puta. Barijere se koriste za sinhronizaciju procesora između izračunavanja vrednosti `myVal` i modifikovanja vrste. Ukoliko prevodilac ignoriše sinhronizacione primitive i ukoliko je `numProcs` takav da matrica `A` stane u keš memoriju procesora, rezultat lokalne analize je da pristupi matrici ispoljavaju vremensku lokalnost po spoljašnjoj petlji. Međutim, to je netačno jer procesori modifikuju matricu u drugom delu petlje, uzrokujući, na taj način, invalidaciju svih vrsta u procesorskoj keš memoriji, izuzev one koja pripada posmatranom procesoru. Predložena modifikacija algoritma detektuje da unutar spoljašnje petlje postoji sinhronizaciona primitiva tako da ne postoji lokalnost; tako, u svakoj iteraciji se inicira dohvatanje podataka unapred. Ovakav pristup je konzervativan, jer nije potrebno dohvatati unapred vrstu koja "pripada" posmatranom procesoru.

```
/* NumProcs = broj procesora */
/* MyProcNum = ID procesora */
/* deljena matrica A; svaki procesor modifikuje jednu vrstu */
shared double A[NumProcs][100];
for(t=0; i<t_max; t++) {
   local double myVal =0.0
   for(p=0; p<NumProcs; p++) {
      for(i=0; i<100; i++)
         myVal+=foo(A[p][i], MyProcNum];
   }
   barrier(B, NumProcs);
   for(i=0; i<100; i++)
      A[MyProcNum][i]+=myVal;
   barrier(B, NumProcs);
}
```

Sl. 2-17. Primer sa eksplicitnom sinhronizacionom primitivom unutar petlje.

U multiprocesorskim sistemima sa invalidacionim protokolima za održavanje koherencije keš memorije od interesa je primena ekskluzivnog dohvatanja podataka unapred. Primer koji ilustruje korišćenje ekskluzivnog dohvatanja podataka unapred je pokazan na Sl. 2-18.

Procesor najpre čita element niza `A[i]`, a potom upisuje novu vrednost. Primena dohvatanja podataka unapred može eliminisati blokiranje procesora usled promašaja u keš memoriji zbog čitanja. Nakon primene klasičnog dohvatanja unapred, keš blok se nalazi u stanju S (*Shared*), a instrukcija čitanja vidi pogodak u keš memoriji. Međutim, upis podatka inicira transakciju na interkonekcionoj mreži kojom procesor postaje ekskluzivni vlasnik podatka `A[i]`. Da bi se izbeglo iniciranje ove transakcije, umesto obične `prefetch` instrukcije treba koristiti instrukciju `prefetch-ex` za ekskluzivno dohvatanje podataka unapred.



Sl. 2-18. Uticaj ekskluzivnog dohvatanja podataka unapred na performanse.
Opis: (a) primer koji ilustruje sekvencu čitanje-upis, (b) izvršavanje sekvence (c) izvršavanje sekvence sa primenom ekskluzivnog dohvatanja podataka unapred. EI – procesor izvršava instrukcije, S – procesor je blokiran usled promašaja u keš memoriji, T – vreme, Shared – podatak se nalazi u stanju deljen, Exclusive – procesor je ekskluzivni vlasnik podatka.

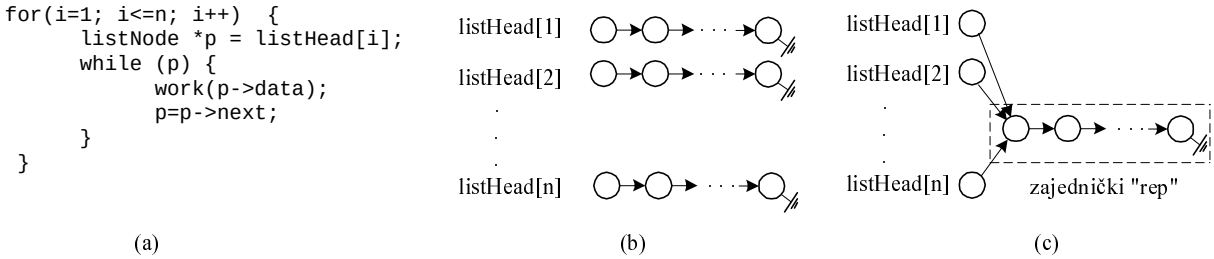
Efikasnost algoritma za selektivno dohvatanje podataka u multiprocesorskim sistemima je analizirana korišćenjem simulatora CC-NUMA multiprocesorskog sistema koji je razvijen po uzoru na DASH multiprocesor. Kao radno opterećenje korišćene su paralelne aplikacije iz skupa SPLASH [Singh*91]. Rezultati eksperimentalne analize pokazali su značajnu efikasnost selektivnog dohvatanja podataka unapred kod CC-NUMA multiprocesora: vreme izvršavanja se redukuje od 6% do 53%, u zavisnosti od aplikacije i parametara memorijskog podsistema. Pri tom, ekskluzivno dohvatanje podataka unapred značajno redukuje saobraćaj na interkonekcionoj mreži u opsegu od 3% do 23%. Takođe, pokazano je da ručno umetanje instrukcija za dohvatanje unapred može dalje povećati efikasnost dohvatanja unapred u multiprocesorskim sistemima.

2.3.3 Dohvatanje podataka unapred u aplikacijama sa rekurzivnim dinamičkim strukturama podataka

Softverski kontrolisano dohvatanje podataka se pokazalo kao izuzetno efikasno u numeričkim aplikacijama sa nizovima kao dominantnim strukturama podataka. U radu [Mowry*97] analiziran je uticaj softverskog dohvatanja podataka unapred u aplikacijama sa dinamičkim strukturama podataka baziranim na pokazivačima (liste, stabla, grafovi, itd). Na bazi izvršene analize predložena su tri algoritma za dohvatanje podataka unapred. Najšire primenljiv algoritam, nazvan *greedy prefetching*, implementiran je u okviru istraživačkog optimizujućeg programskog prevodioca, dok su ostale dve šeme implementirane ručnim umetanjem `prefetch` instrukcija. Uticaj predloženih šema je evaluiran simulacijom modernog superskalarnog mikroprocesora na izabranom skupu aplikacija sa rekurzivnim strukturama podataka. Rezultati simulacione analize pokazuju da se poboljšanje, mereno vremenom izvršavanja u odnosu na slučaj bez primene predloženih algoritama za dohvatanje podataka unapred, kreće između 4% i 45%. Takođe, rezultati pokazuju da prikazani pristup može biti od interesa i u multiprocesorskim sistemima.

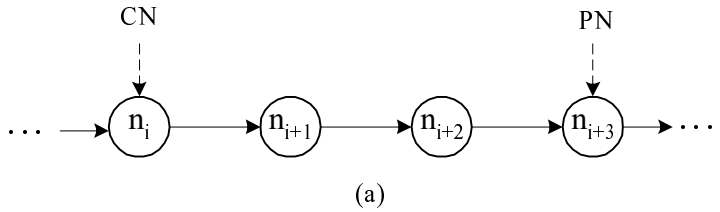
Pored nizova, rekurzivne strukture podataka kao što su stabla, grafovi i liste predstavljaju najčešće korišćene tipove podataka u bazama podataka, grafičkim aplikacijama, itd. Obradom velike količine podataka smanjuje se verovatnoća da podatak bude u keš memoriji u trenutku ponovljenog pristupa tom podatku, tako da je vremenska lokalnost ovih podataka mala. Takođe, rekurzivni podaci, kao dinamičke strukture, ispoljavaju malu prostornu lokalnost, tako da povećavanje dužine keš bloka ne doprinosi smanjivanju broja promašaja u keš memoriji. Tehnike za eliminisanje kašnjenja u pristupu memoriji, kao što su različite optimizacije sa ciljem povećavanja prostorne i vremenske lokalnosti, nisu toliko efikasne kao što je to bio slučaj kod numeričkih aplikacija baziranih na nizovima. Stoga, tehnike za prikrivanje kašnjenja u pristupu memoriji su od velikog značaja u slučaju rekurzivnih struktura podataka. Kašnjenje u slučaju promašaja prilikom upisa se uspešno može prikriti korišćenjem bafera za upise. Pravi izazov je prikrivanje kašnjenja u pristupu memoriji zbog promašaja u keš memoriji prilikom čitanja.

Kao što je već rečeno, algoritam za umetanje `prefetch` instrukcija sastoji se iz dva ključna koraka: analize, koja treba da selektuje pristupe memoriji koji će verovatno rezultovati promašajem u keš memoriji, i umetanja koje podrazumeva reorganizaciju kôda sa ciljem pravovremenog iniciranja dohvatanja podataka unapred, tako da se obezbedi potpuno prikrivanje kašnjenja u pristupu memoriji. Na žalost, proces analize u aplikacijama baziranim na dinamičkim strukturama podataka nije još uvek dovoljno sofisticiran da bi mogao da pravi razliku na primeru koji je ilustrovan na Sl. 2-19. Posmatra se jednostavna sekvenca instrukcija koja vrši obradu elemenata niza listi. Moguća su sledeća dva slučaja: (a) svaki pokazivač na početak liste ukazuje na posebnu listu (Sl. 2-19b), i (b) svi pokazivači imaju istu vrednost, tj. pokazuju na istu listu (Sl. 2-19c). U poslednjem slučaju, ukoliko je kapacitet keš memorije dovoljan da primi sve elemente liste, postoji vremenska lokalnost, pa dohvatanje podataka unapred ima smisla samo u prvom prolazu (i=1).



Sl. 2-19. Obrada listi kada ne postoji i kada postoji vremenska lokalnost.
Opis: (a) Segment kôda sa obradom elemenata liste; (b) Pristup podacima kada ne postoji lokalnost; (c) Pristup podacima kada postoji vremenska lokalnost.

Međutim, problem umetanja `prefetch` instrukcija nije ništa manje složen. Posmatrajmo primer dat na Sl. 2-20. Ukoliko se pretpostavi da je za prikrivanje kašnjenja u pristupu memoriji potrebno vreme za koje se izvrši obrada tri čvora, onda prilikom obrade čvora n_i treba inicirati dohvatanje čvora n_{i+3}. Međutim, problem je što izračunavanje adrese čvora n_{i+3} zahteva izračunavanje adrese čvora n_{i+2}, a to opet zahteva izračunavanje adrese čvora n_{i+1}, itd. Na osnovu toga, primeri prikazani na Sl. 2-20b i Sl. 2-20c praktično imaju isti efekat u pogledu prikrivanja kašnjenja, s tim da primer na Sl. 2-20b može rezultovati većim brojem instrukcija potrebnih da bi se iniciralo dohvatanje podataka unapred. Ovakav scenario je u literaturi poznat pod imenom *pointer-chasing*. Predloženi algoritmi u radu [Mowry*97] zapravo pokušavaju da reše prikazani problem.



(a)

```
while(p) {
   prefetch(p->next->next->next);
   work(p->data);
   p = p->next;
}
```

(b)

```
while(p) {
   prefetch(p->next);
   work(p->data);
   p = p->next;
}
```

(c)

Sl. 2-20. Ilustracija *pointer-chasing* problema.
Opis: (a) Obrada rekurzivne strukture podataka; (b) Dohvatanje unapred čvora sa distancom 3; (c) Dohvatanje unapred čvora sa distancom 1. CN – čvor koji se trenutno obrađuje, PN – čvor koji se želi dohvatiti unapred.

Pretpostavimo da se pristupa čvoru n_i koji se nalazi na adresi A_i i neka se želi unapred dohvatiti čvor n_{i+d} koji se nalazi na adresi A_{i+d}. Distanca d je izabrana tako da omogući potpuno prikrivanje kašnjenja u slučaju promašaja u keš memoriji i iznosi $d = \lceil L/W \rceil$, pri čemu je L očekivano kašnjenje, a W procenjena količina izračunavanja između uzastopnih pristupa memoriji. Adresa čvora n_{i+d} se mora izračunati na osnovu adrese čvora n_i i distance d, tj. $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \end{bmatrix}$, pri čemu je F adresna funkcija. Efikasnost algoritma za umetanje instrukcija za dohvatanje podataka unapred određena je brojem pristupa memoriji potrebnih da bi se izračunala adresa; broj pristupa je označen sa $\|F\|$. Ukoliko se prati lanac pokazivača taj broj odgovara distanci, tj. $\|F\|=d$. Od interesa su algoritmi koji garantuju da je $\|F\|=1$ ili $\|F\|=0$.

Slučaj kada je $\|F\|=1$ znači da čvor n_i sadrži pokazivač na čvor n_{i+d}. Taj pokazivač može biti prirodni ili veštački. Pokazivač je prirodni ukoliko se za određivanje adrese čvora n_{i+d} koristi neki od postojećih pokazivača u samom čvoru n_i. Veštački pokazivač se dodaje naknadno i zahteva dodatni prostor u memoriji, ali može biti tačniji od korišćenja postojećih pokazivača. Slučaj kada je $\|F\|=0$ znači da se adresa čvora n_{i+d} dobija direktnim izračunavanjem na osnovu distance i adrese čvora n_i. Ovakav pristup je moguć kada se, na primer, neka dinamička struktura mapira u niz u memoriji, tako da postoji jedan-na-jedan preslikavanje između elemenata rekurzivne strukture i indeksa elemenata u nizu. U radu [Mowry*97] prikazana su tri algoritma nazvana: (a) *greedy prefetching* koji koristi prirodne pokazivače ($\|F\|=1$), (b) *history-pointer prefetching* koji koristi veštačke pokazivače ($\|F\|=1$), i (c) *data-linearization prefetching* baziran na mapiranju dinamičkih struktura u nizove ($\|F\|=0$).

U daljem tekstu objašnjen je *greedy prefetching*. U slučaju *k*-narne rekurzivne strukture svaki čvor sadrži *k* pokazivača na ostale čvorove. Jedan od *k* čvorova će biti posećen kao sledeći. Prema tome, ostalih *k*-1 pokazivača služe kao prirodni pokazivači na čvorove i mogu se iskoristiti za dohvatanje podataka unapred. Ukoliko nijedan od njih ne ukazuje na čvor n_{i+d}, očekuje se da svaki od njih ukazuje na čvor $n_{i+d'}$, $d' > 0$. Ukoliko je $d' < d$, onda će kašnjenje biti delimično prikriveno; ukoliko je $d' \geq d$, očekuje se da kašnjenje bude potpuno prikriveno.

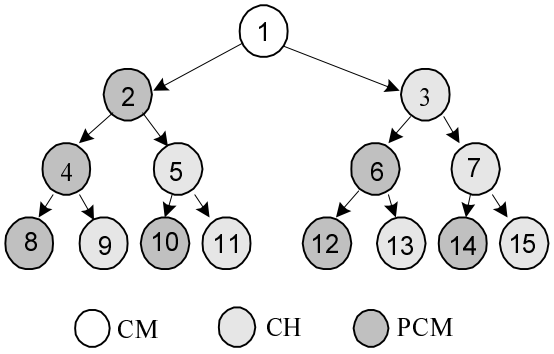
Ilustracija ovog algoritma je prikazana je na primeru obrade binarnog stabla (Sl. 2-21a). Pretpostavimo da je trajanje obrade čvora *process()* jednako polovini vremena potrebnog za

dohvatanje keš bloka iz memorije; da bi prikrili kašnjenje distanca mora biti *d*=2. Sl. 2-21b pokazuje ponašanje keš memorije tokom izvršavanja programa sa Sl. 2-21a. U ovom slučaju kašnjenje je potpuno prikriveno za otprilike polovinu čvorova, a delimično prikriveno za drugu polovinu čvorova. U slučaju *k*-narnog stabla, može se očekivati da kašnjenje bude potpuno prikriveno za otprilike (k-1)/k čvorova. Prednosti ovakvog pristupa su jednostavnost, mala cena merena brojem dodatnih instrukcija i prostora u memoriji i široka primenljivost. Glavni nedostatak je nemogućnost precizne kontrole distance.

Efikasnost predloženog algoritma programskog prevodioca je testirana i u multiprocesorskom sistemu. Razmatra se multiprocesorski sistem sa parametrima koji odgovaraju parametrima sistema koji je korišćen u analizi selektivnog dohvatanja podataka koji je opisan u radu [Mowry94]. Simulaciona analiza je sprovedena na paralelnoj aplikaciji Barnes iz skupa paralelnih aplikacija SPLASH-2 [WooO*95]. U ovoj aplikaciji podaci su organizovani u oktalno stablo. Implementacija opisanog algoritma *greedy-prefetching* rezultovala je značajnim smanjivanjem broja promašaja u keš memoriji, a poboljšanje mereno skraćivanjem vremena izvršavanja iznosi 14%.

```
preorder (treeNode *t ) {
    if (t != NULL ) {
            prefetch ( t-> left );
            prefetch ( t-> right );
            process ( t->data );
            preorder ( t->left );
            preorder ( t->right );
     }
}
```

(a)



(b)

Sl. 2-21. Ilustracija algoritma *greedy prefetching*.
Opis: (a) Procedura *preorder* modifikovana da podrži dohvatanje podataka unapred; (b) ilustracija promašaja u keš memoriji tokom obrade stabla. CM (*Cache Miss*) – promašaji u keš memoriji, CH (*Cache Hit*), PCM (*Partial Cache Miss*) – delimično prikriveni promašaji u keš memoriji.

2.3.4 Potencijal dohvatanja podataka unapred kod multiprocesora sa zajedničkom magistralom

Većina istraživanja tehnika za dohvatanje podataka unapred odnose se na multiprocesorske sisteme sa skalabilnim interkonekcionim mrežama, pre svega na CC-NUMA mašine. U radu [Tulls*93] analizirana je efikasnost dohvatanja podataka unapred u sistemima koji su bazirani na zajedničkoj magistrali. Pretpostavlja se da je programski prevodilac idealan u smislu predviđanja promašaja u keš memoriji. Svaki pristup memoriji koji rezultuje promašajem u keš memoriji augmentira se odgovarajućom `prefetch` instrukcijom. Pored ovog osnovnog pristupa, razmatra se i efikasnost pristupa nastalih modifikacijom polaznog. Tako, razmatra se ekskluzivno dohvatanje podataka unapred, povećavanje *prefetch* distance i ubacivanje redundantnih `prefetch` instrukcija sa ciljem da se poveća efikasnost u slučaju deljenih podataka koji se modifikuju (*write shared data*).

Kao radno opterećenje posmatra se pet paralelnih aplikacija iz skupa SPLASH i SPLASH-2. Modelira se multiprocesor sa 12 procesora koji su povezani magistralom koja podržava

razdvojene transakcije (*split-transactions bus*). Da bi se odredila efikasnost dohvatanja podataka unapred za različite sisteme, pretpostavlja se da je za dohvatanje jednog bloka iz memorije potrebno 100 procesorskih ciklusa (pclk), a da vreme čistog transfera podataka varira između 4pclk i 32pclk, tj. ako je vreme transfera 4pclk, vreme pristupa memoriji je 96pclk. Evaluacija se bazira na simulaciji zasnovanoj na analizi realnih adresnih tragova. Adresni tragovi se propuštaju kroz simulator koji identifikuje sve pristupe koji rezultuju promašajem u keš memoriji. Takvi pristupi se augmentiraju odgovarajućom instrukcijom (`prefetch` ili `prefetch-ex`), tako da se u potpunosti obezbedi prikrivanje kašnjenja u pristupu memoriji.

Dobijeni rezultati pokazuju da je efikasnost dohvatanja podataka unapred pod navedenim uslovima u sistemima za zajedničkom magistralom iznenađujuće mala, izuzev u slučajevima kada je protok na magistrali izuzetno visok, tj. kod sistema kod kojih transfer podataka traje samo 4pclk. U slučaju kada je propusna moć magistrale ograničena, dohvatanje unapred ne garantuje poboljšavanje performanse i pored značajnog smanjivanja broja blokirajućih promašaja u keš memoriji. Pored kontencije na magistrali, ograničavajući faktori za postizanje veće efikasnosti dohvatanja unapred su *false* i *true sharing*, kao i kolizija između unapred dohvaćenih podataka i tekućeg radnog opterećenja, naročito kod keš memorija sa direktnim preslikavanjem. U radu [Tulls*95] je pokazano kako se korišćenjem različitih mehanizama može značajno povećati efikasnost dohvatanja unapred kod multiprocesora sa zajedničkom magistralom: (a) problem promašaja u keš memoriji zbog konflikta između tekućeg i budućeg radnog opterećenja kod keš memorija sa direktnim preslikavanjem u velikoj meri se eliminiše upotrebom *victim* keša [Jouppi90], (b) problem prividnog deljenja podataka (*false sharing*) može se rešiti restruktuiranjem deljenih podataka, i (c) problem niske efikasnosti u odnosu na prave deljene podatke (*true sharing*) može se ublažiti razvojem posebnih tehnika za dohvatanje unapred koje su posebno prilagođene deljenim podacima. Međutim, i pored ovih poboljšanja efikasnost tehnike dohvatanja podataka unapred kod multiprocesora sa zajedničkom magistralom nije naročito značajna.

2.3.5 Algoritam za prosleđivanje podataka kod skalabilnih multiprocesora sa deljenom memorijom

U radu [Koufa*96] predloženo je okruženje za algoritam programskog prevodioca koji za prosleđivanje podataka kod aplikacija koje su paralelizovane na nivou petlji tipa *doall*. Prosleđivanje se ostvaruje `wr-forw` asemblerskom instrukcijom koja je predložena u radu [Pouls*94]. Prevodilac umeće ovu instrukciju umesto obične `write` instrukcije. Ova instrukcija, pored adrese, specificira i procesore buduće korisnike kojima treba proslediti podatke. Oni se obično specificiraju koristeći jedan registar kao bit vektor, a svakom procesoru je pridružen po jedan bit; u zavisnosti od vrednosti bita *i* podaci se prosleđuju procesoru Pi ili ne. Ukoliko je keš blok specificiran instrukcijom u stanju M (*Modified*) inicira se, najpre, transakcija koja ažurira memoriju i blok proglašava deljenim (*Shared*), a potom i transakcija kojom se blok prosleđuje svim procesorima, budućim korisnicima. Ukoliko je keš blok specificiran instrukcijom u stanju S (*Shared*), onda on ostaje u tom stanju; memoriji se prosleđuje samo modifikovana reč, a nakon toga se vrši prosleđivanje procesorima budućim korisnicima. Ako instrukcija `wr-forw` vidi promašaj u keš memoriji onda se najpre dovlači blok iz memorije, modifikuje odgovarajuća reč i nakon toga vrši prosleđivanje.

Sam predloženi algoritam programskog prevodioca nije od interesa za ovo istraživanje, budući da se odnosi na specifično programsko okruženje, gde se paralelizacija vrši na nivou *doall* petlji. Aproksimativna evaluacija je urađena na sledeći način. Koristeći EPG-sim

simulator [Pouls*93], zasnovan na izvršavanju programa, formiraju se adresni tragovi za datu arhitekturu. Na osnovu analize adresnih tragova određuju se `write` instrukcije koje treba da budu zamenjene `wr-forw` instrukcijama, kao i lista procesora budućih korisnika. Nakon toga ponavlja se simulacija. Kao radno opterećenje koriste se paralelne aplikacije iz skupa Perfect Club [Berry*89]. Modeliran je keš-koherentni UMA multiprocesor sa deljenom memorijom koji ima 32 čvora. Za održavanje koherencije keš memorije koristi se invalidacioni protokol. Procesori poseduju privatne keš memorije koji su preko interkonekcione mreže tipa *Omega* povezane sa memorijom.

Dobijeni rezultati pokazali su da primena tehnike prosleđivanja može dati značajne rezultate. Pokazalo se da je tehnika prosleđivanja osetljiva na veličinu privatnih keš memorija. Tako, srednje ubrzanje za razmatrane aplikacije iznosi i do 50% za arhitekture sa velikim keš memorijama, odnosno oko 30% za arhitekture sa relativno malim keš memorijama.

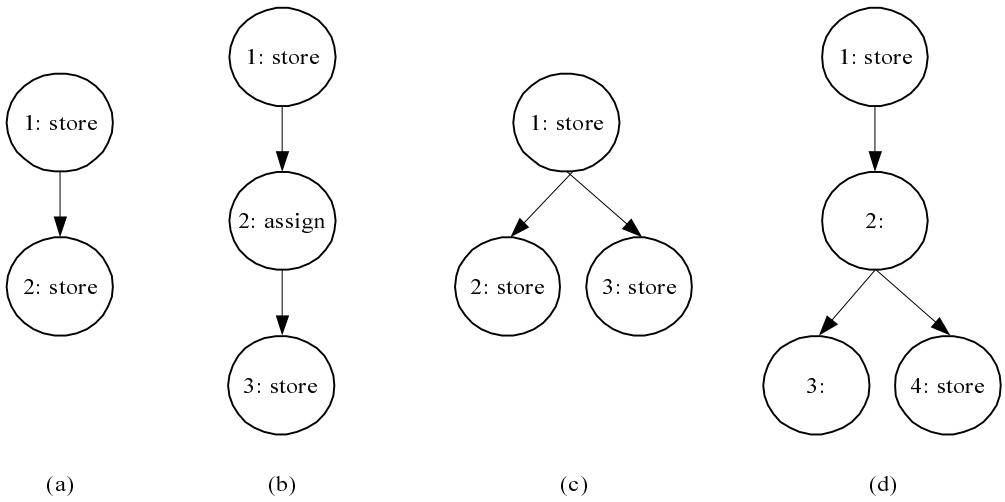
2.3.6 Algoritam za vraćanje modifikovanih deljenih podataka u glavnu memoriju

U radu [Skepp*95] prikazan je jedan algoritam programskog prevodioca koji detektuje poslednji upis u posmatrani keš blok i zamenjuje taj upis instrukcijom koja, pored ažuriranja lokalne keš memorije, inicira ažuriranje glavne memorije. Vraćanje podataka u glavnu memoriju nakon završetka obrade značajno redukuje kašnjenje prilikom čitanja pravih deljenih podataka kod CC-NUMA sistema.

Pretpostavimo sledeći scenario: procesor Pj je ekskluzivni vlasnik bloka Bi, a procesor Pi inicira čitanje bloka Bi. Tokom čitanja bloka Bi, procesor Pi se najpre obraća čvoru domaćinu (*home*) koji je odgovoran za blok Bi. U tom čvoru nalazi se informacija da je procesor Pj ekskluzivni vlasnik bloka, pa se inicira transakcija kojom čvor domaćin zahteva blok Bi. Po prijemu zahteva procesor Pj šalje kopiju bloka čvoru domaćinu. Nakon ažuriranja glavne memorije, čvor domaćin inicira transakciju kojom procesoru Pi prosleđuje traženu kopiju. U ovom slučaju procesor Pi je blokiran za vreme izvršavanja četiri transakcije na interkonekcionoj mreži, a ovaj tip promašaja u keš memoriji naziva se *dirty remote miss*. U slučaju da čvor domaćin poseduje ažurnu kopiju traženog keš bloka, takav promašaj se naziva *clean remote miss*, a kašnjenje u tom slučaju je znatno manje, jer su potrebne samo dve transakcije na interkonekcionoj mreži. Ideja predloženog rešenja je da se umetanjem instrukcija koje ažuriraju *home* čvor, odmah nakon obrade keš bloka, smanji kašnjenje pretvaranjem *dirty remote* promašaja u *clean remote* promašaje. U tom cilju predložene su instrukcije koje iniciraju ažuriranje glavne memorije, nazvane `update` i `store-update`. `Update` instrukcija ažurira glavnu memoriju, a lokalna kopija podatka postaje deljena (*Shared*). `Store-update` instrukcija prvo izvršava običnu `store` instrukciju, a potom `update` instrukciju; ova instrukcija je predložena sa ciljem da se smanji povećavanje broja instrukcija kada je to moguće. U daljem tekstu objašnjen je predloženi algoritam prevodioca.

Pretpostavlja se da prevodilac zna veličinu keš bloka, ali se ne uvode ograničenja u pogledu smeštanja podataka u odnosu na granice između keš blokova. Takođe, pretpostavlja se postojanje predloženih instrukcija `update b,k` i `store-update b,k`, pri čemu su `b` i `k` bazni pokazivač i pomeraj, redom. Radi jednostavnosti diskusije pretpostavimo, za početak, da je keš blok dužine jedne reči. Predloženi algoritam vrši analizu međukôda unutar procedura sa ciljem da nađe poslednju instrukciju koja modifikuje keš blok i iza nje umetne `update` instrukciju ili da poslednju `store` instrukciju zameni sa `store-update` instrukcijom.

Ukoliko ne postoje alternativni putevi u grafu izvršavanja, glavni problem je detektovati poslednji upis u posmatranu reč (u opštem slučaju, poslednji upis u keš blok). Međutim, alternativni putevi u grafu toka problem čine manje trivijalnim. Na Sl. 2-22 je prikazano nekoliko primera grafova toka sa `store` instrukcijama za istu memorijsku lokaciju. Na Sl. 2-22a ilustrovan je jednostavan slučaj gde `store` instrukcija u bazičnom bloku 2 može biti zamenjena `store-update` instrukcijom, dok u slučaju prikazanom na Sl. 2-22b obe `store` instrukcije treba zameniti `store-update` instrukcijama zato što instrukcija `assign` menja bazni pokazivač. U slučaju koji je ilustrovan na Sl. 2-22c svaka od `store` instrukcija u bazičnim blokovima 2 i 3 treba da bude zamenjena `store-update` instrukcijom jer je `store` instrukcija u bazičnom bloku 1 praćena sledećom instrukcijom bez obzira na ishod skoka. Međutim, na Sl. 2-22d je pokazan primer kada postoji sledeća `store` instrukcija samo u slučaju jednog ishoda; u tom slučaju usvojeno je da se umetne novi blok između bazičnih blokova 2 i 3 sa `update` instrukcijom, a da poslednji `store` u bazičnom bloku 4 bude zamenjen `store-update` instrukcijom.



Sl. 2-22. Grafovi toka sa `store` instrukcijama koje upisuju u isti keš blok.
Opis: Čvorovi i usmerene linije predstavljaju bazične blokove i kontrolu toka između bazičnih blokova, redom. Sve `store` instrukcije odnose se na isti blok, a instrukcija `assign` menja sadržaj baznog pokazivača.

Kada je veličina keš bloka veća od jedne reči, javlja se dodatni problem usled nepoznavanja mapiranja u odnosu na granice blokova (ukoliko bi se usvojilo da bazni pokazivač mora ukazivati na početak bloka to bi bilo značajno ograničenje za prevodilac). Radi ilustracije posmatraju se primeri dati na Sl. 2-23. Pretpostavlja se da je veličina keš bloka dve reči i da ne postoji neka `store` instrukcija sa istim baznim pokazivačem u sledećim bazičnim blokovima. U primeru datom na Sl. 2-23a posmatrane reči na adresama (b) i (b+1) mogu pripadati različitim blokovima, pa u tom slučaju prevodilac umeće dve `update` instrukcije. U primeru sa Sl. 2-23b reči na adresama (b+0) i (b+1) mogu pripadati istom bloku, a takođe i reči na adresama (b+1) i (b+2). Stoga prevodilac umeće `update` instrukciju iza druge `store` instrukcije, a iza poslednje `store` instrukcije `store-update` instrukciju. U primeru sa Sl. 2-23c umeću se dve `update` instrukcije na kraju, jer prve dve `store` instrukcije ažuriraju različite blokove. Na kraju, na Sl. 2-23d ilustrovana je sekvenca kada se obe `store` instrukcije mogu zameniti odgovarajućim `store-update` instrukcijama.

Primer	Polazna sekvenca	Modifikovana sekvenca

(a)	`store b, 0`
	`store b, 1`
		`store b, 0`
		`store b, 1`
		`update b,0`
		`update b,2`

(b)	`store b, 0`
	`store b, 1`
	`store b, 2`
		`store b, 0`
		`store b, 1`
		`update b,0`
		`store-update b,2`

(c)	`store b, 0`
	`store b, 2`
	`store b, 1`
		`store b, 0`
		`store b, 2`
		`store b, 1`
		`update b,0`
		`update b,2`

(d)	`store b, 1`
	`store b, 0`
	`store b, 2`
		`store b, 1`
		`store-update b, 0`
		`store-update b, 2`

Sl. 2-23. Primeri primene algoritma za umetanje instrukcija za ažuriranje memorije *home* čvora.

Algoritam je implementiran kao optimizacioni prolaz kroz kôd. Efektivna adresa `store` instrukcije predstavlja se preko baznog pokazivača `b` i pomeraja `k`. Sve `store` instrukcije u okviru jedne procedure su grupisane po klasama, tako da jedna instrukcija pripada klasi $(b,n=\lfloor k/C \rfloor)$, pri čemu je C veličina keš bloka u rečima. Za svaku klasu (b,n) definiše se klasa prethodnik $(b,n-1)$ i klasa sledbenik $(b,n+1)$. Analiza toka podataka (*dataflow analysis*) se odvija unapred (*forward*) i unazad (*backward*). Svaka `store` klasa može da se mapira u dva susedna keš bloka, niži i viši. Svaka `store` instrukcija generiše jedinicu u dijagramu toka (odgovarajuću klasu i reč). Generisana jedinica se propagira unapred i unazad, sve dok bazni pokazivač ne dobije novu vrednost; promena baznog pokazivača znači eliminaciju svih jedinica sa tim baznim pokazivačem. U analizi unapred, jedinica se prenosi na sledeći bazični blok ako je propagirana kroz barem jedan od prethodnih bazičnih blokova; u analizi unazad, jedinica se prenosi u prethodni bazični blok ako je propagirala kroz barem jedan od bazičnih blokova sledbenika. Ako neka jedinica dostigne neku tačku u analizi unapred kaže se da jedinica dostiže tu tačku; ako neka jedinica dostigne neku tačku u analizi unazad kaže se da je jedinica živa u toj tački.

Za svaku *store* klasu (b,n), algoritam prvo određuje tačku u kojoj je niži keš blok modifikovan poslednji put. Takva tačka se označava kao mesto ažuriranja klase (b,n) (*update-site*) i određuje mesto umetanja instrukcije `update` (ili `store-update`) ako ni jedna od klasa (b,n) i $(b,n-1)$ nisu žive. Umetanjem instrukcije `update b, n*C`, niži keš blok je ažuriran, ali ne i viši. Lokalna analiza je demonstrirana na programima koji se sastoje samo od jednog bazičnog bloka, kao u sekvencama prikazanim na Sl. 2-23. U primeru na Sl. 2-23a postoje dve `store` instrukcije koje iniciraju upis u klasu $(b,0)$. Kako klase $(b,0)$ i klasa prethodnik nisu žive iza druge `store` instrukcije, ta tačka predstavlja mesto ažuriranja za niži blok klase $(b,0)$. Tokom analize unazad, algoritam utvrđuje da je modifikovan ne samo niži keš blok, već da postoji mogućnost i da je viši blok modifikovan. U tom slučaju algoritmu se dozvoljava da generiše i klasu sledbenika $(b,1)$. Mesto ažuriranja za ovu klasu je u istoj tački kao i za klasu $(b,0)$. Kao rezultat modifikovana sekvenca sadrži dve `update` instrukcije. Isti rezultat se dobija i za sekvencu prikazanu na Sl. 2-23c. U slučaju da se mesto ažuriranja nalazi neposredno iza upisa u niži keš blok posmatrane klase, `store` instrukcija se može zameniti `store-update` instrukcijom. U primeru na Sl. 2-23b to je moguće za klasu $(b,1)$, a nije moguće za klasu $(b,0)$. Pored toga, `store` instrukcija praćena mestom ažuriranja može biti zamenjena `store-update` instrukcijom ako su ispunjena sledeća dva uslova: (1) klasa sledbenik nije živa u toj tački, i (2) mesto ažuriranja nije odgovorno za ažuriranje višeg keš bloka klase prethodnika.

`Update` ili `store-update` instrukcije se mogu umetati tokom lokalne analize samo ako keš blok nije modifikovan van posmatranog bazičnog bloka. To je slučaj samo ako je klasa eliminisana, pa zatim formirana i opet eliminisana unutar jednog bloka. Ukoliko je klasa eliminisana, pa opet formirana ne mogu se umetati `update` instrukcije, jer postoji mogućnost da je klasa aktivna i u sledećim bazičnim blokovima. U okviru globalne analize rešava se problem umetanja `update` instrukcija. Pretpostavimo da je mesto ažuriranja posmatrane klase u tački *p*. Ukoliko klasa nije eliminisana između početka bazičnog bloka i tačke *p,* onda se takvo mesto ažuriranja zove inicijalno mesto ažuriranja (*initial update site*). Ukoliko klasa nije eliminisana između tačke *p* i kraja bazičnog bloka, onda se takvo mesto ažuriranja naziva konačno mesto ažuriranja (*final update site*). Ukoliko mesto ažuriranja nije inicijalno i nije konačno onda ta klasa mora da je eliminisana, generisana, pa opet eliminisana unutar bazičnog bloka, pa se umetanje `update` instrukcija vrši tokom lokalne analize na način prikazan u prethodnom paragrafu. Na mestu konačnog mesta ažuriranja umeće se odgovarajuća instrukcija ukoliko posmatrana klasa i njen prethodnik nisu živi na kraju posmatranog bazičnog bloka, što se utvrđuje globalnom analizom živih klasa. Na kraju, na mestu inicijalnog mesta ažuriranja (ukoliko nije ujedno i konačno mesto ažuriranja) umeće se odgovarajuća instrukcija u zavisnosti od analize na nivou reči. Prevodilac ne koristi nikakva znanja o komunikaciji između procesora. Uzeto je da se sve klase eliminišu na sinhronizacione operacije *acquire* i *release*.

Efikasnost predloženog algoritma je evaluirana korišćenjem šest paralelnih aplikacija iz skupa SPLASH-2. Razvijen je simulator keš koherentnog CC-NUMA multiprocesora. Primena predloženog algoritma je pokazala visoku efikasnost u otklanjanju *dirty-remote* promašaja i njihovom pretvaranju u *clean-remote* promašaje. U proseku oko 83% originalnih *dirty-remote* promašaja se opisanim algoritmom pretvara u *clean-remote* promašaje. Poboljašanja merena redukcijom vremena izvršavanja se u zavisnosti od aplikacije i parametara arhitekture kreću između 3% i 32%.

2.3.7 Ubrzanje kritičnih sekcija primenom dohvatanja podataka unapred i prosleđivanja podataka

U radu [Tranc*96] prikazane su tehnike za ubrzavanje kritičnih sekcija primenom dohvatanja podataka unapred i prosledjivanjem podataka budućem korisniku. Ubrzavanje kritičnih sekcija ostvaruje se smanjivanjem broja promašaja u keš memoriji tokom izvršavanja kritičnih sekcija. Analizom paralelnih programa sa eksplicitnim sinhronizacionim primitivama *acquire* i *release* uočeno je da se značajan deo vremena izvršavanja programa troši na sinhronizaciju. Za skup od pet paralelnih aplikacija iz skupa SPLASH-2 izmereno je da 24% ukupnog vremena izvršavanja odlazi na sinhronizaciju, kod CC-NUMA multiprocesora. Sa druge strane, kritične sekcije predstavljaju relativno male programske delove koji se lako mogu analizirati sa ciljem da se izbegne čekanje tokom pristupa memoriji usled promašaja u keš memoriji i čekanje nametnuto protokolima za održavanje konzistencije glavne memorije.

Prva optimizacija, nazvana Pref (Sl. 2-24), ima za cilj da redukuje broj blokirajućih promašaja u keš memoriji tokom izvršavanja kritične sekcije. Ovo se ostvaruje dohvatanjem unapred svih deljenih podataka koji se čitaju u kritičnoj sekciji; dohvatanje unapred se inicira neposredno nakon ulaska u kritičnu sekciju. Kako izvršavanje `prefetch` instrukcije ne blokira izvršavanje tekuće programske niti i kako se koristi preklapanje pristupa memoriji, procesor će videti samo kašnjenje u pristupu memoriji tokom pristupa prvom deljenom podatku. Izmeštanjem `prefetch` instrukcija izvan kritične sekcije moguće je eliminisati i

ovo kašnjenje, ali u tom slučaju postoji mogućnost da dohvaćeni podaci budu invalidovani pre trenutka stvarnog korišćenja ukoliko neki drugi procesor modifikuje posmatrane podatke.

Druga optimizacija, nazvana Forw (Sl. 2-24), treba da eliminiše kašnjenje koje preostane nakon primene prve optimizacije. Tako, podaci mogu biti prosledjeni od jednog procesora ka drugom procesoru koji je budući korisnik, pre izlaska prvog procesora iz kritične sekcije. Prosleđivanje podataka se ostvaruje instrukcijama za prosleđivanje koje ne blokiraju izvršavanje programske niti. U idealnom slučaju drugi procesor, nakon ulaska u kritičnu sekciju, nalazi sve potrebne podatke u svojoj keš memoriji. Međutim, prosleđivanje podataka nameće i neke probleme. Pre svega, prvi procesor koji inicira prosleđivanje mora znati identitet drugog procesora kome prosleđuje podatke. Jedan mogući pristup je da u memoriji postoji red čekanja sa procesorima koji čekaju za ulazak u kritičnu sekciju po posmatranom *lock*-u. Procesor vlasnik *lock*-a čita ovaj red i prosleđuje podatke procesoru koji je prvi u listi čekanja. Ukoliko je red čekanja prazan u trenutku izlaska iz kritične sekcije, onda se prosleđivanje preskače ili se inicira vraćanje podataka u glavnu memoriju. Drugi problem se javlja kada se isti *lock* koristi da štiti različite kritične sekcije koje izvršavaju različite operacije nad različitim podacima. U tom slučaju, prosleđivanje može biti beskorisno. Ovaj problem je moguće rešiti tako što se formira unija svih podataka kojima se pristupa u različitim kritičnim sekcijama koje su štićene posmatranim *lock*-om, a prosleđuju se svi podaci iz tako formirane unije. Međutim, ovakav pristup može dovesti do problema da treba proslediti podatak koji se trenutno ne nalazi u lokalnoj keš memoriji; u tom slučaju može se ignorisati posmatrana instrukcija za prosleđivanje, tj., njeno ponašanje odgovara `no-op` instrukciji.

Prve dve šeme za optimizaciju kritičnih sekcija rešavaju problem blokiranja procesora usled promašaja u keš memoriji prilikom čitanja. Kod sistema sa relaksiranim modelima konzistencije memorije promašaji u keš memoriji prilikom upisa ne blokiraju izvršavanje programske niti; međutim, ovi promašaji mogu izazvati blokiranje procesora usled primene protokola za održavanje modela memorijske konzistencije. Tako, na primer, operacija *release* će blokirati procesor čekajući na potvrde invalidacionih poruka poslatih usled modifikacije deljenih podataka. Ovaj problem se može eliminisati dohvatanjem podataka unapred, tako da budu spremni za modifikovanje, tj. da procesor bude ekskluzivni vlasnik posmatranog bloka. Na ovaj način, u trenutku modifikovanja podatka, procesor ne treba da šalje invalidacione poruke. Kombinovanjem ovog pristupa sa šemama za optimizaciju #1 i #2 dobijaju se šeme za ekskluzivno dohvatanje unapred (ExPref) i ekskluzivno prosleđivanje (ExForw), koje su prikazane na Sl. 2-24. Opisane optimizacije dalju bolje rezultate za manje kritične sekcije i kada je kontencija u pristupu *lock* varijablama velika. Ove osobine su tipične za kritične sekcije koje štite pristup kontrolnim strukturama podataka, a ne podacima koji se obrađuju.

Primena opisanih optimizacija ilustrovana je na jednostavnom primeru sa Sl. 2-25. Na Sl. 2-25a prikazana je jednostavna kritična sekcija; izgled kritične sekcije nakon primene optimizacije ExPref prikazana je na slici Sl. 2-25b, a nakon primene optimizacije ExForw na Sl. 2-25c. Primena optimizacija Pref i Forw daje isti rezultat ako se izuzmu `Prefetch_ex` i `Forward_ex` instrukcije. Pretpostavlja se da promenljive `b` i `c` pripadaju istom keš bloku.

Evaluacija predloženih optimizacija bazirana je na simulaciji CC-NUMA multiprocesorskog sistema sa 32 čvora. Svaki čvor sadrži prvi nivo keš memorije sa *write-through* politikom ažuriranja, kapaciteta 4KB i drugi nivo keša sa *write-back* politikom ažuriranja, kapaciteta 64KB. Kao radno opterećenje koriste se paralelne aplikacije iz SPLASH-2 skupa. Dobijeni rezultati pokazuju da se nakon optimizacije kritičnih sekcija posmatrane aplikacije izvršavaju u proseku za 20% brže. Prikazane optimizacije su posebno efikasne u aplikacijama kod kojih

je kašnjenje usled sinhronizacije značajno i kod kojih se glavnina promašaja u keš memoriji dešava u kritičnim sekcijama. Tako, u aplikaciji *MP3D* vreme izvršavanja nakon optimizacije kritičnih sekcija iznosi 48% polaznog vremena izvršavanja. Dobijeni rezultati su pokazali da optimizacije bazirane na prosleđivanju nisu posebno efikasne i ne opravdavaju dodatnu kompleksnost koja je potrebna da se podrži prosleđivanje podataka.

Optimizacija #1 (Pref)
*Neposredno nakon izvršavanja **acquire** operacije dohvatiti unapred sve deljene podatke koji se čitaju u tom kritičnom regionu.*

Optimizacija #2 (Forw)
*Neposredno nakon izvršavanja **acquire** operacije dohvatiti unapred sve deljene podatke koji se čitaju u tom kritičnom regionu. Neposredno pre izvršavanja **release** operacije, ukoliko postoji procesor blokiran na tom lock-u, proslediti deljene podatke tom procesoru.*

Optimizacija #3 (ExPref)
*Neposredno nakon izvršavanja **acquire** operacije dohvatiti unapred, kao ekskluzivne, sve deljene podatke koji će se modifikovati, a dohvati unapred sve preostale deljene podatke koji će se čitati u tom kritičnom regionu.*

Optimizacija #4 (ExForw)
*Neposredno nakon izvršavanja **acquire** operacije dohvatiti unapred, kao ekskluzivne, sve deljene podatke koji će se modifikovati, a dohvati unapred sve preostale deljene podatke koji će se čitati u tom kritičnom regionu. Neposredno pre izvršavanja **release** operacije, ukoliko postoji procesor blokiran na tom lock-u, proslediti, kao ekskluzivne, sve modifikovane podatke i preostale deljene podatke (koji su samo čitani) tom procesoru.*

Sl. 2-24. Optimizacije kritičnih sekcija dohvatanjem podataka unapred i prosleđivanjem podataka.

```
LOCK(l);
     a = b + c;
     d = f->g;
UNLOCK(l);
```
(a)

```
LOCK(l);
     Prefetch(b);
     Prefetch_ex(a);
     Prefetch(f);
     Prefetch(f->g);
     Prefetch_ex(d);
     a = b + c;
     d = f->g;
UNLOCK(l);
```
(b)

```
LOCK(l);
     Prefetch(b);
     Prefetch_ex(a);
     Prefetch(f);
     Prefetch(f->g);
     Prefetch_ex(d);
     a = b + c;
     d = f->g;
     Forward(b, Dest);
     Forward_ex(a, Dest);
     Forward(f, Dest);
     Forward(f->g, Dest);
     Forward_ex(d, Dest);
UNLOCK(l);
```
(c)

Sl. 2-25. Primeri primene optimizacija kritičnih sekcija.
Opis: (a) polazni primer jednostavne kritične sekcije, (b) kritična sekcija nakon primene ExPref optimizacije, (c) kritična sekcija nakon primene ExForw optimizacije.

2.3.8 Efikasnost komunikacionih primitiva iniciranih od strane procesora proizvođača podataka

U radu [Shafi*97] analizirana je efikasnost tehnika koje podržavaju komunikaciju iniciranu od strane procesora proizvođača podataka, kao i efikasnost kombinovane primene ovih tehnika sa dohvatanjem podataka unapred kod CC-NUMA multiprocesora. Za podršku prosleđivanju predložene su dve instrukcije `WriteSend` i `WriteThrough`. Prva instrukcija je neznatno modifikovana verzija `wr-forw` instrukcije i specificira samo jedan procesor kome treba proslediti keš blok. Ukoliko je potrebno izvršiti prosleđivanje većem broju procesora, onda se mora koristiti više odvojenih `WriteSend` instrukcija. Ovaj pristup se opravdava kompleksnošću implementacije sa višestrukim prosleđivanjem i neskalabilnošću bit vektora. `WriteSend` instrukcija obavlja običan upis, a zatim inicira transakciju kojom se

keš blok prosleđuje specificiranom procesoru; novo stanje keš bloka je deljen (*Shared*). Pored ove instrukcije, podržana je i `WriteSendInv` instrukcija koja obezbeđuje ekskluzivno prosleđivanje keš bloka procesoru budućem korisniku, tj. kopija u keš memoriji procesora proizvođača se invaliduje. Međutim, kada sledeći korisnik podataka nije poznat predlaže se korišćenje `WriteThrough` primitive koja ažurira memoriju čvora domaćina (*home*). Ako se sigurno zna da procesor proizvođač podataka neće koristiti keš blok, onda je od interesa korišćenje `WriteThroughInv` instrukcija; na ovaj način može se smanjiti broj invalidacionih poruka na interkonekcionoj mreži.

Kao radno opterećenje u eksperimentima korišćene su paralelne aplikacije iz skupova SPLASH-2 [WooO*95], SPLASH [Singh*91] i Rice Parallel Compiler Group. Predložene instrukcije za prosleđivanje podataka, kao i instrukcije za dohvatanje podataka unapred se umeću u kôd na osnovu jednostavnih heuristika koje se oslanjaju na statičku analizu ponašanja paralelnih aplikacija. Posmatraju se polazni sistem (Base), i sistemi koji uključuju podršku prosleđivanju (RW – *remote writes*), dohvatanju unapred (PF - *prefetching*), kao i sistem koji kombinuje obe tehnike (RW+PF). Modeluje se CC-NUMA multiprocesor sa 32 čvora i privatnim dvonivoskim keš memorijama.

Dobijeni rezultati pokazuju da najbolje performanse ima sistem koji kombinuje primenu dohvatanja unapred i prosleđivanje podataka. Ubrzanje se kreće u opsegu od 10% do 48%, u odnosu na polazni sistem, odnosno između 3% i 28% u odnosu na PF sistem. Pored toga, saobraćaj na interkonekcionoj mreži je manji u odnosu na Base i PF sisteme.

2.3.9 Eksplicitna komunikacija kod multiprocesora sa deljenom memorijom

U radu [Rama*95] predložen je skup primitiva koje dozvoljavaju selektivno ažuriranje keš memorija više procesora u sistemu, kao i dohvatanje unapred bloka podataka proizvoljne dužine. Cilj predloženih primitiva je da se omogući dinamičko prilagođavanje tipa komunikacije (*invalidate vs. update*) u skladu sa karakteristikama aplikacija tokom izvršavanja, kako bi se ostvarilo maksimalno prikrivanje kašnjenja u pristupu memoriji.

Predložene primitive za eksplicitnu komunikaciju prikazane su na Sl. 2-26. Primitiva PSET_WRITE podržava prosleđivanje podataka i odgovara `wr-forw` instrukciji; naime, najpre se vrši ažuriranje memorije čvora domaćina, a nakon toga keš blok se prosleđuje svim procesorima koji su specificirani bit vektorom *pmask*. Primitiva SYNC_WRITE je specijalni slučaj primitive PSET_WRITE i koristi se u radu sa sinhro varijablama. Procesor koji je ekskluzivni vlasnik nekog *lock*-a ovom primitivom inicira prosleđivanje vlasništva nad *lock*-om prvom sledećem procesoru koji čeka na tom *lock*-u, tako što kontroler direktorijuma ažurira blok, invaliduje sve ostale kopije i prosleđuje ekskluzivnu kopiju prvom procesoru koji je blokiran na tom *lock*-u. Primitiva SEL_WRITE (*Selective Write*) je slična PSET_WRITE primitivi, s tim da se skup procesora kojima se prosleđuje keš blok ne specificira eksplicitno bit vektorom, već se određuje dinamički, na osnovu informacija iz direktorijuma. Tako, nakon pristizanja ovog zahteva direktorijum ažurira memoriju i keš memorije svih procesora koji imaju kopiju tog bloka, a na osnovu informacija iz odgovarajućeg ulaza direktorijuma. Primitive RSTREAM i RSTREAM_EX predstavljaju generalizaciju mehanizma dohvatanja podataka unapred. U do sada razmatranim primerima dohvatanje podataka unapred se incira na nivou keš bloka, dok ove primitive omogućavaju specificiranje dužine bloka koji se želi dohvatiti unapred. RSTREAM_EX se koristi za ekskluzivno dohvatanje podataka unapred. Hardver kontrolera direktorijuma treba da po prijemu ovakvog zahteva inicira odgovarajući broj transakcija na interkonekcionoj mreži.

Naziv primitive	Argumenti
PSET_WRITE	address, value, pmask
SYNC_WRITE	address, value, pmask
SEL_WRITE	address, value
RSTREAM	address, #bytes
RSTREAM_EX	address, #bytes

Sl. 2-26. Primitive za eksplicitnu komunikaciju kod multiprocesora sa deljenom memorijom.

Evaluacija efikasnosti predloženih primitiva za eksplicitnu komunikaciju urađena je na primeru paralelnih aplikacija iz skupa SPLASH-2 [WooO*95] i NAS Parallel Benchmark [Bail*91]. Polazne aplikacije su augmentirane predloženim primitivama ručno na osnovu jednostavnih heuristika u skladu sa ponašanjem aplikacija. Na nivou arhitekture modelira se CC-NUMA procesor sa 16 procesora. Eksperimentalna analiza je pokazala da se primenom predloženih primitiva u sistemima sa invalidacionim protokolima vreme blokiranja (*read-stall time*) značajno redukuje i približno odgovara vremenu koje je izmereno u sistemima koji su bazirani na ažuriranju (*update*).

Poglavlje 3

Mehanizam injektiranja

U ovom poglavlju opisana je predložena tehnika injektiranja. U odeljku 3.1 dati su osnovni razlozi za uvođenje predložene tehnike kroz analizu prednosti i mana postojećih tehnika i pregled do sada sprovedenih istraživanja. U odeljku 3.2 definisana je tehnika injektiranja, opisane instrukcije i hardverski resursi za podršku injektiranju i ilustrovana primena injektiranja na jednostavnom primeru. U odeljku 3.3 analizira se primena mehanizma injektiranja i njena efikasnost u poređenju sa dohvatanjem podataka unapred i prosleđivanjem podataka, na primeru jednostavne aplikacije sa izraženim deljenjem podataka. U odeljku 3.4 razmatra se posebno primena i efikasnost mehanizma injektiranja na sinhronizacione primitive *lock*, *unlock* i *barrier*. U odeljku 3.5 dat je kratak osvrt na problem razvoja algoritma programskog prevodioca za podršku mehanizmu injektiranja. U odeljku 3.6 diskutovana su potrebna hardverska proširenja neophodna da bi se podržao mehanizam injektiranja.

3.1 Motivacija

U ovom odeljku navedeni su osnovni razlozi za uvođenje nove tehnike za prikrivanje kašnjenja u pristupu memoriji kod multiprocesorskih sistema sa deljenom memorijom, nazvane mehanizam injektiranja u keš memoriju (*cache injection*). Predložena tehnika ima za cilj poboljšanje efikasnosti u prikrivanju kašnjenja u pristupu memoriji, pre svega kod multiprocesorskih sistema sa zajedničkom magistralom. Tehnika injektiranja koristi dobre osobine postojećih softverskih tehnika za dohvatanje unapred i prosleđivanje podataka i osobinu arhitektura sa zajedničkom magistralom i *snooping* mehanizmom za održavanje koherencije keš memorije. Pri tom, tehnika injektiranja zahteva minimalnu dodatnu hardversku kompleksnost u odnosu na resurse koji su potrebni za podršku postojećim tehnikama. Motivacija za uvođenje tehnike injektiranja data je najpre kroz analizu osobina postojećih tehnika, a potom i kroz osvrt na relevantna istraživanja u ovoj oblasti.

Poredeći osobine dohvatanja unapred i prosleđivanja podataka, treba istaći da dohvatanje podataka unapred omogućuje eliminisanje svih tipova promašaja u keš memoriji (*cold, conflict,* i *coherence*), dok prosleđivanje podataka može eliminisati samo promašaje usled deljenja podataka (*coherence*) i, u nekim slučajevima, promašaje usled prvog pristupa (*cold*). Pri tom, prosleđivanje podataka je potencijalno efikasnije od dohvatanja podataka unapred u

eliminisanju *coherence* promašaja. Glavni razlozi za to su sledeći: (a) prosleđivanje podataka budućem korisniku (ili korisnicima) se može inicirati odmah po završetku obrade podataka, (b) u opštem slučaju, primena prosleđivanja garantuje manje kašnjenje na interkonekcionoj mreži, (c) jednom `forward` instrukcijom se može inicirati slanje podataka ka nekoliko budućih korisnika, dok u slučaju dohvatanja unapred svaki korisnik podataka mora inicirati dohvatanje podataka za sebe. Međutim, prosleđivanje podataka zahteva znatno sofisticiraniju podršku programskog prevodioca, jer određivanje potencijalnih budućih korisnika zahteva kompleksnu analizu različitih programskih niti, za razliku od dohvatanja unapred koje podrazumeva analizu unutar samo jedne programske niti. Pored toga, u slučajevima dinamičkog menjanja uzorka deljenja, tokom izvršavanja paralelnog programa i pristupa sinhronizacionim varijablama nije moguće odrediti procesore buduće korisnike podataka.

Sa druge strane, dohvatanje unapred nije efikasno ukoliko adresa podatka koji se želi dohvatiti unapred nije poznata dovoljno pre trenutka stvarnog korišćenja. U idealnom slučaju dohvatanje podataka unapred treba inicirati tako da podatak pristigne u keš memoriju neposredno pre trenutka stvarnog korišćenja. Međutim, dohvatanje unapred može biti korisno ako je inicirano i posle idealnog trenutka; u tom slučaju kašnjenje u pristupu memoriji se samo delimično prikriva. Ovakva situacija naziva se zakasnelo dohvatanje podataka unapred (*too late issued prefetch*). U vezi sa trenutkom iniciranja dohvatanja podataka unapred može se javiti i problem prerano iniciranog dohvatanja podataka unapred (*too early issued prefetch*). Za razliku od kasno iniciranog dohvatanja unapred koje ipak može doprineti poboljšanju performanse, prerano inicirano dohvatanje ima samo negativne posledice na ukupnu performanse. U tom slučaju može se desiti da unapred dohvaćeni keš blok bude izbačen iz keš memorije usled politike zamene (*cache replacement policy*), ili invalidovan usled upisa nekog drugog procesora u taj keš blok. U oba slučaja rezultat dohvatanja podataka unapred je nepotrebno izvršavanje `prefetch` instrukcije i odgovarajuće transakcije čitanja na interkonekcionoj mreži. Pored toga, u slučaju invalidacije unapred dohvaćenog bloka, ukoliko je procesor koji inicira invalidaciju imao ekskluzivno pravo nad posmatranim blokom pre operacije dohvatanja unapred, onda degradiranje performanse uključuje i vreme za koje je taj procesor blokiran i dodatnu transakciju invalidacije na interkonekcionoj mreži.

Većina ranije pomenutih istraživanja prikazanih u poglavlju 2 ([Mowry94], [Koufa*96], [Shafi*97], [Tranc*96]) fokusirala su se na ispitivanje efikasnosti dohvatanja unapred i prosleđivanja podataka u CC-NUMA i CC-UMA multiprocesorskim sistemima. Međutim, do sada nisu izvedena istraživanja koja bi analizirala korišćenje i efikasnost ovih tehnika u multiprocesorima sa zajedničkom magistralom. Izuzetak je istraživanje prikazano u radu [Tulls*93] u kome se analizira potencijal dohvatanja unapred u multiprocesorima sa zajedničkom magistralom. Dobijeni rezultati su pokazali da je efikasnost dohvatanja unapred u multiprocesorima sa zajedničkom magistralom manja u odnosu na efikasnost u CC-(N)UMA multiprocesorima. Postoje tri osnovna razloga za ovakav rezultat. Prvo, dohvatanjem unapred pokušava se poboljšavanje performanse procesora smanjivanjem broja blokirajućih promašaja u keš memoriji. Smanjivanje broja blokirajućih promašaja često se postiže po cenu uvećavanja ukupnog broja promašaja u keš memoriji, što doprinosi uvećavanju kontencije na internim resursima procesora kao što su keš memorija, baferi, itd., a takođe doprinosi i povećavanju ukupnog saobraćaja na magistrali. Imajući u vidu da su performanse multiprocesora sa zajedničkom magistralom mnogo osetljivije na promene saobraćaja, uvećavanje saobraćaja može doprineti degradiranju ukupne performanse, bez obzira na smanjivanje broja blokirajućih promašaja u keš memoriji. Dalje, dohvatanje podataka unapred može negativno uticati na performanse kada postoji konflikt u keš memoriji između podataka koji se trenutno obrađuju i podataka koji su dohvaćeni unapred. Na kraju,

postojeći algoritmi za dohvatanje podataka unapred nisu toliko efikasni u eliminisanju promašaja koji su posledica invalidacije usled pravog deljenja podataka (*true sharing*). Zapravo, promašaji u keš memoriji usled pristupa pravim deljenim podacima predstavljaju glavnu prepreku postizanju veće efikasnosti dohvatanja podataka unapred, tim pre što njihov uticaj na ukupne performanse paralelnih programa raste sa porastom kapaciteta keš memorija kod modernih mikroprocesora. Sa druge strane, pored nekoliko istraživanja efikasnosti prosleđivanja podataka, do sada nije razvijen neki opšte prihvaćeni algoritam programskog prevodioca koji bi podržao ovu tehniku. Pored toga, do sada nije sprovedeno neko istraživanje u kojem bi se razmatrala efikasnost i implementacija prosleđivanja podataka kod multiprocesora sa zajedničkom magistralom.

Imajući u vidu sve prednosti i nedostatke postojećih tehnika za softversko dohvatanje podataka unapred i prosleđivanje podataka, kao i rezultate istraživanja u ovoj oblasti, u ovom radu predložena je nova tehnika nazvana injektiranje u keš memoriju (*cache injection*). Injektiranje u keš memoriju omogućava prevazilaženje glavnih nedostataka postojećih tehnika kao što su:

- povećana kontencija na memoriji i zajedničkoj magistrali usled dohvatanja podataka unapred i prosleđivanja podataka,
- niska efikasnost dohvatanja unapred za slučaj pravih deljenih podataka,
- nemogućnost da se uvek identifikuju procesori budući korisnici podataka, i
- kompleksnost implementacije prosleđivanja podataka kod multiprocesora sa zajedničkom magistralom.

Kako je tehnika injektiranja, pre svega, namenjena rešavanju problema pravih deljenih podataka i smanjivanju kontencije na memoriji i zajedničkoj magistrali, ona ne isključuje primenu postojećih tehnika za dohvatanje unapred i prosleđivanje podataka. Na ovaj način, primena tehnike injektiranja treba da obezbedi podizanje sveukupne efikasnosti softverskih tehnika za prikrivanje kašnjenja u pristupu memoriji u multiprocesorskim sistemima sa zajedničkom magistralom.

3.2 Mehanizam injektiranja

U ovom poglavlju prikazana je suština tehnike injektiranja u keš memoriju. Koristeći prednosti postojećih tehnika i osobinu arhitektura sa zajedničkom magistralom tehnika injektiranja eliminiše neke od nedostataka postojećih tehnika, kao što su negativan uticaj na deljene podatke, kontencija na magistrali, kompleksnost podrške u programskom prevodiocu i cena koja se plaća usled umetanja novih instrukcija.

Tehnika injektiranja podrazumeva uvođenje novog hardverskog resursa, nazvanog tabela injektiranja (*injection table*), koji je deo keš kontrolera. Na strani procesora potrošača podataka, prevodilac i/ili programer su odgovorni da preko odgovarajućih instrukcija unapred izraze "zainteresovanost" za podatke za koje se očekuje da će biti korišćeni. Ove instrukcije, za razliku od klasičnih `prefetch` instrukcija, ne iniciraju operaciju dohvatanja podataka, već samo inicijalizuju odgovarajuće ulaze tabele injektiranja (*injection table*). Tokom *snooping* faze ciklusa čitanja ili upisa na magistrali keš kontroler proverava sadržaj tabele injektiranja; ukoliko se adresa keš bloka tekuće transakcije na magistrali nalazi u tabeli injektiranja, procesor prihvata taj blok u svoju keš memoriju. Postoje dva osnovna scenarija kada dolazi do injektiranja: tokom ciklusa čitanja (*injection on first read*) i tokom softverski iniciranog ciklusa upisa na magistrali (*injection on write-back*).

Injektiranje tokom ciklusa čitanja. Ovaj scenario je od interesa kada više procesora u bliskom vremenskom intervalu čita iste podatke. U tom slučaju svi procesori inicijalizuju svoje tabele injektiranja pre trenutka stvarnog korišćenja podataka. Kada prvi od procesora čita podatak, imaće promašaj u keš memoriji i iniciraće dohvatanje deljenog keš bloka. Tokom ciklusa čitanja svi ostali procesori koji imaju validan ulaz u tabeli injektiranja sa adresom tog bloka prihvataju podatak u svoju keš memoriju; novo stanje keš bloka je deljen (*Shared*). Tako, tokom jednog ciklusa na magistrali, svi procesori prihvataju blok u svoju keš memoriju, a samo jedan od njih će biti blokiran, tj. videće promašaj u keš memoriji.

Injektiranje tokom ciklusa upisa. U slučaju da se deljeni podaci modifikuju (*write shared data*), mehanizam injektiranja podrazumeva akcije i na strani procesora proizvođača podataka i na strani procesora potrošača podataka. Kao u prethodnom slučaju, procesori potrošači podataka inicijalizuju svoje tabele injektiranja pre trenutka stvarnog korišćenja podataka. Sa druge strane, procesor proizvođač podataka po završetku obrade keš bloka inicira ciklus upisa u memoriju. Tokom ciklusa upisa na magistrali svi procesori koji imaju validan ulaz u svojim tabelama injektiranja prihvataju blok u svoju keš memoriju i proglašavaju ga deljenim (*Shared*).

U odeljku 3.2.1 opisane su predložene instrukcije za podršku mehanizmu injektiranja. Primena mehanizma injektiranja je ilustrovana korišćenjem jednostavnih primera koji su dati u odeljku 3.2.2.

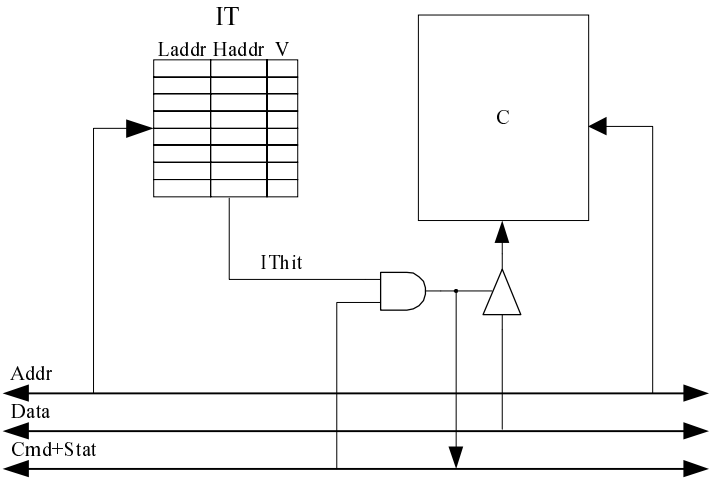
3.2.1 Predložene instrukcije

U ovom odeljku prikazane su predložene instrukcije neophodne za podršku tehnici injektiranja. Najpre su objašnjene instrukcije koje na strani procesora potrošača podataka treba da obezbede inicijalizaciju tabele injektiranja, a potom i instrukcije kojim procesor proizvođač podataka inicira ažuriranje memorije, a time i keš memorije "zainteresovanih" procesora, nakon poslednjeg upisa u posmatrani keš blok. Međutim, pre razmatranja predloženih instrukcija potrebno je ukratko prikazati osnovnu organizaciju predložene tabele injektiranja (Sl. 3-1).

Tabela injektiranja je organizovana kao FIFO bafer sa određenim brojem ulaza (16, 32, 64, itd.). Svaki ulaz u tabeli injektiranja poseduje dva adresna polja (*Laddr, Haddr*) i bit validnosti (*V*). Adresno polje *Laddr* definiše početnu, a polje *Haddr* poslednju adresu bloka (ili niza susednih blokova) koji se želi injektirati. Nakon inicijalizacije adresnih ulaza postavlja se bit validnosti *V*.

Na Sl. 3-2 dat je kratak opis predloženih instrukcija koje se koriste na strani procesora potrošača podataka. Procesor potrošač podataka koristi `OpenWindow` instrukciju za definisanje adresnog opsega jednog keš bloka ili niza susednih keš blokova koji se žele injektirati u keš memoriju. Instrukcija `OpenWindow` specificira početnu (*Laddr*) i poslednju adresu (*Haddr*) keš bloka ili niza susednih keš blokova. Definisanje adresnog prozora koji obuhvata niz susednih keš blokova je naročito od interesa kada pravi deljeni podaci koji se razmenjuju između procesora (*migratory shared data*) ispoljavaju prostornu lokalnost. Izvršavanjem ove instrukcije inicijalizuju se adresna polja odgovarajućeg ulaza tabele injektiranja i postavlja bit validnosti. Treba napomenuti da ova instrukcija inicijalizuje tabelu injektiranja bez obzira da li se specificirani keš blok nalazi u keš memoriji ili ne. Jednom inicijalizovani ulaz u tabeli injektiranja ostaje aktivan sve dok ne bude izbačen iz tabele, tj. dok ne bude reinicijalizovan novom vrednošću ili dok se odgovarajući adresni prozor eksplicitno ne invaliduje. Instrukcija `CloseWindow` obezbeđuje invalidaciju aktivnog ulaza

u tabeli injektiranja. Ova instrukcija specificira samo početnu adresu bloka. Tokom izvršavanja ove instrukcije proverava se da li u tabeli injektiranja postoji aktivan prozor sa specificiranom početnom adresom. Ukoliko je to slučaj, taj ulaz se proglašava nevažećim brisanjem bita validnosti.



Sl. 3-1. Organizacija tabele injektiranja.
Opis: IT (*Injection Table*) – tabela injektiranja, C (*Cache*) – keš memorija, Addr – adresna magistrala, Data – magistrala podataka, Cmd+Stat – upravljačka magistrala, IThit – pogodak u tabeli injektiranja, V – bit validnosti, Laddr – adresno polje koje definiše početnu adresu, Haddr – adresno polje koje definiše poslednju adresu.

`openWindow(Laddr, Haddr):`
Ovom instrukcijom se inicijalizuje prvi slobodan ulaz tabele injektiranja. Instrukcija definiše prvu i poslednju adresu keš bloka ili niza susednih keš blokova. U odgovarajuća polja ulaza tabele injektiranja se smeštaju prva i poslednja adresa adresnog prozora koji se otvara za injektiranje; pored toga, bit validnosti V se postavlja na aktivnu vrednost (V=1).

`CloseWindow(Laddr):`
Proverava se da li postoji neki aktivan ulaz u tabeli injektiranja čije polje sa početnom adresom ima istu vrednost kao Laddr. Ukoliko takav ulaz ne postoji, ova instrukcija je bez dejstva. Ukoliko takav ulaz postoji on se proglašava nevažećim brisanjem bita validnosti (V=0).

Sl. 3-2. Predložene instrukcije za inicijalizaciju i invalidaciju ulaza tabele injektiranja.

Na Sl. 3-3 dat je kratak opis instrukcija kojim se inicira ažuriranje memorije nakon poslednjeg upisa u posmatrani keš blok od strane procesora proizvođača podataka. Instrukcija `Update` inicira ciklus upisa na magistrali, ukoliko se specificirani blok nalazi u keš memoriji u stanju M (*Modified*); u suprotnom, instrukcija se ponaša kao instrukcija bez dejstva. Instrukcija `StoreUpdate` omogućuje prevodiocu da minimizuje broj dodatnih instrukcija tako što poslednju instrukciju upisa u posmatrani keš blok zameni predloženom instrukcijom `StoreUpdate`. Ova instrukcija najpre izvrši običan upis, a zatim inicira operaciju za ažuriranje glavne memorije.

Mehanizam injektiranja obezbeđuje dohvatanje bloka podataka u stanju S (*Shared*). U cilju podrške ekskluzivnom dohvatanju podataka od interesa mogu biti instrukcije `UpdateInv` i `StoreUpdateInv` koje pored iniciranja ciklusa upisa, invaliduju keš blok u keš memoriji procesora proizvođača podataka. Međutim, u tom slučaju treba obezbediti da samo jedan procesor prihvati keš blok i tako postane ekskluzivni vlasnik. U ovoj tezi nije razmatrana primena ovih instrukcija.

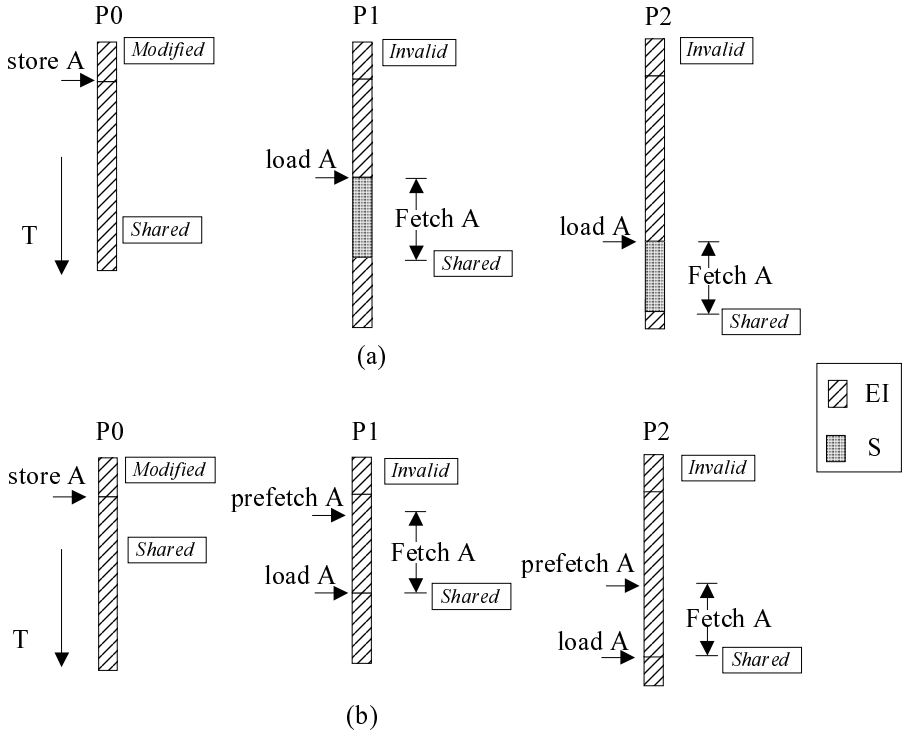
`Update(&A):`
Proverava se da li se podatak sa specificiranom adresom nalazi u keš memoriji. Ukoliko nije, instrukcija je bez dejstva, tj. ponaša se kao noop instrukcija. Ukoliko je keš blok u stanju M (Modified) inicira se write-back operacija na magistrali. Novo stanje bloka je S (Shared).

`StoreUpdate(&A):`
Izvrši se najpre klasična store instrukcija, a zatim se inicira write-back operacija na magistrali. Novo stanje keš bloka je S (Shared).

Sl. 3-3. Predložene instrukcije za ažuriranje memorije nakon poslednjeg upisa.

3.2.2 Primena predloženih instrukcija

U ovom odeljku ilustrovana je primena tehnike injektiranja. Posmatra se jednostavan primer koji demonstrira pravo deljenje podataka, prikazan na Sl. 3-4a. Procesor P0, proizvođač podataka, modifikuje, a procesori P1 i P2, potrošači podataka, čitaju podatak A. Uvedene su sledeće pretpostavke: (a) veličina keš bloka je jedna reč, (b) na početku, procesor P0 je ekskluzivni vlasnik podatka koji se nalazi u stanju M (*Modified*), i (c) srednje vreme trajanja operacije čitanja na magistrali je T_{rdc}. Posmatra se interval od poslednje promene podatka od strane procesora P0 do čitanja podatka od strane procesora P1 i P2. Pokazatelji performanse su ukupno vreme za koje su procesori blokirani i ukupan saobraćaj na magistrali. Procesori P1 i P2 u trenutku čitanja ne nalaze podatak A u svom kešu, pa iniciraju operacije čitanja na magistrali. Za vreme trajanja operacija čitanja, oba procesora su blokirana čekajući na završetak dohvatanja traženog podatka. Ukupno vreme blokiranja je $T_{blocked} = 2 \cdot T_{rdc}$, a ukupni saobraćaj iznosi 2 ciklusa čitanja na magistrali.



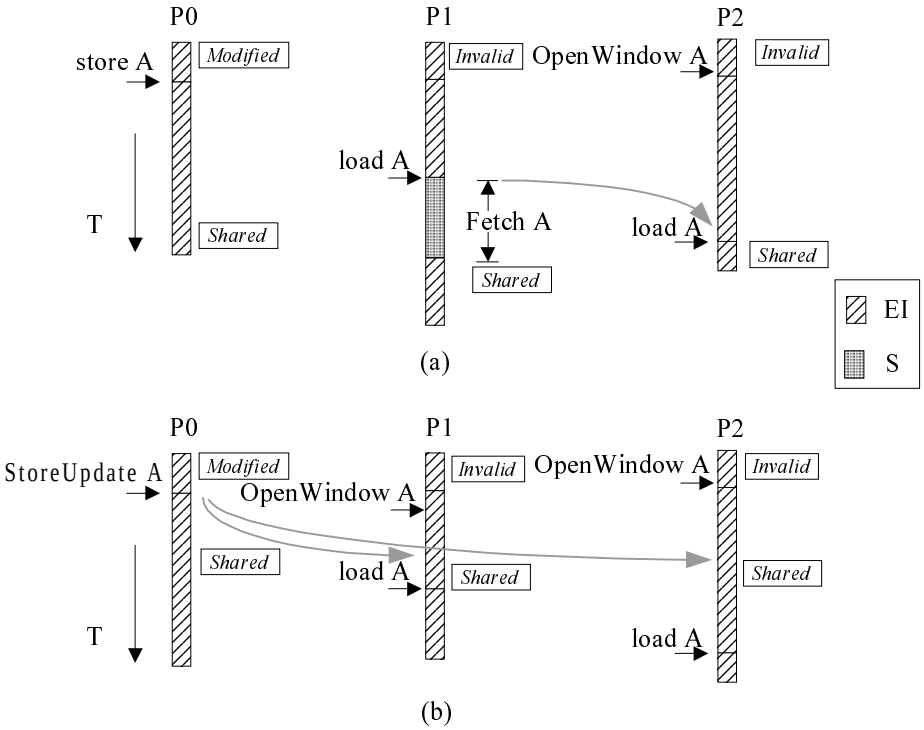
Sl. 3-4. Primer koji demonstrira pravo deljenje podataka.
Opis: (a) polazni primer, (b) nakon primene dohvatanja unapred.

Na Sl. 3-4b prikazan je polazni primer nakon primene klasičnog dohvatanja podataka unapred. Dovoljno pre trenutka stvarnog korišćenja podatka procesori P1 i P2 iniciraju

dohvatanje podataka unapred `prefetch` instrukcijom. Izvršavanjem `prefetch` instrukcije iniciraju se operacije čitanja na magistrali; pri tom, procesori P1 i P2 nastavljaju sa izvršavanjem odgovarajućih programskih niti. Pod pretpostavkom da je dohvatanje podataka unapred inicirano dovoljno unapred da se podatak u trenutku stvarnog korišćenja već nalazi u lokalnoj keš memoriji, a nakon poslednjeg upisa podatka od strane procesora P0, ukupno vreme blokiranja procesora je $T_{blocked} = 0$; međutim, saobraćaj na magistrali ostaje nepromenjen i iznosi 2 ciklusa čitanja.

Na Sl. 3-5a ilustrovana je primena injektiranja tokom ciklusa čitanja. Pretpostavlja se da čitanje podatka `A` od strane procesora P1 prethodi čitanju istog podataka od strane procesora P2, kao i da procesor P2 inicijalizuje tabelu injektiranja instrukcijom `OpenWindow` pre trenutka stvarnog čitanja. Procesor P1 izvršavajući `load` instrukciju ne nalazi podatak u svojoj keš memoriji, pa inicira ciklus čitanja. Tokom *snooping* faze ciklusa čitanja procesor P2 vidi da adresa bloka koji se dohvata iz memorije odgovora adresi u tabeli injektiranja, pa prihvata blok u svoju keš memoriju; novo stanje keš bloka je S (*Shared*). Tako, u trenutku stvarnog izvršavanja procesor P2 nalazi podatak u svojoj keš memoriji, pa je ukupno vreme blokiranja $T_{blocked} = T_{rdc}$, a ukupni saobraćaj na magistrali je 1 ciklus čitanja.

Na Sl. 3-5b prikazan je polazni primer, modifikovan da podrži injektiranje tokom ciklusa upisa na magistrali. Procesori P1 i P2 koristeći `OpenWindow` instrukciju inicijalizuju svoje tabele injektiranja. Procesor P0 umesto poslednje `store` instrukcije umeće `StoreUpdate` instrukciju koja inicira ciklus upisa na magistrali. Tokom ciklusa upisa na magistrali, procesori P1 i P2 vide da se adresa tekućeg bloka koji se vraća u memoriju nalazi u njihovim tabelama injektiranja, pa prihvataju podatak `A` u svoje keš memorije. Tako, u trenutku stvarnog korišćenja procesori P1 i P2 nalaze blok `A` u svojoj keš memoriji, pa nema blokiranja izvršavanja programske niti, tj. $T_{blocked} = 0$. Ukupni saobraćaj na magistrali iznosi 1 ciklus upisa. U prikazanom primeru uzeto je da procesor P2 inicijalizuje tabelu injektiranja pre trenutka završetka obrade podatka od procesora P0. U ovom slučaju ne dolazi do neželjenih posledica kao kod dohvatanja podataka unapred.



Sl. 3-5. Polazni primer nakon primene injektiranja u keš.
Opis: (a) Injektiranje tokom ciklusa čitanja, (b) Injektiranje tokom softverski iniciranog ažuriranja glavne memorije.

3.3 Primer primene injektiranja na prave deljene podatke

U ovom odeljku razmatra se primena postojećih tehnika dohvatanja unapred i prosleđivanja podataka budućim korisnicima i predložene tehnike injektiranja na jednostavnom primeru koji demonstrira pravo deljenje podataka. Pri tom, izvedena je približna analiza performanse koja podrazumeva određivanje ukupnog broja blokirajućih i neblokirajućih promašaja u keš memoriji i zahteva za invalidacijom, kao i saobraćaja koji se generiše na zajedničkoj magistrali.

Odeljak 3.3.1 prikazuje polazni primer i definiše uslove i pretpostavke analize koja sledi. Odeljci 3.3.2, i 3.3.3 prikazuju polazni primer nakon primene tehnika za dohvatanje podataka unapred i prosleđivanje podataka, redom, a odeljak 3.3.4 sadrži polazni primer nakon kombinovane primene obe postojeće tehnike. Odeljak 3.3.5 sadrži opis polaznog segmenta programa nakon primene mehanizma injektiranja. Uporedni prikaz rezultata dobijenih približnom analizom performanse prikazan je u odeljku 3.3.6.

3.3.1 Polazni primer (Base)

Na Sl. 3-6 prikazan je segment paralelnog programa koji ilustruje pravo deljenje podataka. Svaki od `NumProcs` procesora modifikuje po jednu vrstu matrice `A` na osnovu vrednosti `myVal` koja se izračunava na osnovu cele matrice `A`; ceo postupak se ponavlja `t_max` puta. Barijere se koriste za sinhronizaciju procesora između faze izračunavanja vrednosti `myVal` i faze u kojoj se modifikuju vrste matrice. Detaljna objašnjenja posmatranog primera data su u odeljku 2.3.2 (Sl. 2-17).

```
shared double A[NumProcs][100];
for(t=0; i<t_max; t++) {
   local double myVal =0.0
   for(p=0; p<NumProcs; p++) {
      for(i=0; i<100; i++)
         myVal+=foo(A[p][i], MyProcNum];
   }
   barrier(B, NumProcs);
   for(i=0; i<100; i++)
      A[MyProcNum][i]=goo(A[MyProcNum][i],myVal);
   barrier(B, NumProcs);
}
```

Sl. 3-6. Paralelni program koji ilustruje pravo deljenje podataka.

Usvojene su sledeće pretpostavke. Posmatrani paralelni program se izvršava na multiprocesorskom sistemu sa zajedničkom magistralom koji koristi MESI *write-back invalidate* protokol za održavanje koherencije keš memorije. Dužina keš linije je 16B, a elementi matrice A se smeštaju u memoriju po vrstama, tako da elementi A[i][2j] i A[i][2j+1] pripadaju istom bloku u keš memoriji. Za prikrivanje kašnjenja usled čitanja jednog keš bloka iz memorije potrebno je 6 iteracija unutrašnje petlje (po promenljivoj `i`) originalnog paralelnog programa. Adrese su dužine 4B, a komanda na magistrali je dužine 1B. Takođe, pretpostavlja se da je kapacitet keš memorije procesora dovoljan da prihvati celu matricu A.

Približna analiza performanse memorijskog podsistema podrazumeva određivanje ukupnog broja blokirajućih i neblokirajućih promašaja u keš memoriji i zahteva za invalidacijom, kao i saobraćaja koji se generiše na zajedničkoj magistrali. Radi jednostavnosti analiziraju se samo pristupi deljenoj matrici A. U prvom delu spoljašnje petlje po promenljivoj `t` svaki procesor čita celu matricu A, pa kako keš blok sadrži dva elementa matrice, a elementima se pristupa po vrstama, ukupan broj blokirajućih promašaja u kešu je $NumProcs \cdot 50$ u prvoj iteraciji, odnosno $(NumProcs-1)\cdot 50$ u svim ostalim iteracijama. U trenutku pristizanja na prvu barijeru, svi procesori imaju matricu A u svom kešu, i to u stanju S (*Shared*). U drugom delu petlje svakom procesoru je pridružena jedna vrsta matrice koju modifikuje na osnovu prethodno izračunate vrednosti `myVal`. Upis u parne elemente pridružene vrste matrice inicira invalidaciju kopija tog bloka u keš memorijama svih ostalih procesora; prema tome, svaki procesor u jednoj iteraciji spoljašnje petlje inicira 50 operacija za invalidaciju. Treba napomenuti da u multiprocesorima sa sekvencijalnim modelom konzistencije memorije, operacija invalidacije blokira izvršavanje programske niti za vreme trajanja te operacije na magistrali. Tako, ukupan broj blokirajućih promašaja u keš memoriji jednog procesora usled čitanja elemenata matrice A iznosi $N_{BRM} = [NumProcs + (t\_max-1)\cdot(NumProcs-1)]\cdot 50$; ukupan broj blokirajućih invalidacija je $N_{BINV} = 50 \cdot t\_max$, ukoliko se koristi sekvencijalna memorijska konzistencija; u slučaju *release* modela memorijske konzistencije operacije za invalidaciju ne blokiraju izvršavanje programske niti.

Na osnovu usvojenih pretpostavki operacija čitanja na zajedničkoj magistrali uzima 21B (4B za adresu, 1B za komandu i 16B za podatke), a operacija invalidacije uzima 5B (4B za adresu i 1B za komandu). Ukupan saobraćaj na magistrali tokom izvršavanja paralelnog programa je $Traffic = [N_{RM}\cdot 21 + N_{INV}\cdot 5]\cdot NumProcs$. Važan parametar performanse je vreme blokiranja procesora usled promašaja u keš memoriji ili operacije invalidacije. Vreme čekanja jednog procesora približno se može izračunati kao $T_{stall} = N_{BRM}\cdot T_{RM} + N_{BINV}\cdot T_{INV}$, pri čemu su T_{RM} i T_{INV} prosečna vremena trajanja operacija čitanja i invalidacije, redom, merena brojem procesorskih ciklusa.

3.3.2 Primena dohvatanja podataka unapred (Pref)

Na Sl. 3-7 prikazan je posmatrani paralelni program nakon umetanja instrukcija za dohvatanje podataka unapred. Koristi se tehnika softverske protočnosti i činjenica da je za prikrivanje kašnjenja u pristupu memoriji potrebno 6 iteracija unutrašnje petlje originalnog paralelnog programa. Na ovaj način eliminišu se svi blokirajući promašaji u keš memoriji u prvom delu iteracije. Analogno polaznom primeru, u trenutku pristizanja na prvu barijeru svi procesori imaju matricu `A` u svom kešu u stanju S (*Shared*). U drugom delu petlje procesor inicira invalidaciju kopija vrste koju modifikuje u keš memorijama svih ostalih procesora. Treba napomenuti da dohvatanje unapred elemenata vrste matrice koja pripada datom procesoru ima smisla samo u prvoj iteraciji spoljašnje petlje; u svim ostalim iteracijama `prefetch` instrukcija pronalazi elemente matične vrste u lokalnoj keš memoriji, tako da se neće inicirati zahtev na magistrali, tj. ponašaće se kao `noop` instrukcija. Dodatna transformacija originalnog koda može obezbediti eliminaciju redundantnih `prefetch` instrukcija. Međutim, ukoliko je kompleksnost takve transformacije znatno veća od gubitka usled izvršavanja nepotrebnih `prefetch` instrukcija, onda je bolje ne vršiti dodatne transformacije.

Kako je već ranije rečeno, ukoliko se koristi sekvencijalni model konzistencije memorije, svaki upis u parne elemente vrste inicira operaciju invalidacije koja blokira izvršavanje programske niti na datom procesoru. Međutim, blokiranje procesora se može izbeći primenom `prefetch-ex` instrukcije. Ukoliko pretpostavimo da su dve iteracije unutrašnje petlje (po promenljivoj `i`) dovoljne da prikriju trajanje transakcije za invalidaciju, prestruktuiranjem petlje dobija se kôd prikazan na Sl. 3-8.

```
shared double A[NumProcs][100];
for(t=0; i<t_max; t++) {
   local double myVal =0.0
   for(p=0; p<NumProcs; p++) {
      for(i=0; i<6; i+=2)
         prefetch(&A[p][i]);
      for(i=0; i<94; i+=2) {
         prefetch(&A[p][i+6]);
         myVal+=foo(A[p][i], MyProcNum];
         myVal+=foo(A[p][i+1], MyProcNum];
      }
      for(i=94; i<100; i+=2) {
         myVal+=foo(A[p][i], MyProcNum];
         myVal+=foo(A[p][i+1], MyProcNum];
      }
   }
   barrier(B, NumProcs);
   for(i=0; i<100; i++)
      A[MyProcNum][i]=goo(A[MyProcNum][i],myVal);
   barrier(B, NumProcs);
}
```

Sl. 3-7. Paralelni program nakon primene algoritma za selektivno umetanje `prefetch` instrukcija.

Primenom tehnike dohvatanja podataka u prvom delu petlje skoro u potpunosti se izbegava blokiranje procesora usled promašaja u keš memoriji prilikom čitanja elemenata matrice `A`. Takođe, primenom ekskluzivnog dohvatanja podataka unapred izbegava se blokiranje procesora tokom trajanja operacija invalidacije, pa je $T_{stall} = 0$. Međutim, ukupni saobraćaj na magistrali generisan tokom izvršavanja modifikovanog paralelnog programa je isti kao u polaznom primeru. Pored toga, prestruktuiranje kôda dovelo je do značajnog povećavanja dužine programa, a povećala se i kontencija na internim resursima usled izvršavanja dodatnih instrukcija za dohvatanje podataka unapred.

```
   ...
   barrier(B, NumProcs);
   prefetch-ex(&A[MyNumProc][0]);
   for(i=0; i<98; i+=2) {
     prefetch-ex(&A[MyNumProc][i+2]);
     A[MyProcNum][i]=goo(A[MyProcNum][i],myVal);
     A[MyProcNum][i+1]=goo(A[MyProcNum][i+1],myVal);
   }
   A[MyProcNum][98]=goo(A[MyProcNum][98],myVal);
   A[MyProcNum][99]=goo(A[MyProcNum][99],myVal);
  barrier(B, NumProcs);
}
```

Sl. 3-8. Prestruktuiranje kôda sa ciljem da se izbegne zaustavljanje procesora tokom invalidacije.

3.3.3 Primena prosleđivanja podataka (Forw)

Na Sl. 3-9 prikazan je paralelni program nakon umetanja odgovarajućih instrukcija za prosleđivanje podataka. Instrukcija `forward(&a,0)` prosleđuje keš blok sa adresom `a` u keš memoriju procesora P0; pri tom, novo stanje keš bloka je S (*Shared*). U ovom primeru je usvojeno da instrukcija za prosleđivanje dozvoljava specificiranje samo jednog odredišnog procesora; stoga, za svaki odredišni procesor potrebna je po jedna `forward` instrukcija. Nakon što je završena modifikacija jednog keš bloka inicira se prosleđivanje tog bloka svim ostalim procesorima; pri tom, izbegava se auto prosleđivanje. Treba napomenuti da nije neophodno eliminisanje instrukcija koje iniciraju auto prosleđivanje; naime, može se usvojiti implementacija instrukcije `forward` koja se u slučaju da procesor inicira prosleđivanje podataka samom sebi ponaša se kao `noop` instrukcija. Prosleđivanje svakom procesoru zasebno dovodi do zagušenja *write* bafera, što opet može dovesti do blokiranja procesora. Sa druge strane, implementacija instrukcije za prosleđivanje sa više odredišnih procesora, kao što je `forward(&A[MyNumProc][i],0,1,...NumProcs)`, rešava problem zagušenja *write* bafera, ali unosi značajnu dodatnu hardversku kompleksnost u implementaciju jedinice koja je odgovorna za iniciranje operacija na magistrali i same zajedničke magistrale.

```
shared double A[NumProcs][100];
for(t=0; i<t_max; t++) {
   local double myVal =0.0
   for(p=0; p<NumProcs; p++) {
      for(i=0; i<100; i++)
         myVal+=foo(A[p][i], MyProcNum];
   }
   barrier(B, NumProcs);
   for(i=0; i<100; i+=2)
     A[MyProcNum][i]=goo(A[MyProcNum][i],myVal);
     A[MyProcNum][i+1]=goo(A[MyProcNum][i+1],myVal);
     for(int j=0; j<NumProcs; j++)
       if(j!=MyProcNum)
         forward(&A[MyNumProcNum][i], j);
   barrier(B, NumProcs);
}
```

Sl. 3-9. Paralelni program nakon umetanja instrukcija za prosleđivanje.

Primenom tehnike prosleđivanja podataka, u idealnom slučaju, eliminiše se glavnina blokirajućih promašaja u keš memoriji prilikom čitanja elemenata matrice `A`. Međutim, u prvoj iteraciji spoljašnje petlje podaci se ne nalaze u keš memoriji, pa je ukupan broj blokirajućih promašaja jednog procesora prilikom čitanja, $N_{BRM} = NumProcs \cdot 50$. Takođe, tehnika prosleđivanja podataka ne rešava problem blokiranja procesora tokom invalidacije, pa je $N_{BINV} = 50 \cdot t\_max$. Na osnovu usvojenih pretpostavki `forward` instrukcija inicira operaciju na magistrali koja uzima 21B (4B za adresu, 1B za komandu i identifikator

odredišnog procesora i 16B za podatke). Ukupan saobraćaj na magistrali iznosi $Traffic = [N_{RM} \cdot 21 + N_{INV} \cdot 5 + N_{FWD} \cdot 21] \cdot NumProcs$, pri čemu je N_{FWD} broj transakcija prosleđivanja na magistrali; $N_{FWD} = 50 \cdot (NumProcs - 1) \cdot t\_max$.

Ukupni saobraćaj na magistrali tokom izvršavanja programa sa Sl. 3-9 je neznatno veći u poređenju sa polaznim primerom (Base) i primerom koji se dobija primenom dohvatanja podataka unapred (Pref). To je posledica nepotrebnog prosleđivanja u poslednjoj iteraciji spoljašnje petlje (`t=t_max-1`). Ovaj problem se može rešiti izdvajanjem poslednje iteracije, a odgovarajući program je prikazan na Sl. 3-10. Ovom modifikacijom saobraćaj na magistrali je isti kao u Base i Pref primerima.

```
shared double A[NumProcs][100];
for(t=0; i<t_max-1; t++) {
   local double myVal =0.0
   for(p=0; p<NumProcs; p++) {
      for(i=0; i<100; i++)
         myVal+=foo(A[p][i], MyProcNum];
   }
   barrier(B, NumProcs);
   for(i=0; i<100; i+=2) {
     A[MyProcNum][i]=goo(A[MyProcNum][i],myVal);
     A[MyProcNum][i+1]=goo(A[MyProcNum][i+1],myVal);
     for(int j=0; j<NumProcs; j++)
        if(j!=MyProcNum)
           forward(&A[MyNumProcNum][i], j);
   }
   barrier(B, NumProcs);
}
local double myVal =0.0
for(p=0; p<NumProcs; p++) {
   for(i=0; i<100; i++)
      myVal+=foo(A[p][i], MyProcNum];
   }
barrier(B, NumProcs);
for(i=0; i<100; i+=2){
   A[MyProcNum][i]=goo(A[MyProcNum][i],myVal);
   A[MyProcNum][i+1]=goo(A[MyProcNum][i+1],myVal);
}
barrier(B, NumProcs);
```

Sl. 3-10. Paralelni program bez redundantnih prosleđivanja.

3.3.4 Kombinovana primena dohvatanja unapred i prosleđivanja podataka (Pref+Forw)

Primena tehnike prosleđivanja podataka ne omogućava eliminisanje promašaja prilikom čitanja elemenata matrice `A` u prvoj iteraciji spoljašnje petlje (`t=0`); takođe, ova tehnika ne omogućava rešenje problema zaustavljanja procesora usled invalidacije deljenih kopija prilikom upisa u blok u stanju S (*Shared*). Međutim, tehnike dohvatanja unapred i prosleđivanja podataka mogu se kombinovati sa ciljem da se izbegnu navedeni problemi. Rezultujući program dobijen izdvajanjem prve (`t=0`) i poslednje (`t=t_max`) spoljašnje iteracije prikazan je na Sl. 3-11.

```
shared double A[NumProcs][100];
// prolog: t=0
local double myVal =0.0
for(p=0; p<NumProcs; p++) {
   for(i=0; i<6; i+=2)
      prefetch(&A[p][i]);
   for(i=0; i<94; i+=2) {
      prefetch(&A[p][i+6]);
      myVal+=foo(A[p][i], MyProcNum];
      myVal+=foo(A[p][i+1], MyProcNum];
   }
   for(i=94; i<100; i+=2) {
      myVal+=foo(A[p][i], MyProcNum];
      myVal+=foo(A[p][i+1], MyProcNum];
   }
}
barrier(B, NumProcs);
prefetch-ex(&A[MyNumProc][0]);
for(i=0; i<98; i+=2) {
   prefetch-ex(&A[MyNumProc][i+2]);
   A[MyProcNum][i]=goo(A[MyProcNum][i],myVal);
   A[MyProcNum][i+1]=goo(A[MyProcNum][i+1],myVal);
   for(int j=0; j<NumProcs; j++)
     if(j!=MyProcNum)
        forward(&A[MyNumProcNum][i], j);
}
A[MyProcNum][98]=goo(A[MyProcNum][98],myVal);
A[MyProcNum][99]=goo(A[MyProcNum][99],myVal);
for(int j=0; j<NumProcs; j++)
   if(j!=MyProcNum)
     forward(&A[MyNumProcNum][i], j);
barrier(B, NumProcs);
// glavno telo petlje
for(t=1; i<t_max-1; t++) {
   for(p=0; p<NumProcs; p++) {
      for(i=0; i<100; i++)
         myVal+=foo(A[p][i], MyProcNum];
   }
   barrier(B, NumProcs);
   for(i=0; i<100; i+=2){
     A[MyProcNum][i]=goo(A[MyProcNum][i],myVal);
     A[MyProcNum][i+1]=goo(A[MyProcNum][i+1],myVal);
     for(int j=0; j<NumProcs; j++)
       if(j!=MyProcNum)
          forward(&A[MyNumProcNum][i], j);
   }
   barrier(B, NumProcs);
}
// epilog: t=t_max
for(p=0; p<NumProcs; p++) {
   for(i=0; i<100; i++)
      myVal+=foo(A[p][i], MyProcNum];
}
barrier(B, NumProcs);
for(i=0; i<100; i++)
   A[MyProcNum][i]=goo(A[MyProcNum][i],myVal);
barrier(B, NumProcs);
```

Sl. 3-11. Kombinovanje prosleđivanja podataka sa dohvatanjem unapred.

3.3.5 Primena injektiranja (Inject)

Sve prethodne tehnike su bile jako efikasne u redukovanju broja blokirajućih promašaja u keš memoriji. Takođe, ekskluzivno dohvatanje podataka unapred je efikasno u eliminisanju vremena čekanja prilikom modifikovanja deljenog podatka u stanju S (*Shared*) za multiprocesore sa sekvencijalnim modelom memorijske konzistencije. Međutim, ukupni saobraćaj nije redukovan. Takođe, primena nekih tehnika, na primer prosleđivanja podataka, može dovesti do blokiranja izvršavanja programske niti usled zagušenja na nekom od internih

resursa procesora, na primer bafera za odloženi upis (*write buffer*). Pored toga, cena koja se plaća zbog povećavanja dužine koda usled reorganizacije nije zanemarljiva.

U posmatranom primeru procesor P0 je proizvođač elemenata nulte vrste matrice A, procesor P1 je proizvođač prve vrste matice A, itd. U sledećoj iteraciji svi procesori čitaju sve elemente matrice A. Ovakav tip deljenja podataka poznat je pod nazivom Proizvođač-Potrošač (*producer-consumer*). Primenom tehnike injektiranja, procesori "potrošači" predviđaju svoje buduće potrebe, ali pri tom ne iniciraju dohvatanje podataka unapred. Instrukcija `OpenWindow` definiše opseg adresa deljenih podataka za koje se očekuje da će biti korišćeni; početna i krajnja adresa adresnog prozora se smeštaju u tabelu injektiranja. U zavisnosti da li se inicira ažuriranje glavne memorije na strani proizvođača ili ne, moguća su dva pristupa: injektiranje tokom ciklusa upisa i injektiranje tokom ciklusa čitanja, redom.

Na Sl. 3-12 je prikazan modifikovan paralelni program proširen instrukcijama `OpenWindow(Laddr, Haddr)`i `CloseWindow(Laddr)`, tako da podrži injektiranje tokom ciklusa čitanja. Svaki procesor otvara adresni prozor koji obuhvata celu matricu A, tako da jedan procesor inicira ciklus čitanja na magistrali, a svi ostali tokom tog ciklusa prihvataju podatak u svoju keš memoriju. Na taj način, u svakoj iteraciji spoljašnje petlje svi procesori zajedno vide $NumProcs \cdot 50$, odnosno ukupan prosečan broj promašaja u keš memoriji po jednom procesoru je $N_{BRM} = 50 \cdot t\_max$. Broj blokirajućih invalidacija je $N_{BINV} = 50 \cdot t\_max$. Ukupan saobraćaj na magistrali je: $Traffic = [N_{BRM} \cdot 21 + N_{BINV} \cdot 5] \cdot NumProcs$.

Broj blokirajućih promašaja se može dalje smanjiti uvođenjem podrške za injektiranje tokom ciklusa ažuriranja memorije. Odgovarajući paralelni program je prikazan na Sl. 3-13. Kao i u prethodnom primeru, svaki procesor otvara adresni prozor koji obuhvata celu matricu A. Pored toga, drugi deo petlje modifikovan je tako da podrži ažuriranje glavne memorije koristeći instrukcije `Update`. Tokom ciklusa ažuriranja glavne memorije svi procesori nadgledaju zajedničku magistralu i proveravaju da li tekuća adresa na magistrali pripada nekom od otvorenih adresnih prozora u tabeli injektiranja; ukoliko je to slučaj, vrši se injektiranje podataka sa magistrale u lokalnu keš memoriju. U ovom slučaju prosečan broj promašaja u keš memoriji jednog procesora iznosi $N_{BRM} = 50$. Broj blokirajućih invalidacija je $N_{BINV} = 50 \cdot t\_max$. Ukupan saobraćaj na magistrali je:

$Traffic = [N_{BRM} \cdot 21 + N_{BINV} \cdot 5 + N_{WB} \cdot 21] \cdot NumProcs$, pri čemu je N_{WB} broj transakcija ažuriranja glavne memorije; $N_{WB} = 50 \cdot t\_max$.

```
shared double A[NumProcs][100];
OpenWindow(&A[0][0], &A[NumProcs-1][99]);
for(t=0; i<t_max; t++) {
   local double myVal =0.0
   for(p=0; p<NumProcs; p++) {
      for(i=0; i<100; i++)
         myVal+=foo(A[p][i], MyProcNum];
   }
   barrier(B, NumProcs);
   for(i=0; i<100; i++)
      A[MyProcNum][i]+=myVal;
   barrier(B, NumProcs);
}
CloseWindow(&A[0][0], &A[NumProcs-1][99]);
```

Sl. 3-12. Paralelni program modifikovan da podrži injektiranje tokom ciklusa čitanja.

```
shared double A[NumProcs][100];
OpenWindow(&A[0][0], &A[NumProcs-1][99]);
for(t=0; i<t_max; t++) {
   local double myVal =0.0
   for(p=0; p<NumProcs; p++) {
      for(i=0; i<100; i++)
         myVal+=foo(A[p][i], MyProcNum];
   }
   barrier(B, NumProcs);
   for(i=0; i<100; i+=2) {
      A[MyProcNum][i]+=myVal;
      A[MyProcNum][i+1]+=myVal;
      Update(&A[MyNumProcNum][i]);
   }
   barrier(B, NumProcs);
}
CloseWindow(&A[0][0], &A[NumProcs-1][99]);
```

Sl. 3-13. Paralelni program modifikovan da podrži injektiranje tokom ciklusa upisa.

Problem promašaja tokom prve spoljašnje iteracije (`t=0`) može se rešiti i kombinovanjem predloženog rešenja sa tehnikom dohvatanja unapred; međutim, imajući u vidu ukupan broj promašaja u keš memoriji, ovde to nije od preteranog interesa. Primenom ekskluzivnog dohvatanja podataka unapred moguće je eliminisati zaustavljanje procesora usled invalidacije. Izdvajanjem poslednje iteracije (`t=t_max`) može se izbeći ažuriranje glavne memorije u poslednjoj spoljašnjoj iteraciji. Kôd dobijen kombinovanjem dohvatanja unapred i tehnike injektiranja prikazan je na Sl. 3-14.

```
OpenWindow(&A[0][0], &A[NumProcs-1][99]);
for(t=0; i<t_max-1; t++) {
   for(p=0; p<NumProcs; p++) {
      for(i=0; i<100; i++)
         myVal+=foo(A[p][i], MyProcNum];
   }
   barrier(B, NumProcs);
   prefetch-ex(&A[MyNumProc][0]);
   for(i=0; i<98; i+=2) {
     prefetch-ex(&A[MyNumProc][i+2]);
     A[MyProcNum][i]=goo(A[MyProcNum][i],myVal);
     A[MyProcNum][i+1]=goo(A[MyProcNum][i+1],myVal);
     Update(&A[MyNumProcNum][i]);
   }
   A[MyProcNum][98]=goo(A[MyProcNum][98],myVal);
   A[MyProcNum][99]=goo(A[MyProcNum][99],myVal);
   Update(&A[MyNumProcNum][98]);
   barrier(B, NumProcs);
}
for(p=0; p<NumProcs; p++) {
   for(i=0; i<100; i++)
      myVal+=foo(A[p][i], MyProcNum];
}
barrier(B, NumProcs);
   prefetch-ex(&A[MyNumProc][0]);
   for(i=0; i<98; i+=2) {
     prefetch-ex(&A[MyNumProc][i+2]);
     A[MyProcNum][i]=goo(A[MyProcNum][i],myVal);
     A[MyProcNum][i+1]=goo(A[MyProcNum][i+1],myVal);
   }
   A[MyProcNum][98]=goo(A[MyProcNum][98],myVal);
   A[MyProcNum][99]=goo(A[MyProcNum][99],myVal);
barrier(B, NumProcs);
CloseWindow(&A[0][0], &A[NumProcs-1][99]);
```

Sl. 3-14. Paralelni program nakon kombinovanja tehnika injektiranja i ekskluzivnog dohvatanja unapred.

3.3.6 Uporedni prikaz efikasnosti razmatranih tehnika

U ovom odeljku dat je uporedni prikaz vremena blokiranja jednog procesora T_{stall} i ukupnog saobraćaja na magistrali *Traffic* tokom izvršavanja prikazanih verzija paralelnog programa: Base, Pref, Forw, Pref+Forw, InjectFR (*injection on first read*), InjectWB (*injection on write-back*), Inject+Pref. Takođe, navedena je kvalitativna procena kompleksnosti zahtevanih modifikacija polaznog programa. Usvojene su sledeće pretpostavke. Broj procesora (ujedno i broj vrsta deljene matrice A) je $NumProcs = 16$, $t\_max = 10$, $T_{RM} = 40\,\text{pclk}$ (procesorskih ciklusa), $T_{INV} = 5\,\text{pclk}$. Rezultati su prikazani na Sl. 3-15. Polazna verzija paralelnog programa (Base) prikazana je na Sl. 3-6. Paralelni programi prošireni instrukcijama za dohvatanje podataka unapred (Pref) i ekskluzivnim dohvatanjem unapred (Pref-Ex) prikazani su na Sl. 3-7 i Sl. 3-8, redom. Paralelni program sa neredundantnim prosleđivanjem podataka (Forw) prikazan je na Sl. 3-10, a paralelni program koji kombinuje primenu tehnika dohvatanja podataka unapred i prosleđivanja (Forw+Pref) prikazan je na Sl. 3-11. Primeri sa injektiranjem InjectFR i InjectWB prikazani su na Sl. 3-12 i Sl. 3-13, redom, a primer koji kombinuje injektiranje i ekskluzivno dohvatanje podataka unapred (Inject+Pref-Ex) prikazan je na Sl. 3-14.

	Base	Pref-Ex	Forw	Forw+ Pref-Ex	InjectFR	InjectWB	Inject+ Pref-Ex
T_{stall} [x10^3pclk]	304,5	≈0	34,5	≈0	22,5	4,5	2
Saobraćaj [x10^6B]	2,5768	2,5768	2,5768	2,5768	0,2017	0,2017	0,2017
Složenost koda	0	>>	>>	>>	0	>	>>

Sl. 3-15. Uporedni prikaz performanse različitih verzija paralelnog programa.
Opis: T_{stall} - vreme blokiranja jednog procesora, *Saobraćaj* - ukupni saobraćaj na magistrali i *Složenost koda* – kvalitativna procena složenosti umetnutog koda (>> - modifikacija koda je značajna, > - modifikacija koda je umerena, 0 – modifikacija koda je minimalna).

Dobijeni rezultati pokazuju da predložena tehnika injektiranja omogućuje značajnu redukciju vremena blokiranja i posebno saobraćaja na magistrali, što u velikoj meri određuje sveukupne performanse. Parametri T_{stall}, *Saobraćaj* i *Složenost koda* daju samo približnu procenu performanse; detaljna evaluacija simulacionom analizom omogućuje merenje uticaja predložene tehnike injektiranja na sveukupne performanse.

3.4 Injektiranje i sinhronizacija paralelnih programa

U odeljku 3.3 razmatrana je efikasnost tehnike injektiranja kada pravi deljeni podaci ispoljavaju Proizvođač-Potrošač tip deljenja podataka. U ovom poglavlju demonstrirana je primena tehnike injektiranja na sinhronizacione promenljive koje se koriste u implemenataciji primitiva za sinhronizaciju kao što su *lock, unlock* i *barrier*.

U odeljku 3.4.1 dat je osvrt na definicije i različite pristupe implementaciji sinhronizacionih primitiva *lock, unlock,* i *barrier*. U odeljku 3.4.2 analizirana je primena mehanizma injektiranja na sinhronizacione operacije *lock* i *unlock,* a u odeljku 3.4.3 na sinhronizacionu operaciju *barrier*.

3.4.1 Sinhronizacija paralelnih programa

Određivanje nivoa hardverske/softverske podrške sinhronizacionim primitivama predstavlja jednu od najinteresantnijih i najatraktivnijih oblasti istraživanja u oblasti multiprocesorskih sistema. Hardverska podrška garantuje visoke performanse, dok softverska podrška obezbeđuje visoku fleksibilnost i adaptabilnost. Gotovo sve sinhronizacione operacije počivaju na nekoj vrsti atomskih *read-modify-write* primitiva što podrazumeva čitanje, modifikovanje i ažuriranje memorijske lokacije, bez intervencije neke druge memorijske operacije. Za paralelne programe od posebnog značaja su sinhronizacione primitive *lock* i *unlock* koje se koriste u implementaciji kritičnih regiona, kao i primitiva *barrier* koja se koristi za globalnu sinhronizaciju.

Osnovne komponente nekog sinhronizacionog događaja su *acquire* metod, algoritam čekanja (*waiting algorithm*) i *release* metod. *Acquire* metod definiše način na koji neki proces pokušava da stekne pravo na sinhronizaciju, na primer da uđe u kritični region. Algoritam čekanja definiše ponašanje procesa dok čeka na sinhronizaciju; npr., ukoliko se već neki drugi proces nalazi u kritičnoj sekciji, posmatrani proces mora nekako čekati dok *lock* ne postane slobodan. *Release* metod definiše kako neki proces omogućava drugim procesima da izađu iz sinhronizacionog događaja; na primer, implementacija *unlock* operacije kojom se oslobađa *lock*, ili metod kojim poslednji pristigli proces na barijeru dozvoljava svim ostalim blokiranim procesima da nastave izvršavanje.

U opštem slučaju, postoje dva osnovna algoritma čekanja: zaposleno čekanje (*busy-waiting*) i blokiranje (*blocking*). Zaposleno čekanje znači da se proces nalazi u petlji u kojoj stalno proverava vrednost npr. *lock* varijable, sve dok ne nađe da je *lock* slobodan. Blokiranje znači da se proces blokira, a procesor prepušta nekom drugom procesu. *Release* omogućava buđenje blokiranog procesa. Konkretno okruženje određuje koji je algoritam bolji. U opštem slučaju, blokiranje podrazumeva veće troškove, jer suspendovanje i buđenje procesa podrazumeva pozivanje operativnog sistema, ali se zato omogućuje da procesor obavlja koristan posao izvršavanjem drugog procesa. Zaposlenim čekanjem se izbegavaju troškovi prilikom suspendovanja i buđenja procesa, ali se cena plaća procesorskim vremenom i kontencijom na memoriji usled pristupa tokom čekanja u petlji. Detaljna diskusija o osobinama navedenih algoritama čekanja data je u [Culler*98].

U odeljku 3.4.1.1 razmatraju se tipične implementacije *lock* i *unlock* primitiva kod multiprocesora sa zajedničkom magistralom. U odeljku 3.4.1.2 prikazana je jedna tipična implementacija *barrier* primitive.

3.4.1.1 Sinhronizacione primitive *Lock* i *Unlock*

Operacije *lock* i *unlock* obezbeđuju međusobno isključivanje procesa prilikom ulaska u kritični region. Postoji veliki broj algoritama koji se koriste u implementaciji *lock* i *unlock* sinhronizacionih primitiva. Najjednostavniji algoritmi su efikasni u uslovima male kontencije, ali neefikasni kada postoji visoka kontencija. Nasuprot tome, sofisticirani algoritmi garantuju relativno visoke performanse u uslovima visoke kontencije, ali su zato neefikasni u uslovima male kontencije. Jednostavni algoritmi za implementaciju *lock* i *unlock* primitiva su bazirani na korišćenju procesorskih instrukcija koje garantuju atomsko modifikovanje memorijske lokacije.

Stanje *lock*-a se pamti se u pridruženoj memorijskoj lokaciji (*lock* varijabla). Vrednost pridružene memorijske lokacije određuje stanje *lock*-a: 0 – *lock* je slobodan, 1 – *lock* je zauzet. Proces koji želi da dobije *lock* proverava vrednost *lock* varijable; ukoliko je *lock*

slobodan, proces pokušava da dobije *lock* tako što ga proglašava zauzetim upisujući jedinicu u *lock* varijablu. Ukoliko je *lock* zauzet proces čeka da *lock* postane slobodan koristeći algoritam čekanja. *Unlock* operacija na kraju kritične sekcije prosto upisuje vrednost 0 u *lock* varijablu i time *lock* postaje slobodan.

Tipična implementacija *lock* primitive počiva na korišćenju operacije za atomsko modifikovanje memorijske lokacije. U tom slučaju redosled koraka je sledeći: čita se specificirana memorijska lokacija i podatak smešta u registar, a druga vrednost, definisana instrukcijom ili dobijena kao neka funkcija pročitane vrednosti, smešta se u tu lokaciju. Pri tom, memorijske operacije čitanja i upisa su nedeljive, tj. ne može se izvršiti ni jedna druga memorijska operacija između njih. Tipična instrukcija koja se koristi u implementaciji sinhronizacionih operacija je `exch` koja atomski razmenjuje sadržaje specificiranog registra i memorijske lokacije. Implementacija *lock* i *unlock* operacija korišćenjem `exch` instrukcije prikazana je na Sl. 3-16.

```
          loadi R2, #1
lockit:   exch R2, location          /* atomska operacija*/
          bnez R2, lockit            /* provera vrednosti */

unlock:   store location, #0         /* upis 0 */
```

Sl. 3-16. Implementacija *lock* i *unlock* primitiva korišćenjem atomske `exch` instrukcije.

U implementaciji *lock* operacija kod starijih multiprocesora često je korišćena `test&set` instrukcija koja atomski testira vrednost specificirane memorijske lokacije i postavlja novu vrednost ako je uslov testa ispunjen. Tako, `test&set` instrukcija koja testira da li je vrednost specificirane memorijske lokacije 0 i postavlja vrednost na 1 može se koristiti na sličan način kao i instrukcija `exch`. U keš-koherentnim multiprocesorskim sistemima prikazane implementacije *lock* operacija su nedovoljno efikasne. Naime, problem sa `exch` instrukcijom je što svako ispitivanje vrednosti *lock* varijable rezultuje operacijom upisa u keš blok koji sadrži *lock* varijablu, bez obzira da li je *lock* slobodan ili nije; kako se posmatrani keš blok nalazi u keš memoriji nekog drugog procesora koji je poslednji izvršio `exch` instrukciju nad posmatranom *lock* varijablom, to svaka `exch` instrukcija rezultuje promašajem u keš memoriji usled upisa (*write miss*). Svaki procesor koji čeka na *lock*, repetitivno u petlji izvršava `exch` instrukciju koja uključuje čitanje i upis. Ovakva implementacija *lock* operacije dovodi do zagušenja na zajedničkoj magistrali i degradiranja sveukupne performanse u uslovima kada više procesora istovremeno pokušava da dobije *lock*. Pored toga, zagušenje zajedničke magistrale usporava i napredovanje procesora koji se nalazi u kritičnoj sekciji, što opet značajno produžava vreme boravka u kritičnoj sekciji.

Da bi se izbeglo zagušenje magistrale mogu se učiniti dve stvari: (a) redukovati broj izdatih zahteva za *lock*-om tokom čekanja ili (b) koristiti instrukciju kojom se izbegava generisanje saobraćaja na magistrali u uslovima kada je *lock* zauzet. Prvi pristup podrazumeva da se nakon svakog neuspelog pokušaja da se dobije *lock* umetne čekanje pre ponovnog zahteva (*spin-lock with backoff*). Dužina čekanja može biti fiksna ili funkcija broja neuspelih pokušaja [Crumm*91]. Drugi pristup podrazumeva modifikovanje algoritma čekanja tako da svi procesori koji čekaju da *lock* postane slobodan testiraju lokalnu keš kopiju *lock* varijable, sve dok *lock* ne postane slobodan. Po oslobađanju *lock*-a procesori ulaze u nadmetanje za *lock* izvršavajući `exch` instrukciju. Samo jedan procesor izlazi kao pobednik nadmetanja, dok ostali ponovo ulaze u stanje čekanja na *lock*. Ovakav pristup, nazvan `test-and-exch`, podrazumeva da se najpre vrši testiranje *lock* varijable korišćenjem obične `load` instrukcije, a ako je *lock* slobodan onda se pokušava dobijanje *lock*-a instrukcijom `exch`. *Lock* i *unlock*

operacije napisane u pseudo-asemblerskom jeziku koristeći navedeni pristup prikazane su na Sl. 3-17.

```
lockit:   load R2, location          /* čitanje lock varijable */
          bnz R2, lockit             /* provera vrednosti */
          loadi R2, #1               /* čitanje 1 */
          exch R2, location          /* atomska operacija */
          bnz reg, lockit            /* ako lock nije dobijen, ponovi postupak */

unlock:   store location, #0         /* upis 0 */
```

Sl. 3-17. Implementacija *Lock* i *unlock* primitiva korišćenjem `test-and-exch` pristupa.

Međutim, i ova realizacija *lock* operacije ima nedostataka. Pre svega, nedostatak se ogleda u činjenici da nakon oslobađanja *lock*-a svi procesori koji su čekali na tom *lock*-u vide da je *lock* slobodan i približno u isto vreme izvršavaju `exch` instrukciju, mada će samo jedan od njih dobiti *lock*. Na taj način generiše se značajan saobraćaj na zajedničkoj magistrali. Stoga je od interesa izbegavanje neuspešnih pokušaja da se dobije *lock* koji generišu transakcije invalidacije na magistrali. Takođe, od interesa je postojanje podrške koja bi omogućila implementiranje širokog spektra atomskih operacija tipa *read-modify-write* kao što su *test&set*, *fetch&op* i *compare&swap*, umesto da se obezbede posebne instrukcije za svaku od navedenih operacija.

Moderni mikroprocesori poseduju instrukcije koje omogućavaju prevazilaženje oba navedena problema [Culler*98]. Koristi se par instrukcija za rad za varijablama za sinhronizaciju. Prva instrukcija `load-locked` (`load-linked`, `ll`) se koristi za čitanje sinhronizacione varijable iz memorije u registar. Iza ove instrukcije može doći proizvoljan broj različitih instrukcija kojim se menja pročitana vrednost u registru. Poslednja instrukcija u sekvenci instrukcija je druga specijalna instrukcija nazvana `store-conditional` (`sc`). Ova instrukcija upisuje vrednost registra nazad u memorijsku lokaciju, **ako i samo ako** ni jedan drugi procesor nije modifikovao vrednost te memorijske lokacije od trenutka kada je izvršena `ll` instrukcija nad tom memorijskom lokacijom. *Lock* i *unlock* operacije napisane u pseudo-asemblerskom jeziku koristeći `ll` i `sc` instrukcije prikazane su na Sl. 3-18.

```
lockit:   ll R2, location            /* load-linked čitanje lokacije location */
          bnz R2, lockit             /* ako je lock zauzet, pokušaj ponovo */
          load R2, #1
          sc location, R2            /* upis 1, uslovno */
          beqz R2, lockit            /* ako je sc neuspešan, pokušaj ponovo */

unlock:   store location, #0         /* upis 0 */
```

Sl. 3-18. Implementacija *lock* i *unlock* primitiva korišćenjem `ll` i `sc` instrukcija.

Jedna moguća implementacija ovih instrukcija je sledeća. Adresa specificirana instrukcijom `ll` upisuje se u poseban registar procesora, tzv. *link register*. U slučaju prekida ili invalidacije keš bloka čija adresa odgovara adresi u *link* registru, sadržaj *link* registra se briše. Instrukcija `sc` proverava da li adresa specificirana tom instrukcijom odgovara adresi u *link* registru. Ako je to slučaj, `sc` instrukcija je izvršena uspešno (dobija se *lock*), inače, instrukcija nije izvršena uspešno, tj. *lock* je zauzet.

3.4.1.2 Sinhronizaciona primitiva *Barrier*

Pored *lock* i *unlock* sinhronizacionih primitiva u paralelnim programima se često koristi primitiva za globalnu sinhronizaciju *barrier*. Barijera `BARRIER(B,N)` obezbeđuje globalnu sinhronizaciju N procesa nad barijerom B. Kada neki proces naiđe na barijeru, proverava se da

li je to poslednji (N-ti) proces koji je pristigao na barijeru; ukoliko nije, proces se zaustavlja čekajući u petlji da svih N procesa pristigne na barijeru. Kada poslednji proces pristigne na barijeru uradi se *release* svih N procesa. Tipična implementacija barijere se bazira na korišćenju dve *lock* varijable [Patte*96]. Za rad sa barijerama koriste se makroi prikazani na Sl. 3-19. Makro `BARDEC(B)` definiše strukturu podataka koja je pridružena barijeri `B`. *Lock* varijabla `counterlock` obezbeđuje kritičnu sekciju u kojoj se ažurira broj procesora pristiglih na barijeru koji se pamti u varijabli `sleepers`. *Lock* varijabla `sleeplock` obezbeđuje čekanje sve dok svi procesori ne stignu na barijeru. Makro BARINIT(B) vrši inicijalizaciju varijabli barijere: `counterlock=0` (slobodan), `sleeplock=1` (zauzet), `sleepers=0`. Makro `BARRIER(B, N)` inicira globalnu sinhronizaciju N procesa na barijeri B.

```
struct BarrierStruct {
        LOCKDEC(counterlock);
        LOCKDEC(sleeplock);
        int sleepers;
        };
...
#define BARDEC(B)               struct BarrierStruct B;
#define BARINIT(B)              sys_barrier_init(&B);
#define BARRIER(B,N)    sys_barrier(&B, N);
```

Sl. 3-19. Makroi za rad sa barijerama.

Postoji više različitih implementacija barijere. Na Sl. 3-20 prikazana je jedna tipična implementacija preuzeta iz ANL (*Argonne National Laboratory*) skupa makroa [Magdic97a]. Prvih (N-1) procesora koji dolaze na barijeru inkrementira varijablu `sleepers` u kritičnoj sekciji koja je štićena *lock*-om `counterlock`, a potom se blokiraju na *lock*-u `sleeplock`. Poslednji N-ti procesor koji dolazi na barijeru izvrši `lock(B->counterlock)` i inkrementira varijablu `sleepers`; kako detektuje da je poslednji pristigli procesor na barijeru, započinje *release* proces tako što dekrementira varijablu `sleepers` i oslobađa `sleeplock` dozvoljavajući na taj način sledećem procesoru da otpočne izlazak iz barijere. Sledeći procesor koji vidi `sleeplock` slobodan ulazi u kritičnu sekciju u kojoj dekrementira varijablu `sleepers` i ponovo oslobađa `sleeplock`. Postupak se ponavlja sve dok poslednji procesor ne izađe iz barijere. Poslednji procesor koji izlazi iz barijere oslobađa `counterlock` koji je zauzet tokom cele *release* faze (zauzet je od poslednje pristiglog procesa na barijeru), dok `sleeplock` ostaje zauzet. Ovakvo stanje `counterlock=0` (slobodan), `sleeplock=1` (zauzet), `sleepers=0` odgovara inicijalnom stanju barijere.

```
void sys_barrier(struct BarrierStruct *B, int N) {
        LOCK(B->counterlock)
        (B->sleepers)++;
        if (B->sleepers < N ) {
                UNLOCK(B->counterlock)
                LOCK(B->sleeplock)
                B->sleepers--;
                if(B->sleepers > 0) UNLOCK(B->sleeplock)
                else UNLOCK(B->counterlock)
                }
        else    {
                B->sleepers--;
                if(B->sleepers > 0) UNLOCK(B->sleeplock)
                else UNLOCK(B->counterlock)
                }
        }
```

Sl. 3-20. Jedna implementacija barijere.

3.4.2 Primena injektiranja u implementaciji sinhronizacionih operacija *lock* i *unlock*

U ovom odeljku data je približna kvantitativna analiza uticaja mehanizma injektiranja na poboljšanje performanse sinhronizacionih primitiva kod multiprocesorskih sistema sa zajedničkom magistralom. Za ilustraciju primene injektiranja na sinhronizacione operacije posmatra se sledeća sekvenca pseudokôda prikazana na Sl. 3-21. Kritični region se modelira prostim kašnjenjem (bez realnog posla) koje je određeno parametrom `d`. Posmatra se izvršavanje takve kritične sekcije, pri čemu je broj procesora u sistemu *N*. Pretpostavimo sledeći pojednostavljeni scenario događaja. Svi procesori pokušavaju da dobiju *lock*; samo jedan od njih dobija *lock*, dok ostali ulaze u stanje čekanja, pokušavajući da dobiju *lock*. Pretpostavimo da je broj neuspelih pokušaja da se dobije *lock* za svaki procesor u stanju čekanja (vreme odgovara izvršavanju jednog kritičnog regiona) jednak *k*. Nakon izvršavanja koda u kritičnom regionu, procesor vlasnik *lock*-a operacijom `unlock` oslobađa *lock*. Preostalih (*N*-1) procesora započinje nadmetanje za *lock*, a samo jedan od njih dobija *lock* i ulazi u kritičnu sekciju, dok (*N*-2) procesora ulazi u stanje čekanja, pokušavajući da dobije *lock*. Pretpostavimo, opet, da je broj neuspelih pokušaja da se dobije *lock* po jednom procesoru jednak *k* (ovo je pojednostavljenje jer *k* može da zavisi od broja procesora u stanju čekanja). Procesor vlasnik *lock*-a oslobađa *lock* operacijom `unlock`, a postupak se opet ponavlja na opisani način sve dok svih *N* procesora ne prođe kroz kritični region.

```
lock(L);
   critical-section(d);
unlock(L);
```

Sl. 3-21. Pseudokôd kritične sekcije.

Približna procena performanse izvodi se određivanjem generisanog saobraćaja na magistrali i vremena blokiranja procesora tokom izvršavanja kritičnog regiona sa Sl. 3-21. Posmatra se multiprocesorski sistem sa zajedničkom magistralom sa MESI *write-back invalidate* protokolom za održavanje koherencije keš memorije. U naredna dva odeljka analizirane su performanse test primera pre i nakon primene injektiranja za dve implementacije *lock* primitiva; `test&exch` se razmatra u odeljku 3.4.2.1, a `ll-sc` u odeljku 3.4.2.2.

3.4.2.1 Test-and-exch *lock*

U ovom odeljku data je približna procena performanse *test&exch* implementacije *lock* operacije, pre i nakon primene injektiranja. Na Sl. 3-22 dat je redosled relevantnih događaja tokom izvršavanja polaznog kritičnog regiona (Base) sa Sl. 3-21. Ne umanjujući opštost može se pretpostaviti da je redosled izvršavanja kritičnih sekcija sledeći: P_0, P_1, P_2, ..., P_{N-1}. Kolona *Korak* opisuje relevantna stanja tokom izvršavanja posmatrane kritične sekcije, prema usvojenim pretpostavkama. Kolona *Procesor(i)* definiše procesore koji iniciraju određeni događaj. Kolona *Broj[Događaj]* sadrži ukupan broj relevantnih događaja; mogući događaji su: [RdC]–*ReadCycle*, [RdXC]–*ReadExCycle*, [InvC]–*InvalidateCycle*, [WbC]–*WriteBackCycle*.

Korak	Instrukcija	Procesor(i)	Broj[Događaj]	Komentar
acquire	lock:: load reg, L	P_0, P_1, P_2, ..., P_{N-1}	N[RdC]	P_0 pobeđuje u nadmetanju za *lock*
	lock:: exch L	P_0	[InvC]	
	lock:: exch L	P_1, P_2, ..., P_{N-1}	(N-1)[RdXC]	
busy waiting	lock:: load reg, L	P_1, P_2, ..., P_{N-1}	(N-1)[RdC]	P_0 izvršava kritičnu sekciju; ostali procesori testiraju *lock*
unlock	unlock:: store L, #0	P_0	[RdXC]	P_0 oslobađa *lock*
acquire	lock:: load reg, L	P_1, P_2, ..., P_{N-1}	(N-1)[RdC]	P_1 pobeđuje u nadmetanju za *lock*
	lock:: exch L	P_1	[InvC]	
	lock:: exch L	P_2, ..., P_{N-1}	(N-2)[RdXC]	
busy waiting	lock:: load reg, L	P_2, P_3, ..., P_{N-1}	(N-2)[RdC]	P_1 izvršava kritičnu sekciju; ostali procesori testiraju *lock*
unlock	unlock:: store L, #0	P_1	[RdXC]	P_1 oslobađa *lock*
...	....	...	...	...
acquire	lock:: load reg, L	P_{N-1}	[RdC]	P_{N-1} dobija *lock*
	lock:: exch L	P_{N-1}	[InvC]	
busy waiting	---	---	---	P_{N-1} izvršava kritičnu sekciju
unlock	unlock:: store L, #0	P_{N-1}	----	P_{N-1} oslobađa *lock*

Sl. 3-22. Izvršavanje polazne kritične sekcije u slučaju *test&exch* implementacije *lock* primitive.

Na Sl. 3-23 prikazana je kritična sekcija nakon umetanja odgovarajućih instrukcija za inicijalizaciju tabele injektiranja. Pre ulaska u kritični region procesori inicijalizuju tabelu injektiranja sa adresom *lock* varijable *L*. Nakon izlaska iz kritičnog regiona procesori invaliduju odgovarajući ulaz u tabeli injektiranja. Na Sl. 3-24 dat je redosled događaja tokom izvršavanja posmatranog kritičnog regiona kada se primenjuje mehanizam injektiranja i to injektiranje tokom ciklusa čitanja (InjectFR). Treba napomenuti da pored dodavanja instrukcija za inicijalizaciju tabele injektiranja nema nikakvih drugih izmena. Implementacija *lock* primitiva je ista kao u polaznom slučaju. Međutim, kako se nakon upisa *lock* promenljive ta vrednost obično čita od strane drugih procesora, može se razmotriti i modifikovanje instrukcije `exch` tako da podrži ažuriranje svih drugih procesora i glavne memorije (`exch+Update`). Opis relevantnih akcija tokom izvršavanja kritičnog regiona za slučaj modifikovane `exch` instrukcije dat je na Sl. 3-25 (InjectWB).

```
OpenWindow(L);
lock(L);
   critical-section(d);
unlock(L);
CloseWindiow(L);
```

Sl. 3-23. Kritična sekcija sa podrškom mehanizmu injektiranja.

Uporedni prikaz broja relevantnih događaja za tri posmatrana slučaja Base, InjectFR i InjectWB prikazan je na Sl. 3-26. Dobijeni rezultati ukazuju da mehanizam injektiranja omogućuje eliminisanje kvadratne zavisnosti broja blokirajućih promašaja od broja procesora i time značajno utiče na performanse sinhronizacionih primitiva.

Korak	Instrukcija	Procesor(i)	Broj[Događaj]	Komentar
acquire	lock:: load reg, L	P_0, P_1, P_2, ..., P_{N-1}	[RdC]	P_0 pobeđuje u nadmetanju za *lock*
	lock:: exch L	P_0	[InvC]	
	lock:: exch L	P_1, P_2, ..., P_{N-1}	(N-1)[RdXC]	
busy waiting	lock:: load reg, L	P_1, P_2, ..., P_{N-1}	[RdC]	P_0 izvršava kritičnu sekciju; ostali procesori testiraju *lock*
unlock	unlock:: store L, #0	P_0	[RdXC]	P_0 oslobađa *lock*
acquire	lock:: load reg, L	P_1, P_2, ..., P_{N-1}	[RdC]	P_1 pobeđuje u nadmetanju za *lock*
	lock:: exch L	P_1	[InvC]	
	lock:: exch L	P_2, ..., P_{N-1}	(N-2)[RdXC]	
busy waiting	lock:: load reg, L	P_2, P_3, ..., P_{N-1}	[RdC]	P_1 izvršava kritičnu sekciju; ostali procesori testiraju *lock*
unlock	unlock:: store L, #0	P_1	[RdXC]	P_1 oslobađa *lock*
...	....	...	...	...
acquire	lock:: load reg, L	P_{N-1}	[RdC]	P_{N-1} dobija *lock*
	lock:: exch L	P_{N-1}	[InvC]	
busy waiting	---	---	---	P_{N-1} izvršava kritičnu sekciju
unlock	unlock:: store L, #0	P_{N-1}	----	P_{N-1} oslobađa *lock*

Sl. 3-24. Izvršavanje kritične sekcije sa injektiranjem u slučaju *test&exch* implementacije *lock* primitive.

Korak	Instrukcija	Procesor(i)	Broj[Događaj]	Komentar
acquire	lock:: load R2, L	P_0, P_1, P_2, ..., P_{N-1}	[RdC]	P_0 vidi promašaj; ostali procesori dobijaju podatak injektiranjem
	lock:: exch R2, L	P_0	[InvC], [WbC]	modifikovani test&set inicira *update*
	lock:: exch R2, L	P_1, P_2, ..., P_{N-1}	(N-1)[WbC], (N-1)[InvC]	
busy waiting	lock:: load R2, L	P_1, P_2, ..., P_{N-1}	---	P_0 izvršava kritičnu sekciju; ostali procesori testiraju *lock*
unlock	unlock:: store L, #0	P_0	[InvC], [WbC]	P_0 oslobađa *lock*
acquire	lock:: load R2, L	P_1, P_2, ..., P_{N-1}	---	
	lock:: exch R2. L	P_1	[InvC], [WbC]	
	lock:: exch R2, L	P_2, ..., P_{N-1}	(N-2)[WbC], (N-2)[InvC]	
busy waiting	lock:: load R2, L	P_2, P_3, ..., P_{N-1}	---	P_1 izvršava kritičnu sekciju; ostali procesori testiraju *lock*
unlock	unlock:: store L, #0	P_1	[InvC], [WbC]	P_1 oslobađa *lock*
...	....	...	...	...
acquire	lock:: load R2, L	P_{N-1}	---	P_{N-1} dobija *lock*
	lock:: exch R2, L	P_{N-1}	[InvC], [WbC]	
busy waiting	---	---	---	P_{N-1} izvršava kritičnu sekciju
unlock	unlock:: store L, #0	P_{N-1}	[InvC], [WbC]	P_{N-1} oslobađa *lock*

Sl. 3-25. Izvršavanje kritične sekcije sa injektiranjem u slučaju *test&exch* implementacije *lock* primitive sa modifikovanom `exch` instrukcijom.

Ex.	RdC	RdXC	InvC	WbC
Base	N^2	$N \cdot (N+1)/2$	N	-
InjectFR	$2 \cdot N - 1$	$N \cdot (N+1)/2$	N	-
InjectWb	1	-	$N \cdot (N+1)/3$	$N \cdot (N+1)/3$

Sl. 3-26. Saobraćaj na magistrali tokom izvršavanja kritičnih sekcija sa *test&exch* implementacijom *lock* primitive.
Opis: *Base* – polazna verzija test primera, *InjectFR* – verzija sa injektiranjem tokom ciklusa čitanja (klasična `exch` instrukcija) i *InjectWb* – verzija sa injektiranjem tokom softverski iniciranog ciklusa ažuriranja (modifikovana `exch` instrukcija).

3.4.2.2 LL-SC *lock*

U ovom odeljku razmatraju se performanse *lock* operacija implementiranih korišćenjem `ll-sc` para instrukcija, pre i nakon primene tehnike injektiranja. Posmatra se izvršavanje kritične sekcije prikazane na Sl. 3-21. Na Sl. 3-27 dat je redosled relevantnih događaja tokom izvršavanja polaznog kritičnog regiona pod ranije definisanim uslovima (Base), a na Sl. 3-28 redosled događaja tokom izvršavanja kritične sekcije sa podrškom mehanizmu injektiranja i nemodifikovanim instrukcijama `ll` i `sc` (InjectFR).

Korak	Instrukcija	Procesor(i)	Broj[Događaj]	Komentar
acquire	lock:: ll R2, L	P_0, P_1, P_2, ..., P_{N-1}	N[RdC]	P_0 pobeđuje u nadmetanju za *lock*
	lock:: sc L, R2	P_0	[InvC]	
	lock:: sc L, R2	P_1, P_2, ..., P_{N-1}	---	
busy waiting	lock:: ll R2, L	P_1, P_2, ..., P_{N-1}	(N-1)[RdC]	P_0 izvršava kritičnu sekciju; ostali procesori testiraju *lock*
unlock	unlock:: store L, #0	P_0	[InvC]	P_0 oslobađa *lock*
acquire	lock:: ll R2, L	P_1, P_2, ..., P_{N-1}	(N-1)[RdC]	P_1 pobeđuje u nadmetanju za *lock*
	lock:: sc L, R2	P_1	[InvC]	
	lock:: sc L, R2	P_2, ..., P_{N-1}	----	
busy waiting	lock:: ll R2, L	P_2, P_3, ..., P_{N-1}	(N-2)[RdC]	P_1 izvršava kritičnu sekciju; ostali procesori testiraju *lock*
unlock	unlock:: store L, #0	P_1	[InvC]	P_1 oslobađa *lock*
...	....	...	...	...
acquire	lock:: ll R2, L	P_{N-1}	[RdC]	P_{N-1} dobija *lock*
	lock:: sc L, R2	P_{N-1}	[InvC]	
busy waiting	---	---	---	P_{N-1} izvršava kritičnu sekciju
unlock	unlock:: store L, #0	P_{N-1}	----	P_{N-1} oslobađa *lock*

Sl. 3-27. Izvršavanje polazne kritične sekcije u slučaju `ll-sc` implementacije *lock* primitive.

Ukoliko se `sc` instrukcija modifikuje tako da nakon uspešnog izvršavanja inicira ažuriranje glavne memorije i keš memorija procesora koji imaju inicijalizovane tabele injektiranja, redosled relevantnih događaja će biti kao na Sl. 3-29 (InjectWB). Uporedni prikaz broja relevantnih događaja tokom izvršavanja kritične sekcije za tri posmatrana pristupa Base, InjectFR i InjectWB prikazan je na Sl. 3-30. Slično kao kod prethodne implementacije *lock* primitive, pokazuje se da je mehanizam injektiranja vrlo efikasan u redukovanju broja potrebnih transakcija na magistrali.

Korak	Instrukcija	Procesor(i)	Broj[Događaj]	Komentar
acquire	lock:: ll R2, L	P_0, P_1, P_2, ..., P_{N-1}	[RdC]	P_0 pobeđuje u nadmetanju za *lock*
	lock:: sc L, R2	P_0	[InvC]	
	lock:: sc L, R2	P_1, P_2, ..., P_{N-1}	---	
busy waiting	lock:: ll R2, L	P_1, P_2, ..., P_{N-1}	[RdC]	P_0 izvršava kritičnu sekciju; ostali procesori testiraju *lock*
unlock	unlock:: store L, #0	P_0	[InvC]	P_0 oslobađa *lock*
acquire	lock:: ll R2, L	P_1, P_2, ..., P_{N-1}	(N-1)[RdC]	P_1 pobeđuje u nadmetanju za *lock*
	lock:: sc L, R2	P_1	[InvC]	
	lock:: sc L, R2	P_2, ..., P_{N-1}	----	
busy waiting	lock:: ll R2, L	P_2, P_3, ..., P_{N-1}	[RdC]	P_1 izvršava kritičnu sekciju; ostali procesori testiraju *lock*
unlock	unlock:: store L, #0	P_1	[InvC]	P_1 oslobađa *lock*
...	....	...	...	...
acquire	lock:: ll R2, L	P_{N-1}	[RdC]	P_{N-1} dobija *lock*
	lock:: sc L, R2	P_{N-1}	[InvC]	
busy waiting	---	---	---	P_{N-1} izvršava kritičnu sekciju
unlock	unlock:: store L, #0	P_{N-1}	----	P_{N-1} oslobađa *lock*

Sl. 3-28. Izvršavanje kritične sekcije sa injektiranjem u slučaju `ll-sc` implementacije *lock* primitive.

Korak	Instrukcija	Procesor(i)	Broj[Događaj]	Komentar
acquire	lock:: ll R2, L	P_0, P_1, P_2, ..., P_{N-1}	[RdC]	P_0 pobeđuje u nadmetanju za *lock*
	lock:: sc L, R2	P_0	[InvC], [WbC]	
	lock:: sc L, R2	P_1, P_2, ..., P_{N-1}	---	
busy waiting	lock:: ll R2, L	P_1, P_2, ..., P_{N-1}	---	P_0 izvršava kritičnu sekciju; ostali procesori testiraju *lock*
unlock	unlock:: store L, #0	P_0	[InvC], [WbC]	P_0 oslobađa *lock*
acquire	lock:: ll R2, L	P_1, P_2, ..., P_{N-1}	---	P_1 pobeđuje u nadmetanju za *lock*
	lock:: sc L, R2	P_1	[InvC], [WbC]	
	lock:: sc L, R2	P_2, ..., P_{N-1}	---	
busy waiting	lock:: ll R2, L	P_2, P_3, ..., P_{N-1}	---	P_1 izvršava kritičnu sekciju; ostali procesori testiraju *lock*
unlock	unlock:: store L, #0	P_1	[InvC], [WbC]	P_1 oslobađa *lock*
...	....	...	...	...
acquire	lock:: ll R2, L	P_{N-1}	[InvC], [WbC]	P_{N-1} dobija *lock*
	lock:: sc L, R2	P_{N-1}	[InvC], [WbC]	
busy waiting	---	---	---	P_{N-1} izvršava kritičnu sekciju
unlock	unlock:: store L, #0	P_{N-1}	----	P_{N-1} oslobađa *lock*

Sl. 3-29. Izvršavanje kritične sekcije sa injektiranjem u slučaju `ll-sc` implementacije *lock* primitive sa modifikovanom `sc` instrukcijom.

Ex.	RdC	RdXC	InvC	WbC
Base	N^2	-	$2 \cdot N - 1$	-
InjectFR	$2 \cdot N - 1$	-	$2 \cdot N - 1$	-
InjectWb	1	-	$2 \cdot N - 1$	$2 \cdot N - 1$

Sl. 3-30. Saobraćaj na magistrali tokom izvršavanja različitih verzija test primera baziranih na `ll-sc` implementaciji *lock* primitive.
Opis: *Base* – polazna verzija test primera, *InjectFR* – verzija sa injektiranjem tokom ciklusa čitanja (klasična verzija `sc` instrukcije) i *InjectWb* – verzija sa injektiranjem tokom softverski iniciranog ciklusa ažuriranja (modifikovana `sc` instrukcija).

3.4.3 Primena injektiranja u implementaciji globalne sinhronizacione primitive *barrier*

Na Sl. 3-20 prikazana je jedna implementacija barijere. Kako se u implementaciji barijere koriste dve *lock* promenljive, poboljšanje primenom mehanizma injektiranja je posledica poboljšanja samih implementacija *lock* i *unlock* primitiva.

Pored toga, moguće je modifikovati implementaciju barijere u cilju da se podrži injektiranje deljene promenljive `sleepers` u kojoj se čuva broj procesora pristiglih na barijeru u *acquire* fazi, odnosno broj procesora koji su izašli iz barijere u *release* fazi. U tom cilju, nakon svake operacije koja inkrementira/dekrementira varijablu `sleepers` inicira se operacija ažuriranja glavne memorije instrukcijom `Update`. Pri tom, treba napomenuti da umetanje instrukcije `Update` može da bude redundantno, ukoliko se varijabla `sleepers` i `counterlock` ili `sleeplock` nalaze u istom keš bloku; u tom slučaju ažuriranje varijable `sleepers` može da nastane kao sporedni efekat ažuriranja *lock* varijabli koje je inicirano izvršavanjem modifikovanih *lock* i *unlock* operacija. Modifikovana implementacija barijere prikazana je na Sl. 3-31.

Primena mehanizma injektiranja kod primitiva *barrier* podrazumeva umetanje odgovarajućih instrukcija za inicijalizaciju tabele injektiranja. Na početku, potrebno je inicijalizovati tabelu injektiranja tako da se obezbedi injektiranje cele strukture podataka koja je pridružena primitivi *barrier* (Sl. 3-32). Na kraju dela programa u kome se javlja barijera vrši se deaktiviranje posmatranog ulaza tabele injektiranja.

Imajući u vidu obimnost broja pretpostavki koje treba uvesti da bi se dala približna procena performanse klasične i modifikovane barijere, kao i činjenicu da procena zavisi od implementacija *lock* i *unlock* operacija u ovom odeljku se ne analizira efikasnost mehanizma injektiranja. Performanse različitih implementacija barijera razmatraju se u delu koji je posvećen simulacionoj analizi.

```
void sys_barrier(struct BarrierStruct *B, int N) {
        LOCK(B->counterlock)
        (B->sleepers)++;
        update(B->sleepers);
        if (B->sleepers < N ) {
                UNLOCK(B->counterlock)
                LOCK(B->sleeplock)
                B->sleepers--;
                update(B->sleepers);
                if (B->sleepers > 0)
                        UNLOCK(B->sleeplock)
                else
                        UNLOCK(B->counterlock)
                }
        else    {
                B->sleepers--;
                update(B->sleeprs);
                if (B->sleepers > 0)
                        UNLOCK(B->sleeplock)
                else
                        UNLOCK(B->counterlock)
                }
        }
```

Sl. 3-31. Modifikovana implementacija barijere koja podržava mehanizam injektiranja.

```
OpenWindow(B->counterlock, B->sleepers);
....
Barrier(B, N);
   .......
Barrier(B, N);
........
CloseWindiow(B->counterlock);
```

Sl. 3-32. Deo programa koji obezbeđuje podršku mehanizmu injektiranja za barijere.

3.5 Podrška mehanizmu injektiranja u programskom prevodiocu

Pitanje potencijalne podrške mehanizmu injektiranja od strane programskog prevodioca za paralelne programe ostaje glavno pitanje koje treba razmatrati u budućim istraživanjima. Cilj ove teze je da predloži novi mehanizam i da ispita opravdanost takvog mehanizma na osnovu ručnog umetanja predloženih instrukcija u skladu sa statičkom analizom paralelnih aplikacija i deljenja podataka. U ovom odeljku dat je osvrt na pitanja podrške mehanizmu injektiranja.

Sa gledišta kompleksnosti podrške programskog prevodioca mehanizmu injektiranja najjednostavniji pristup je korišćenje makroa pridruženih predloženim instrukcijama `OpenWindow`, `CloseWindow`, `Update` i `StoreUpdate`. U tom slučaju odgovornost za umetanje ovih makroa ima programer. Međutim, pristup u kome programer preuzima odgovornost za podršku tehnikama za prikrivanje kašnjenja u pristupu memoriji nije naročito popularan zbog dodatnog posla koji se nameće programeru, i pored toga što je ta podrška često trivijalna i ne zahteva nikakve dodatne informacije od programera koje već nisu sadržane u njegovoj glavi u procesu programiranja paralelne aplikacije, naročito u aplikacijama sa eksplicitnom sinhronizacijom.

Pri tom, treba napomenuti da se podrška injektiranju kod sinhronizacionih varijabli *lock, unlock* i *barrier* može u potpunosti realizovati makroima. Naime, kako je pokazano u odeljku 3.4.2 podrška injektiranju kod kritičnih regiona se bazira na tome da se pre ulaska u kritični region otvori ulaz u tabeli injektiranja koji sadrži adresu *lock* varijable; po izlasku iz kritične sekcije taj ulaz se invaliduje. Sličan mehanizam se koristi kod sinhronizacije tipa *barrier*. Na osnovu toga, jasno je da se odgovarajuće instrukcije za podršku injektiranju mogu ugraditi u

makroe za sinhronizaciju. Tako, prva instrukcija *lock* makroa treba da bude `OpenWindow` koja otvara prozor u tabeli injektiranja. Dalje, poslednja instrukcija *unlock* makroa treba da bude instrukcija `CloseWindow` koja invaliduje odgovarajući ulaz u tabeli injektiranja. U zavisnosti od implementacije sinhronizacione operacije *barrier* (vidi odeljak 3.4.1.2) mogu se koristiti modifikovane implementacije *lock* i *unlock* primitiva. Drugi pristup je da se u makro sa Sl. 3-20 na početak i na kraj dodaju odgovarajuće `OpenWindow` i `CloseWindow` instrukcije (vidi odeljak 3.4.3).

Iz gore navedenog, jasno je da se problem podrške mehanizmu injektiranja sinhronizacionim varijablama može uspešno rešiti korišćenjem makroa. Prema tome, podrška injektiranju pravih deljenih podataka ostaje jedini pravi izazov za programskog prevodioca.

Jedan mogući pristup podrazumeva da se analiza ograniči na kritične sekcije i epohe (kôd obuhvaćen uzastopnim primitivama za globalnu sinhronizaciju *barrier*) [Tranc*96]. Prvi korak je da se unutar svake sinhronizacione celine odrede dva skupa podataka: READ_SET koji obuhvata podatke koji se samo čitaju i WRITE_SET koji obuhvata podatke koji se upisuju. Sledeći korak podrazumeva analizu između sinhronizacionih celina, sa ciljem da se detektuju deljenja podataka od interesa za mehanizam injektiranja. Najpre se posmatraju sinhronizacione celine koje se uporedo izvršavaju na različitim procesorima (epohe). Proverava se da li je presek skupova READ_SET procesora Pi i Pj neprazan skup. Ukoliko je to slučaj, na početak te sinhronizacione celine treba umetnuti instrukcije za inicijalizaciju tabele injektiranja za podatke koji pripadaju tom preseku, a na kraju sinhronizacione celine instrukcije koje invaliduju odgovarajuće ulaze tabele injektiranja. Nakon toga, vrši se analiza između različitih sinhronizacionih celina koje se izvršavaju na različitim procesorima sa ciljem da se otkrije da li postoji deljenje podataka tipa Proizvođač-Potrošač. Ukoliko je to slučaj na strani procesora proizvođača umeću se `Update` instrukcije. Kada se jednom detektuju podaci koji se dele, za samo umetanje `Update` instrukcije može se koristiti algoritam koji je opisan u odeljku 2.3.6 [Skepp*95].

U daljim istraživanjima od interesa mogu biti rezultati prikazani u radovima [Jerem*95], [Tous*95]. U radu [Jerem*95] koji razmatra algoritme programskog prevodioca za redukovanje prividnog deljenja (*false sharing*) razvijen je algoritam koji vrši analizu pristupa deljenim podacima za svaku programsku nit posebno. U radu [Tous*95] predloženi su algoritmi programskog prevodioca za korišćenje adaptivnih protokola za održavanje keš koherencije.

3.6 Hardverska podrška mehanizmu injektiranja

Hardverska podrška predloženom mehanizmu injektiranja podrazumeva sledeće elemente: (a) proširenje skupa instrukcija predloženim instrukcijama, (b) implementaciju tabele injektiranja unutar keš kontrolera i (c) proširenje BCU (*Bus Control Unit*) jedinice keš kontrolera tako podrži injektiranje keš bloka sa magistrale podataka.

U odeljku 3.2.1 opisane su predložene instrukcije za podršku mehanizmu injektiranja: `OpenWindow`, `CloseWindow`, `Update` i `StoreUpdate`. Pri tom, treba napomenuti da instrukcije `Update` i `StoreUpdate` zapravo nisu nove, jer neki moderni komercijalni mikroprocesori (DEC Alpha, PowerPC, itd) podržavaju slične instrukcije koje omogućuju ažuriranje memorije modifikovanom vrednošću koja se nalazi u keš memoriji. Stoga, stvarno nove instrukcije su `OpenWindow` i `CloseWindow`. Instrukcija `OpenWindow` inicijalizuje tabelu injektiranja i pri tom definiše početnu i krajnju adresu bloka podataka koji se želi

injektirati. Kako formati instrukcija modernih RISC procesora obično ne podržavaju dva adresna polja, ova instrukcija se može implementirati kao dve instrukcije: `OWL` koja definiše donju adresu bloka i `OWH` koja definiše gornju granicu bloka. Ova implementacija je korišćena u eksperimentalnoj analizi. Instrukcija `OWL` pronalazi slobodan ulaz u tabeli injektiranja, ili ukoliko ne postoji ni jedan slobodan ulaz prepisuje najranije inicijalizovani ulaz. U oba polja se upisuje adresa keš bloka koji je specificiran adresnim poljem. Na taj način, ukoliko se injektiranje vrši na nivou jednog keš bloka, nema potrebe za instrukcijom `OWH`. Međutim, ukoliko se specificira injektiranje bloka podataka koji obuhvata više susednih keš blokova, `OWH` instrukcija definiše gornju adresu bloka i upisuje je u *Haddr* ulaz tabele injektiranja. Instrukcija `CloseWindow` definiše samo adresu *Laddr*. Ukoliko u tabeli injektiranja postoji ulaz sa adresnim poljem koje odgovara specificiranoj adresi, ulaz se invaliduje, tj. proglašava nevažećim.

Organizacija tabele injektiranja i njena veza sa sistemskom magistralom su već objašnjeni na Sl. 3-1 u odeljku 3.2.1. Važno pitanje u implementaciji tabele injektiranje je njen kapacitet. Eksperimentalna analiza je pokazala da je za većinu aplikacija tabela injektiranja sa 64 ulaza dovoljna, mada se mogu naći aplikacije koje zahtevaju veći broj ulaza. Međutim, imajući u vidu da je napredak tehnologije obezbedio projektantima površinu na čipu koja prevazilazi njihove potrebe problem implementacije tabele injektiranja nije ograničavajući faktor. Pored toga, ne postoji ni problem brzine odziva tabele tokom *snooping* ciklusa na magistrali jer se tabela injektiranja nalazi u keš kontroleru na čipu čije je brzina znatno veća od trajanja ciklusa na magistrali.

Na kraju, BCU jedinica treba da podrži samo injektiranje sa magistrale podataka prihvatanjem keš bloka koji se prenosi preko magistrale podataka i njegovim smeštanjem u odgovarajući blok keš memorije. Ovo se ostvaruje proširivanjem kontrolnog dela BCU jedinice. Takođe, treba napomenuti da injektiranje keš bloka sa magistrale može da bude razlog izbacivanja bloka iz keš memorije; ukoliko je izbačeni blok modifikovan, on se smešta u bafer za odložene upise, a odatle ga BCU jedinica iznosi na magistralu i vraća u glavnu memoriju. Što se tiče izmena na sistemskoj magistrali, jedini dodatak je postojanje posebne linije INJECT na koju procesori tokom *snooping* faze postavljaju aktivnu vrednost, ukoliko u njihovoj tabeli injektiranja postoji validan ulaz sa adresom koja odgovara adresi tekućeg bloka koji se čita iz memorije ili vraća u memoriju.

Poglavlje 4

Eksperimentalna metodologija

U ovom poglavlju objašnjena je eksperimentalna metodologija korišćena u cilju verifikacije predloženog mehanizma injektiranja u keš memoriju kod multiprocesorskih sistema baziranih na zajedničkoj magistrali. Evaluacija predložene tehnike se bazira na simulacionoj analizi zasnovanoj na realnom izvršavanju paralelnih test programa (*execution-driven simulation*). Simulacija se vrši korišćenjem programskog alata Limes koji je razvijen na Elektrotehničkom fakultetu u Beogradu [Magdic97]; detalji u vezi ovog programskog alata dati su u odeljku 4.1. Detaljan opis organizacije i funkcija memorijskog podsistema razmatranog multiprocesora sa zajedničkom magistralom i MESI *write-back* invalidacionom protokolu dati su u odeljku 4.2. Simulator idealnog memorijskog podsistema PRAM-MESI koji se koristi za preliminarnu analizu performanse i simulator realnog memorijskog podsistema MESI-SPLIT koji se koristi u egzaktnoj analizi performanse opisani su u odeljku 4.3.

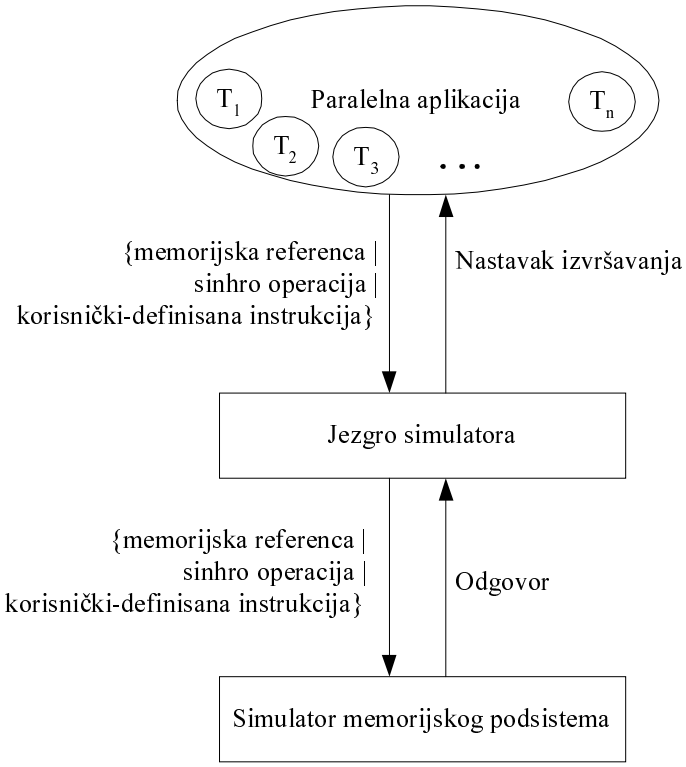
Kao radno opterećenje koriste se originalno razvijena sinhronizaciona jezgra, originalno razvijene aplikacije i aplikacije preuzete iz skupa paralelnih programa SPLASH-2 [WooO*95]. Simulaciona analiza podrazumeva poređenje relevantnih pokazatelja performanse originalnog programa i programa koji je proširen predloženim instrukcijama za podršku mehanizmu injektiranja. Umetanje predloženih instrukcija se vrši ručno, na osnovu jednostavnih heuristika koje se baziraju na analizi statičke strukture paralelnog programa. U odeljku 4.4 opisane su aplikacije i postupak umetanja instrukcija za podršku injektiranju.

4.1 Programski alat LIMES

Programski alat Limes (*Linux Memory Simulator*) je pre svega namenjen simulaciji multiprocesorskih sistema na PC računarima pod operativnim sistemom Linux, a može se koristiti i za evaluaciju paralelnih algoritama. Limes omogućuje simulaciju zasnovanu na izvršavanju paralelnih programa (*execution-driven simulation*) [Magdic97] i simulaciju zasnovanu na korišćenju adresnih tragova (*trace-driven simulation*) [Ikodi99].

Simulaciono okruženje zasnovano na izvršavanju paralelnih programa obuhvata tri celine čiji je odnos ilustrovan na Sl. 4-1. Paralelna aplikacija koja je predviđena da se izvršava na N procesora se sastoji od N programskih niti T_1, T_2, ...T_n koje se izvršavaju u paraleli. Limes podržava programerski model lakih programskih niti (*lightweight threads*), što znači da svih

N programskih niti deli isti virtuelni adresni prostor; svaka programska nit poseduje privatnu stek memoriju. Tekuća verzija programskog alata Limes ne podržava migraciju programskih niti po procesorima, tj. programska nit T_1 je pridružena procesoru P_1, a programska nit T_n procesoru P_n. Same aplikacije su regularni C ili C++ programi koji koriste ANL (*Argonne National Lab*) skup makroa za kontrolu programskih niti, alokaciju deljenih promenljivih, sinhronizaciju i podršku razvoju paralelnih programa [Magdic97a].



Sl. 4-1. Organizacija programskog alata Limes.

Drugu celinu čini simulator memorijskog podsistema (*Memory System Simulator*) koji opisuje ponašanje konkretnog memorijskog podsistema nekog multiprocesora, počev od TLB jedinice, preko jednog ili više nivoa keš memorije, magistrale, glavne memorije, pa do interkonekcionih mreža koje povezuju različite klastere.

Jezgro simulatora (*Simulator Kernel*) je smešteno između paralelne aplikacije i simulatora memorijskog podsistema. Jezgro simulatora prihvata od aplikacije događaje od interesa kao što su pristupi deljenoj i privatnoj memoriji, sinhro operacije i korisnički definisane instrukcije, zaustavlja izvršavanje programske niti koja je generisala zahtev od interesa, prosleđuje ga simulatoru memorijskog podsistema koji u skladu sa vrstom događaja simulira ponašanje memorijskog podsistema i zatim generiše odgovor na zahtev; nakon toga, jezgro simulatora prihvata odgovor i dozvoljava nastavak izvršavanja programske niti.

U odeljku 4.1.1 dat je detaljniji opis programskog okruženja Limes, uključujući proces prevođenja i opis samog toka simulacije. Funkcionisanje simulatora memorijskog podsistema je opisano u odeljku 4.1.2, a jedan primer organizacije simulatora memorijskog podsistema je dat u odeljku 4.1.3.

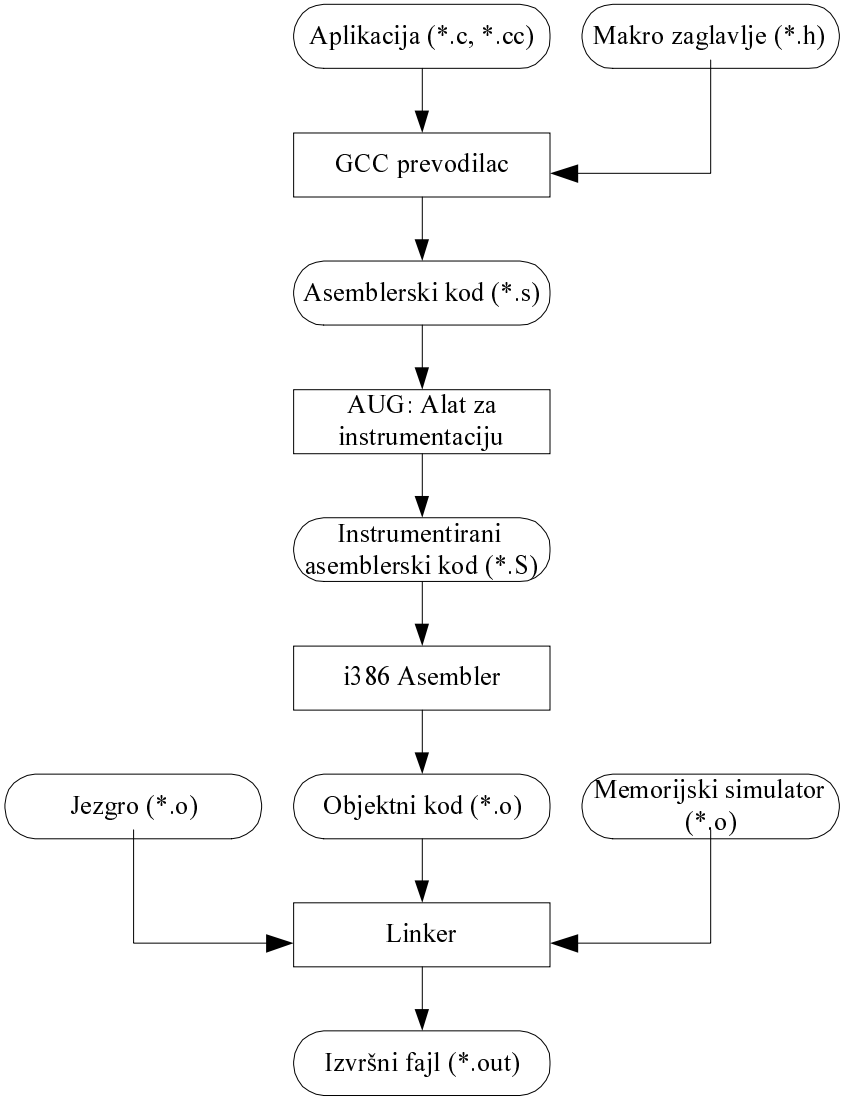
4.1.1 Formiranje simulatora i tok simulacije

Paralelna aplikacija, jezgro simulatora i simulator memorijskog podsistema se prevode i povezuju u jedan izvršni fajl. Postupak prevođenja je ilustrovan na Sl. 4-2. Paralelne aplikacije su regularni C i C++ programi prošireni skupom ANL makroa koji podržavaju kontrolu programskih niti, alokaciju deljenih promenljivih, sinhronizaciju i podršku razvoju i ispravljanju grešaka kod paralelnih programa. Prvi korak u formiranju simulatora je prevođenje paralelne aplikacije u asemblerski kôd korišćenjem prevodioca GCC. Sledeći korak podrazumeva instrumentaciju asemblerskog koda umetanjem odgovarajućih poziva jezgru simulatora. U opštem slučaju, alat za instrumentaciju koda AUG dozvoljava tri nivoa instrumentacije. Nivo 0 instrumentira samo sinhronizacione primitive i korisnički definisane asemblerske instrukcije; nivo 1 obuhvata nivo 0 i sve upise i čitanja koji se odnose na deljene podatke, a nivo 2 uključuje još i pristupe lokalnim podacima. Kako većina programa poziva standardne bibliotečne funkcije, Limes uključuje podršku instrumentaciji bibliotečnih funkcija koje kao argument imaju pokazivače na podatke. Da bi se izbegla kolizija sa bibliotečnim funkcijama koje se pozivaju iz jezgra simulatora ili simulatora memorijskog podsistema, instrumentirane funkcije imaju prefiks *sys_*, a uvodi se i niz makroa koji pozive bibliotečnih funkcija iz aplikacije preusmeravaju na pozive unapred instrumentiranih funkcija. Kako se programski alat Limes izvršava na PC računarima koji su bazirani na procesorima sa CISC skupom instrukcija (i386/387), proces instrumentacije je nešto komplikovaniji u poređenju sa sličnim sistemima koji se izvršavaju na radnim stanicama baziranim na RISC procesorima sa *Load-Store* arhitekturom [Golds93]. Detalji koji se odnose na proces instrumentacije dati su u priručniku za korisnike programskog alata Limes [Magdic97a]. Nakon instrumentacije i asembliranja, objektni kôd paralelne aplikacije se povezuje sa odgovarajućim objektnim kodom jezgra simulatora i simulatora memorijskog podsistema u jedan izvršni fajl.

Tokom procesa simulacije jezgro simulatora kontroliše izvršavanje programskih niti paralelne aplikacije i simulatora memorijskog podsistema. Izvršava se originalni kôd jedne od programskih niti paralelne aplikacije sve dok se ne dođe do događaja koji je od globalnog interesa (sinhronizacione operacije, memorijske reference ili korisnički definisane instrukcije), kada kontrolu preuzima jezgro simulatora. Jezgro simulatora pamti parametre tekućeg zahteva i vreme kada je zahtev iniciran, proverava koja od programskih niti ima najranije vreme simulacije i predaje kontrolu toj programskoj niti. Postupak se ponavlja sve dok ima aktivnih programskih niti. Nakon toga, opet u skladu sa vremenom iniciranja, simulatoru memorijskog podsistema se predaju generisani zahtevi i ažurira vreme izvršavanja odgovarajuće programske niti, a proces se nastavlja sve do završetka svih programskih niti paralelnog programa.

Opisani postupak je ilustrovan na konkretnom primeru koji sledi. Posmatraju se dve programske niti T0 i T1 koje se izvršavaju na procesorima P0 i P1, redom. Pretpostavimo, dalje, da programska nit T0 izvršava instrukcije koje nemaju globalni značaj sve do trenutka t0=50 kada inicira čitanje deljenog podatka; programska nit T1 u trenutku t1=20 izvršava upis deljenog podatka. Jezgro simulatora na početku vidi dve spremne programske niti sa tekućim vremenima koja odgovaraju početnom trenutku t0=0 i t1=0. Pretpostavimo da jezgro simulatora kontrolu predaje najpre programskoj niti T0 koja se izvršava sve do trenutka t0=50, kada generiše čitanje deljenog podatka. Kontrolu preuzima jezgro simulatora koje vidi da postoji zahtev simulatoru memorijskog podsistema u trenutku t0=50 i aktivna programska nit T1 sa tekućim vremenom t1=0; kontrola se predaje programskoj niti T1 koja izvršava instrukcije sve do trenutka t1=20, kada generiše deljeni upis. Jezgro ponovo preuzima kontrolu i vidi dva zahteva inicirana u trenucima t0=50 i t1=20; kako su obe programske niti

blokirane, jezgro simulatora poziva simulator memorijskog podsistema i prosleđuje mu najranije inicirani zahtev, u ovom slučaju zahtev od procesora P1. Pretpostavimo da je za kompletiranje ovog zahteva potrebno 10 vremenskih jedinica. Nakon toga, tekuće vreme programske niti T1 je t1=30, pa jezgro simulatora ponovo predaje kontrolu programskoj niti T1. Pretpostavimo da će ova programska nit generisati deljeni upis u trenutku t1=100. Jezgro simulatora sada vidi da postoje dva zahteva u trenucima t0=50 i t1=100; kako zahtev od programske niti T0 prethodi zahtevu od programske niti T1, ovaj zahtev se predaje simulatoru memorijskog podsistema. Nakon pristizanja odgovora simulacija je nastavlja na opisani način.



Sl. 4-2. Formiranje simulatora u programskom paketu Limes: proces prevođenja i povezivanja.

U prethodnom primeru je pomenut problem merenja vremena tokom simulacije. Jezgro simulatora čuva tekuće vreme za svaku programsku nit posebno. U programskom okruženju Limes usvojena je pretpostavka da svaka instrukcija koja nema globalni značaj traje jedan ciklus takta, što znači da se simulira idealna protočna obrada. Međutim, alat za instrumentaciju AUG omogućuje da se svakoj instrukciji koja nije globalna pridruži odgovarajući broj procesorskih ciklusa, što omogućava realističniju simulaciju. Takođe, ovaj broj se može i smanjivati da bi se eventualno simulirali procesori sa mogućnošću paralelnog izvršavanja više instrukcija. Sa druge strane, svi globalni događaji se simuliraju, pa vreme trajanja instrukcija globalnog značaja zavisi od konkretnog simulatora memorijskog

podsistema. Relacije između jezgra simulatora i simulatora memorijskog podsistema su bliže objašnjene u narednom odeljku.

4.1.2 Komunikacija simulatora memorijskog podsistema sa jezgrom

Simulator memorijskog podsistema opisuje ponašanje svih relevantnih delova multiprocesora koji logički počinju od tačke gde procesori generišu svoje zahteve memoriji i u opštem slučaju obuhvata TLB jedinicu, keš memoriju, magistralu, glavnu memoriju, interkonekcionu mrežu, itd. Simulator memorijskog podsistema je u potpunosti nezavisan od paralelnih aplikacija i jezgra simulatora, a za komunikaciju sa jezgrom simulatora koristi unapred definisan interfejs koji, opet, ne zavisi od konkretnog opisa samog memorijskog podsistema.

Jezgro simulatora je odgovorno da simulatoru memorijskog podsistema u odgovarajućem trenutku prosledi zahtev, tako da redosled događaja odgovara redosledu kod realnog multiprocesora. Sa svoje strane, simulator memorijskog podsistema prihvata zahtev i započinje simulaciju. Nakon svakog simuliranog ciklusa takta, simulator memorijskog podsistema vraća kontrolu jezgru simulatora koje na osnovu dobijenog odgovora odlučuje o sledećoj akciji. Postoje dva osnovna moguća odgovora na tekući zahtev: (a) zahtev je opslužen (*satisfied*) ili (b) zahtev još nije opslužen (*stalled*). Ako je zahtev opslužen, programska nit može nastaviti sa izvršavanjem instrukcija do sledećeg globalnog događaja; ukoliko je procesor blokiran, jezgro simulatora će pozvati simulator memorijskog podsistema u sledećem ciklusu takta, i tako redom sve dok zahtev ne bude opslužen. Dakle, simulator memorijskog podsistema zapravo predstavlja jedan konačni automat (*finite-state-machine*).

U cilju boljeg razumevanja simulatora memorijskog podsistema posmatra se jednostavan primer simulacije keš podsistema multiprocesora sa zajedničkom magistralom. Posmatraju se karakteristični slučajevi *read-hit, read-miss,* i *write-miss*. Pretpostavimo da se instrukcija koja inicira odgovarajući događaj izvršava u trenutku t=1000. Dalje, pretpostavimo da pristup keš memoriji u slučaju pogotka traje jedan ciklus takta.

Read-hit. Jezgro prosleđuje zahtev simulatoru memorijskog podsistema. Simulator proverava keš memoriju i vidi da se traženi podatak nalazi u keš memoriji. Ažuriraju se informacije o poslednjem pristupu keš bloku, postavlja indikacija da je zahtev opslužen i vraća kontrola jezgru simulatora. Na osnovu usvojene pretpostavke simulator u istom ciklusu takta vraća signal *satisfied,* tako da jezgro simulatora dozvoljava nastavak izvršavanja programske niti sa tekućim vremenom t=1001.

Read-miss. Najpre se proveravaju tagovi (*tags*) keš memorije i kako se podatak ne nalazi u keš memoriji, postavlja se zahtev arbitratoru magistrale za izlazak na magistralu. Kako zahtev nije opslužen u tekućem ciklusu takta, kontrola se vraća jezgru simulatora i postavlja signal *stalled*. Jezgro simulatora će ponovo pozvati simulator memorijskog podsistema sa tekućim vremenom t=1001. U zavisnosti da li je dobijena dozvola za izlazak na magistralu ili ne, simulator započinje ciklus na magistrali ili ostaje u stanju čekanja na dozvolu od arbitratora magistrale. Nakon simulacije drugog ciklusa takta kontrola se ponovo vraća jezgru simulatora. Na taj način, jezgro simulatora će pozivati simulator memorijskog podsistema sve dok se ne završi operacija čitanja. Pretpostavimo da će se kroz 15 vremenskih jedinica traženi podatak naći u keš memoriji. U trenutku t=1015 simulator ažurira odgovarajuće informacije, postavlja odgovor *satisfied* i ponovo vraća kontrolu jezgru. Jezgro simulatora vidi da je tekući zahtev opslužen, pa odgovarajuću programsku nit smatra spremnom za nastavak izvršavanja sa tekućim vremenom t=1016.

Write-miss. Proveravaju se tagovi keš memorije i kako traženi podatak nije u keš memoriji generiše se zahtev arbitratoru magistrale. Kako upis podatka ne zahteva nikakvu povratnu informaciju od simulatora memorijskog podsistema, simulator odmah po prijemu zahteva postavlja odgovor *satisfied*. Po prijemu signala *satisfied* jezgro simulatora dozvoljava nastavak izvršavanja odgovarajuće programske niti sa tekućim vremenom t=1001. Pretpostavimo da ova programska nit generiše sledeći zahtev od interesa u trenutku t=1050. Jezgro simulatora ponovo dobija kontrolu i poziva memorijski simulator; kao parametri prosleđuju se tekuće vreme t=1050 i vreme poslednjeg poziva t=1001. Kako simulacija prethodnog zahteva nije završena, simulator memorijskog podsistema izvršava ciklus po ciklus akcije predviđene za slučaj promašaja u kešu prilikom upisa. Pri tom, jezgro simulatora poziva simulator memorijskog podsistema repetitivno sve dok se ne završi simulacija. Pretpostavimo da će zahtev biti u potpunosti opslužen u trenutku t=1020. Kako sledeći zahtev dolazi u trenutku t=1050, to simulator "preskače" cikluse takta u intervalu od t=1021 do t=1049. Simulacija se nastavlja na uobičajeni način od trenutka t=1051.

Opisani pristup pokazuje dva važna elementa koji čine Limes veoma efikasnim alatom za simulaciju multiprocesorskih sistema. Prvo, simulator memorijskog podsistema se poziva ako i samo ako postoji barem jedan procesor koji je blokiran ili ukoliko postoji novi zahtev. Drugo, jezgro simulatora u odgovarajućim slučajevima dozvoljava da se jednim pozivom simulatora memorijskog simulatora obavi simulacija više ciklusa takta bez vraćanja kontrole jezgru simulatora posle svakog ciklusa takta; na taj način, smanjuje se broj promena konteksta između procesora i memorijskog simulatora.

4.1.3 Jedan primer organizacije simulatora memorijskog podsistema

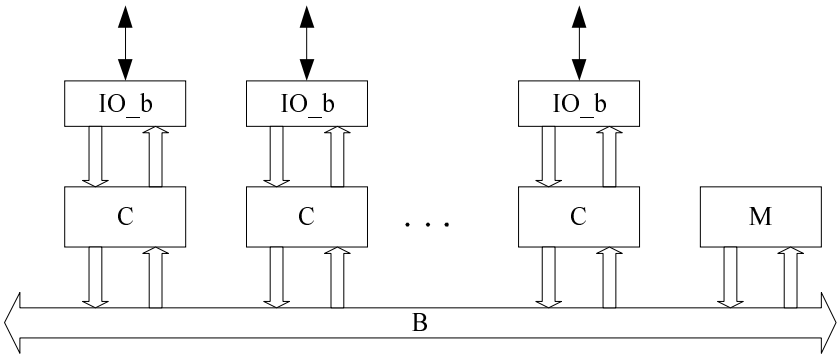
U opštem slučaju arhitekta koji modelira konkretni memorijski podsistem ima potpunu slobodu u pogledu organizacije simulatora memorijskog podsistema. Jedino što se pri tom mora poštovati jeste unapred definisan interfejs prema jezgru simulatora [Magdic97a]. Međutim, od interesa je pre svega modularni pristup koji garantuje proširivost, lako održavanje, ispravljanje grešaka i čitljivost. U tekstu koji sledi ukratko su objašnjeni osnovni elementi ovakvog pristupa.

Simulator memorijskog podsistema se projektuje tako da sadrži više modula, pri čemu svaki modul predstavlja automat sa konačnim brojem stanja. Pri tom, tok podataka između različitih modula se ostvaruje preko portova. Modul čita potrebne informacije sa svojih ulaznih portova, izvršava odgovarajuće operacije na osnovu trenutnog stanja i ulaza i u skladu sa rezultatima prelazi u novo stanje i postavlja informacije na izlazne portove. Na ovaj način ni jedan modul ne mora biti svestan postojanja drugih modula u sistemu. U sistemu postoji samo jedan modul koji poznaje topologiju celog sistema i brine o razmeni podataka između modula koji komuniciraju tako što modul koji poznaje topologiju sistema čita podatke sa izlaznih portova jednog modula i upisuje ih u ulazne portove drugog modula.

U daljem tekstu razmatra se multiprocesorski sistem sa *N* procesora sa prvim nivoom keš memorije koja se nalazi na čipu, keš kontrolerom koji je baziran na *snooping* protokolu za održavanje koherencije keš memorije, magistralom i memorijom. Relevantni moduli posmatranog memorijskog podsistema multiprocesora sa zajedničkom magistralom su prikazani na Sl. 4-3. Sistem čine *N IO_Buf* modula, *N Cache* modula, *Bus* i eventualno *Memory* modul.

IO_Buf modul odvaja sve ostale module memorijskog podsistema od interfejsa prema jezgru simulatora. Ovaj modul prihvata zahtev i transformiše ga u oblik koji je prihvatljiv *Cache*

modulu. Pored toga, ovaj modul prihvata odgovor od *Cache* modula i prosleđuje ga jezgru simulatora. *Cache* modul sadrži opis keš memorije i odgovarajuće kontrolne jedinice keš kontrolera. Ovaj modul preko svojih ulaznih portova prihvata zahteve od procesora koji dolaze preko *IO_Buf* modula i signale sa zajedničke magistrale. Na bazi ulaznih vrednosti i trenutnog stanja postavljaju se odgovarajuće vrednosti na izlazne portove prema *IO_Buf* modulu i zajedničkoj magistrali. *Bus* modul sadrži kontrolne linije, adresnu i magistralu podataka. Pored toga, ovaj modul je odgovoran i za arbitraciju zahteva za magistralu koji pristižu od procesora. U opštem slučaju, *Memory* modul komunicira sa zajedničkom magistralom preko odgovarajućih portova. Ovaj modul može da modelira specifičnu organizaciju memorijskih modula, međutim u većini slučajeva se smatra apstraktnim modulom, jer se relevantni parametri kao što je vreme pristupa memoriji mogu simulirati u okviru *Cache* modula.



Sl. 4-3. Moduli simulatora memorijskog podsistema multiprocesora sa zajedničkom magistralom.
Opis: C (*Cache*) – keš memorija, IO_b (*IO buffer*) – ulazno/izlazni bafer, M (*Memory*) – memorija, B (*Bus*) – magistrala.

Simulator memorijskog podsistema u svakom ciklusu takta poziva module odgovarajućim redosledom. Na početku poziva se *N IO_Buf* modula koji prihvataju zahteve od jezgra simulatora. Nakon toga, poziva se *N Cache* modula koji odrađuju simulaciju u tom ciklusu takta, pa *Bus* modul koji prihvata signale koje su postavili *Cache* moduli na svoje portove prema magistrali. Na kraju ponovo se poziva *N IO_Buf* modula koji prihvataju odgovore od *Cache* modula i prosleđuju ih jezgru simulatora. Ovakav redosled omogućava da se, na primer, pogodak u kešu završi u jednom ciklusu takta.

4.2 Organizacija memorijskog podsistema

Simulaciona analiza se bazira na multiprocesorskom sistemu sa zajedničkom magistralom koji koristi MESI *write-back* invalidacioni protokol za održavanje koherencije keš memorije. MESI protokol je opisan u odeljku 4.2.1. Magistrala multiprocesorskog sistema podržava razdvojene transakcije (*split-transactions bus*) kod ciklusa čitanja iz memorije. U odeljku 4.2.2 objašnjena je organizacija magistrale koja podržava razdvojene transakcije. U odeljku 4.2.3 opisana je organizacija i funkcionisanje keš kontrolera.

4.2.1 MESI protokol

Pretpostavimo da dva procesora Pi i Pj poseduju kopiju istog keš bloka u svojim keš memorijama. Problem koherencije keš memorije nastaje kada npr. procesor Pi izvrši upis u posmatrani keš blok. Nakon upisa podatka u keš memoriju procesora Pi u sistemu će postojati

različite kopije istog keš bloka. Da bi se to sprečilo, procesor Pi treba da: (a) invaliduje kopiju keš bloka u keš memoriji procesora Pj i tako postane ekskluzivni vlasnik ili (b) ažurira kopiju u keš memoriji procesora Pj i glavnu memoriju. U zavisnosti od preduzete akcije protokoli se svrstavaju u dve grupe (a) invalidacione (*invalidate*) i (b) bazirane na ažuriranju (*update*). Kod protokola koji su bazirani na ažuriranju svaki upis u keš blok inicira transakciju kojom se ažuriraju sve ostale kopije tog keš bloka, što rezultuje velikim saobraćajem na interkonekcionoj mreži. U slučaju višestrukih uzastopnih upisa u istu reč ili keš blok, ovaj saobraćaj nema nikakav korisni efekat. Uticaj kontencije na interkonekcionoj mreži i potreba za povezivanjem većeg broja procesora učinili su invalidacione protokole dominantnim [Patte*96].

U pogledu politike ažuriranja glavne memorije kod multiprocesorskih sistema sa keš memorijom moguća su dva osnovna pristupa: (a) *write-through* koji podrazumeva ažuriranje glavne memorije posle svakog upisa i (b) *write-back* koji podrazumeva odloženo ažuriranje na nivou keš blokova. Potreba da se redukuje saobraćaj na magistrali da bi se povećala skalabilnost i performanse nameću korišćenje keš memorija sa odloženim ažuriranjem (*write-back*).

Kod multiprocesorskih sistema sa zajedničkom magistralom održavanje koherencije keš memorije je olakšano zahvaljujući osobinama zajedničke magistrale. Kada jedan procesor inicira neku operaciju na magistrali, svi drugi procesori nadgledaju magistralu (*snooping*) i ako je potrebno preduzimaju akcije kojim održavaju koherenciju podataka u keš memoriji. MESI *write-back* invalidacioni protokol baziran na monitorisanju zahteva na magistrali je najraširenij protokol za održavanje koherencije keš memorije kod multiprocesora sa zajedničkom magistralom. Ovaj protokol je implementiran u nekoliko različitih varijanti kod gotovo svih modernih mikroprocesora kao što su PentiumII, PentiumPro, PowerPC 601, MIPS 10000, itd. U literaturi je poznat i pod imenom Illinois protokol, prema imenu Univerziteta na kome je prvi put predložen (Illinois at Urbana-Champaign) [Papa*84].

Kod multiprocesora koji podržava MESI *write-back* invalidacioni protokol keš blok se u svakom trenutku može naći u jednom od četiri stanja: *Modified* (M) – blok je modifikovan i posmatrani procesor jedini ima validnu kopiju tog keš bloka u celom sistemu; kopije u keš memorijama ostalih procesora i glavnoj memoriji su invalidovane; *Exclusive* (E) – blok je nemodifikovan, tj. kopija ovog bloka je ista kao i kopija u glavnoj memoriji, ali posmatrani procesor je jedini koji ima kopiju ovog bloka u svojoj keš memoriji; *Shared* (S) – više procesora u svojim privatnim keš memorijama poseduje nemodifikovanu kopiju keš bloka; *Invalid* (I) – keš blok je invalidovan.

Pre detaljnog razmatranja MESI protokola definisano je okruženje i uvedene su neophodne pretpostavke. Procesor inicira dva tipa zahteva: čitanje PrRd (*ProcessorRead*) i upis PrWr (*ProcessorWrite*). Na magistrali su mogući sledeći ciklusi: dohvatanje keš bloka radi čitanja RdC (*ReadBusCycle*), dohvatanje keš bloka radi upisa RdXC (*ReadExclusiveBusCycle*), invalidaciju ostalih kopija keš bloka InvC (*InvalidateBusCycle*) i vraćanje keš bloka u memoriju WbC (*WriteBackBusCycle*). Ciklusi na magistrali se sastoje iz tri dela: (1) arbitracije u kojoj procesor postavlja zahtev i dobija dozvolu za magistralu, (2) iznošenja adrese i komande na adresnu magistralu i upravljačke linije, redom, i (3) prenosa podataka. Tokom relevantnih ciklusa na magistrali (RdC, RdXC i InvC) svi procesori nadgledaju magistralu, proveravaju da li se traženi keš blok nalazi u njihovim keš memorijama, i ako je to slučaj preduzimaju odgovarajuće akcije.

Na Sl. 4-4 prikazani su dijagrami prelaza za MESI *write-back* invalidacioni protokol. Zbog jednostavnosti se ne razmatra izbacivanje iz keš memorije modifikovanih keš blokova usled

politike zamene. Na levoj strani su prikazani prelazi usled procesorskih zahteva PrRd i PrWr, a na desnoj strani prelazi usled relevantnih operacija na magistrali (RdC, RdXC, InvC). U tekstu koji sledi analizira se dijagram prelaza MESI protokola prilikom čitanja PrRd i upisa PrWr, redom.

Čitanje

Procesor Pi inicira čitanje keš bloka koji se ne nalazi u keš memoriji; može se smatrati da se takav blok nalazi u stanju I (*Invalid*), tj. stanje I uključuje stvarno invalidovane keš blokove i one koji nisu prisutni u keš memoriji. Keš kontroler inicira ciklus dohvatanja keš bloka radi čitanja RdC. Svi ostali keš kontroleri nadgledaju magistralu i kada vide da je iniciran ciklus čitanja izvršavaju *snooping* svojih keš memorija. Tokom *snooping* ciklusa mogući su sledeći slučajevi:

(a) procesor Pj ne poseduje traženi keš blok u svojoj keš memoriji, tj. vidi *snoop miss*. Keš kontroler procesora Pj ne preduzima nikakvu akciju;
(b) procesor Pj poseduje traženi keš blok u svojoj keš memoriji u stanju S. Keš blok ostaje u stanju S (SC:S→S), a keš kontroler procesora obično ne preduzima nikakvu akciju dozvoljavajući glavnoj memoriji da odgovori iznošenjem traženog keš bloka na magistralu;
(c) procesor Pj poseduje traženi keš blok u svojoj keš memoriji u stanju E. Keš kontroler menja stanje keš bloka u S (SC:E→S). U zavisnosti od implementacije keš kontroler procesora Pj može da iznese traženi keš blok na magistralu ili da prepusti glavnoj memoriji da odgovori na ovaj zahtev;
(d) procesor Pj poseduje traženi keš blok u svojoj keš memoriji u stanju M. Keš kontroler menja stanje u S (SC:M→S) i iznosi keš blok na magistralu (*flush*) umesto memorije, jer on jedini poseduje validnu kopiju traženog keš bloka. Glavna memorija i keš kontroler procesora Pi prihvataju keš blok. Ova implementacija podrazumeva podršku direktnom transferu između keš kontrolera; ukoliko ne postoji podrška za takav transfer, tekući RdC ciklus procesora Pi se poništava, a keš kontroler procesora Pj inicira ciklus WbC kojim ažurira glavnu memoriju. Kada procesor Pi ponovo inicira poništeni ciklus dohvatanja RdC glavna memorija iznosi keš blok na magistralu.

Novo stanje keš bloka u keš memoriji procesora Pi je E (PC:I→E) ukoliko je on jedini procesor koji u svom kešu poseduje kopiju tog keš bloka (tj., svi procesori su imali *snoop miss*), odnosno S ukoliko još neki procesor poseduje kopiju keš bloka u svojoj keš memoriji (PC:I→S). Čitanje keš bloka koje rezultuje pogotkom u keš memoriji ne menja stanje keš bloka niti inicira neki ciklus na magistrali, bez obzira da li se keš blok nalazi u stanju S, E ili M (PC:S→S, PC:E→E, PC:M→M).

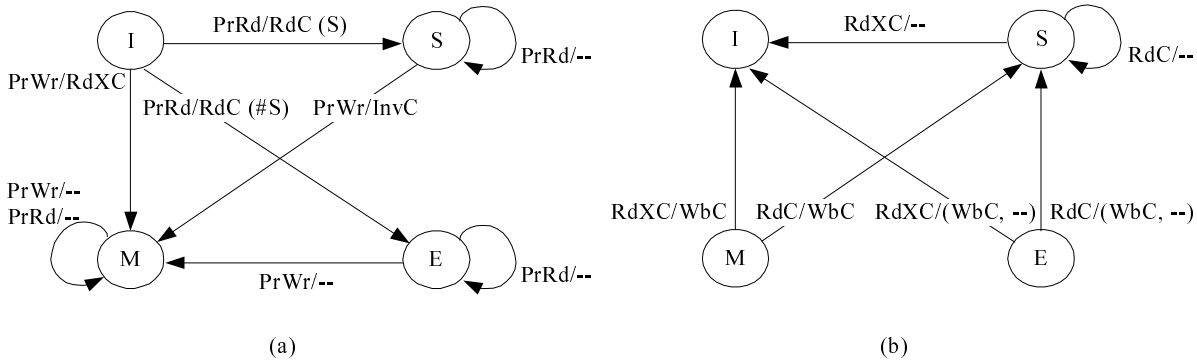
Upis

Procesor Pi inicira upis u keš blok koji se ne nalazi u keš memoriji. Keš kontroler inicira ciklus dohvatanja keš bloka radi upisa RdXC. Svi ostali keš kontroleri nadgledaju magistralu i kada vide da je iniciran ciklus ekskluzivnog čitanja izvršavaju *snooping* svojih keš memorija. Tokom *snooping* ciklusa mogući su sledeći slučajevi:

(a) procesor Pj ne poseduje traženi keš blok u svojoj keš memoriji, tj. vidi *snoop miss*. Keš kontroler procesora Pj ne preduzima nikakvu akciju;
(b) procesor Pj poseduje traženi keš blok u svojoj keš memoriji u stanju S. Keš blok se invaliduje (SC:S→I), a keš kontroler procesora obično ne preduzima nikakvu drugu akciju, dozvoljavajući glavnoj memoriji da odgovori iznošenjem traženog keš bloka na magistralu;

(c) procesor Pj poseduje traženi keš blok u svojoj keš memoriji u stanju E; keš kontroler menja stanje keš bloka u I (SC:E→I). U zavisnosti od implementacije keš kontroler procesora Pj može da iznese traženi keš blok na magistralu ili da prepusti glavnoj memoriji da odgovori na ovaj zahtev;
(d) procesor Pj poseduje traženi keš blok u svojoj keš memoriji u stanju M. Keš kontroler menja stanje u I (SC:M→I) i iznosi keš blok na magistralu (*flush*) umesto memorije, jer on jedini poseduje validnu kopiju traženog keš bloka. Keš kontroler procesora Pi prihvata keš blok. Ova implementacija podrazumeva podršku direktnom transferu između keš kontrolera; ukoliko ne postoji podrška za takav transfer tekući RdXC ciklus procesora Pi se poništava, a keš kontroler procesora Pj inicira ciklus WbC kojim ažurira glavnu memoriju. Kada procesor Pi ponovo inicira poništeni ciklus ekskluzivnog dohvatanja RdXC, glavna memorija iznosi keš blok na magistralu.

Novo stanje keš bloka u keš memoriji procesora Pi je M (PC:I→M). Ukoliko se keš blok u koji se upisuje već nalazi u keš memoriji, moguća su sledeća tri slučaja: (a) keš blok se nalazi u stanju E, (b) keš blok se nalazi u stanju S i (c) keš blok se nalazi u stanju M. Ukoliko se blok nalazi u stanju E, keš kontroler menja stanje u M (PC:E→M) i pri tom nije potreban nikakav ciklus na magistrali jer je procesor Pi već ekskluzivni vlasnik keš bloka. Ako se keš blok nalazi u stanju S, to znači da je moguće da se nalazi u keš memorijama drugih procesora, pa keš kontroler procesora Pi inicira ciklus invalidacije na magistrali InvC. Novo stanje keš bloka je M (PC:S→M). Tokom ciklusa invalidacije keš kontroleri procesora koji poseduju keš blok u stanju S invaliduju kopije (SC:S→I). Ako se keš blok nalazi u stanju M, onda keš kontroler ne inicira nikakvu transakciju, a stanje ostaje nepromenjeno (PC:M→M).



Sl. 4-4. Dijagrami stanja/prelaza za MESI protokol.
Opis: (a) PC (*Processor Controller*): procesorski deo kontrolera; (b) SC (*Snoop Controller*): *snoop* deo kontrolera.

4.2.2 Magistrala sa podrškom razdvojenim transakcijama

U praksi se sreću dve organizacije magistrale kod multiprocesora sa zajedničkom magistralom: magistrale sa nedeljivim transakcijama (*atomic bus*) i magistrale koje podržavaju razdvojene transakcije (*split-transactions bus*). Kod multiprocesora sa atomskom magistralom jedan procesor drži magistralu sve dok se ne završi kompletno dohvatanje traženog keš bloka u keš memoriju, tokom ciklusa čitanja na magistrali RdC i RdXC. To vreme uključuje vreme koje je potrebno za iniciranje zahteva, arbitraciju za dobijanje magistrale, izvršavanje *snooping* ciklusa, vreme pristupa memoriji i vreme koje je potrebno za transfer keš bloka po magistrali podataka. Na ovaj način svi drugi procesori koji trenutno čekaju na magistralu ostaju blokirani sve dok se ne završi tekući ciklus. Kako preopterećenje

zajedničke magistrale može u velikoj meri da degradira ukupne performanse multiprocesora sa zajedničkom magistralom, to je bio razlog da se potraže modernije organizacije zajedničke magistrale koje po cenu veće kompleksnosti eliminišu problem preopterećenosti magistrale.

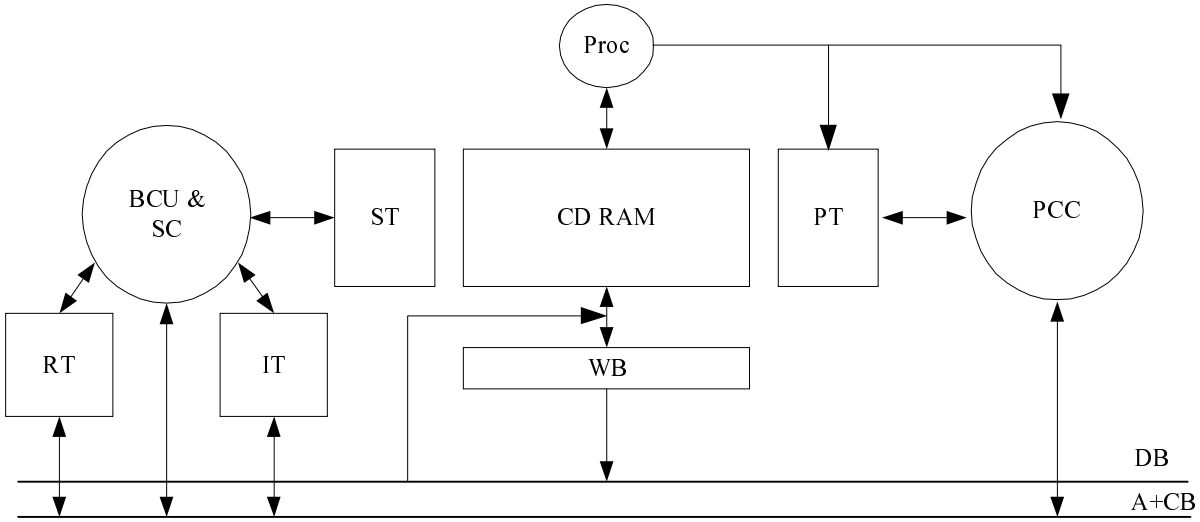
Kod multiprocesora sa zajedničkom magistralom koja podržava razdvojene transakcije ciklusi koji zahtevaju odgovor od udaljenog uređaja, keš memorije drugog procesora ili glavne memorije, podeljeni su u dve faze: *request* i *response* fazu. U prvoj fazi keš kontroler nakon dobijanja magistrale inicira ciklus dohvatanja keš bloka na magistrali, postavljanjem odgovarajuće adrese na adresnu magistralu i komande na upravljačke linije magistrale. Svi drugi procesori izvršavaju *snooping* ciklus i postavljaju relevantne kontrolne linije magistrale. Nakon toga keš kontroler oslobađa magistralu dozvoljavajući na taj način da i drugi procesori iniciraju svoje zahteve. Memorija ili keš kontroler drugog procesora iniciraju ciklus vraćanja nakon što je blok spreman. Na ovaj način je omogućeno da više procesora uporedo ima aktivne zahteve. Preklapanje vremena pristupa memoriji različitih procesora dovodi do efikasnijeg korišćenja magistrale, a time i do veće performanse. Kada glavna memorija ili keš memorija drugog procesora ima spreman traženi keš blok, ona zahteva magistralu podataka i traženi keš blok se preko magistrale podataka prosleđuje procesoru koji je inicirao zahtev (*response* faza). Ciklusi čitanja RdC i ekskluzivnog čitanja RdXC se realizuju kao razdvojene transakcije, dok transakcije InvC, WbRC i WbSC poseduju samo prvu *request* fazu, jer ne zahtevaju nikakve podatke ili potvrdu memorije ili keš kontrolera drugih procesora.

Međutim, razdvajanje pojedinih transakcija na magistrali unosi dodatnu kompleksnost u dizajn memorijskog podsistema multiprocesora. Pored toga, razdvajanje transakcija može dovesti do konflikta u zahtevima, pa je potrebna podrška njihovom razrešavanju. Naime, do konflikta dolazi kada jedan procesor inicira zahtev za ekskluzivnim čitanjem, a potom, pre završetka dohvatanja podataka (pre *response* faze) jedan ili više drugih procesora inicira zahtev za čitanjem i/ili ekskluzivnim čitanjem. Nakon izvršavanja ovih operacija imali bi situaciju da više procesora istovremeno poseduje ekskluzivne kopije keš bloka, što nije dozvoljeno protokolom za održavanje koherencije keš memorije. Da bi se izbegla ovakva situacija postoji poseban resurs koji se zove tabela aktivnih zahteva (*Request Table*). Pre započinjanja nekog ciklusa na magistrali, svaki procesor proverava da li je tekući zahtev u koliziji sa nekim trenutno aktivnim zahtevom. Ukoliko je to slučaj, keš kontroler ulazi u stanje čekanja dok se ne završi aktivni zahtev.

4.2.3 Organizacija keš kontrolera

Ključni element simuliranog memorijskog podsistema multiprocesora sa zajedničkom magistralom je kontroler keš memorije jednog procesora (čvora). Struktura razmatranog keš kontrolera je prikazana na Sl. 4-5. Kontroler keš memorije se sastoji od dve kontrolne jedinice: PCC (*Processor Cache Controller*) i BCU&SCC (*Bus Controller Unit & Snoop Cache Controller*). Kontrolne jedinice su organizovane kao nezavisni automati sa konačnim brojem stanja koji rade uporedo u svakom ciklusu takta. PCC jedinica je odgovorna za prihvatanje zahteva od procesora *Proc* i njihovu obradu. Jedinica BCU&SCC je odgovorna za iniciranje ciklusa na magistrali i monitorisanje relevantnih ciklusa iniciranih od strane drugih procesora. Da bi se obezbedio istovremeni pristup kontrolnih jedinica tagovima keš memorije, svaka jedinica ima svoju kopiju tagova (PT i ST). Keš kontroler sadrži bafer za odložene upise (WB) kapaciteta jednog keš bloka. Za podršku razdvojenim transakcijama na magistrali koristi se tabela aktivnih zahteva (RT). Mehanizam injektiranja je podržan kroz poseban resurs, tabelu injektiranja (IT). U posmatranoj implementaciji se pretpostavlja da svaki procesor može da ima samo jednu aktivnu operaciju na magistrali, pa je broj ulaza tabele

zahteva jednak broju procesora, tj. svakom procesoru je pridružen po jedan ulaz u tabeli aktivnih zahteva. Svaki keš kontroler poseduje po jednu kopiju tabele aktivnih zahteva, tako da se tabeli pristupa lokalno.

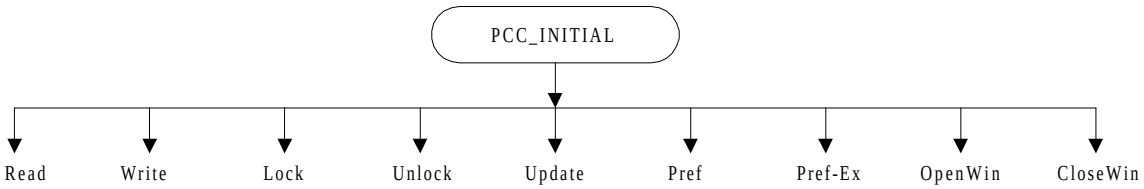


Sl. 4-5. Struktura keš kontrolera.
Opis: Proc (*Processor*) – procesor, BCU&SC (*Bus Control Unit & Snoop Controller*), PCC (*Processor Cache Controller*) – procesorski deo kontrolera, ST (*Snoop Tags*) – tagovi *snoop* dela kontrolera, PT (*Processor Tags*) – tagovi procesorskog dela kontrolera, CD (*Cache Data RAM*) – keš memorija, IT (*Injection Table*) – tabela injektiranja, RT (*Request Table*) – tabela zahteva, WB (*WriteBack Buffer*) – bafer za odložene upise, DB (*Data Bus*) – magistrala podataka, A+CB (*Address + Control Bus*).

Funkcionisanje keš kontrolera je opisano kroz detaljan opis odgovarajućih kontrolnih jedinica PCC i BCU&SCC koje su opisane u odeljcima 4.2.3.1 i 4.2.3.2, redom.

4.2.3.1 Jedinica PCC

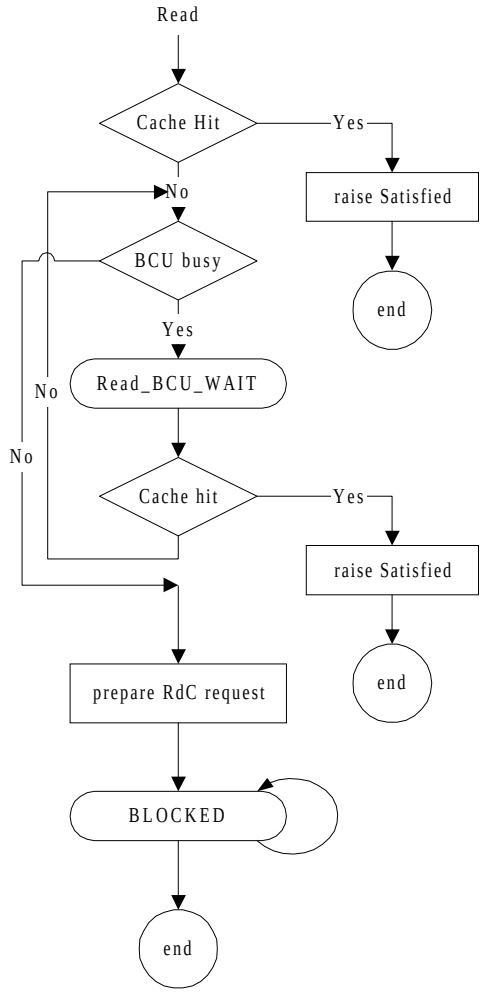
Jedinica PCC inicijalno se nalazi u stanju PCC_INITIAL, spremna da prihvati sledeći zahtev od procesora (Sl. 4-6). Nakon prihvatanja zahteva od procesora, jedinica PCC započinje obradu u zavisnosti od tipa zahteva. Pored osnovnih zahteva kao što su Read, Write, Lock, i Unlock, jedinica PCC prihvata korisnički definisane operacije koje podržavaju mehanizam injektiranja OpenWin, CloseWin i Wrback (Update), kao i korisnički definisane operacije za podršku drugim tehnikama za prikrivanje kašnjenja u pristupu memoriji kao što su Pref i Pref-Ex. U daljem tekstu je opisano ponašanje kontrolne jedinice PCC tokom obrade svakog od pomenutih zahteva.



Sl. 4-6. Inicijalno stanje PCC kontrolne jedinice i zahtevi koji dolaze od procesora.

Read. Dijagram toka jedinice PCC tokom obrade Read zahteva prikazan je na Sl. 4-7. Keš kontroler najpre proverava da li se specificirana reč nalazi u keš memoriji procesora. Ukoliko je reč u keš memoriji, keš kontroler u istom simuliranom ciklusu generiše signal završetka

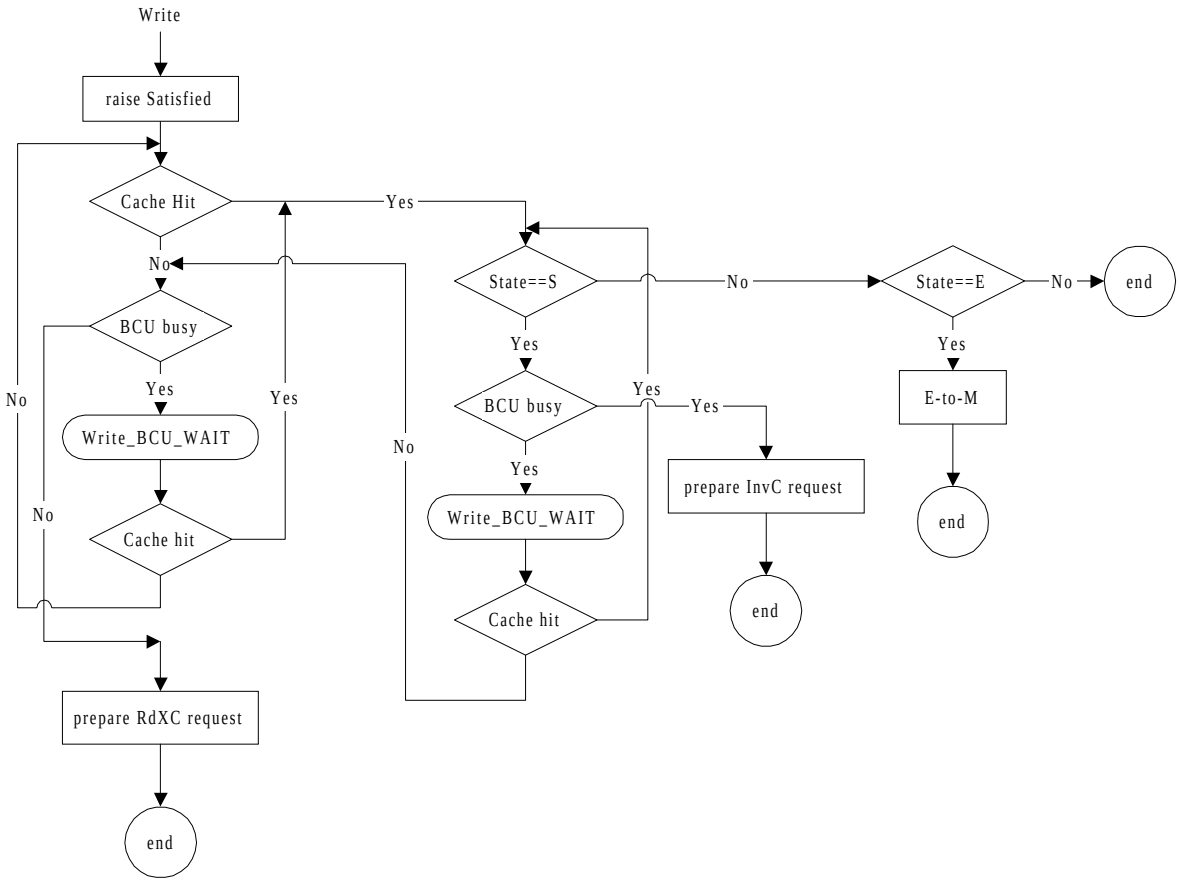
obrade postavljajući signal *Satisfied*. Ukoliko se podatak ne nalazi u keš memoriji, keš kontroler inicira dohvatanje keš bloka. Kako je BCU jedinica odgovorna za iniciranje i monitorisanje transakcija na magistrali, najpre se proverava da li je BCU jedinica spremna da prihvati zahtev za čitanje na magistrali RdC. Jedinica BCU je spremna da prihvati zahtev ukoliko je prethodno iniciran ciklus završen, a jedinica se nalazi u početnom stanju BCU_INITIAL. Ukoliko BCU jedinica nije spremna da prihvati novi zahtev, PCC jedinica prelazi u stanje čekanja na spremnost BCU jedinice Read_BCU_WAIT. PCC jedinica ostaje u ovom stanju sve dok BCU jedinica ne postane spremna, tj. dok ne završi obradu prethodno iniciranog ciklusa na magistrali. U stanju READ_BCU_WAIT jedinica PCC proverava da li se traženi keš blok nalazi u keš memoriji, u svakom ciklusu takta. Naime, zahvaljujući mehanizmu injektiranja keš blok se može naći u keš memoriji i pre iniciranja ciklusa na magistrali. U tom slučaju generiše se signal *Satisfied* i prelazi u početno stanje PCC_INITIAL, bez iniciranja ciklusa RdC. Ako keš blok i dalje nije u keš memoriji, zahtev se predaje BCU jedinici, a PCC jedinica prelazi u stanje BLOCKED u kome ostaje sve dok se traženi podatak ne nađe u keš memoriji.



Sl. 4-7. Dijagram toka Read zahteva jedinice PCC.

Write. Dijagram toka jedinice PCC tokom obrade Write zahteva prikazan je na Sl. 4-8. Odmah po prihvatanju zahteva jedinica PCC generiše signal *Satisfied* koji dozvoljava nastavak izvršavanja procesorske niti koja je inicirala zahtev, a potom nastavlja sa simulacijom ponašanja memorijskog podsistema. Proverava se da li se specificirani keš blok nalazi u keš memoriji. Ukoliko se ne nalazi, inicira se ciklus ekskluzivnog dohvatanja keš

bloka RdXC. U tom cilju, proverava se li je BCU jedinica spremna da prihvati zahtev ili nije. Ukoliko nije, PCC jedinica ulazi u stanje Write_BCU_WAIT i u tom stanju ostaje sve dok jedinica BCU ne postane spremna da prihvati novi zahtev. Ukoliko se specificirani keš blok već nalazi u keš memoriji u stanju S, po protokolu treba inicirati ciklus invalidacije InvC. Ovaj zahtev se na uobičajeni način predaje jedinici BCU ukoliko je jedinica spremna, inače se prelazi u stanje čekanja Write_BCU_WAIT dok jedinica BCU ne postane spremna da primi novi zahtev. Slično kao kod stanja Read_BCU_WAIT, jedinica PCC u svakom ciklusu stanja Write_BCU_WAIT proverava da li su uslovi za iniciranje RdXC ili InvC ciklusa ispunjeni. Tokom provere u stanju čekanja moguće je da se promeni zahtev BCU jedinici: prvobitno inicirani ciklus RdXC se može zameniti InvC i obratno, InvC sa RdXC. U slučaju da se specificirani keš blok nalazi u keš memoriji u stanju E, po protokolu za održavanje koherencije keš memorije PCC jedinica menja stanje u M bez iniciranja nekog ciklusa na magistrali.



Sl. 4-8. Dijagram toka Write zahteva jedinice PCC.

Lock. U prethodnom poglavlju diskutovane su različite implementacije sinhronizacione operacije *lock* (odeljak 3.4.1.1). U simulacionoj analizi koristi se *test&exch* implementacija sinhronizacione primitive *lock*. Ova implementacija podrazumeva da se najpre izvrši obično čitanje, pa tako jedinica PCC prolazi kroz niz koraka kao na Sl. 4-7, s tim da se ne generiše signal *Satisfied*. Nakon toga vrši se provera vrednosti specificirane *lock* varijable. Ukoliko je *lock* slobodan, procesor pokušava da izvrši `exch` operaciju kojom se čita vrednost *lock* promenljive i upisuje jedinica. Redosled koraka tokom upisa jedinice odgovara redosledu koji je opisan na Sl. 4-8, s tim da se signal *Satisfied* generiše tek nakon upisa, ukoliko je instrukcija `exch` izvršena uspešno, tj. procesor je dobio *lock*. Ukoliko *lock* nije slobodan, ulazi se u stanje Lock_SLEEP u kome se ostaje određeni broj simuliranih ciklusa takta;

trajanje stanja Lock_SLEEP se može specificirati na početku simulacije kao parametar. Nakon isteka predviđenog vremena keš kontroler ponovo pokušava da dobije *lock* prolazeći ponovo kroz opisani niz koraka.

Unlock. Operacija Unlock podrazumeva upis nule u specificiranu *lock* varijablu. Stoga, operacija Unlock se ponaša kao obična Write operacija, s tim da se posebno vodi računa da se oslobađa *lock*, upisom 0 u *lock* promenljivu.

Wrback (**Update**). Ovaj zahtev je posledica izvršavanja instrukcija za softverski inicirano ažuriranje glavne memorije i keš memorija drugih procesora korišćenjem mehanizma injektiranja. Slično operaciji upisa, prvi korak u opsluživanju ovog zahteva je generisanje signala *Satisfied* kojim se dozvoljava nastavak izvršavanja programske niti u sledećem ciklusu takta, bez obzira na rezultat izvršavanja ove operacije. Proverava se da li keš memorija posmatranog procesora poseduje specificirani keš blok u stanju M i ukoliko je to slučaj, proverava se spremnost BCU jedinice da prihvati zahtev WbC; ako je BCU jedinica zauzeta drugim zahtevom, čeka se u stanju Wrback_BCU_WAIT. Ukoliko se specificirani keš blok nalazi u stanju S ili se uopšte ne nalazi u keš memoriji posmatranog procesora, onda se ova operacija ponaša kao operacija bez dejstva.

Prefetch. Ovaj zahtev je posledica izvršavanja `prefetch` instrukcije i njegovo izvršavanje u potpunosti odgovara izvršavanju običnog Read zahteva, s tim da se ne blokira izvršavanje programske niti, tj. odmah po prihvatanju zahteva generiše se signal *Satisfied*.

Prefetch-Ex. Ovaj zahtev je posledica izvršavanja instrukcije `prefetch-ex`. Ponašanje keš kontrolera u opsluživanju ovog zahteva je kao u slučaju običnog Write zahteva, s tim da je novo stanje dohvaćenog keš bloka E.

OpenWin. Ovaj zahtev je uzrokovan izvršavanjem predložene instrukcije za inicijalizaciju tabele injektiranja `OpenWindow`. Jedinica PCC prihvata specificirani adresni opseg koji se smešta u prvi slobodan ulaz tabele injektiranja. Nakon toga, generiše se signal *Satisfied* i tako završava obrada zahteva u jednom ciklusu takta.

CloseWin. Zahtev je posledica izvršavanja instrukcije `CloseWindow`. Jedinica PCC proverava da li u tabeli injektiranja postoji validan ulaz sa specificiranim adresom. Ukoliko je to slučaj, odgovarajući ulaz se proglašava nevažećim; u suprotnom, instrukcija se ponaša kao instrukcija bez dejstva. U istom simuliranom ciklusu takta generiše se signal *Satisfied*.

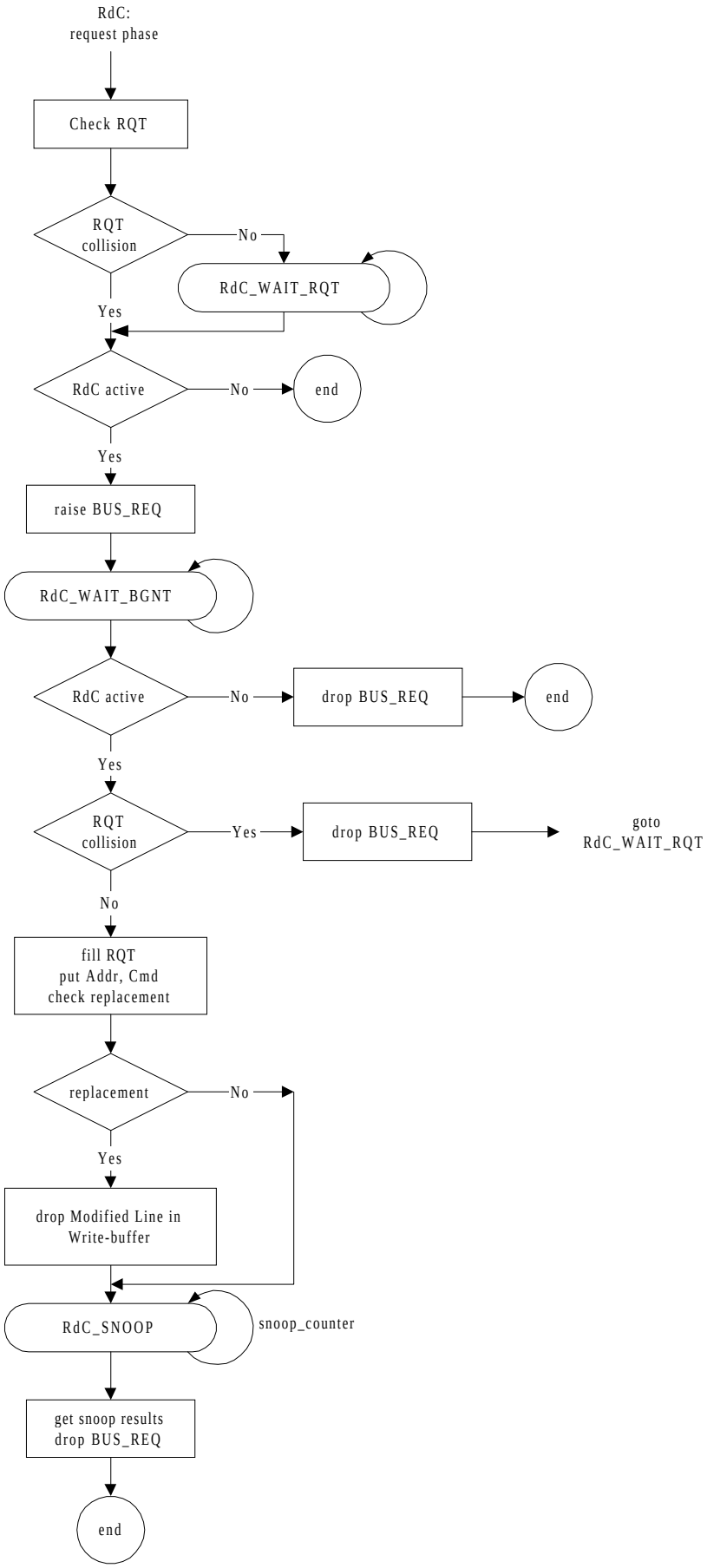
4.2.3.2 Jedinica BCU & SCC

BCU jedinica je odgovorna za izvršavanje sledećih ciklusa na magistrali: RdC - čitanje keš bloka iz memorije ili keš memorije drugog procesora, RdXC – ekskluzivno čitanje keš bloka iz memorije ili keš memorije drugog procesora, InvC – invalidacija kopija keš bloka u keš memorijama drugih procesora, WbRC – vraćanje modifikovanog keš bloka u glavnu memoriju uzrokovano izbacivanjem keš bloka usled politike zamene u keš memoriji, WbSC – ažuriranje kopije keš bloka u glavnoj memoriji i keš memorijama zainteresovanih procesora i WbIC – vraćanje modifikovanog keš bloka u glavnu memoriju uzrokovano mehanizmom injektiranja. U tekstu koji sledi opisani su relevantni ciklusi na magistrali.

RdC. Ciklus čitanja keš bloka se sastoji iz dve faze: postavljanja zahteva (*request*) i prihvatanja odgovora (*response*). Redosled glavnih koraka u fazi postavljanja zahteva ilustrovan je dijagramom toka na Sl. 4-9. Najpre se proverava sadržaj tabele aktivnih zahteva. Ukoliko je neki procesor inicirao zahtev na magistrali koji je u koliziji sa tekućim RdC zahtevom posmatranog procesora, onda BCU jedinica keš kontrolera ulazi u stanje

RdC_WAIT_RQT. U ovom stanju BCU jedinica ostaje sve dok se zahtev koji pravi koliziju ne završi i time izbriše iz tabele aktivnih zahteva. Keš kontroler može ostati u ovom stanju više simuliranih ciklusa takta. U opštem slučaju, moguće je da se usled akcija za održavanje koherencije keša ili mehanizma injektiranja promene početni uslovi dok se BCU jedinica nalazi u nekom stanju čekanja, tako da tekući zahtev BCU jedinice više nije od interesa. Na primer, ovakav scenario je moguć ukoliko se u tokom stanja čekanja mehanizmom injektiranja traženi keš blok nađe u keš memoriji. Stoga se nakon svakog stanja čekanja na neki događaj vrši provera da li su uslovi za iniciranje nekog ciklusa na magistrali još uvek validni. Ako jesu, postavlja se zahtev za magistralu i prelazi u stanje čekanja na dozvolu RdC_WAIT_BGNT. Ponovo, po pristizanju dozvole vrši se provera da li je zahtev još uvek validan. Ukoliko nije, oslobađa se magistrala i završava tekuća operacija. Ukoliko jeste, na adresnu magistralu se postavlja adresa traženog keš bloka a na kontrolne linije magistrale odgovarajući signali. Pored toga, u odgovarajući ulaz tabele aktivnih zahteva upisuju se parametri tekućeg zahteva (adresa i tip), a novo stanje PCC jedinice je RdC_SNOOP. Kako je aktivan signal Snoop_Req to svi keš kontroleri izvršavaju *snooping* ciklus i postavljaju odgovarajuće signale na kontrolnu magistralu. Trajanje *snooping* ciklusa je parametar simulacije koji se može menjati; u konkretnom modelu pretpostavlja se da *snooping* ciklus traje dva ciklusa takta. Rezultati *snooping* ciklusa se pamte, a keš kontroler oslobađa magistralu.

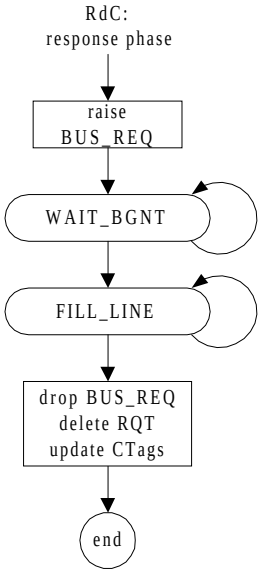
U konkretnom slučaju ne razmatraju se organizacioni detalji memorijskih modula, već se pretpostavlja idealizovani model po kome kontroler memorije odgovara na zahtev prilikom čitanja nakon vremena pristupa memoriji (MRC - *Memory Read Cycle*). Stoga se obraćanje memoriji može simulirati unutar keš kontrolera. U intervalu između postavljanja zahteva i dobijanja odgovora od memorije, drugi procesori mogu inicirati cikluse na magistrali koji nisu u koliziji sa tekućim zahtevom RdC. Nakon isteka vremena pristupa memoriji keš kontroler najpre postavlja zahtev za magistralu. Po dobijanju magistrale započinje faza prihvatanja odgovora od memorije (*response*) u kojoj se vrši transfer keš bloka preko magistrale podataka. Relevantni koraci ove faze prikazani su na Sl. 4-10. Trajanje transfera zavisi od veličine keš bloka i širine magistrale podataka. Širina magistrale podataka je parametar simulatora i može se menjati u opsegu od 32 bita, pa do veličine koja odgovara veličini keš bloka. Vreme koje protekne između prenosa dve uzastopne reči na magistrali predstavlja takođe parametar simulacije koji se može menjati. Nakon završetka ciklusa ukida se zahtev za magistralu, poništava odgovarajući ulaz u tabeli zahteva i ažuriraju stanja dohvaćenog keš bloka. Pored toga, aktivira se jedinica PCC koja je blokirana tokom dohvatanja keš bloka, tako da može da generiše signal *Satisfied* koji dozvoljava nastavak izvršavanja programske niti. Pri tom, treba napomenuti da je implementirana tehnika koja dozvoljava rano aktiviranje jedinice PCC. Naime, jedinici PCC se dozvoljava nastavak izvršavanja čim se dohvati tražena reč, pre završetka dohvatanja celog keš bloka.



Sl. 4-9. Dijagram toka RdC ciklusa na magistrali: faza postavljanja zahteva (*request phase*).

RdXC. Ciklus ekskluzivnog čitanja keš bloka se realizuje na sličan način kao i ciklus običnog čitanja na magistrali RdC. Male razlike postoje tokom *snooping* ciklusa u pogledu relevantnih linija koje se postavljaju na magistralu, kao i tokom faze prihvatanja odgovora, jer u slučaju ekskluzivnog dohvatanja nema interakcije sa jedinicom PCC.

InvC. Ciklus invalidacije se inicira kada procesor pokuša upis u keš blok koji se nalazi u stanju S (*Shared*). Da bi se omogućio upis, potrebno je da se prethodno invaliduju sve ostale kopije posmatranog keš bloka u keš memorijama drugih procesora. Redosled koraka tokom ciklusa invalidacije prikazan je na Sl. 4-11. Prvi korak je provera da li u tabeli aktivnih zahteva postoji neki zahtev koji je u koliziji sa tekućim. Ukoliko je to slučaj, BCU jedinica prelazi u stanje InvC_WAIT_RQT u kom ostaje sve dok se ciklus na magistrali koji pravi koliziju ne završi. Pre postavljanja zahteva za magistralu proverava se da li je zahtev za invalidacijom još uvek aktivan. Naime, moguće je da u međuvremenu neki drugi procesor invaliduje keš blok u keš memoriji procesora koji inicira zahtev za invalidacijom; u tom slučaju, tekući zahtev treba promeniti u ekskluzivno čitanje RdXC. Ukoliko je zahtev i dalje validan, postavlja se zahtev za magistralu arbitratoru magistrale i prelazi u stanje čekanja na dozvolu InvC_WAIT_BGNT. Po dobijanju dozvole ponovo se proverava da li je zahtev još uvek aktivan i da li sada postoji kolizija sa nekim od zahteva u tabeli aktivnih zahteva. Ukoliko je to slučaj, ukida se zahtev za magistralom i prelazi u stanje čekanje InvC_WAIT_RQT. Ako ne postoji kolizija, na adresnu magistralu se postavlja adresa keš bloka koji se invaliduje, a na upravljačke linije signali koji iniciraju invalidaciju. Nakon završetka *snooping* ciklusa na magistrali ukida se zahtev na magistralom i time završava ciklus invalidacije.

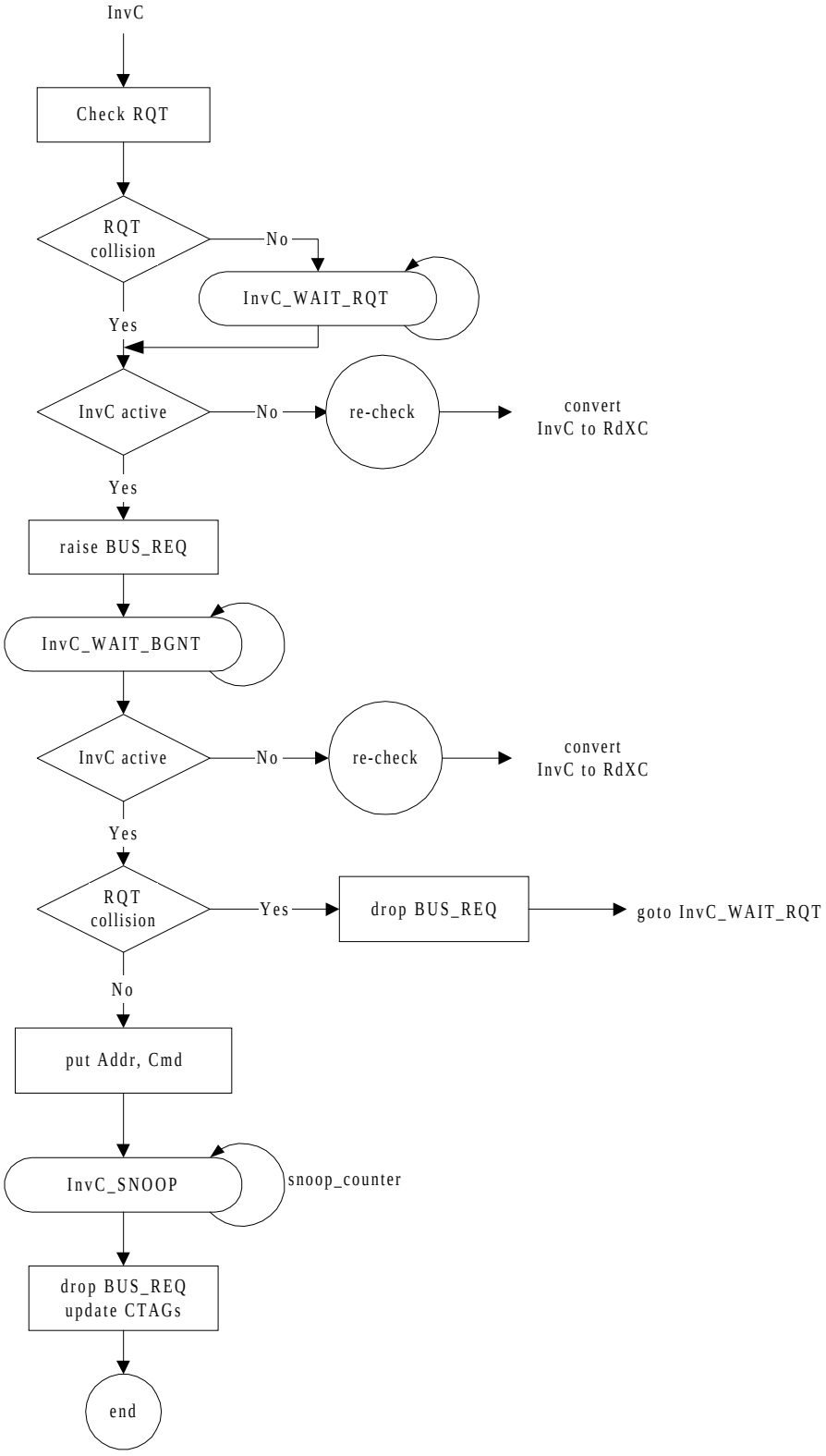


Sl. 4-10. Dijagram toka RdC ciklusa na magistrali: faza prihvatanja odgovora (*response phase*).

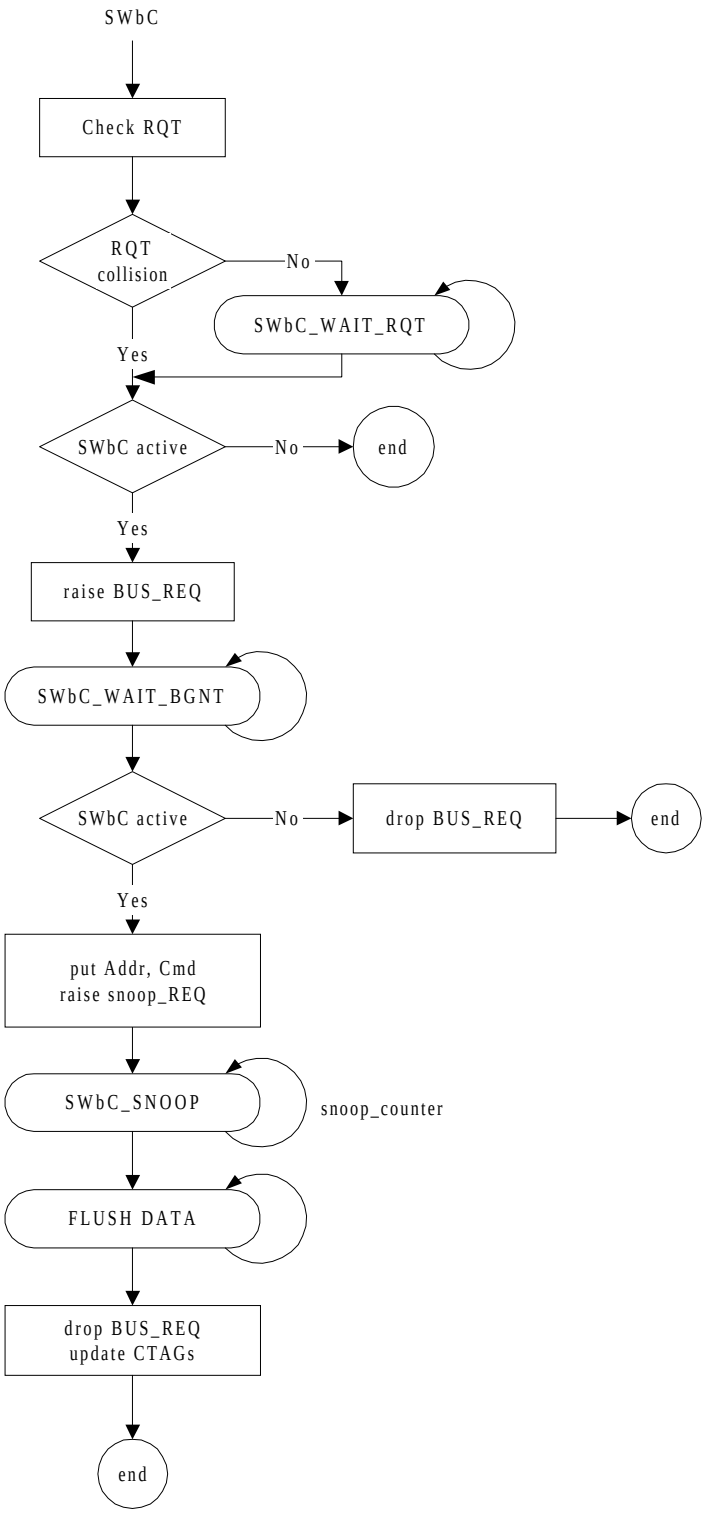
SWbC. Redosled relevantnih koraka tokom softverski iniciranog ciklusa ažuriranja glavne memorije i keš memorija procesora sa validnim ulazom u tabeli injektiranja prikazan je na Sl. 4-12. Prvi korak je provera tabele aktivnih zahteva sa ciljem da se izbegne kolizija sa nekim od trenutno aktivnih zahteva na magistrali. U slučaju kolizije sa nekim od zahteva u tabeli aktivnih zahteva, BCU jedinica realizuje čekanje u stanju SWbC_WAIT_RQT. Nakon provere validnosti zahteva, postavlja se zahtev za magistralom i prelazi u stanje čekanja na dozvolu SWbC_WAIT_BGNT. Po dobijanju dozvole i provere validnosti zahteva započinje *snooping* ciklus na magistrali; tokom ovog ciklusa keš kontroleri drugih procesora proveravaju da li postoje uslovi za primenu mehanizma injektiranja. Nakon toga, BCU

jedinica iznosi modifikovani keš blok na magistralu podataka i ažurira kopiju keš bloka u glavnoj memoriji. Istovremeno, keš kontroleri procesora koji imaju aktivan ulaz u tabeli injektiranja sa adresom posmatranog keš bloka prihvataju ovaj blok u svoju keš memoriju. Na kraju treba dati važnu napomenu o trajanju upisa keš bloka u glavnu memoriju. Pretpostavlja se da na strani glavne memorije postoje baferi dovoljnog kapaciteta koji prihvataju keš blok, tako da se upis u memoriju završava odmah po prenosu keš bloka preko magistrale podataka.

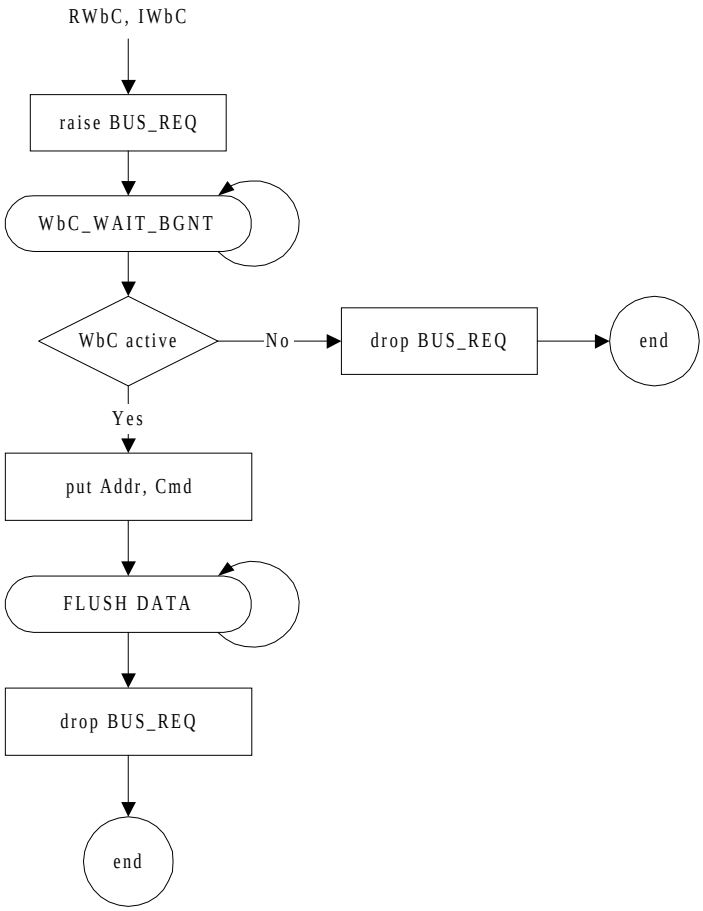
RWbC, **IWbC**. Ciklusi RWbC i IWbC realizuju ažuriranje glavne memorije modifikovanom vrednošću keš bloka koji je izbačen iz keš memorije usled politike zamene. U zavisnosti da li je izbacivanje keš bloka iz keš memorije posledica obrade regularnog zahteva od posmatranog procesora ili injektiranja, govori se o RWbC ili IWbC ciklusima, redom. Različite oznake za jedan isti ciklus na magistrali su uvedene u cilju lakšeg procenjivanja uticaja mehanizma injektiranja na broj keš blokova koji se izbaci iz keš memorije. U opštem slučaju, BCU jedinica postavlja zahtev za magistralu i prelazi u stanje čekanja na dozvolu WbC_WAIT_BGNT. Po dobijanju dozvole proverava se da li je zahtev još uvek validan i ukoliko je to slučaj započinje iznošenje modifikovanog keš bloka na magistralu podataka. Trajanje transfera podataka je kao i u prethodnim slučajevima određeno dužinom keš bloka i širinom magistrale podataka.



Sl. 4-11. Dijagram toka InvC ciklusa na magistrali.



Sl. 4-12. Dijagram toka SWbC ciklusa na magistrali.



Sl. 4-13. Dijagram toka RWbC ciklusa na magistrali.

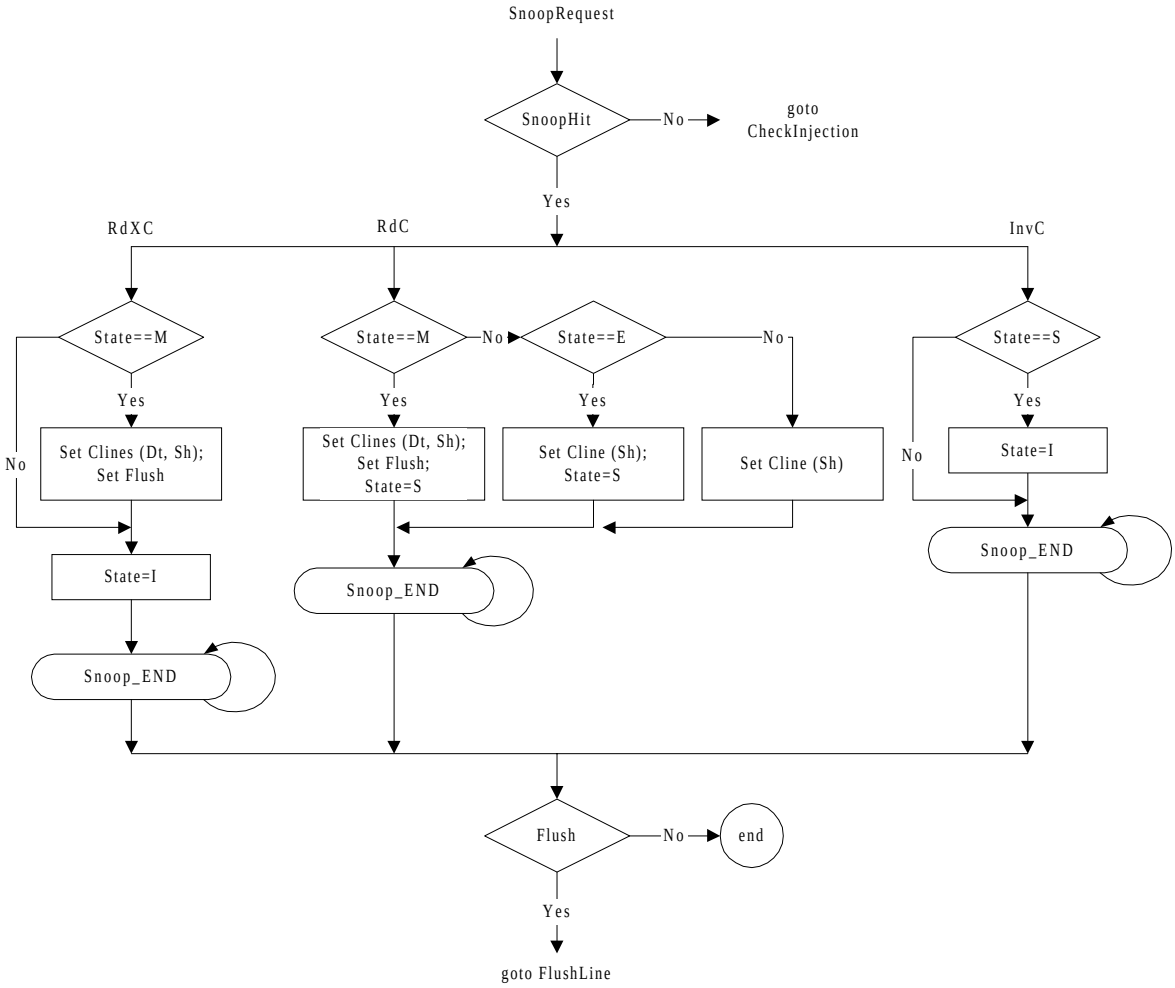
Do sada je opisano funkcionisanje BCU jedinice kada ona inicira cikluse RdC, RdXC, InvC, SWbC, RWbC i SWbC na zajedničkoj magistrali. Međutim, BCU&SCC jedinica obuhvata i jedinicu SCC (*Snoop Cache Controller*) koja je odgovorna za izvršavanje *snooping* ciklusa kada keš kontroler nekog drugog procesora inicira ciklus na magistrali. Pored toga, SCC jedinica je odgovorna i za proveru da li postoje uslovi za injektiranje i ako postoje, za realizaciju injektiranja u keš memoriju. U daljem tekstu opisano je funkcionisanje ovog dela kontrolera čiji je dijagram toka dat na Sl. 4-14.

Kada keš kontroler vidi aktivan *snooping* ciklus koji nije on inicirao, prvi korak je provera da li se u njegovoj keš memoriji nalazi keš blok čija se adresa nalazi na adresnoj magistrali (*SnoopHit*). Ukoliko to nije slučaj, prelazi se na deo koji proverava uslove za injektiranje (*CheckInjection*) koji će biti kasnije objašnjen; inače, dekoduje se tip ciklusa na magistrali. *Snooping* ciklus može biti posledica običnog ciklusa čitanja RdC, ekskluzivnog čitanja RdXC ili ciklusa invalidacije InvC.

RdXC. Ukoliko keš kontroler poseduje modifikovanu kopiju keš bloka (stanje M), najpre se postavljaju odgovarajuće linije kontrolne magistrale (Sh-*Shared* i Dt-*Dirty*) i indikacija da BCU jedinica kontrolera treba da inicira ciklus kojim iznosi keš blok na magistralu (*Flush*). Keš kontroler prelazi u stanje Snoop_END u kome ostaje sve dok se ne završi *snooping* ciklus na magistrali. Novo stanje keš bloka je I.

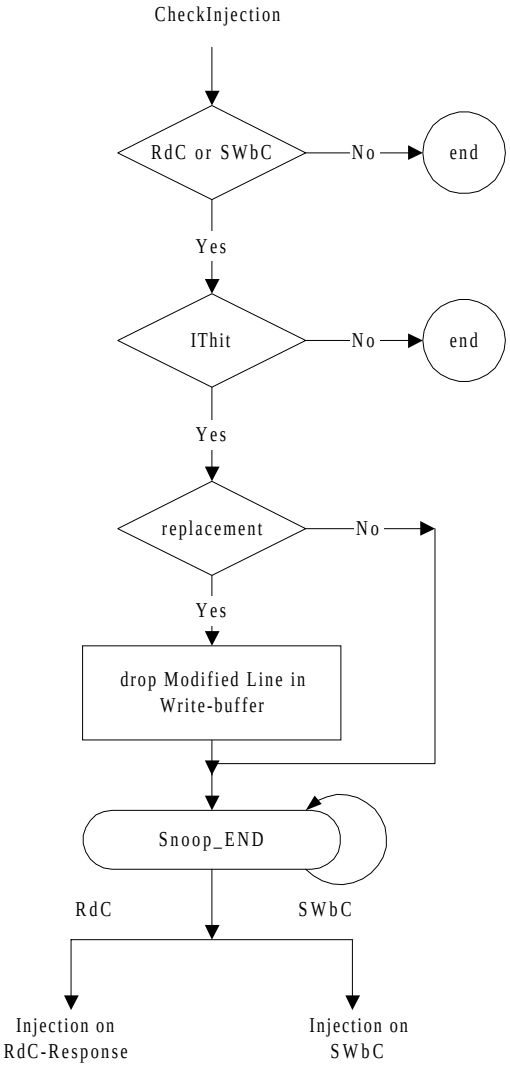
RdC. U zavisnosti od stanja keš bloka, keš kontroler na upravljačku magistralu postavlja odgovarajuće signale; u slučaju da je keš blok u stanju M, postavljaju se relevantne linije i indikacija da BCU jedinica treba da inicira ciklus kojim iznosi keš blok na magistralu (*Flush*). Prelazi se u stanje Snoop_END, a novo stanje keš bloka je S.

InvC. U slučaju invalidacije keš kontroler invaliduje svoju deljenu kopiju i prelazi u stanje Snoop_END u kome ostaje do završetka *snooping* ciklusa.



Sl. 4-14. Dijagram toka *snooping* ciklusa na magistrali.

Ukoliko keš kontroler utvrdi da keš memorija ne poseduje keš blok čija se adresa nalazi na magistrali, izvršava se provera da li postoje uslovi za primenu mehanizma injektiranja. Injektiranje je moguće tokom ciklusa čitanja RdC i ciklusa softverski iniciranog ažuriranja memorije SWbC. U tom slučaju proverava se da li se adresa keš bloka nalazi u nekom validnom ulazu tabele injektiranja. Ukoliko je to slučaj (*IThit*), proverava se da li injektiranje keš bloka dovodi do izbacivanja nekog modifikovanog keš bloka. U tom slučaju, modifikovan keš blok se prebacuje u bafer za odloženi upis (*WriteBuffer*). Nakon toga prelazi se u stanje u kome se čeka završetak *snooping* ciklusa. Samo injektiranje keš bloka se obavlja u fazi prihvatanja odgovora ciklusa čitanja RdC ili tokom transfera podataka ciklusa SWbC.



Sl. 4-15. Dijagram toka provere da li postoje uslovi za injektiranje.

4.3 Simulatori multiprocesorskog sistema

U cilju eksperimentalne verifikacije predloženog mehanizma u ovoj tezi korišćena su dva originalno razvijena simulatora memorijskog podsistema multiprocesora sa zajedničkom magistralom koji je opisan u odeljku 4.2.3. Za preliminarnu analizu efikasnosti predložene tehnike koristi se PRAM-MESI simulator, a za ispitivanje stvarnog uticaja predložene tehnike na vreme izvršavanja paralelnog programa korišćen je MESI-SPLIT simulator posmatranog memorijskog podsistema.

4.3.1 PRAM-MESI

PRAM-MESI simulator opisuje idealan memorijski podsistem multiprocesora sa zajedničkom magistralom. Reč "idealan" znači da se u ovom simulatoru ne razmatraju vremena trajanja pojedinih memorijskih operacija, a takođe se ne razmatra ni kontencija na zajedničkim resursima, pre svega na magistrali. Pretpostavlja se pojednostavljeni model u kome se svaka memorijska operacija, bez obzira da li se radi o čitanju ili upisu, o pogotku ili promašaju u keš

memoriji, izvršava u jednom ciklusu takta. U svim ostalim elementima memorijski podsistem je detaljno simuliran.

Ovakav simulator je podesan za preliminarno ispitivanje efikasnosti predloženog mehanizma injektiranja iz sledećih razloga. Mada ne omogućuje tačno merenje vremenskih odnosa, omogućuje iscrpno vođenje statistike o relevantnim događajima. Kako se keš memorija u potpunosti simulira, ovakav simulator omogućuje merenje procenta promašaja u keš memoriji, ukupnog saobraćaja na magistrali i sl. Rezultati dobijeni ovakvom analizom su nezavisni od implementacionih detalja koji se moraju definisati kod opisa realnog memorijskog podsistema kao što su interna organizacija keš kontrolera, parametri magistrale, organizacija memorije, itd. Pored toga, značajan momenat je i brzina ovakvih simulatora. Naime, kako se svaki relevantni događaj izvršava u jednom ciklusu takta vreme koje je potrebno za simulaciju je značajno manje u poređenju sa vremenom koje je potrebno kod simulatora koji opisuju realni memorijski podsistem.

Za razliku od ostalih memorijskih operacija koje se završavaju u jednom ciklusu takta, *lock* primitiva je realizovana u PRAM-MESI simulatoru na sledeći način. Koristi se *test&exch* pristup (vidi Sl. 3-17). Čita se keš blok i proverava vrednost *lock* varijable; ukoliko je *lock* slobodan, procesor dobija *lock*, proglašava ga zauzetim i završava simulaciju, sve to u jednom ciklusu takta; međutim, ukoliko *lock* nije slobodan, posmatrani procesor treba da se blokira i da nakon nekog vremena ponovo pokuša da dobije *lock*. Zaustavljanje procesora se realizuje tako što simulator ne generiše signal *Satisfied* sve dok ne dobije *lock*. Ukoliko *lock* nije slobodan, simulator memorijskog podsistema prelazi u stanje u kome čeka nekoliko ciklusa takta, pre ponovnog pokušaja da se dobije *lock*. Trajanje stanja čekanja na *lock* je parametar simulacije i može se definisati na početku simulacije. U svim eksperimentima baziranim na PRAM-MESI simulatoru uzeto je da stanje čekanja Lock_Wait traje četiri ciklusa takta. Nakon isteka četiri ciklusa takta, ponovo se simulira ciklus čitanja i proverava vrednost *lock* varijable. Ukoliko je sada *lock* slobodan, onda se generiše signal *Satisfied* i time dozvoljava nastavak izvršavanje programske niti; u suprotnom, ponovo se prelazi u stanje čekanje Lock_Wait.

Pored perioda Lock_Wait, u PRAM-MESI simulatoru mogu se definisati parametri koji definišu keš memoriju: kapacitet, asocijativnost i veličinu keš bloka. Dalje, pretpostavlja se da je kapacitet tabele injektiranja dovoljno veliki da ne dolazi do izbacivanja validnih ulaza tokom inicijalizacije. Tokom simulacije vodi se ekstenzivna statistika o relevantnim događajima. Za svaki zahtev (Read, Write, Lock, Unlock, Pref, Pref-ex i Update) vodi se statistika o broju promašaja i pogodaka u keš memoriji, broju izbačenih keš blokova usled politike zamene i injektiranja, broju iniciranih ciklusa na magistrali, itd; takođe, izračunava se i ukupni procenat promašaja u keš memoriji i zbirna statistika o saobraćaju na zajedničkoj magistrali.

4.3.2 MESI-SPLIT

MESI-SPLIT simulator u potpunosti opisuje ponašanje realnog memorijskog podsistema koji je opisan u odeljku 4.2.3. Za razliku od PRAM-MESI simulatora, detaljno se simuliraju svi resursi realnog memorijskog podsistema, počev od kontencije na internim resursima keš kontrolera, pa do zajedničke magistrale. Budući da se svaki zahtev memorijskom simulatoru simulira u potpunosti, ovakav simulator se koristi pre svega za merenje ukupnog vremena izvršavanja, ukupnog vremena blokiranja programske niti, vremena provedenog u pojedinim stanjima keš kontrolera, itd. Pored toga, vodi se statistika o procentu promašaja u keš memoriji, pre svega parcijalna po tipovima zahteva, a takođe i globalna posmatrajući ukupno

sve zahteve zajedno. Takođe, simulator omogućava merenje ukupnog saobraćaja na magistrali.

U MESI-SPLIT simulatoru mogu se definisati sledeći parametri: širina magistrale podataka (DATA_BUS_WIDTH), vreme pristupa memoriji tokom čitanja keš bloka (MRC – MEMORY_READ_CYCLE), vreme koje protekne između prenosa dve uzastopne reči na magistrali podataka tokom ciklusa čitanja (MEMORY_READ_CYCLE2), vreme pristupa memoriji tokom ciklusa upisa (MEMORY_WRITE_CYCLE), vreme između prenosa dve uzastopne reči keš bloka tokom ciklusa upisa (MEMORY_WRITE_CYCLE2), trajanje *snooping* ciklusa (SNOOP_WAIT), vreme koje protekne do sledećeg pokušaja da se dobije *lock* (LOCK_BUSY_SLEEP_COUNT). Pored ovih parametara, mogu se definisati i parametri koji definišu keš memoriju: veličina keš memorije (CACHE_SIZE), kapacitet keš bloka (CACHE_LINE_SIZE) i asocijativnost (CACHE_WAY). Kao u slučaju PRAM-MESI simulatora, pretpostavlja se da je veličina tabele injektiranja dovoljna da ni u jednoj aplikaciji ne dolazi do kolizije prilikom inicijalizacije tabele injektiranja.

4.4 Aplikacije i primena mehanizma injektiranja

U ovom odeljku opisane su aplikacije koje se koriste kao radno opterećenje u simulacionoj analizi. Za svaku aplikaciju najpre je dat kratak funkcionalni opis, a zatim i glavnih struktura podataka; nakon toga, objašnjena su relevantna deljenja podataka i implementacija podrške mehanizmu injektiranja.

Provera efikasnosti mehanizma injektiranja kod sinhronizacionih operacija *lock, unlock* i *barrier* bazirana je na korišćenju originalno razvijenih sintetičkih sinhronizacionih jezgara LTEST i BTEST. Provera efikasnosti predloženog mehanizma injektiranja u slučaju paralelnih aplikacija se bazira na originalno razvijenim aplikacijama PC, MM i Jacobi, i aplikacijama Radix, LU, FFT i Ocean koje su preuzete iz SPLASH-2 skupa paralelnih programa [WooO*95]. Aplikacije PC, MM i Jacobi su relativno jednostavni primeri koji reprezentuju karakteristična deljenja podataka koja se mogu sresti u većim i komplikovanijim aplikacijama. Aplikacije preuzete iz skupa SPLASH-2 koji je razvijen na Univerzitetu Stanford reprezentuju deljenja podataka koja se mogu sresti kod naučnih i inženjerskih aplikacija u oblastima sortiranja, digitalne obrade slike i govora, dinamike fluida, itd.

Pre razmatranja svake pojedinačne aplikacije u odeljku 4.4.1 dat je osvrt na postupak umetanja instrukcija za podršku injektiranju. U odeljku 4.4.2 opisana su sinhronizaciona jezgra LTEST i BTEST. Paralelne aplikacije PC, MM i Jacobi opisane su u odeljcima 4.4.3, 4.4.4 i 4.4.5, redom, a aplikacije Radix, LU, FFT i Ocean u odeljcima 4.4.6, 4.4.7, 4.4.8 i 4.4.9, redom.

4.4.1 Modifikacija polaznih programa

Modifikacija paralelnih aplikacija da bi se podržao mehanizam injektiranja vrši se ručno na osnovu statičke analize strukture paralelnih programa i tipa deljenja podataka između procesora. U ovom odeljku ukratko su navedene osnovne heuristike koje su korišćene u ručnoj modifikaciji programa.

U slučaju jednostavnih primera kao što su LTEST i BTEST modifikacija koda je trivijalna i podrazumeva umetanje na početak programa instrukcije za inicijalizaciju tabele injektiranja tako da se podrži injektiranje svih sinhronizacionih varijabli, odnosno keš blokova koji sadrže te varijable.

Kod paralelnih aplikacija PC, MM, Jacobi, Radix, LU, FFT i Ocean modifikacija koda podrazumeva sledeći postupak. Prvi korak je uključivanje podrške za sve sinhronizacione *lock* i *barrier* varijable. Kako je uključivanje ove podrške jednostavno (vidi odeljak 3.5) ono neće biti posebno razmatrano u delu koji je posvećen svakoj aplikaciji ponaosob. Drugi korak podrazumeva podršku injektiranju kod pravih deljenih podataka. Ukoliko postoji deljenje podataka tipa 1-Proizvođač-N-Potrošača (N>1) uključuje se podrška injektiranju tokom ciklusa čitanja. To znači da modifikacija koda treba da obezbedi pravovremenu inicijalizaciju tabele injektiranja kod procesora potrošača podataka, a takođe i invalidaciju ulaza po završetku faze u kojoj je takav tip komunikacije prisutan. Ukoliko postoji deljenje podataka tipa 1-Proizvođač-1-Potrošač, tj. komunikacija se odvija samo između dva procesora, modifikacija se vrši tako da podrži injektiranje tokom softverski iniciranog ciklusa ažuriranja glavne memorije. To podrazumeva modifikaciju koda na strani procesora proizvođača i procesora potrošača podataka.

Treba napomenuti da u slučaju deljenja tipa 1-Proizvođač-N-Potrošača podrška injektiranju tokom softverski iniciranog ciklusa ažuriranja glavne memorije potencijalno dovodi do poboljšanja efikasnosti u odnosu na primenjeni pristup koji podrazumeva samo injektiranje tokom ciklusa čitanja. Međutim, kompleksnost ručno modifikovanog koda i cena koja se plaća usled broja dodatih instrukcija su osnovni razlozi zašto je usvojen pristup da se u slučaju postojanja više korisnika koristi injektiranje samo tokom ciklusa čitanja.

4.4.2 LTEST & BTEST

Efikasnost mehanizma injektiranja kod sinhronizacionih primitiva *lock&unlock* i *barrier* proverava se korišćenjem originalno razvijenih primera LTEST i BTEST, redom, koji opisuju sintetičko radno opterećenje.

Sinhronizaciono jezgro LTEST je razvijeno po uzoru na slične primere korišćene u evaluaciji *lock* primitiva u radovima [LimAg94], [Shafi*97] (Sl. 4-16). LTEST se sastoji od `for` petlje u kojoj se nalazi samo jedna kritična sekcija čije je izvršavanje štićeno jednom *lock* varijablom `indexLock`. Trajanje kritične sekcije i segmenta koda između dva uzastopna zahteva za ulazak u kritičnu sekciju se simulira softverskim kašnjenjem bez realnog posla; kašnjenje je određeno vrednošću parametara C i D. Primena injektiranja ostvaruje se umetanjem instrukcije `OWL(indexLock)` na početak sinhronizacionog jezgra.

```
for(i=0; i<I; i++) {
   LOCK(indexLock);        /* enter the critical section */
      /*C – determines the size of critical section */
      for(j=0; j<C; j++);
   UNLOCK(indexLock);      /* leave the critical section */
      /* D – delay parameter */
     d=(1.0(rand()/MAX_RAND)*D
     for(j=0; j<d; j++);
};
```

Sl. 4-16. Sinhronizaciono jezgro LTEST.
Opis: Parametar C određuje trajanje kritične sekcije (bez realnog posla). Nakon *unlock* operacije kašnjenje koje se menja po uniformnoj raspodeli u opsegu [0, D] simulira vreme koje protekne pre nego što isti procesor ponovo zatraži *lock*. Parametar I definiše broj iteracija petlje.

Sinhronizaciono jezgro BTEST koristi se u evaluaciji mehanizma injektiranja kod primitive za globalnu sinhronizaciju *barrier*. BTEST sadrži `for` petlju u kojoj se simulira jedna epoha (Sl. 4-17), čije je trajanje određeno parametrom *t*. Injektiranje je podržano umetanjem instrukcije `OpenWindow(b→counterlock, b→sleepers)` na početak sinhro jezgra.

```
for(i=0; i<I; i++) {
   /*t – determines the size of an epoche */
    t = Tmin + (1.0*rand()/RAND_MAX)*(Tmax-Tmin);
    for(j=0; j<t; j++);
   BARRIER(b, numProc);
};
```

Sl. 4-17. Sinhronizaciono jezgro BTEST.
Opis: Parametar t određuje trajanje epohe (bez realnog posla) po uniformnoj raspodeli u opsegu [Tmin, Tmax], a parametar I definiše broj iteracija.

4.4.3 PC

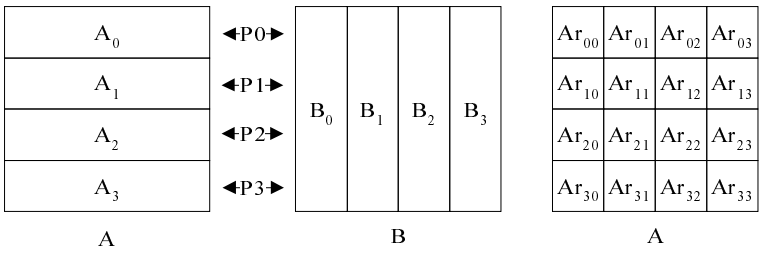
Originalno razvijena aplikacija PC ilustruje Proizvođač-Potrošač tip deljenja podataka; aplikacija je bazirana na primeru sa Sl. 3-6 koji je preuzet iz doktorske teze [Mowry94] (odeljak 3.3.1). Glavna struktura deljenih podataka je matrica dimenzija $M \times N$ koja se obrađuje paralelno u više iteracija. Dekompozicija problema je izvršena tako da je svakom procesoru pridružena submatrica dimenzija $(M/P) \cdot N$, pri čemu P predstavlja broj procesora u sistemu.

Glavno telo aplikacije PC čini `for` petlja koja se izvršava u I iteracija. Svaka iteracija se sastoji iz dva dela, međusobno razdvojena primitivom za globalnu sinhronizaciju. U prvom delu svaki procesor izračunava promenljivu `myVal` na osnovu vrednosti svih elemenata matrice i rednog broja procesora. U drugom delu svaki procesor modifikuje njemu pridružen deo matrice na osnovu izračunate vrednosti `myVal`. U sledećoj iteraciji svi procesori čitaju celu deljenu matricu. Tako, ako se posmatra jedna submatrica, na nivou susednih iteracija postoji deljenje podataka tipa 1-Proizvođač:(P-1)-Potrošača.

Polazna aplikacija je modifikovana da podrži mehanizam injektiranja tokom ciklusa čitanja, na način koji je pokazan na Sl. 3-12. U ovom slučaju, jedina modifikacija je umetanje instrukcija koje inicijalizuju tabelu injektiranja na početku programa; opseg adresa obuhvata adresni opseg deljene matrice i adresni opseg strukture podataka pridružene sinhronizacionoj primitivi *barrier*.

4.4.4 MM

Aplikacija MM predstavlja paralelnu verziju množenja dvodimenzionalnih matrica. Aplikacija MM prihvata kvadratne matrice `A` i `B` dimenzija $N \cdot N$, nalazi njihov proizvod, a rezultat smešta u matricu `A`. Dekompozicija je izvršena tako da je svaki procesor odgovoran za izračunavanje submatrice matrice `A` dimenzija $(N/P) \cdot N$, pri čemu P predstavlja broj procesora. Na Sl. 4-18 ilustrovan je primer množenja matrice kada je broj procesora u sistemu P=4. U posmatranom primeru procesor P0 izračunava submatrice $\texttt{Ar}_{00}$, $\texttt{Ar}_{01}$, $\texttt{Ar}_{02}$ i $\texttt{Ar}_{03}$, a procesor P3 izračunava $\texttt{Ar}_{30}$, $\texttt{Ar}_{31}$, $\texttt{Ar}_{32}$ i $\texttt{Ar}_{33}$, itd; pri tom, $Ar_{00} = A_0 \cdot B_0$, $Ar_{01} = A_0 \cdot B_1$, $Ar_{03} = A_0 \cdot B_3$, itd. Prema tome, procesor P0 tokom izračunavanja pridružene submatrice čita submatricu `A0` i celu matricu `B`, procesor P1 čita submatricu `A1` i celu matricu `B`, itd.

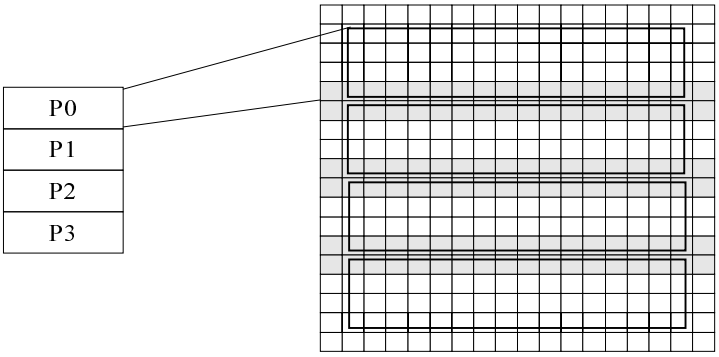


Sl. 4-18. Množenje matrica.

Kako svi procesori čitaju celu matricu B, primena mehanizma injektiranja tokom ciklusa čitanja elemenata matrice B može značajno redukovati saobraćaj na magistrali, a time i poboljšati performanse. Jedina modifikacija koja je potrebna za podršku injektiranju je umetanje jedne instrukcije za inicijalizaciju tabele injektiranja na početku programa; opseg adresa za koje se dozvoljava injektiranje odgovara opsegu adresa matrice B.

4.4.5 Jacobi

Aplikacija Jacobi je razvijena po uzoru na iterativne rešavače parcijalnih diferencijalnih jednačina [Amza*96], [Culler*98]. Glavna struktura podataka je dvodimenzionalna matrica sa $(N+2)\times(N+2)$ elemenata koja se iterativno obrađuje u I iteracija; pri tom, N predstavlja broj internih, negraničnih elemenata posmatrane matrice. Dekompozicija je izvršena tako da svaki procesor obrađuje submatricu kapaciteta $(N/P)\cdot N$. U svakoj iteraciji vrši se ažuriranje negraničnih elemenata submatrice na osnovu tekuće vrednosti i vrednosti 4 susedna elementa koji se nalaze ispod, iznad, levo i desno od posmatranog elementa. Na Sl. 4-19 prikazana je dvodimenzionalna matrica sa 16x16 negraničnih elemenata; podaci su dekomponovani na 4 submatrice, tako da je svakom procesoru pridružena po jedna submatrica kapaciteta 4x16. Elementi matrice koji se dele između različitih procesora su označeni sivom bojom. Da bi u nekoj iteraciji npr. procesor P0 izračunao element matrice Grid[3][0] (posmatraju se samo negranični elementi matrice) potrebno je da se pročita vrednost Grid[4][0] koju je modifikovao procesor P1 u prethodnoj iteraciji. Stoga, svaki procesor poseduje lokalne kopije deljenih vrsta matrice Grid, smeštene u pomoćnom nizu Scratch. Svaka iteracija sadrži dve faze međusobno razdvojene primitivom za globalnu sinhronizaciju. U prvoj fazi izračunavaju se elementi deljene matrice Grid koje ne koriste drugi procesori i lokalne kopije graničnih vrsta. U drugoj fazi, lokalne kopije graničnih vrsta se pridružuju matrici Grid, tako da će u prvoj fazi sledeće iteracije svaki procesor videti ažurne vrednosti elemenata matrice Grid.



Sl. 4-19. Jacobi: dekompozicija i deljenje podataka.

U ovom slučaju postoji deljenje podataka tipa 1-Proizvođač-1-Potrošač; za razliku od prethodna dva primera kada je postojalo više potrošača podataka u ovom primeru postoji

samo jedan potrošač. Postojanje samo jednog potrošača podataka čini da primena injektiranja tokom ciklusa čitanja nema smisla, ali može se koristiti injektiranje tokom softverski iniciranog ciklusa upisa u memoriju. Da bi se podržao ovaj tip injektiranja potrebno je izvršiti akcije i na strani procesora proizvođača i na strani procesora potrošača podataka. Procesor proizvođač podataka, nakon druge faze u kojoj ažurira granične elemente matrice Grid, instrukcijom `Update` inicira ažuriranje glavne memorije i keš memorija procesora koji će koristiti te podatke u prvoj fazi sledeće iteracije. Stoga, svaki procesor na početku programa inicijalizuje tabelu injektiranja, tako da prihvati podatke koji pripadaju graničnim vrstama submatrica susednih procesora. Tako, procesor P0 prihvata prvu vrstu submatrice koja pripada procesoru P1, procesor P1 prihvata poslednju vrstu submatrice koja pripada procesoru P0 i prvu vrstu submatrice koja pripada procesoru P2, itd.

4.4.6 Radix

Paralelna aplikacija RADIX, preuzeta iz skupa paralelnih programa SPLASH-2 [WooO*95], izvršava sortiranje celobrojnih ključeva po metodi koja je u literaturi poznata kao *radix* [Tanen*90]. Sortiranje po metodi *radix* je bazirano na vrednosti cifara u pozicionim brojnim sistemima. Jedna moguća implementacija ove metode sortira ključeve polazeći od cifre najmanje težine ka cifri najveće težine. Tako, ukoliko se posmatraju dvocifreni celi brojevi u decimalnom brojnom sistemu, brojevi se najpre sortiraju prema cifri jedinica, pa zatim po cifri desetica. U svakoj iteraciji sortiranja brojevi se grupišu u klase prema vrednosti posmatrane cifre; u slučaju decimalnog brojnog sistema postoji 10 klasa (0, 1, 2, ... 9). Iščitavanjem polaznog niza polazeći od klase 0 do klase 9 dobija se preuređen niz takav da se cifre jedinica nalaze u rastućem redosledu. Polazeći od novog niza postupak se ponavlja i za cifru desetica. Ponovnim iščitavanjem elemenata niza od klase 0 do klase 9 dobija se uređen niz u rastućem redosledu.

Opisani postupak ilustrovan je sledećim primerom. Posmatra se niz celih brojeva (13, 14, 45, 78, 92, 35, 67, 22, 91, 38). Brojevi se grupišu u 10 klasa prema vrednosti cifre jedinica. Tako, grupa "0" nema ni jedan član, grupa "1" ima jedan član 91, itd. Iščitavanjem elemenata niza po klasama dobija se sledeći niz uređen prema ciframa jedinica (91, 92, 22, 13, 14, 45, 35, 67, 78, 38). U sledećem koraku brojevi se grupišu prema cifri desetica po klasama. Npr., grupa "3" ima članove 35 i 38, grupa "6" ima jedan član 67, itd. Iščitavanjem elemenata niza redom po grupama dobija se uređen niz (13, 14, 22, 35, 38, 45, 67, 78, 91, 92).

Na Sl. 4-20 dato je jezgro glavne procedure Slave_Sort() koja se izvršava paralelno na svim procesorima. Nizovi `key[0]` i `key[1]` sadrže ključeve koje treba sortirati, a niz `rank` sadrži globalni histogram. Takođe, svakom procesoru pridružen je niz `rank_me` koji čuva lokalni histogram za svaku iteraciju sortiranja i niz `rank_ff` koji čuva globalni histogram svakog procesora.

Kod paralelne implementacije sortiranja po metodi *radix* svakom procesoru se pridružuje jedan deo niza. Proces sortiranja se odvija u više iteracija (petlja po `loopnum`) u zavisnosti od osnove *radix* i maksimalne moguće vrednosti celobrojnog ključa. Svaka iteracija se odvija u tri faze. U prvoj fazi svaki procesor najpre određuje lokalni histogram na bazi elemenata niza koji su pridruženi tom procesoru, a zatim se ažurira i globalni histogram. U tom delu iteracije čitaju se elementi pridruženog niza i izdvaja cifra po kojoj se vrši uređivanje i na osnovu toga određuje lokalni histogram. Ovo je ilustrovano sa READ{key[from][key_start:key_stop-1]} i WRITE{rank_me[0:radix-1]}. Nakon toga, ažurira se globalni histogram *rank*. Radi efikasnosti koristi se preklopljeno ažuriranje, tako da svi procesori uporedo ažuriraju pojedine segmente niza; međusobno isključivanje je ostvareno

nizom *lock* varijabli. Ova faza se završava globalnom sinhronizacijom. U drugoj fazi svaki procesor na osnovu globalnog histograma *rank* i lokalnih histograma procesora koji mu prethode *rank_me* određuje novi histogram *rank_ff* koji određuje novo mesto svakog elementa pridruženog niza u nizu ključeva. Na kraju, u trećoj fazi, na osnovu *rank_ff* histograma svaki procesor formira novi međuniz. Postupak se ponavlja u sledećoj iteraciji.

Modifikacija polazne aplikacije podrazumeva umetanje odgovarajućih instrukcija za podršku injektiranju na mestima gde postoji pravo deljenje podataka. Tako, u prvom delu iteracije koristi se injektiranje globalnog histograma *rank*. Histogram *rank* se ažurira od strane svih procesora, i to različiti delovi histograma se ažuriraju uporedo. Tako, u prvoj iteraciji po *kk* procesor P0 ažurira prvi deo niza *rank*, P1 ažurira drugi deo, itd. U sledećoj iteraciji P0 ažurira drugi deo niza *rank*, procesor P1 treći deo, itd. U poslednjoj iteraciji procesor P0 ažurira poslednji deo niza *rank*, a poslednji procesor ažurira prvi deo niza *rank*. Imajući to u vidu, svaki procesor inicijalizuje tabelu injektiranja za prihvatanje elemenata niza *rank* koje trenutno ažurira prvi sledeći procesor. Posmatrano lokalno, ovde postoji 1-Proizvođač-1-Potrošač tip deljenja, pa procesor proizvođač podataka treba da inicira ažuriranje instrukcijom `Update`, nakon upisa poslednje reči u keš blok (Sl. 4-20). U drugom delu iteracije svaki procesor izračunava svoj *rank_ff* na osnovu globalnog histograma *rank* i lokalnih histograma *rank_me* svih procesora sa nižim rednim brojem. Tako, svi procesori čitaju niz *rank* i *rank_me* procesora P0, (N-1) procesor čita *rank_me* procesora P1, itd. Stoga, postoji više čitalaca, pa se može koristiti injektiranje tokom ciklusa čitanja. Svaki procesor inicijalizuje tabelu injektiranja prema podacima koje će koristiti. U poslednjem delu petlje postoji intenzivno deljenje podataka. Međutim, u ovom slučaju svaki procesor upisuje element niza na odgovarajuće mesto u novom nizu prema svom globalnom histogramu *rank_ff*. Ukoliko veličina keš linije odgovara jednoj reči tada primena injektiranja daje velika poboljšanja. Međutim, u ostalim slučajevima to ne daje dobre rezultate usled izražene neregularne komunikacije svako-sa-svakim i prividnog deljenja koje nastaje kao posledica ove komunikacije.

```
key[0] = (int *) G_MALLOC(num_keys*sizeof(int));
key[1] = (int *) G_MALLOC(num_keys*sizeof(int));
rank = (int *) G_MALLOC(radix*sizeof(int));
size = number_of_processors*(radix*sizeof(int)+sizeof(int *));
rank_me = (int **) G_MALLOC(size);
   ...
void slave_sort() {
   ...
/* Do 1 iteration per digit.  */
OWL(global->barrier_rank.lock); OWH(global->barrier_rank.sleepers);
OWL(global->section_lock[0]); OWH(global->section_lock[number_of_processors]);
   ...
for (loopnum=0;loopnum<max_num_digits;loopnum++) {
  ...
/* generate histograms based on one digit */
  READ{key[from][key_start:key_stop-1]};
  WRITE{rank_me[0:radix-1]};
  /* open win for rank partition */
  OWL(rank_partition[0]); OWH(rank_partition[number_of_processors-1]);
  for ( kk = 0; kk < number_of_processors; kk++) {
   Ind = (MyNum+kk);
   if (Ind >= number_of_processors) Ind -= number_of_processors;
   OWL(rank[rank_partition[Ind+1]]); OWH(rank[rank_partition[Ind+2]]);
   ALOCK(global->section_lock,Ind);
    for (k=rank_partition[Ind]; k < rank_partition[Ind+1]; k++) {
     rank[k] = key_density[k] + rank[k];
     if(k%4==3) UPDATE(rank[k-3]);}
   AULOCK(global->section_lock,Ind);
   CWL(rank[rank_partition[Ind+1]]);
  }
  CWL(rank_partition[0]);
  BARRIER(global->barrier_rank, number_of_processors);
  OWL(rank[0]); OWH(rank[radix-1]);    /* open window for rank */
  for(i=0; i<MyNum; i++) {OWL(rank_me[i][0]);OWH(rank_me[i][radix-1]);}
  READ{rank[0:radix-1]};
  READ{rank_me[0:myNum-1][0:radix}};
  WRITE{rank_ff[MyNum][0:radix-1]};
  CWL(rank[0]); for(i=0; i<MyNum; i++) CWL(rank_me[i][0]); /* close rank win */
  BARRIER(global->barrier_rank, number_of_processors)
  /* put it in order according to this digit */
  READ(key[from][key_start:key_stop-1]};
  READ{rank_ff[MyNum][0:radix]};
  WRITE(key[to][somewhere[0:num_of_keys-1]};...
  BARRIER(global->barrier_rank, number_of_processors)
} /* for */
...
BARRIER(global->barrier_rank, number_of_processors)
CWL(global->barrier_rank.lock); CWL(global->section_lock[0]);
```

Sl. 4-20. Ilustracija kôda paralelne aplikacije RADIX uključujući instrukcije za podršku mehanizmu injektiranja.

4.4.7 LU

Paralelna aplikacija LU, preuzeta iz skupa paralelnih programa SPLASH-2 [WooO*95], izvršava triangularizaciju kvadratnih matrica (svi elementi ispod ili iznad glavne dijagonale su jednaki nuli) [Tanen*90], koristeći blokovski pristup. Kod paralelne implementacije ove aplikacije matrica A dimenzija $n \times n$ je podeljena na $B \times B$ submatrica dimenzija $b \times b$ ($n = b \times B$). Svaki blok je pridružen jednom procesoru koji je odgovoran za ažuriranje tog bloka tokom procesa triangularizacije. Da bi se redukovala komunikacija između procesora koristi se dvodimenzionalna raštrkana dekompozicija. Radi ilustracije na Sl. 4-21 je prikazana dekompozicija podataka u slučaju matrice dimenzija 64×64 sa veličinom bloka od 8×8, kada je broj procesora u sistemu P=8.

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	0	1	2	3	0	1	2	3
1	4	5	6	7	4	5	6	7
2	0	1	2	3	0	1	2	3
3	4	5	6	7	4	5	6	7
4	0	1	2	3	0	1	2	3
5	4	5	6	7	4	5	6	7
6	0	1	2	3	0	1	2	3
7	4	5	6	7	4	5	6	7

Sl. 4-21. 2D raštrkana dekompozicija kod aplikacije LU.
Opis: Matrica dimenzija 64×64 je podeljena na 64 (8×8) submatrice kapaciteta 8×8. Svaki kvadrat predstavlja jedan blok, a broj unutar kvadrata predstavlja ID procesora koji je pridružen tom bloku.

Na Sl. 4-22 prikazani su relevantni delovi kôda procedure *lu* u kojoj se vrši triangularizacija, uključujući i delove umetnutog koda za podršku mehanizmu injektiranja. U prvoj fazi prve iteracije procesor P0 vrši triangularizaciju bloka B(0,0) pozivom procedure *lu0*, dok su svi ostali procesori blokirani na barijeri `Global->start`. Nakon sinhronizacije na barijeri započinje druga faza u kojoj se ažuriraju blokovi nulte vrste {B(1,0), ..., B(7,0)} pozivom procedure *bdiv* i nulte kolone {B(0,1), ..., B(0,7)} pozivom procedure *bmodd*. Procesori P0, P1, P2 i P3 su odgovorni za ažuriranje blokova nulte vrste, a procesori P0 i P4 za blokove nulte kolone. U procesu ažuriranja svaki procesor čita i piše blok koji trenutno obrađuje i samo čita dijagonalni blok B(0,0). Po završetku druge faze svi procesori se sinhronizuju na sledećoj barijeri `Global->start`. U trećoj fazi vrši se ažuriranje svih ostalih blokova počev od prve vrste i prve kolone {B(1,1), ..., B(7,7)}, tj. svih blokova koji ne pripadaju prvoj vrsti i prvoj koloni, pozivom procedure *bmod*. Pri ažuriranju bloka B(I,J) procesor koji je odgovoran za taj blok čita blokove B(I,0) i B(0,J), a čita i piše blok B(I,J). Po završetku ove faze svi procesori se sinhronizuju na barijeri i započinje sledeća iteracija. U prvoj fazi sledeće iteracije procesor P5 ažurira dijagonalni blok B(1,1); u drugoj fazi ažuriraju se elementi prve vrste i prve kolone desno i ispod dijagonalnog elementa B(1,1), redom; u poslednjoj fazi ažuriraju se blokovi počev od bloka B(2,2) do B(7,7). Proces napredovanja po dijagonali se nastavlja dok se ceo proces ne završi.

U opštem slučaju procesor P*i* u prvoj fazi *k-te* iteracije ažurira dijagonalni blok B(K,K). U drugoj fazi odgovarajući procesori ažuriraju blokove duž *k-te* kolone i *k-te* vrste desno i ispod dijagonalnog elementa, redom. Kako u toj fazi više procesora čita blok B(K,K) koji je modifikovan u prethodnoj fazi, to je pogodno obezbediti da svi procesori inicijalizuju tabelu injektiranja tako da podrže injektiranje bloka B(K,K). Na početku druge faze vrši se analiza da li je procesor odgovoran za neki od blokova *k*-te vrste i *k*-te kolone, i ako jeste, inicijalizuje se tabela injektiranja (Sl. 4-22). U prvoj iteraciji posmatranog primera sa Sl. 4-21 inicijalizaciju tabele injektiranja vrše procesori P1, P2, P3, i P4. Mada je procesor P0 vlasnik bloka B(0,0) pokazalo se da je korisno da i procesori vlasnici nekog bloka inicijalizuju tabelu injektiranja, posebno za keš memorije malog kapaciteta. Tako, kada jedan iz grupe ovih procesora inicira čitanje neke reči iz bloka B(0,0) svi procesori će injektiranjem prihvatiti tu reč. Alternativni pristup je korišćenje mehanizma injektiranja tokom softverski iniciranog ciklusa ažuriranja. Ovaj pristup podrazumeva da svi procesori potrošači podataka inicijalizuju tabele injektiranja u prvoj fazi, a procesor proizvođač na kraju prve faze instrukcijama `Update` inicira ažuriranje keš memorija zainteresovanih procesora.

Važan problem je inicijalizacija tabele injektiranja na strani procesora potrošača podataka. Da bi se podržalo injektiranje npr. bloka B(0,0) iz posmatranog primera treba inicijalizovati tabelu injektiranja tako da sadrži adresni opseg ovog bloka. Međutim, u posmatranoj

implementaciji aplikacije LU cela matrica se alocira tako da zauzima kontinualni adresni prostor, a kako su elementi matrice u memoriji smešteni po vrstama, to sam blok B(0,0) ne zauzima kontinulani adresni prostor. Zbog toga se inicijalizacija tabele injektiranja vrši po vrstama bloka B, pa u posmatranom primeru treba inicijalizovati 8 ulaza tabele injektiranja. Ovakav pristup može dovesti do zauzimanja svih ulaza tabele injektiranja, što u slučaju tabela injektiranja malog kapaciteta može umanjiti efikasnost mehanizma injektiranja. Takođe, ovo utiče i na povećanu kompleksnost umetnutog koda za inicijalizaciju tabele injektiranja. Ovaj problem se može lako rešiti prealokacijom, tako da svaki blok sadrži kontinualni adresni prostor.

Na kraju druge faze vrši se invalidacija odgovarajućih ulaza tabele injektiranja koristeći instrukciju `CWL` (CloseWindow). U trećoj fazi postoji najveći potencijal za korišćenje mehanizma injektiranja. U toj fazi procesori ažuriraju redom blokove {B(K+1,K+1), ...B(B,B)} koji se nalaze desno i ispod dijagonalnog bloka B(K,K), a da pri tom ne pripadaju *k*-toj vrsti i *k*-toj koloni. Ako neki procesor ažurira blok B(I,J), on pri tom čita blokove B(K,J) i B(I,K) koji su ažurirani u prethodnoj fazi. Kako više procesora čita blokove B(K,J) i B(I,K), to treba inicijalizovati tabelu injektiranja tako da podrži injektiranje za te blokove. Pri tom, pokazalo se korisnim da se na početku treće faze izvrši potpuna inicijalizacija tabele injektiranja. Alternativni put je da se inicijalizacija vrši u glavnoj petlji u kojoj se poziva funkcija *bmod*; međutim, u tom slučaju se redukuje efikasnost injektiranja imajući u vidu činjenicu da obrada pojedinih blokova od strane različitih procesora nije orkestrirana, tj. moglo bi se desiti da neki procesor nema otvoren adresni opseg u trenutku kada neki drugi procesor čita podatke od interesa za taj procesor. Kako jedan procesor ažurira redom pripadajuće blokove po kolonama, to po završetku ažuriranja svih blokova u toj koloni J mogu se invalidovati ulazi u tabeli injektiranja koji obezbeđuju injektiranje bloka B(K,J). Ulaze u tabeli injektiranja koji obuhvataju blok B(I,K) treba ostaviti aktivne do kraja obrade jer se može desiti da injektiranje bude opet od koristi ukoliko su već injektirani podaci izbačeni iz keš memorije, posebno kod keš memorija malog kapaciteta.

```
.....
void lu(n, bs, MyNum, lc, dostats)
{
...
  /* main body */
  for (k=0, K=0; k<n; k+=bs, K++) {
    kl = k+bs; if (kl>n) {kl = n;}
    diagowner = BlockOwner(K, K);
    if (diagowner == MyNum) { /* factorize diagonal block */
      A = &(a[k+k*n]); lu0(A, kl-k, strI);}  /* READ/WRITE{b(K,K)}; */
    BARRIER(Global->start, P);
    D = &(a[k+k*n]);
    inject=0;
    for (i=kl, I=K+1; i<n; i+=bs, I++)
      if(BlockOwner(I,K)==MyNum)||(BlockOwner(K,I)==MyNum) {inject=1; break;}
    if (inject) for (i=0; i<bs; i++) {OWL(D[i*n]); OWH(D[i*n+bs-1])};
    for (i=kl, I=K+1; i<n; i+=bs, I++) {
      if (BlockOwner(I, K) == MyNum) {  /* parcel out blocks */
       ...; A = &(a[i+k*n]);
       bdiv(A,D,strI,n,il-i,kl-k);} /*READ{b(K,K),b(I,K)}; WRITE{b(I,K)};*/
    }
    for (j=kl, J=K+1; j<n; j+=bs, J++) { /* modify row k by diagonal block */
      if (BlockOwner(K, J)==MyNum) {     /* parcel out blocks */
        ...; A = &(a[k+j*n]);  /* READ{b(K,K), b(K,J)}; WRITE{b(K,J)};*/
        bmodd(D, A, kl-k, jl-j, n, strI);}
    }
    if (inject) { for (i=0; i<bs; i++) CWL(D[i*n]); inject=0;}
    BARRIER(Global->start, P);
    /* open windows */
    for (i=kl, I=K+1; i<n; i+=bs, I++) {
      colowner = BlockOwner(I,K); A = &(a[i+k*n]);
      for (j=kl, J=K+1; j<n; j+=bs, J++) {
        if (BlockOwner(I, J) == MyNum) {
           inject=1; B = &(a[k+j*n]);
           for (itmp=0; itmp<bs; itmp++)
             {OWL(B[itmp*n]); OWH(B[itmp*n+bs-1])}; }
      }
      if (inject) {
         for (itmp=0; itmp<bs; itmp++) {OWL(A[itmp*n]); OWH(A[itmp*n+bs-1])};
         inject=0; }
    }
    /* modify subsequent block columns */
    for (i=kl, I=K+1; i<n; i+=bs, I++) {
      ...; colowner = BlockOwner(I,K); A = &(a[i+k*n]);
      for (j=kl, J=K+1; j<n; j+=bs, J++) {
        ...
        if (BlockOwner(I, J) == MyNum) {  /* parcel out blocks */
          B = &(a[k+j*n]); C = &(a[i+j*n]);
          bmod(A,B,C,il-i,jl-j,kl-k,n); /*READ{b(I,K),b(K,J)};WRITE{b(I,J)};*/
        }
      }
      if (inject) {
         for (itmp=0; itmp<bs; itmp++) {CWL(A[itmp*n]); CWL(A[itmp*n])}
         inject=0; }
    } /* end for(j) */
  } /* end for(i) */
}
```

Sl. 4-22. Ilustracija kôda paralelne aplikacije LU sa instrukcjama za podršku injektiranju.

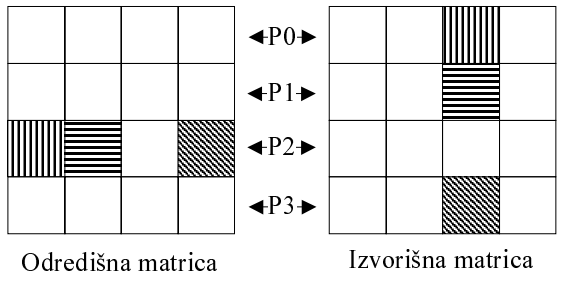
4.4.8 FFT

Paralelna aplikacija FFT (*Fast Fourier Transform*) za brzu Furijeovu transformaciju predstavlja reprezentativni uzorak izračunavanja koja se sreću kod širokog opsega aplikacija, počev od digitalne obrade slike i govora pa do modelovanja klimatskih uticaja. Posmatrana verzija paralelne aplikacije implementira jednodimenzionalni FFT algoritam u šest koraka prikazan na Sl. 4-23a.

Radno opterećenje paralelne aplikacije FFT čini *n* kompleksnih brojeva na koje treba primeniti Furijeovu transformaciju i *n* kompleksnih brojeva koje se nazivaju korenima jedinstva (*roots of unity*). Oba pomenuta skupa podataka su organizovana u matrice dimenzija $\sqrt{n} \times \sqrt{n}$ koje se nazivaju matrica podataka i matrica korena jedinstva, redom. Svaki procesor je odgovoran za izračunavanja svog dela matrice podataka dimenzija $\left(\sqrt{n}/p\right) \times \sqrt{n}$, pri čemu P predstavlja broj procesora. Komunikacija između procesora se odvija jedino u fazi transponovanja matrice podataka (*Transpose phase*), na način koji je ilustrovan na Sl. 4-23b. Posmatra se slučaj sa 4 procesora. Na slici je ilustrovano deljenje podataka kada procesor P2 izvršava deo koda koji vrši transponovanje pridruženog dela matrice.

1. Transponovanje matrice podataka
2. 1D FFT na svakoj vrsti matrice podataka
3. Transformacija matrice podataka korišćenjem matrice korena jedinstva
4. Transponovanje matrice
5. 1D FFT na svakoj vrsti matrice podataka
6. Transponovanje matrice podataka

(a) Algoritam u šest koraka



(b) FFT faza transponovanja matrice

Sl. 4-23. FFT: Algoritam i ilustracija faze transponovanja matrice podataka.

Na Sl. 4-24 dati su relevantni delovi kôda paralelne aplikacije FFT. Svaki procesor izvršava proceduru *SlaveStart*. Nakon inicijalizacije podataka poziva se procedura FFT1D u kojoj se izvršava FFT algoritam u šest koraka (Sl. 4-23a). U koracima 2., 3. i 5. vrši se obrada matrice podataka i to tako da svaki procesor modifikuje samo onaj deo matrice podataka koji mu je pridružen. U koracima 1., 4. i 6. vrši se transponovanje matrice podataka. Posmatrajmo primer koji je prikazan na Sl. 4-23. Procesor P2 u fazi transponovanja čita podatke čiji su ekskluzivni vlasnici procesori P0, P1 i P3, pa prema tome vidi promašaje u keš memoriji. Međutim, primena mehanizma injektiranja može doprineti eliminisanju ovakvih promašaja. Kako je deljenje podataka u ovom slučaju 1-Proizvođač-1-Potrošač, potrebno je da se na strani proizvođača podataka inicira ažuriranje keš memorija drugih procesora i glavne memorije instrukcijama `Update`, dok potrošač podataka treba da inicijalizuje tabelu injektiranja tako da prihvati podatke potrebne u fazi transponovanja. Tako, u proceduri *SlaveStart* umeću se instrukcije koje treba da podrže injektiranje koje rešava problem prvog transponovanja matrice (1. korak algoritma FFT). Pored toga, nakon koraka 3. i 5. FFT algoritma, umeću se instrukcije koje iniciraju ažuriranje memorije i keš memorija procesora (Sl. 4-24). Na strani potrošača, na primer, procesor P2 treba da inicijalizuje tabelu injektiranja tako da prihvati osenčene blokove od procesora P0, P1 i P3, redom (Sl. 4-24).

```
int N;                  /* N = 2^M                                  */
int rootN;              /* rootN = N^1/2                            */
double *x;              /* x is the original time-domain data       */
double *trans;          /* trans is used as scratch space           */
double *umain;          /* umain is roots of unity for 1D FFTs      */
double *umain2;         /* umain2 is entire roots of unity matrix */
int pad_length;

void OpenWindow(double *x, int MyNum, int MyFirst, int MyLast) {
  for(j=0; j<MyFirst; j++) {
    i=2*j*(rootN+pad_length)+MyNum*(rootN/P)*2; OWL(x[i]); OWH(x[i+2*rootN/P-1]);}
  for(j=MyLast; j<rootN; j++) {
    i=2*j*(rootN+pad_length)+MyNum*(rootN/P)*2; OWL(x[i]); OWH(x[i+2*rootN/P-1]);}
}
void CloseWindow(double *x, int MyNum, int MyFirst, int MyLast) {
  for(j=0; j<MyFirst; j++) {
    i=2*j*(rootN+pad_length)+MyNum*(rootN/P)*2; CWL(x[i]);}
  for(j=MyLast; j<rootN; j++) {
    i=2*j*(rootN+pad_length)+MyNum*(rootN/P)*2; CWL(x[i]);}
}
void SlaveStart()
{
  ...
  BARRIER(Global->start, P);
  OWL(umain[0]); OWH(umain[2*(rootN-1)-1]);
  upriv = (double *) malloc(2*(rootN-1)*sizeof(double));
  for (i=0;i<2*(rootN-1);i++) upriv[i] = umain[i];
  CWL(umain[0]);
  MyFirst = rootN*MyNum/P; MyLast = rootN*(MyNum+1)/P;
  OpenWindow(x, MyNum, MyFirst, MyLast); /*to support injection for Transpose*/
  TouchArray(....);
  CloseWindow(x, MyNum, MyFirst, MyLast);
  BARRIER(Global->start, P);
  FFT1D(...); /*perform forward FFT*/
  ...
}
void FFT1D(..., x, scratch, upriv, umain2, MyNum, ..., MyFirst, MyLast, pad_length,
P, ...)
{
  BARRIER(Global->start, P);
  Transpose(n1, x, scratch, MyNum, MyFirst, MyLast, pad_length);/*x into scratch*/
  OpenWindow(scratch, MyNum, MyFirst, MyLast);
  for (j=MyFirst; j<MyLast; j++) { /* do n1 1D FFTs on columns */
    FFT1DOnce(..., &scratch[2*j*(n1+pad_length)]);
    TwiddleOneCol(..., &scratch[2*j*(n1+pad_length)], pad_length);
    for(i=0; i<rootN; i++) /* cache line is 32B */
       if ((i<MyNum*rootN/P) || (i>=(MyNum+1)*rootN/P))
          if(i%2==1) WRBACK(scratch[2*j*(n1+pad_length)+2*i+1]);
  }
  CloseWindow(scratch, MyNum, MyFirst, MyLast);
  BARRIER(Global->start, P);
  Transpose(n1, scratch, x, MyNum, MyFirst, MyLast, pad_length);
  OpenWindow(x, MyNum, MyFirst, MyLast);
  for (j=MyFirst; j<MyLast; j++) {   /* do n1 1D FFTs on columns again */
    FFT1DOnce(direction, m1, n1, upriv, &x[2*j*(n1+pad_length)]);
    if (direction == -1) Scale(n1, N, &x[2*j*(n1+pad_length)]);
    for(i=0; i<rootN; i++)
       if ((i<MyNum*rootN/P) || (i>=(MyNum+1)*rootN/P))
          if(i%2==1) WRBACK(x[2*j*(n1+pad_length)+2*i+1]);
  }
  BARRIER(Global->start, P);
  Transpose(n1, x, scratch, MyNum, MyFirst, MyLast, pad_length);
  BARRIER(Global->start, P);
  ... /* copy columns from scratch to x */
  BARRIER(Global->start, P);
}
```

Sl. 4-24. Relevantni delovi koda aplikacije FFT sa ručno umetnutim instrukcijama za podršku injektiranju.

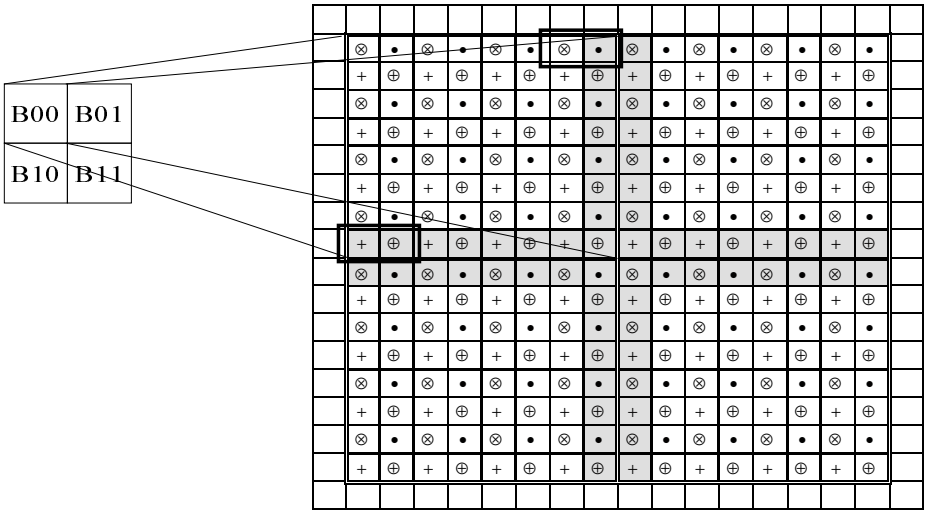
4.4.9 Ocean

Ocean je paralelna aplikacija koja simulira vodene struje u okeanu i predstavlja reprezentativnu paralelnu aplikaciju za izračunavanja u oblasti dinamike fluida. Svakom poprečnom preseku okeanskog bazena pridruženo je nekoliko promenljivih koje modeluju brzinu, temperaturu, pritisak i trenje. Svaka promenljiva je diskretizovana i predstavljena dvodimenzionalnim nizom $(n+2)\times(n+2)$ tačaka, pri čemu n predstavlja broj internih negraničnih tačaka. Nakon inicijalizacije, aplikacija obavlja simulaciju po vremenskim koracima koji se definišu kao parametar aplikacije. U svakom koraku rešavaju se eliptične diferencijalne jednačine koristeći *Red-Black Gauss-Seidel Multigrid* tehniku [Singh*91], [WooO*95]. Najveći deo vremena aplikacija provodi u izračunavanju diferencijalnih jednačina.

Glavninu struktura podataka čini 25 dvodimenzionalnih matrica. Svakom procesoru statički su pridružene odgovarajuće submatrice. Veći deo komunikacije između procesora se odvija prilikom izračunavanja ivičnih elemenata kada se koriste podaci koji pripadaju drugim procesorima. Detaljan opis algoritma i struktura podataka dat je u [Singh*91].

Na slici Sl. 4-25 ilustrovano je deljenje podataka u fazi rešavanja parcijalnih diferencijalnih jednačina, na primeru kada je broj negraničnih tačaka n=16, a broj procesora P=4. Podaci su raspodeljeni tako da je procesor P0 odgovoran za submatricu u gornjem levom uglu (B00), procesor P1 za izračunavanje submatrice u gornjem desnom uglu (B01), procesor P2 za submatricu u donjem levom uglu (B10), a procesor P3 za submatricu u donjem desnom uglu (B11). U svakom koraku algoritma ažurira se vrednost svakog pojedinačnog elementa na osnovu vrednosti susedna 4 elementa. Da bi se povećala efikasnost algoritma svi elementi su podeljeni u dve grupe: crvene označene simbolima $\otimes$ i $\oplus$, i crne označene simbolima $\bullet$ i +. Najpre se izračunavaju elementi jedne grupe, a zatim elementi druge grupe. Za izračunavanje elementa iz crne grupe koriste se vrednosti samo crvenih elemenata i obrnuto. Podaci koji se dele između procesora su osenčeni sivom bojom na slici.

Mehanizam injektiranja je primenjen na sve sinhronizacione *lock* i *barrier* varijable. Inače, u aplikaciji Ocean definisano je 6 *lock* i 19 *barrier* varijabli. Primena injektiranja na prave podatke implementirana je samo u fazi rešavanja parcijalnih diferencijalnih jednačina koja je ilustrovana na Sl. 4-25. Kako deljenje podataka odgovara tipu 1-Proizvođač-1-Potrošač, primenjuje se injektiranje tokom softverski iniciranog ciklusa ažuriranja. Procesor proizvođač podataka, instrukcijom `Update`, inicira ažuriranje keš memorije procesora potrošača keš blokom koji će se koristiti u izračunavanju elemenata u sledećoj iteraciji; u skladu sa mehanizmom injektiranja procesor potrošač podataka mora da na odgovarajući način inicijalizuje svoju tabelu injektiranja. Tako, u konkretnom primeru procesor P0 treba da inicira ažuriranje keš memorije procesora P3 poslednjom vrstom submatrice B00 i keš memorije procesora P1 poslednjom kolonom submatrice B00. Procesori P3 i P1 prethodno treba da inicijalizuju svoje tabele injektiranja za prihvatanje navedenih podataka. U prolazu kada se izračunavaju samo crni elementi vrši se ažuriranje crnih elemenata, a nakon izračunavanja crvenih elemenata ažuriraju se deljeni crveni elementi.



Sl. 4-25. Ocean: Ilustracija deljenja podataka u fazi rešavanja parcijalnih diferencijalnih jednačina.

U vezi sa ovako implementiranim injektiranjem treba prodiskutovati kompleksnost koda koji se umeće za podršku injektiranja. U opštem slučaju, svaki procesor deli podatke sa susedna četiri procesora koji izračunavaju elemente submatrice koje se nalaze iznad (Up), ispod (Bottom), levo (Left) i desno (Right) od submatrice koja je pridružena posmatranom procesoru; ivični procesori nemaju uvek sve susede. U originalnoj aplikaciji matrice podataka su alocirane tako da se podaci smeštaju po vrstama, zauzimajući kontinualni adresni prostor. Procesor treba da inicijalizuje tabelu injektiranja tako da prihvati poslednju vrstu submatrice pridružene procesoru Up i prvu vrstu submatrice pridružene procesoru Bottom. Kako ovi podaci zauzimaju kontinualni adresni prostor to je za svaki od njih je potreban po jedan ulaz u tabeli injektiranja. Procesor, takođe, treba da prihvati poslednju kolonu submatrice pridružene procesoru Left i prvu kolonu submatrice pridružene procesoru Right. Međutim, kolone ne zauzimaju kontinualni adresni prostor, pa se za svaki element mora inicijalizovati poseban ulaz u tabeli injektiranja.

Poglavlje 5

Rezultati

U ovom poglavlju prikazani su rezultati simulacione analize. U odeljku 5.1 razmatra se efikasnost mehanizma injektiranja u implementaciji sinhronizacionih primitiva *lock&unlock* i *barrier*. U odeljku 5.2 analizira se efikasnost mehanizma injektiranja na primeru originalno razvijenih paralelnih aplikacija PC, MM i Jacobi i aplikacija Radix, LU, FFT i Ocean preuzetih iz skupa paralelnih programa SPLASH-2. U odeljku 5.3 dat je kratak pregled dobijenih rezultata.

5.1 Mehanizam injektiranja kod sinhronizacionih primitiva

U ovom odeljku prikazani su rezultati simulacione provere efikasnosti mehanizma injektiranja za sinhronizacione primitive *lock&unlock* i *barrier*. Efikasnost mehanizma injektiranja u implementaciji sinhronizacione primitive *lock* proverava se na originalno razvijenom primeru LTEST, a primitive *barrier* na originalno razvijenom primeru BTEST. Eksperimenti su bazirani na simulatoru realnog memorijskog podsistema MESI-SPLIT (odeljak 4.3.2). MESI-SPLIT simulator podržava klasičnu *test&exch* implementaciju *lock* primitive koja je objašnjena u odeljku 3.4.1.1. Parametri memorijskog podsistema prikazani su na Sl. 5-1. Programskom prevodiocu GCC se zadaje parametar koji garantuje maksimalnu optimizaciju kôda, a alatu za instrumentaciju AUG naredba za prvi nivo instrumentacije.

Kao mera performanse koristi se vreme koje protekne od trenutka postavljanja zahteva za dobijanje *lock*-a (*lock request*), pa do trenutka dobijanja *lock*-a (*lock grant*), LAT (*Lock Acquire Time*) i ukupno vreme izvršavanja test primera, ET (*Execution Time*). Meri se LAT i ET za originalnu verziju primera B(ase) i verziju programa koja uključuje podršku mehanizmu injektiranja I(nject).

Rezultati provere efikasnosti mehanizma injektiranja kod sinhronizacione primitive *lock* prikazani su u odeljku 5.1.1, a kod sinhronizacione primitive *barrier* u odeljku 5.1.2.

MESI-SPLIT: Parametri memorijskog podsistema	
CacheSize	32KB
CacheLineSize	32B (8W)
DataBusWidth	8B (2W)
MemoryReadCycle	20/1 i 100/1 pclk
MemoryWriteCycle	1/1 pclk
SnoopCycle	2 pclk
LockSleepCounter	5 pclk

Sl. 5-1. Relevantni parametri simuliranog memorijskog podsistema.

5.1.1 Lock&Unlock

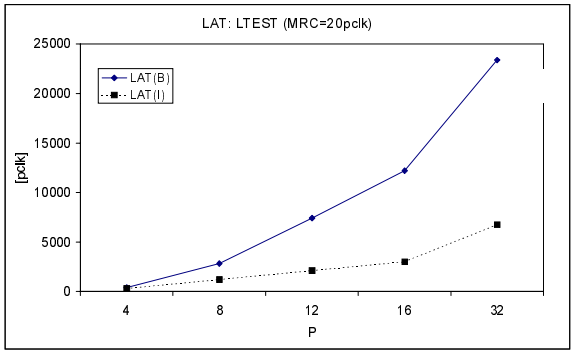
U eksperimentima se analiziraju srednje vreme potrebno za dobijanje *lock*-a (LAT) i vreme izvršavanja (ET) polaznog primera B(ase) i primera sa injektiranjem I(nject), kada je broj procesora u sistemu P=4, 8, 12, 16 i 32. Parametri LTEST primera su sledeći: svaki procesor inicira po 1000 zahteva za ulazak u kritičnu sekciju (I=1000), trajanje kritične sekcije je fiksno i iznosi 200 procesorskih ciklusa [pclk] (C=62), a vreme koje protekne od trenutka izlaska iz kritične sekcije pa do trenutka ponovnog zahteva za ulazak u kritičnu sekciju menja se po uniformnoj raspodeli u opsegu od 0 do 1000 pclk (D=300).

Na Sl. 5-2a i Sl. 5-2b prikazani su vreme dobijanja *lock*-a LAT i normalizovano vreme izvršavanja NET, redom, kada je vreme pristupa memoriji prilikom čitanja MRC=20pclk. Poboljšanje mereno relativnim smanjivanjem vremena LAT prema jednačini 100*(LAT(B)-LAT(I))/LAT(B) kreće se u opsegu od 27% za sistem sa P=4 procesora, do 75% za sistem sa P=16 procesora, odnosno oko 72% za sistem sa P=32 procesora. Mehanizam injektiranja redukuje i ukupno vreme izvršavanja test primera od 12% za sistem sa P=4 procesora do 79% za sistem sa P=32 procesora. Na Sl. 5-2c i Sl. 5-2d prikazani su vreme dobijanja *lock*-a LAT i normalizovano vreme izvršavanja NET, redom, kada je vreme pristupa memoriji prilikom čitanja MRC=100pclk. U ovom slučaju, mehanizam injektiranja redukuje vreme LAT u opsegu od 66% za sistem sa P=4 procesora do 77% za sistem sa P=32 procesora, a vreme izvršavanja se redukuje od 48% za sistem sa P=4 procesora do 84% za sistem sa P=32 procesora.

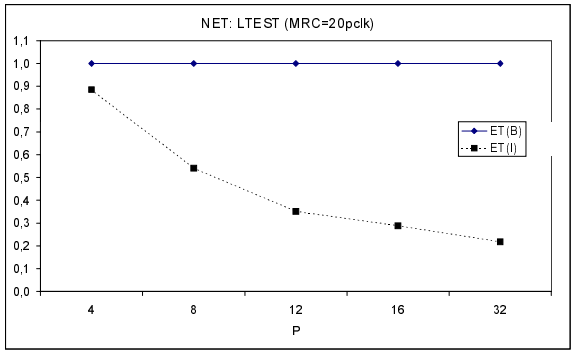
Na Sl. 5-3 prikazani su rezultati za slučaj kada je trajanje kritičnog regiona za red veličine manje nego u prethodnom primeru i iznosi 20pclk (C=8); svi ostali parametri test primera ostaju nepromenjeni (I=1000, D=300). U ovom slučaju mehanizam injektiranja redukuje vreme LAT od 36% za sistem sa P=4 procesora do 93% za sistem sa P=16 procesora i 90% za sistem sa P=32 procesora, kada je MRC=20pclk, odnosno od 76% za sistem sa P=4 procesora do 82% za sistem sa P=32 procesora, kada je MRC=100pclk. Vreme izvršavanja se redukuje od 6% za sistem sa P=4 procesora do 92% za sistem sa P=32 procesora, kada je MRC=20pclk, odnosno između 60% za sistem sa P=4 procesora i 88% za sistem sa P=32 procesora kada je MRC=100pclk.

Dobijeni rezultati potvrđuju očekivanja da efikasnost mehanizma injektiranja kod sinhronizacionih primitiva raste sa porastom broja procesora. Pri tom, vreme dobijanja *lock* primitive LAT zavisi gotovo linearno od broja procesora u sistemu. Odstupanje koje se pojavljuje kada je broj procesora u sistemu P=32 je posledica zagušenja zajedničke magistrale. Rezultati pokazuju da efikasnost mehanizma injektiranja raste sa porastom vremena pristupa memoriji tokom čitanja, MRC. Takođe, efikasnost raste sa skraćivanjem trajanja kritičnih sekcija; naime, kada je trajanje kritične sekcije kraće, iskorišćenost

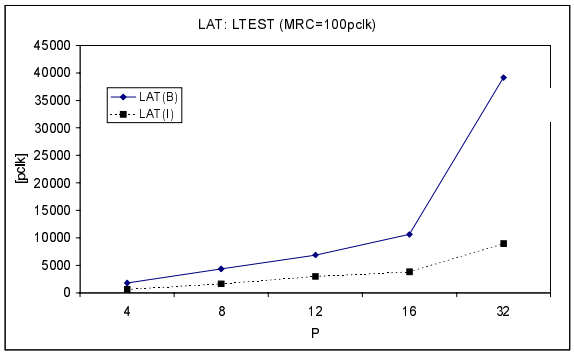
magistrale je veća, pa relaksiranje saobraćaja u slučaju mehanizma injektiranja dodatno utiče na poboljšanje performanse.



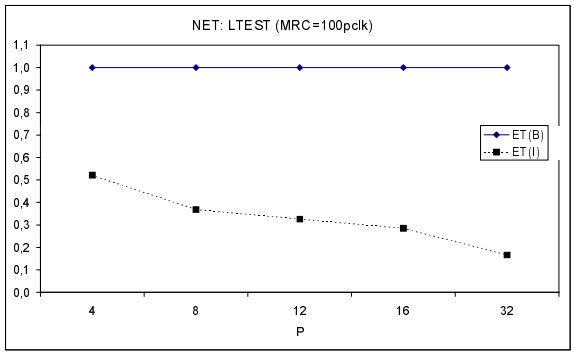
(a)



(b)

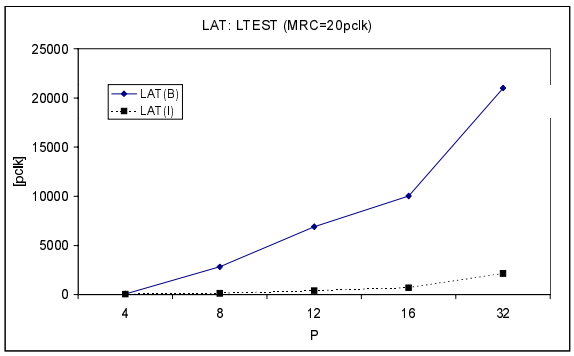


(c)

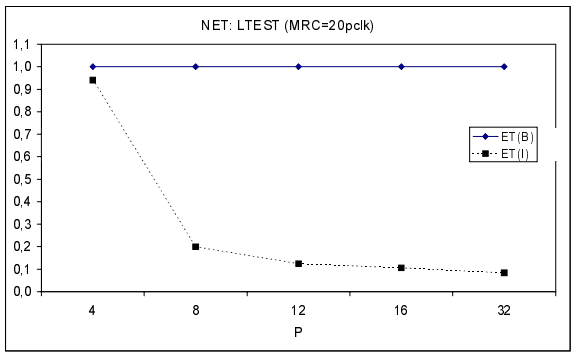


(d)

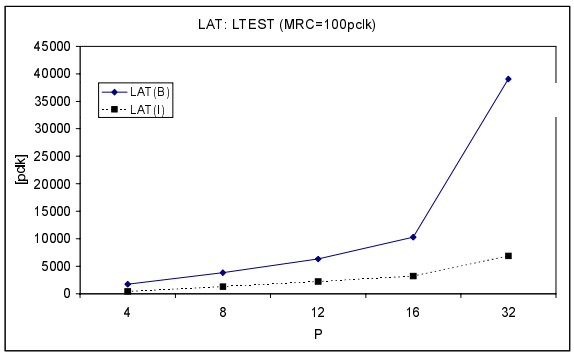
Sl. 5-2. LTEST: LAT i NET; (C=62, D=300).



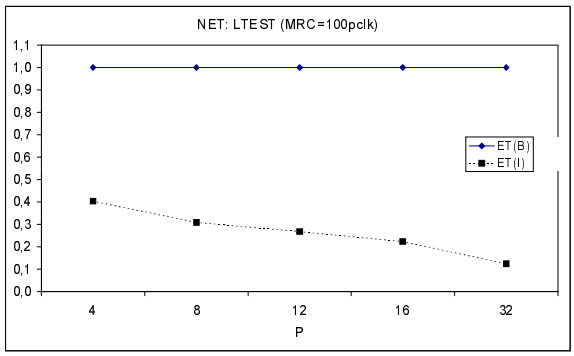
(a)



(b)



(c)



(d)

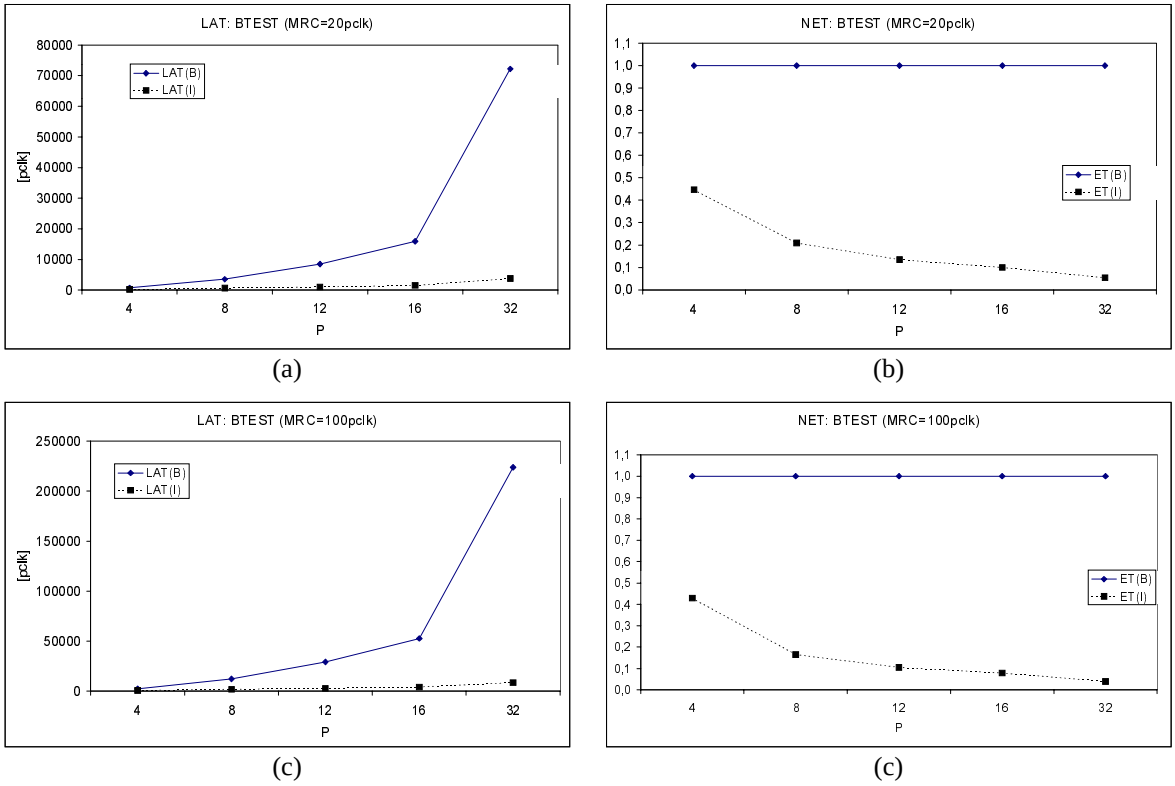
Sl. 5-3. LTEST: LAT i NET; (C=8, D=300).

5.1.2 Barrier

U cilju simulacione provere efikasnosti tehnike injektiranja kod sinhronizacione primitive *barrier* razvijeno je sinhronizaciono jezgro BTEST. MESI-SPLIT simulator podržava klasičnu implementaciju barijere koja je data na Sl. 3-20 (odeljak 3.4.1.2). Posmatra se polazna verzija sinhronizacionog jezgra BTEST sa Sl. 3-20 B(ase), i verzija koja uključuje podršku injektiranju I(nject). Parametri realnog memorijskog podsistema su prikazani na Sl. 5-1; broj procesora u sistemu se menja i uzima sledeće vrednosti P=4, 8, 12, 16 i 32. Parametri sinhronizacionog jezgra BTEST su sledeći: broj iteracija po jednom procesoru I=100, trajanje jedne epohe je određeno parametrom Tmin=Tmax=40, tj. iznosi oko 120pclk.

Na Sl. 5-4a i Sl. 5-4b prikazani su vreme LAT i normalizovano vreme izvršavanja NET, redom, u zavisnosti od broja procesora kada je MRC=20pclk. Mehanizam injektiranja redukuje vreme dobijanja *lock*-a od 67% za sistem sa P=4 procesora do 94,5% za sistem sa P=32 procesora. Vreme izvršavanja se redukuje od 56% za sistem sa P=4 procesora do 94,5% za sistem sa P=32 procesora. Na Sl. 5-4c i Sl. 5-4d prikazani su rezultati kada je MRC=100pclk. Vreme dobijanja *lock*-a se redukuje od 63% za sistem sa P=4 procesora do 96% za sistem sa P=32 procesora, a normalizovano vreme izvršavanja od 57% za sistem sa P=4 procesora do 96% za sistem sa P=32 procesora.

Prema očekivanju mehanizam injektiranja je izuzetno efikasan u redukciji vremena LAT i ET. Pri tom, efikasnost tehnike injektiranja raste sa porastom broja procesora u sistemu, a takođe i sa porastom vremena pristupa memoriji usled čitanja, MRC. U eksperimentima je analizirana i modifikovana implementacija barijere koja je pokazana na Sl. 3-31. Dobijeni rezultati pokazuju da umetanje dodatnih `Update` instrukcija ne doprinosi značajno skraćivanju vremena LAT i ukupnog vremena izvršavanja u odnosu na rešenje I. Osnovni razlozi za to su sledeći: (a) umetanje dodatnih instrukcija povećava dužinu koda, a time i vreme izvršavanja, (b) dodatne instrukcije povećavaju kontenciju na internim resursima keš kontrolera, i (c) umetnute `Update` instrukcije ne pronalaze uvek odgovarajući keš blok u stanju M(odified), jer je moguće da neki drugi procesor pre trenutka ažuriranja inicira dohvatanje odgovarajućeg keš bloka; u ovom slučaju umetnute `Update` instrukcije se ponašaju kao `noop` instrukcije.



(a) (b)

(c) (c)

Sl. 5-4. BTEST: LAT i NET; (Tmin=N=40, Tmax=X=40, I=100).

5.2 Mehanizam injektiranja kod paralelnih aplikacija

U ovom odeljku prikazani su rezultati simulacione analize koja ima za cilj da ispita efikasnost predloženog mehanizma injektiranja na odabranom skupu paralelnih aplikacija. Posmatraju se tri originalno razvijene jednostavne paralelne aplikacije PC, MM i Jacobi i četiri aplikacije preuzete iz skupa paralelnih programa SPLASH-2: Radix, LU, FFT i Ocean (odeljak 4.4).

Kod aplikacija PC, MM i Jacobi analiziraju se relevantni parametri performanse za polaznu aplikaciju (B) i aplikaciju koja uključuje podršku injektiranju sinhronizacionih varijabli i pravih deljenih podataka (Isd). Kod aplikacija iz skupa SPLASH-2 analiziraju se relevantni parametri performanse za polaznu aplikaciju (B), aplikaciju koja uključuje podršku injektiranju samo za sinhronizacione varijable (Is), i aplikaciju koja uključuje podršku injektiranju kako za sinhronizacione varijable tako i za prave deljene podatke (Isd).

Simulaciona analiza se sastoji iz dva dela. U prvom delu vrši se preliminarno ispitivanje efikasnosti tehnike injektiranja poređenjem performanse polaznog primera (B) i primera koji uključuje podršku injektiranju za sinhronizacione varijable i prave deljene podatke (Isd), korišćenjem simulatora idealnog memorijskog podsistema PRAM-MESI. Naime, zbog specifičnosti PRAM-MESI modela rezultati ispitivanja efikasnosti mehanizma injektiranja primenjenog samo na sinhronizacione varijable (Is) vrlo malo se razlikuju od rezultata za polazne aplikacije, pa se preliminarna analiza vrši samo za verzije aplikacija koje uključuju podršku injektiranju za prave deljene podatke i sinhronizacione varijable (Isd). Kao mera performanse koristi se procenat promašaja u keš memoriji (MR - *Miss Rate*) i saobraćaj na magistrali (BT – *Bus Traffic*) u zavisnosti od kapaciteta keš memorije. Pri tom, u okviru ukupnog saobraćaja na magistrali posebno se posmatraju transakcije RdC, RdXC, InvC,

RWbC, SWbC i IWbC objašnjene u odeljku 4.2.2. Relevantni parametri memorijskog podsistema su dati na Sl. 5-5.

PRAM-MESI: Parametri memorijskog podsistema	
CacheSize	16KB – 1024KB
CacheLineSize	32B (8W)
CacheWay	2
LockSleepCounter	4 pclk

Sl. 5-5. Relevantni parametri memorijskog podsistema opisanog PRAM-MESI simulatorom.

Drugi deo simulacione analize obuhvata analizu stvarnog uticaja tehnike injektiranja na ukupne performanse korišćenjem simulatora realnog memorijskog podsistema opisanog MESI-SPLIT simulatorom (vidi odeljke 4.2.3 i 4.3.2). Kao mera performanse posmatra se vreme izvršavanja (ET – *Execution Time*) i vreme blokiranja procesora usled čitanja (RST – *Read Stall Time*). Vreme izvršavanja i vreme blokiranja procesora u slučaju paralelnih aplikacija sa mehanizmom injektiranja su normalizovani prema vrednosti dobijenoj za polazne aplikacije koje ne uključuju mehanizam injektiranja. Normalizovano vreme izvršavanja (NET - *Normalized Execution Time*) paralelne aplikacije sa injektiranjem se izračunava na sledeći način: $NET_I = ET_I / ET_B$, pri čemu je ET_I vreme izvršavanja paralelne aplikacije sa injektiranjem, a ET_B vreme izvršavanja polazne aplikacije. Slično, normalizovano vreme blokiranja procesora usled promašaja u keš memoriji prilikom čitanja (*NRST – Normalized Read Stall Time*) kod aplikacije sa injektiranjem iznosi $NRST_I = RST_I / RST_B$, pri čemu je RST_I vreme blokiranja procesora kod aplikacije sa injektiranjem, a RST_B vreme blokiranja kod polazne aplikacije. Pored toga, analizira se ubrzanje (SU – *SpeedUp*) koje pokazuje zavisnost vremena izvršavanja od broja procesora u sistemu. Za polaznu aplikaciju bez primene injektiranja $SU_B(n) = ET_B(n) / ET_B(1)$, pri čemu je $ET_B(n)$ vreme izvršavanja polazne paralelne aplikacije sa *n* procesora u sistemu (*n*=2, 4, 8, 16 i 32), a $ET_B(1)$ je vreme izvršavanja na jednoprocesorskom sistemu. Za aplikaciju sa injektiranjem $SU_I(n) = ET_I(n) / ET_B(1)$, pri čemu je $ET_I(n)$ vreme izvršavanja paralelne aplikacije sa injektiranjem kada je broj procesora u sistemu *n*. Poboljšanja usled mehanizma injektiranja obično se predstavljaju u relativnom iznosu; tako, za *SU* se koristi sledeća formula $100*(SU_I - SU_B)/SU_B$. Na isti način se meri poboljšanje u slučaju drugih parametara performanse, kao što su vreme izvršavanja *ET*, vreme blokiranja *RST*, procenat promašaja u keš memoriji *MR*, itd.

Parametri realnog memorijskog podsistema prikazani su na Sl. 5-6. Dužina keš bloka je CacheLineSize=32B, asocijativnost keš memorije je CacheWay=2, kapacitet tabele injektiranja je IATsize=128 ulaza, širina magistrale podataka je DataBusSize=8B. Vreme odziva memorije na cikluse RdC i RdXC se menja od 20pclk do 120pclk. Nakon započinjanja *response* faze ovih ciklusa svaka sledeća reč (ili više reči u zavisnosti od širine magistrale podataka) se prosleđuje tokom dva procesorska ciklusa takta, sve dok se ne prenese ceo keš blok. Pretpostavlja se da se tokom transfera podataka ciklusa WbRC, WbSC i WbIC keš blok prenosi odmah nakon *snooping* faze u potrebnom broju ciklusa takta. *Snooping* faza traje SnoopCycle=2pclk, a vreme koje protekne između neuspešnog dobijanja *lock* primitive i sledećeg pokušaja dobijanja *lock*-a iznosi LockSleepCounter=5pclk. Svi eksperimenti za izračunavanje NET_I, $NRST_I$ i *SU* su ponovljeni za dve karakteristične veličine keš memorije, sa ciljem da se odredi osetljivost mehanizma injektiranja prema veličini keš memorije. Kod određivanja NET_I i $NRST_I$ eksperimenti se ponavljaju za različite vrednosti kašnjenja u pristupu memoriji prilikom čitanja MRC (*Memory Read Cycle*), MRC=20, 40, 60, 80, 100 i

120pclk, kada je broj procesora u sistemu P=8 i P=16. U svim eksperimentima alat za augmentaciju AUG vrši prvi nivo instrumentacije koda (Level 1), a prevodiocu *gcc* se prosleđuje parametar koji garantuje maksimalnu optimizaciju koda.

MESI-SPLIT: Parametri memorijskog podsistema	
CacheSize	64/128KB, 1024KB
CacheLineSize	32B (8W)
DataBusWidth	8B (2W)
MemoryReadCycle	20/1 - 120/1 pclk
MemoryWriteCycle	1/1 pclk
SnoopCycle	2 pclk
LockSleepCounter	5 pclk
IATsize	128

Sl. 5-6. Relevantni parametri realnog memorijskog podsistema opisanog MESI-SPLIT simulatorom.

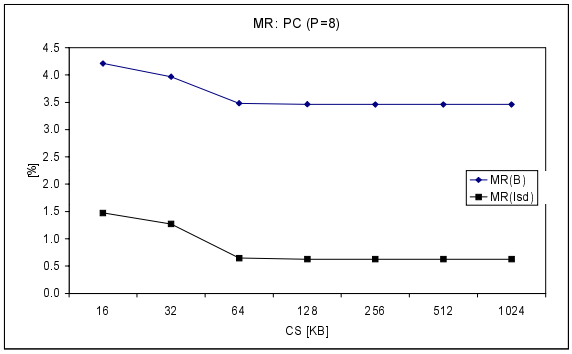
U tekstu koji sledi prikazani su dobijeni rezultati za svaku aplikaciju posebno. Za svaku aplikaciju dati su rezultati preliminarne analize na bazi simulatora PRAM-MESI, a potom rezultati simulacione analize zasnovane na korišćenju simulatora realnog memorijskog podsistema MESI-SPLIT. Tako, paralelne aplikacije PC, MM i Jacobi se analiziraju u odeljcima 5.2.1, 5.2.2 i 5.2.3, redom. Rezultati dobijeni analizom paralelnih aplikacija Radix, LU, FFT i Ocean prikazani su u odeljcima 5.2.4, 5.2.5, 5.2.6, 5.2.7, redom.

5.2.1 PC

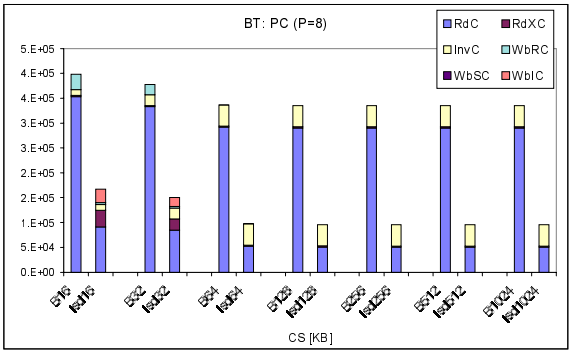
U ovom odeljku prikazani su rezultati simulacione analize za aplikaciju PC koja je opisana u odeljku 4.4.3. U eksperimentima se koriste sledeći parametri aplikacije: *M*=128, *N*=128 i *I*=20. U odeljku 5.2.1.1 prikazani su rezultati preliminarne analize korišćenjem PRAM-MESI simulatora, a u odeljku 5.2.1.2 prikazani su rezultati analize na bazi realnog simulatora MESI-SPLIT.

5.2.1.1 PC: PRAM-MESI

Na Sl. 5-7a i Sl. 5-7b prikazani su procenat promašaja u keš memoriji (MR) i saobraćaj na magistrali (BT) u zavisnosti od kapaciteta keš memorije, redom, kada je broj procesora u sistemu P=8. Mehanizam injektiranja redukuje procenat promašaja u keš memoriji (100*(MR(B)-MR(Isd))/MR(B)), u opsegu od 65% za keš memoriju kapaciteta 16KB do 82% za keš memoriju kapaciteta 1024KB. Blaga zavisnost procenta promašaja u keš memoriji od kapaciteta keš memorije ukazuje da je većina promašaja u keš memoriji posledica invalidacije deljenih podataka (*coherence misses*). Mehanizam injektiranja je upravo najefikasniji u redukciji ovog tipa promašaja u keš memoriji. Smanjivanje broja promašaja u keš memoriji se ostvaruje uz značajnu redukciju saobraćaja na magistrali, pre svega transakcija čitanja RdC. U slučaju keš memorija malog kapaciteta mehanizam injektiranja doprinosi povećavanju broja RdXC transakcija, ali je to povećavanje daleko manje od smanjivanja broja blokirajućih transakcija RdC, tj. važi RdC(B)>>RdC(Isd)+RdXC(Isd). Takođe, treba napomenuti da i pored značajnog broja modifikovanih keš blokova koji su izbačeni usled mehanizma injektiranja kod keš memorija malog kapaciteta, mehanizam injektiranja ne doprinosi ukupnom povećavanju izbačenih keš blokova, jer je RWbC(B)≈RWbC(Isd)+IWbC(Isd).



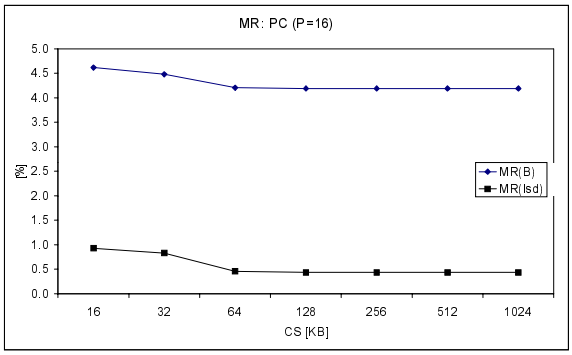
(a)



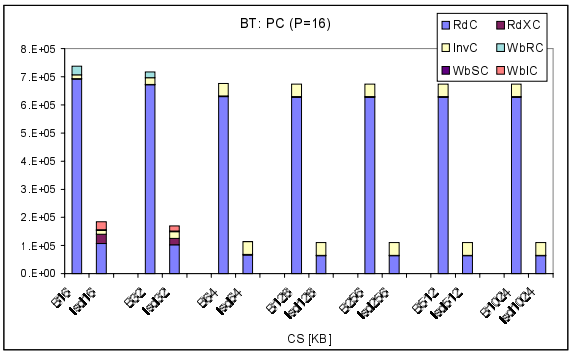
(b)

Sl. 5-7. PC: MR i BT; P=8.

Na Sl. 5-8a i Sl. 5-8b prikazani su procenat promašaja u keš memoriji i saobraćaj na magistrali u zavisnosti od kapaciteta keš memorije, redom, kada je broj procesora u sistemu P=16. Primenom injektiranja procenat promašaja u keš memoriji se redukuje u opsegu od 80% za keš memoriju kapaciteta 16KB do 89% za keš memoriju kapaciteta 1024KB. Za keš memorije velikog kapaciteta praktično se eliminišu promašaji usled ograničenog kapaciteta keš memorije (*capacity misses*) jer je WbRC≈0. Poređenjem procenta promašaja, npr. za keš memoriju kapaciteta 1024KB, za sistem sa P=8 i P=16 procesora dolazi se do očekivanog zaključka da efikasnost injektiranja tokom ciklusa čitanja raste sa porastom broja procesora u sistemu.



(a)



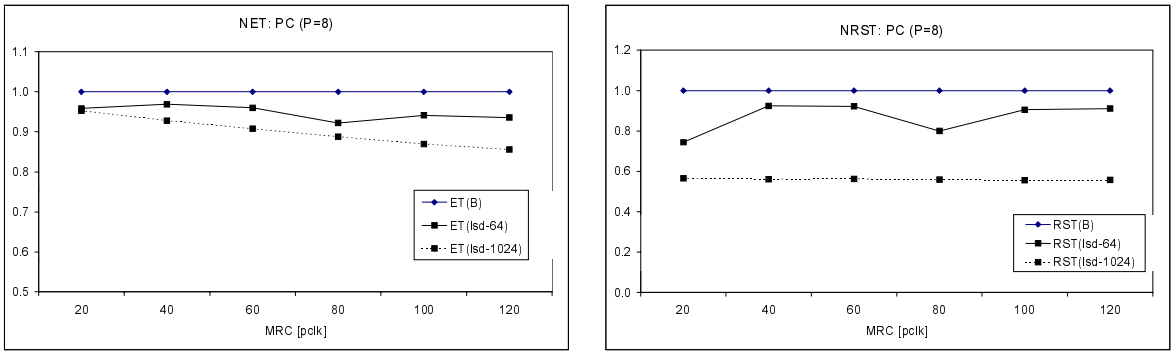
(b)

Sl. 5-8. PC: MR i BT; P=16.

5.2.1.2 PC: MESI-SPLIT

Na Sl. 5-9a i Sl. 5-9b prikazani su normalizovano vreme izvršavanja NET i normalizovano vreme blokiranja prilikom čitanja NRST, redom, za sistem sa P=8 procesora, kada je kapacitet keš memorije CacheSize=64KB (Isd-64) i CacheSize=1024KB (Isd-1024). Za sistem sa keš memorijom kapaciteta 64KB vreme izvršavanja se redukuje u opsegu od 3% do 8% (ET(Isd-64)), u zavisnosti od vremena pristupa memoriji; međutim, oblik krive ET(Isd-64) ukazuje na nepravilnu zavisnost vremena izvršavanja od vremena pristupa memoriji. Ovakvo ponašanje je rezultat, pre svega, kontencije na internim resursima i njihove zavisnosti od kašnjenja u pristupu memoriji, a takođe i relativno velikog broja modifikovanih keš blokova koji se izbacuju iz keš memorije usled mehanizma injektiranja, u uslovima kada je kapacitet keš memorije relativno mali. Vreme blokiranja procesora, takođe pokazuje nepravilnu zavisnost od vremena pristupa memoriji prilikom čitanja, a procenat redukcije je od 7% do 25% (kriva RST(Isd-64)). Za sistem sa keš memorijom kapaciteta 1024KB (Isd-1024) primena injektiranja redukuje vreme izvršavanja u opsegu od 4,5% kada je MRC=20pclk do

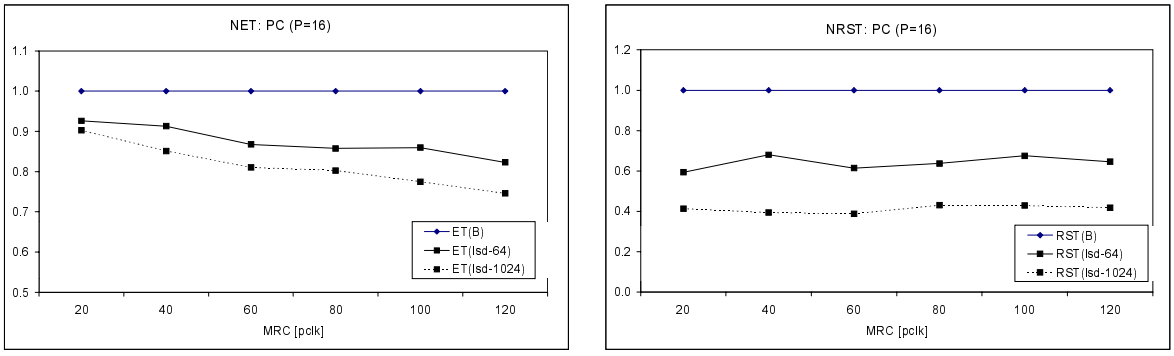
14,5% kada je MRC=100pclk (ET(Isd-1024)), dok se vreme blokiranja procesora redukuje za oko 56% i ne zavisi od vremena pristupa memoriji (RST(Isd-1024)).



(a) (b)

Sl. 5-9. PC: NET i NRST; P=8.

Na Sl. 5-10 prikazani su rezultati kada je broj procesora u sistemu P=16. Za sistem sa keš memorijom kapaciteta 64KB mehanizam injektiranja (Isd-64) redukuje vreme izvršavanja od 8% kada je MRC=20pclk do 17% kada je MRC=100pclk (ET(Isd-64)), a vreme blokiranja od 32% do 40% (RST(Isd-64)). Za sistem sa keš memorijom kapaciteta 1024KB (Isd-1024) vreme izvršavanja se redukuje od 10% kada je MRC=20pclk do 25% kada je MRC=100pclk (ET(Isd-1024)), a vreme blokiranja za oko 60% (RST(Isd-1024)).



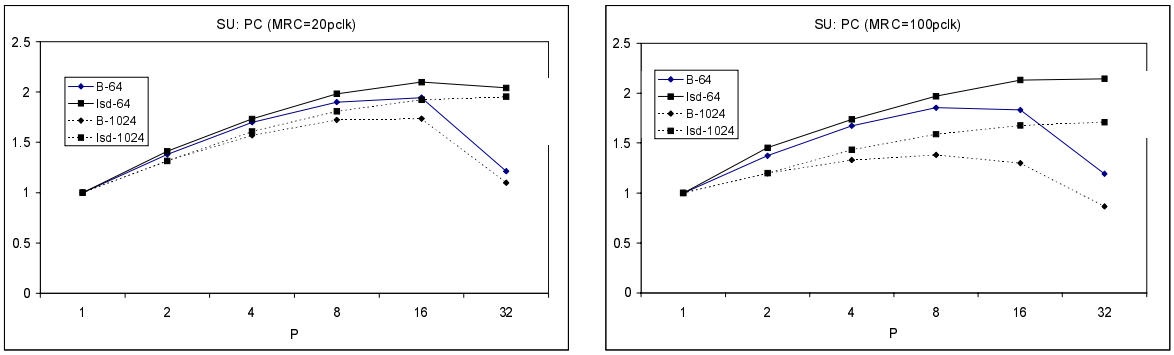
(a) (b)

Sl. 5-10. PC: NET i NRST; P=16.

Za ovu aplikaciju potvrđena su očekivanja da efikasnost mehanizma injektiranja raste sa porastom kapaciteta keš memorije, jer se time smanjuje verovatnoća kolizije u keš memoriji između injektiranih keš blokova i keš blokova koji se obrađuju u trenutku injektiranja. Takođe, efikasnost mehanizma injektiranja raste sa porastom broja procesora u sistemu.

Na Sl. 5-11a i Sl. 5-11b prikazano je ubrzanje SU u sistemu sa keš memorijom kapaciteta 64KB i 1024KB i različitim vremenima pristupa memoriji prilikom čitanja, MRC=20pclk i MRC=100pclk, redom, za polaznu aplikaciju (B-64 i B-1024) i aplikaciju sa injektiranjem (Isd-64 i Isd-1024). Apsolutni iznosi veličine SU nisu od interesa, jer u ovom primeru dimenzija problema raste sa porastom broja procesora; međutim, relativni odnos vremena izvršavanja polazne aplikacije i aplikacije sa injektiranjem omogućava korektnu procenu efikasnosti mehanizma injektiranja. Na osnovu dobijenih rezultata za aplikaciju PC mogu se izvesti sledeći zaključci:
(a) mehanizam injektiranja poboljšava performanse bez obzira na kapacitet keš memorije i vreme pristupa memoriji tokom ciklusa čitanja.

(b) procenat poboljšanja raste sa porastom broja procesora. Tako, u sistemu sa keš memorijom kapaciteta 64KB i vremenom pristupa MRC=20pclk, mehanizam injektiranja poboljšava performanse za 2,13% u sistemu sa P=2 procesora, odnosno za 40,6% u sistemu sa P=32 procesora.
(c) procenat poboljšanja raste sa porastom kapaciteta keš memorije. U sistemu sa P=32 procesora i vremenom pristupa od MRC=20pclk, mehanizam injektiranja poboljšava performanse za 40,6% u sistemu sa keš memorijom kapaciteta 64KB, odnosno za 43,8% u sistemu sa keš memorijom kapaciteta 1024KB.
(d) procenat poboljšanja raste sa porastom vremena pristupa memoriji. Tako, u sistemu sa keš memorijom kapaciteta 1024KB i P=32 procesora, mehanizam injektiranja poboljšava performanse za 43,8% kada je MRC=20pclk, odnosno za 49,4% kada je MRC=100pclk.



(a) (b)

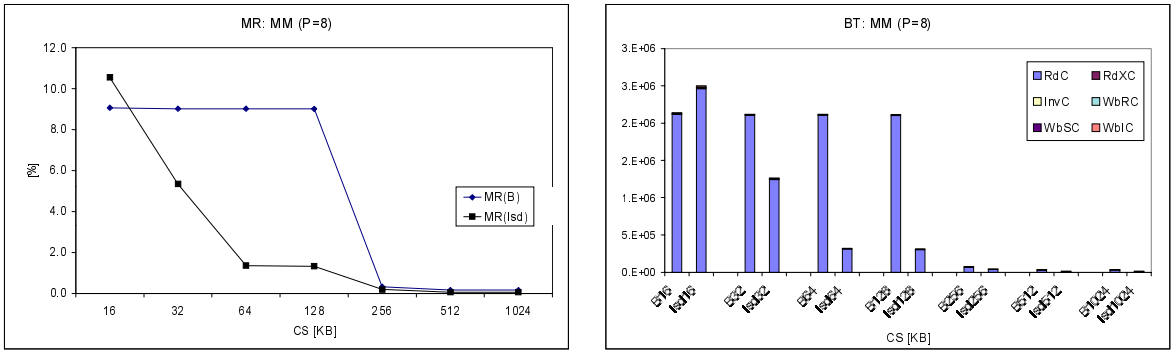
Sl. 5-11. PC: SU.

5.2.2 MM

U ovom odeljku prikazani su rezultati simulacione analize za aplikaciju MM koja je opisana u odeljku 4.4.4. U eksperimentima koji slede posmatra se množenje kvadratnih matrica dimenzija 128×128. U odeljku 5.2.2.1 prikazani su rezultati preliminarne analize bazirane na PRAM-MESI simulatoru, a u odeljku 5.2.2.2 prikazani su rezultati analize na bazi realnog simulatora MESI-SPLIT.

5.2.2.1 MM: PRAM-MESI

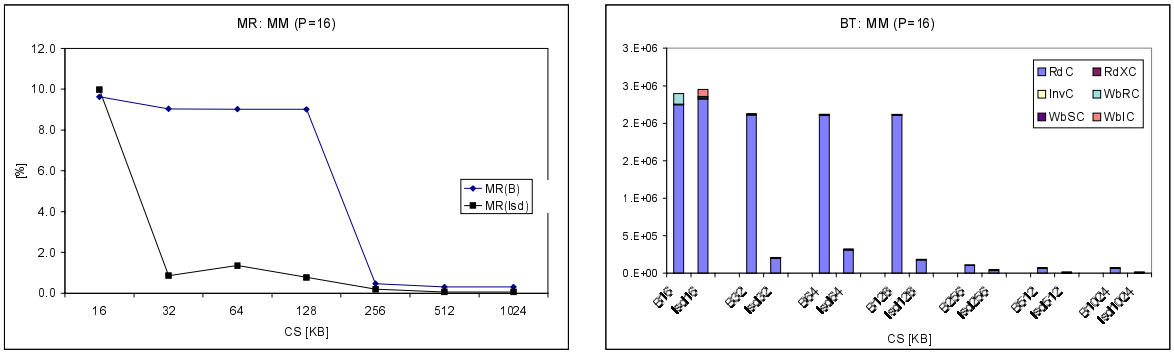
Na Sl. 5-12a i Sl. 5-12b prikazani su procenat promašaja u keš memoriji i saobraćaj na magistrali u zavisnosti od kapaciteta keš memorije, kada je broj procesora u sistemu P=8. Dobijeni rezultati pokazuju da u slučaju keš memorije kapaciteta 16KB mehanizam injektiranja dovodi do blagog povećanja procenta promašaja u keš memoriji, a takođe raste i ukupni saobraćaj na magistrali. Ovo je posledica kolizije podataka koji se injektiraju i podataka koji se trenutno obrađuju, u uslovima kada je kapacitet keš memorije izuzetno mali. Međutim, kod sistema sa keš memorijom kapaciteta 32KB i više, mehanizam injektiranja značajno doprinosi redukovanju procenta promašaja u keš memoriji, a takođe i ukupnog saobraćaja. Pri tom, redukcija procenta promašaja u keš memoriji je najizraženija za keš memorije kapaciteta 64KB i 128KB i iznosi oko 85%. Za keš memorije većeg kapaciteta (CacheSize>=256KB) svi podaci se nalaze u keš memoriji, pa se procenat promašaja redukuje od 42% do 64%. Velika poboljšanja za keš memorije kapaciteta 64KB i 128KB posledica su višestrukih injektiranja.



(a) (b)

Sl. 5-12. MM: MR i BT; P=8.

Na Sl. 5-13 prikazani su rezultati kada je broj procesora u sistemu P=16. Slično prethodnom primeru, ali u manjoj meri, mehanizam injektiranja povećava procenat promašaja u keš memoriji kapaciteta 16KB, dok u svim ostalim slučajevima doprinosi značajnom redukovanju procenta promašaja. Tako, procenat promašaja u keš memoriji se redukuje od 85% do 90%, za keš memorije kapaciteta od 32KB do 128KB, odnosno 79% za keš memorije kapaciteta 512KB i 1024KB. Ova poboljšanja su praćena redukovanjem broja RdC transakcija na magistrali.



(a) (b)

Sl. 5-13. MM: MR i BT; P=16.

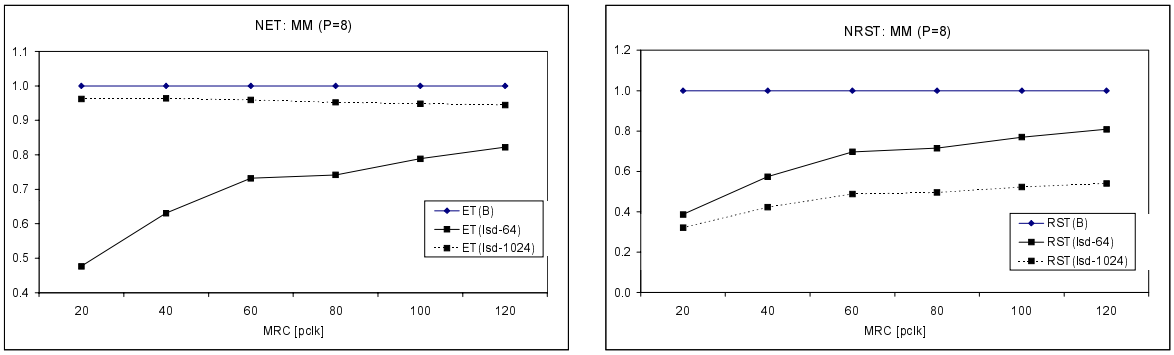
5.2.2.2 MM: MESI-SPLIT

Na Sl. 5-14a i Sl. 5-14b prikazani su normalizovano vreme izvršavanje NET i normalizovano vreme blokiranja prilikom čitanja NRST, redom, za sistem sa P=8 procesora. Posmatraju se polazna aplikacija i aplikacija sa podrškom injektiranju kada je kapacitet keš memorije CacheSize=64KB (Isd-64) i CacheSize=1024KB (Isd-1024). Mehanizam injektiranja redukuje vreme izvršavanja između 52% i 18%, odnosno između 4% i 5,5% u zavisnosti od kašnjenja u pristupu memoriji za sisteme sa keš memorijom kapaciteta 64KB, odnosno 1024KB, redom. Vreme blokiranja RST se redukuje od 61% do 19% za sistem sa keš memorijom kapaciteta 64KB, odnosno od 67% do 46% za sistem sa keš memorijom kapaciteta 1024KB.

Dobijeni rezultati pokazuju da je relativni procenat poboljšanja vremena izvršavanja na ovom primeru veći za sisteme sa malom keš memorijom. To se objašnjava činjenicom da mehanizam injektiranja omogućava višestruka injektiranja keš blokova koji se izbacuju iz keš memorije zbog malog kapaciteta. Kako se elementi matrice B samo čitaju (*read-only data*), mehanizam injektiranja omogućuje da se izbačeni keš blokovi matrice B ponovo injektiraju u keš memoriju.

Kriva ET(Isd-64) ukazuje na jedan interesantan fenomen kada povećavanje kašnjenja u pristupu memoriji smanjuje efikasnost mehanizma injektiranja. Međutim, kada je kapacitet keš memorije 1024KB (Isd-1024) efikasnost tehnike injektiranja blago raste sa porastom kašnjenja u pristupu memoriji. Sa druge strane, posmatrajući odnos krivih RST(Isd-64) i RST(Isd-1024), uočava se da je procenat redukcije kašnjenja veći kod sistema sa keš memorijom kapaciteta 1024KB. Ovaj, na izgled čudan fenomen, može se objasniti uvidom u pokazatelje ponašanja memorijskog podsistema koji inače ovde nisu posebno diskutovani. Naime, u sistemu sa keš memorijom kapaciteta 64KB i vremenom pristupa MRC=20pclk, vreme izvršavanja polazne aplikacije B-64 je u potpunosti određeno saobraćajem na magistrali, jer je procenat iskorišćenja magistrale 99,96%. Međutim, u sistemu sa MRC=100pclk iskorišćenje magistrale iznosi 58,57%, pa u ovom slučaju vreme izvršavanja nije u tolikoj meri određeno saobraćajem na magistrali. Mehanizam injektiranja značajno redukuje saobraćaj na magistrali, pa iskorišćenje magistrale u slučaju rešenja Isd-64 iznosi 45% kada je MRC=20pclk, odnosno 15% kada je MRC=100pclk. Sa druge strane, u sistemu sa keš memorijom kapaciteta 1024KB iskorišćenost magistrale kod polazne aplikacije (B-1024) u znatno manjoj meri zavisi od vremena pristupa memoriji i kreće se oko 11% kada je MRC=20pclk, odnosno oko 10% kada je MRC=100pclk; kod primera koji uključuje podršku injektiranju iskorišćenost magistrale je oko 4%. Ovaj rezultat pokazuje još jednu vrlo značajnu osobinu mehanizma injektiranja da u uslovima zagušenja magistrale redukovanjem saobraćaja na magistrali doprinosi značajnom povećavanju performanse.

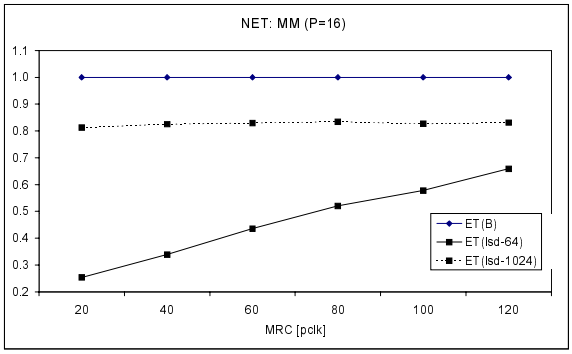
Ukupno, doprinos tehnike injektiranja poboljšanju performanse kod sistema sa keš memorijom kapaciteta 1024KB (Isd-1024) je procentualno manji jer nema višestrukog injektiranja keš blokova matrice B (jednom injektirani, podaci ostaju u keš memoriji); takođe, kako saobraćaj na magistrali ne predstavlja usko grlo u sistemu B-1024, redukovanje saobraćaja nema toliki uticaj na performanse.



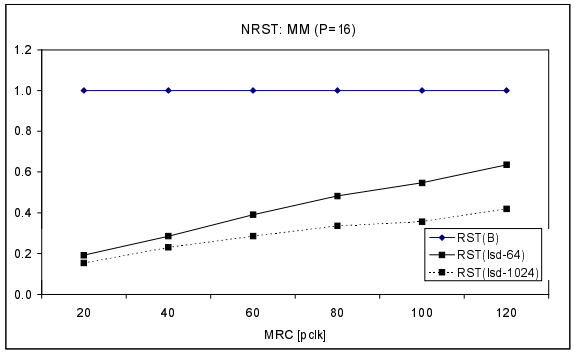
(a) (b)

Sl. 5-14. MM: NET i NRST; P=8.

Na Sl. 5-15 prikazani su rezultati kada je broj procesora u sistemu P=16. Mehanizam injektiranja redukuje vreme izvršavanja od 75% (MRC=20pclk) do 34% (MRC=100pclk), odnosno između 19% (MRC=20pclk) i 17% (MRC=100pclk) za sisteme sa keš memorijom kapaciteta 64KB, odnosno 1024KB, redom. Vreme blokiranja RST se redukuje od 80% (MRC=20pclk) do 36% (MRC=100pclk) za sistem sa keš memorijom kapaciteta 64KB, odnosno od 85% (MRC=20pclk) do 58% (MRC=20pclk) za sistem sa keš memorijom kapaciteta 1024KB. Poboljšanja performanse u odnosu na sistem sa P=8 procesora su posledica povećavanja stepena deljenja elemenata matrice B. Analiza trenda i međusobnog odnosa krivih data u slučaju sistema sa P=8 procesora važi i za sistem sa P=16 procesora.



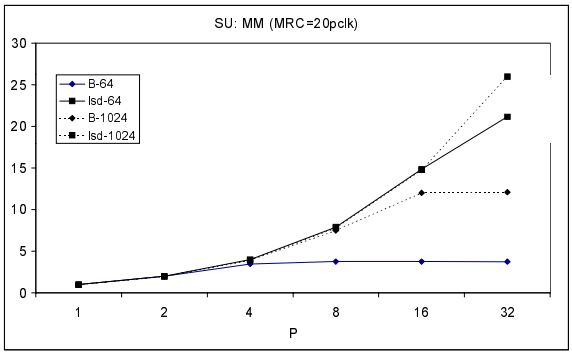
(a)



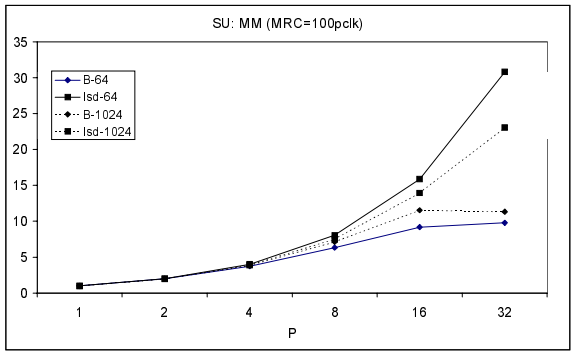
(b)

Sl. 5-15. MM: NET i NRST; P=16.

Na Sl. 5-16a i Sl. 5-16b prikazano je ubrzanje SU za sisteme za keš memorijom kapaciteta 64KB i 1024KB, kada je kašnjenje u pristupu memoriji MRC=20pclk i MRC=100pclk, redom. Na osnovu dobijenih rezultata mogu se dati sledeći zaključci:
(a) mehanizam injektiranja uvek doprinosi poboljšanju performanse bez obzira na kapacitet keš memorije i vreme pristupa memoriji.
(b) procenat poboljšanja raste sa porastom broja procesora. Tako, u sistemu sa keš memorijom kapaciteta 64KB i vremenom pristupa MRC=20pclk, mehanizam injektiranja poboljšava performanse za 0,64% u sistemu sa P=2 procesora, odnosno za 82,3% u sistemu sa P=32 procesora. U sistemu sa keš memorijom kapaciteta 1024KB poboljšanje performanse iznosi 0% za sistem sa P=2 procesora, odnosno 53,5% za sistem sa P=32 procesora.
(c) zbog pojave višestrukih injektiranja procenat poboljšanja je veći kod sistema sa malom keš memorijom. Tako, u sistemu sa P=32 procesora i vremenom pristupa memoriji MRC=20pclk, poboljšanje performanse iznosi 82,3% za sistem za keš memorijom kapaciteta 64KB, odnosno 53,5% za sistem za keš memorijom kapaciteta 1024KB.
(d) zbog zagušenja na magistrali u sistemu sa keš memorijom kapaciteta 64KB, mehanizam injektiranja je efikasniji za sisteme sa vremenom pristupa od MRC=20pclk nego u sistemima sa MRC=100pclk. Na primer, u sistemu sa P=32 procesora, poboljšanje performanse iznosi 82,3% kada je MRC=20pclk, odnosno 68,3% kada je MRC=100pclk.



(a)



(b)

Sl. 5-16. MM: SU.

5.2.3 Jacobi

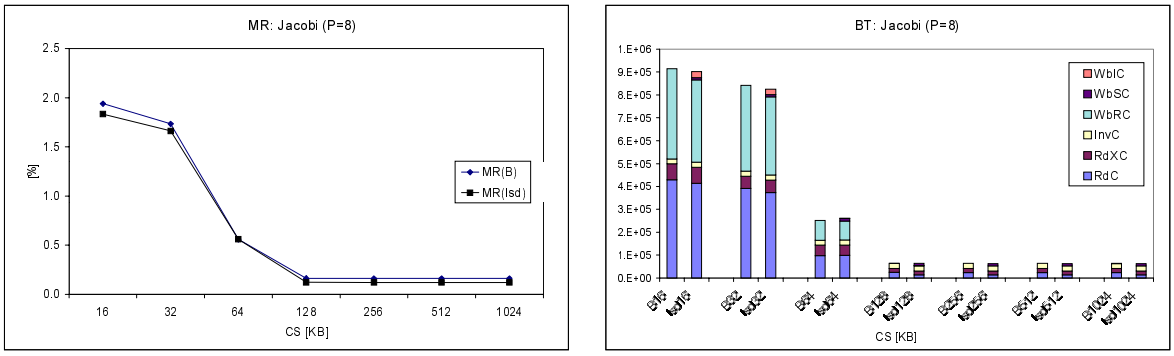
U ovom odeljku prikazani su rezultati simulacione analize za aplikaciju Jacobi koja je opisana u odeljku 4.4.5. Radno opterećenje koje se koristi u eksperimentima koji slede definisano je sledećim parametrima: N=256 i I=20. Preliminarna analiza efikasnosti mehanizma injektiranja

na bazi PRAM-MESI simulatora data je u odeljku 5.2.3.1, a simulaciona analiza bazirana na realnom simulatoru memorijskog podsistema MESI-SPLIT data je u odeljku 5.2.3.2.

5.2.3.1 Jacobi: PRAM-MESI

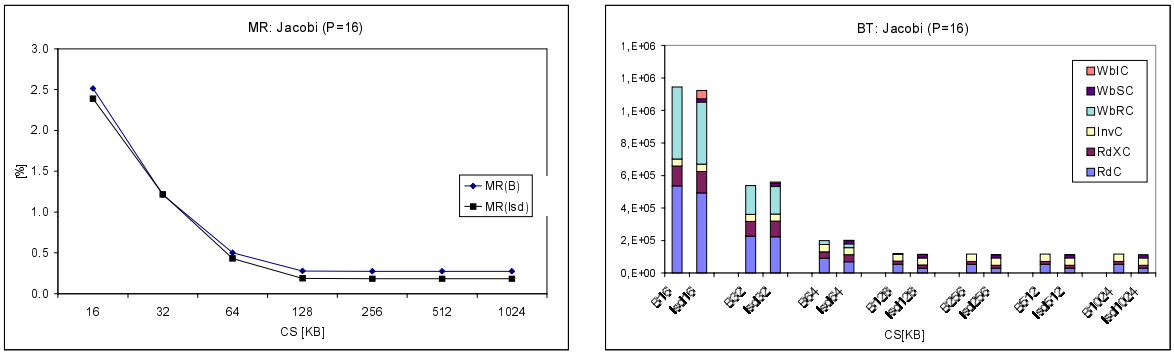
Na Sl. 5-17a i Sl. 5-17b prikazani su procenat promašaja u keš memoriji i saobraćaj na magistrali u zavisnosti od kapaciteta keš memorije, kada je broj procesora u sistemu P=8. Procenat promašaja u keš memoriji se redukuje od 5,5% za sistem sa keš memorijom kapaciteta 16KB do 25% za sistem sa keš memorijom kapaciteta 1024KB; za sistem sa keš memorijom kapaciteta 64KB mehanizam injektiranja ne redukuje procenat promašaja u keš memoriji. Pri tom, saobraćaj na magistrali ostaje približno isti u polaznom primeru i primeru sa injektiranjem, s tim da je razlika između broja RdC ciklusa približno jednaka broju softverski iniciranih ažuriranja SWbC, tj. važi sledeće SWbC(Isd)≈RdC(B)-RdC(Isd). Na ovaj način eliminišu se blokirajući promašaji u keš memoriji koji su posledica čitanja, a RdC transakcije se zamenjuju SWbC transakcijama na magistrali. Takođe, za sisteme sa keš memorijom malog kapaciteta važi da je ukupan broj izbačenih keš blokova u slučaju sa injektiranjem približno jednak broju izbačenih keš blokova u polaznom slučaju, tj. važi sledeća jednačina RWbC(B)≈RWbC(I)+IWbC(I).

Na Sl. 5-18 prikazani su rezultati kada je broj procesora u sistemu P=16. Mehanizam injektiranja redukuje procenat promašaja u keš memoriji od 5% do 33%, u zavisnosti od kapaciteta keš memorije. Zaključci u vezi saobraćaja dati za sistem sa P=8 procesora važe i u sistemu sa P=16 procesora.



(a) (b)

Sl. 5-17. Jacobi: MR i BT; P=8.



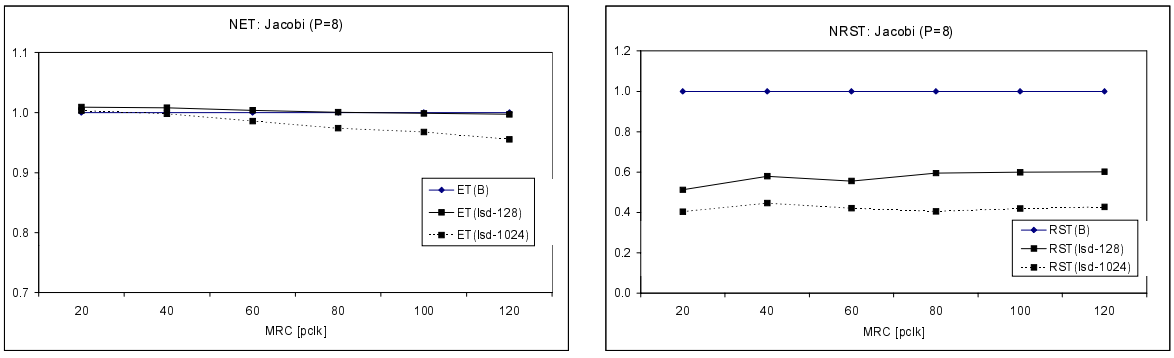
(a) (b)

Sl. 5-18. Jacobi: MR i BT; P=16.

5.2.3.2 Jacobi: MESI-SPLIT

Na Sl. 5-19a i Sl. 5-19b prikazani su normalizovano vreme izvršavanja NET i normalizovano vreme blokiranja prilikom čitanja NRST, redom, za sistem sa P=8 procesora. Posmatra se izvršavanje polazne verzije (B) i verzije koja uključuje podršku mehanizmu injektiranja za sistem sa keš memorijom kapaciteta CacheSize=128KB (Isd-128) i CacheSize=1024KB (Isd-1024).

Mehanizam injektiranja doprinosi značajnoj redukciji vremena blokiranja procesora RST, i to od 48% do 40% u zavisnosti od vremena pristupa memoriji tokom čitanja za sistem sa keš memorijom kapaciteta 128KB, odnosno oko 60% za sistem sa keš memorijom kapaciteta 1024KB. Međutim, i pored značajne redukcije vremena blokiranja, vreme izvršavanja se praktično ne redukuje ili se redukuje vrlo malo. Tako, vreme izvršavanja sa sistem sa keš memorijom kapaciteta 128KB se neznatno pogoršava za manje od 1% za male vrednosti kašnjenja u pristupu memoriji, odnosno neznatno poboljšava za veće vrednosti kašnjenja u pristupu memoriji. Ovo se može objasniti sledećim razlozima: (a) kako je procenat promašaja u keš memoriji vrlo mali (oko 0,16% za B-128) učešće vremena blokiranja u ukupnom vremenu izvršavanja nije značajno, pa i redukcija tog vremena nema značajniji uticaj na ukupno vreme izvršavanja, (b) mali kapacitet keš memorije rezultuje kolizijom podataka koji se injektiraju u keš memoriju i podataka koji se trenutno koriste, i (c) podrška injektiranju na strani procesora proizvođača podataka zahteva modifikaciju koda, koja nije beznačajna imajući u vidu mali broj linija kôda originalne aplikacije Jacobi. Kriva ET(Isd-1024) pokazuje da se u sistemu sa keš memorijom kapaciteta 1024KB vreme izvršavanja redukuje od 0% do 4,5% u zavisnosti od vremena pristupa memoriji tokom čitanja.

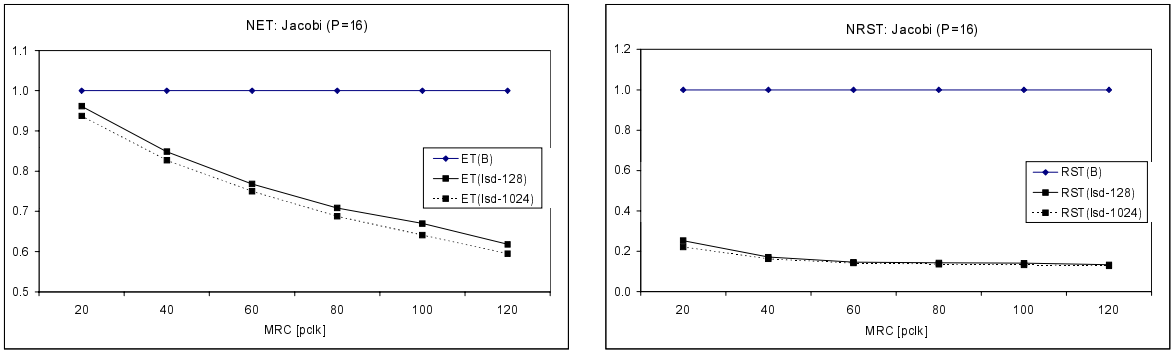


(a) (b)

Sl. 5-19. Jacobi: NET i NRST; P=8.

Na Sl. 5-20 prikazani su rezultati kada je broj procesora u sistemu P=16. Vreme blokiranja se redukuje od 74% do 86% za sistem sa keš memorijom kapaciteta 128KB, odnosno od 78% do 87% za sistem sa keš memorijom kapaciteta 1024KB. U ovom slučaju, mehanizam injektiranja značajnije redukuje vreme izvršavanja, i to od 4% do 38% za sistem sa keš memorijom kapaciteta 128KB, odnosno od 6% do 40% za sistem sa keš memorijom kapaciteta 1024KB.

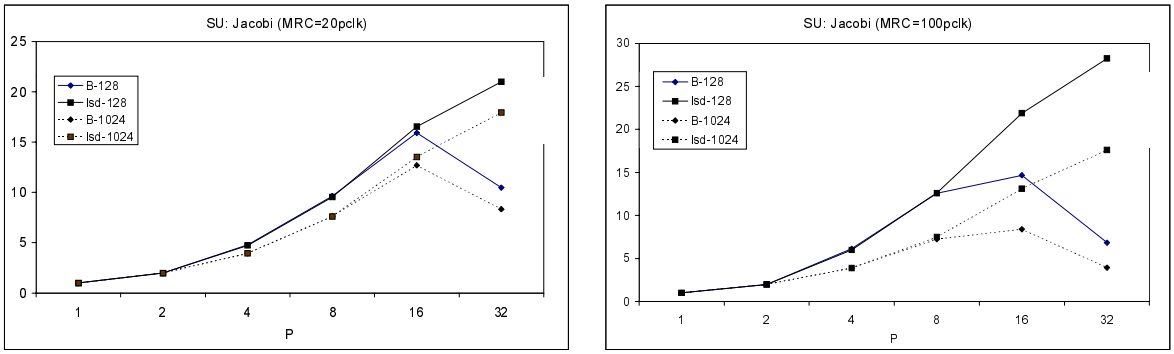
Dobijeni rezultati pokazuju da efikasnost mehanizma injektiranja raste sa porastom broja procesora čak i u ovom primeru kada postoji samo tip deljenja 1-Proizvođač-1-Potrošač, a injektiranje se vrši samo tokom softverski iniciranog ciklusa ažuriranja glavne memorije. Ovo povećanje efikasnosti mehanizma injektiranja je, pre svega, posledica povećavanja količine deljenih podataka na koje se može primeniti mehanizam injektiranja.



(a) (b)

Sl. 5-20. Jacobi: NET i NRST; P=16.

Na Sl. 5-21a i Sl. 5-21b prikazano je ubrzanje SU za sisteme za keš memorijom kapaciteta 128KB i 1024KB kada je kašnjenje u pristupu memoriji MRC=20pclk i MRC=100pclk, redom. Na osnovu dobijenih rezultata mogu se dati sledeći zaključci:
(a) mehanizam injektiranja neznatno degradira performanse kod sistema sa malim brojem procesora P=2 i 4. Tako u sistemu sa P=4 i CacheSize=128KB mehanizam injektiranja produžava vreme izvršavanja za nešto ispod 1% kada je MRC=20pclk, odnosno oko 2% kada je MRC=100pclk. Kada je broj procesora u sistemu P≥8, mehanizam injektiranja poboljšava performanse.
(b) procenat poboljšanja raste sa porastom broja procesora. Tako, u sistemu sa keš memorijom kapaciteta 1024KB i vremenom pristupa MRC=20pclk, mehanizam injektiranja poboljšava performanse za 6,2% u sistemu sa P=16 procesora, odnosno za 53,6% u sistemu sa P=32 procesora.
(c) procenat poboljšanja raste sa porastom kapaciteta keš memorije. Na primer, u sistemu sa P=32 procesora i vremenom pristupa memoriji MRC=20pclk, poboljšanje performanse iznosi 50% za sistem za keš memorijom kapaciteta 128KB, odnosno 53,6% za sistem za keš memorijom kapaciteta 1024KB.
(d) efikasnost mehanizma injektiranja raste sa porastom vremena pristupa memoriji. Na primer, u sistemu sa P=32 procesora i kapacitetom keš memorije od 128KB, poboljšanje performanse iznosi 50% kada je MRC=20pclk, odnosno 75,8% kada je MRC=100pclk.



(a) (b)

Sl. 5-21. Jacobi: SU.

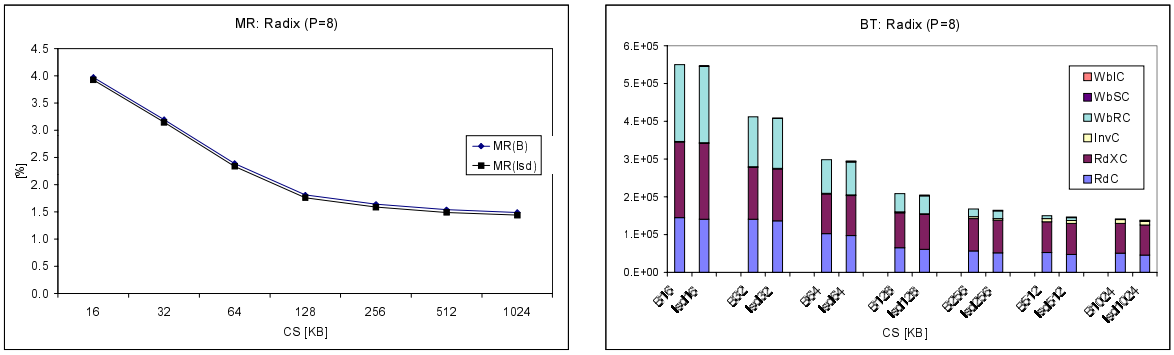
5.2.4 Radix

U ovom odeljku prikazani su rezultati simulacione analize za aplikaciju Radix koja je opisana u odeljku 4.4.6. Rezultati preliminarne analize efikasnosti mehanizma injektiranja koja se

bazira na simulatoru PRAM-MESI prikazani su u odeljku 5.2.4.1, a rezultati analize bazirane na simulatoru realnog memorijskog podsistema MESI-SPLIT u odeljku 5.2.4.2. Radno opterećenje je definisano sledećim parametrima: broj celobrojnih ključeva koji se sortiraju je 128K (131072), osnova je 256 (*radix*), a celobrojni ključevi uzimaju vrednosti iz skupa od 0 do 2^{31}.

5.2.4.1 Radix: PRAM-MESI

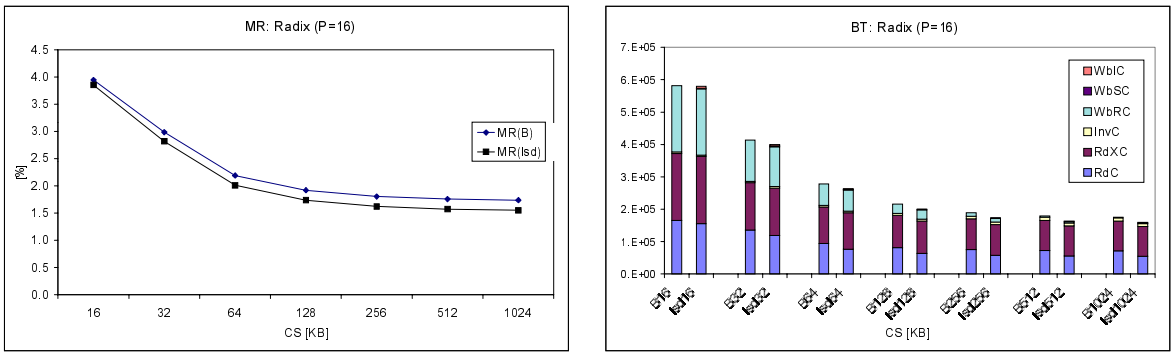
Na Sl. 5-22a i Sl. 5-22b prikazani su procenat promašaja u keš memoriji i saobraćaj na magistrali u zavisnosti od kapaciteta keš memorije, kada je broj procesora u sistemu P=8. Mehanizam injektiranja redukuje procenat promašaja u keš memoriji u relativnom iznosu od 1,2% do 3,4% u zavisnosti od kapaciteta keš memorije. Ovo smanjenje procenta promašaja u keš memoriji ostvareno je uz redukciju broja RdC ciklusa na magistrali. Pri tom, broj ostalih transakcija na magistrali ostaje nepromenjen. Treba napomenuti da je ovo poboljšanje ostvareno uz minimalno uvećanje dužine koda za manje od 0,2%.



(a) (b)

Sl. 5-22. Radix: MR i BT; P=8.

Na Sl. 5-23 prikazani su rezultati kada je broj procesora u sistemu P=16. Apsolutno smanjenje procenta promašaja u keš memoriji iznosi oko 0,185 i približno je konstantno za različite veličine keš memorije, izuzev za keš memoriju kapaciteta 16KB, kada je apsolutni iznos smanjenja procenta promašaja ispod 0,1. Relativno smanjivanje procenta promašaja u keš memoriji je od 2,3% do 10,6%, u zavisnosti od kapaciteta keš memorije. Razlika u stepenu poboljšanja između sistema sa P=8 i P=16 procesora je posledica povećavanja količine deljenih podataka na koje se primenjuje injektiranje i povećavanja broja procesora koji istovremeno čitaju deljene podatke, posebno nizove *rank* i *rank_me*.



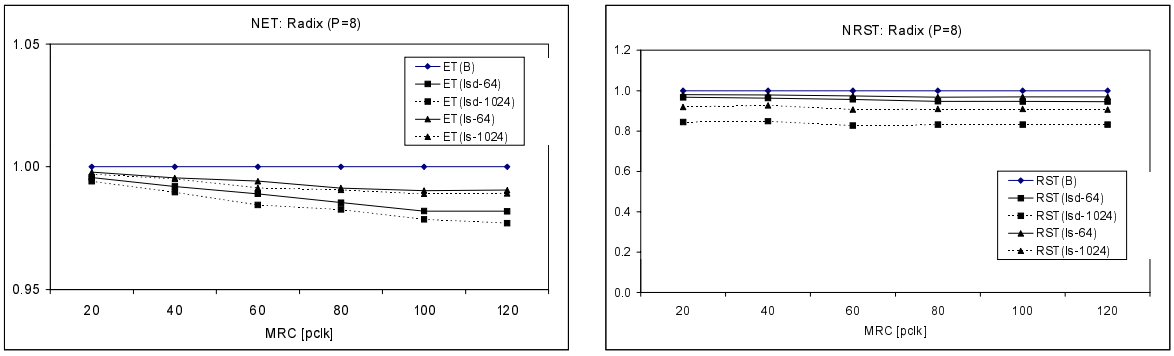
(a) (b)

Sl. 5-23. Radix: MR i BT; P=16.

5.2.4.2 Radix: MESI-SPLIT

U eksperimentalnoj analizi efikasnosti mehanizma injektiranja posmatraju se originalna aplikacija B, verzija koja podržava injektiranje samo za sinhronizacione varijable Is i verzija aplikacije koja podržava injektiranje kako sinhronizacionih varijabli, tako i pravih deljenih podataka Isd. Analiziraju se dva memorijska podsistema sa keš memorijom kapaciteta CacheSize=64KB (B-64, Is-64, Isd-64) i CacheSize=1024KB (B-1024, Is-1024, Isd-1024).

Na Sl. 5-24a i Sl. 5-24b prikazani su normalizovano vreme izvršavanja NET i normalizovano vreme blokiranja prilikom čitanja NRST u zavisnosti od vremena pristupa memoriji prilikom čitanja MRC, redom, za sistem sa P=8 procesora. Za sistem sa keš memorijom kapaciteta 64KB, rešenje Is redukuje vreme blokiranja RST(Is-64) od 2% (MRC=20pclk) do 3% (MRC=120pclk), a rešenje Isd (RST(Isd-64)) od 3,3% (MRC=20pclk) do 5,5% (MRC=120pclk). Rešenje Is redukuje vreme izvršavanja (ET(Is-64)) od 0,3% (MRC=20pclk) do 1% (MRC=120pclk), a rešenje Isd (ET(Isd-64)) od 0,5% (MRC=20pclk) do 1,8% (MRC=120pclk). Za sistem sa keš memorijom kapaciteta 1024KB, rešenje Is redukuje vreme blokiranja RST(Is-1024) od 8% (MRC=20pclk) do 9,3% (MRC=120pclk), a rešenje Isd (RST(Isd-1024)) od 15,5% (MRC=20pclk) do 16,8% (MRC=120pclk). Rešenje Is redukuje vreme izvršavanja (ET(Is-1024)) od 0,3% (MRC=20pclk) do 1,1% (MRC=120pclk), a rešenje Isd (ET(Isd-1024)) od 0,6% (MRC=20pclk) do 3,2% (MRC=120pclk). Relativno mali procenat poboljšanja je posledica malog procenta pravih deljenih podataka na koje se primenjuje mehanizam injektiranja. Naime, većina promašaja u keš memoriji se dešava u trećoj fazi svake iteracije kada se na osnovu izračunatih lokalnih histograma izračunava nova pozicija svakog ključa u nizu. U ovoj fazi, kako je već objašnjeno, primena injektiranja ima smisla samo kada veličina keš bloka odgovara veličini jedne reči. Inače, mehanizam injektiranja može dovesti do pogoršanja performanse usled nepravog deljenja podataka (*false sharing*).

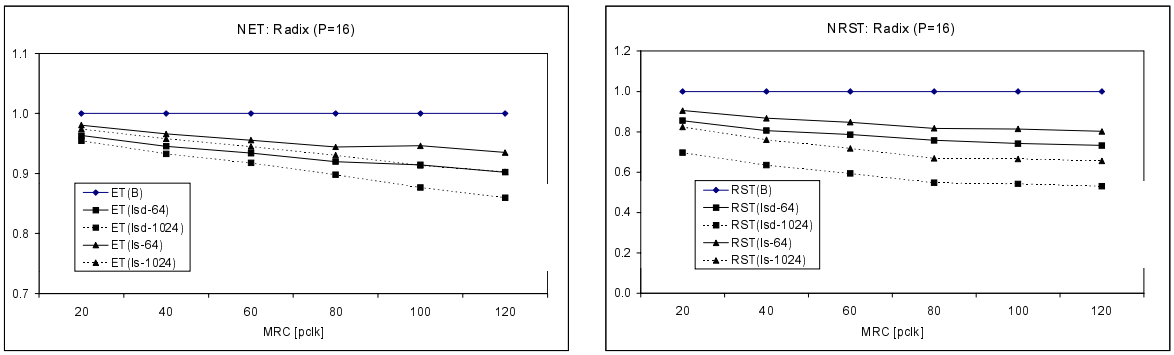


(a) (b)

Sl. 5-24. Radix: NET i NRST; P=8.

Na Sl. 5-25 prikazani su rezultati kada je broj procesora u sistemu P=16. Za sistem sa keš memorijom kapaciteta 64KB, rešenje Is redukuje vreme blokiranja RST(Is-64) od 9,5% (MRC=20pclk) do 20% (MRC=120pclk), a rešenje Isd (RST(Isd-64)) od 14,5% (MRC=20pclk) do 27% (MRC=120pclk). Rešenje Is redukuje vreme izvršavanja (ET(Is-64)) od 2% (MRC=20pclk) do 6,5% (MRC=120pclk), a rešenje Isd (ET(Isd-64)) od 3,7% (MRC=20pclk) do 10% (MRC=120pclk). Za sistem sa keš memorijom kapaciteta 1024KB, rešenje Is redukuje vreme blokiranja RST(Is-1024) od 17,5% (MRC=20pclk) do 34,5% (MRC=120pclk), a rešenje Isd (RST(Isd-1024)) od 30% (MRC=20pclk) do 47% (MRC=120pclk). Rešenje Is redukuje vreme izvršavanja ET(Is-1024) od 2,5% (MRC=20pclk) do 10% (MRC=120pclk), a rešenje Isd (ET(Isd-1024)) od 4,5% (MRC=20pclk) do 14%

(MRC=120pclk). U poređenju sa rezultatima dobijenim za sistem sa P=8 procesora primećuje se značajan porast efikasnosti tehnike injektiranja. Ovo poboljšanje je, pre svega, posledica povećavanja procenta deljenih podataka, a takođe i povećavanja broja procesora koji dele iste podatke.



(a) (b)

Sl. 5-25. Radix: NET i NRST; P=16.

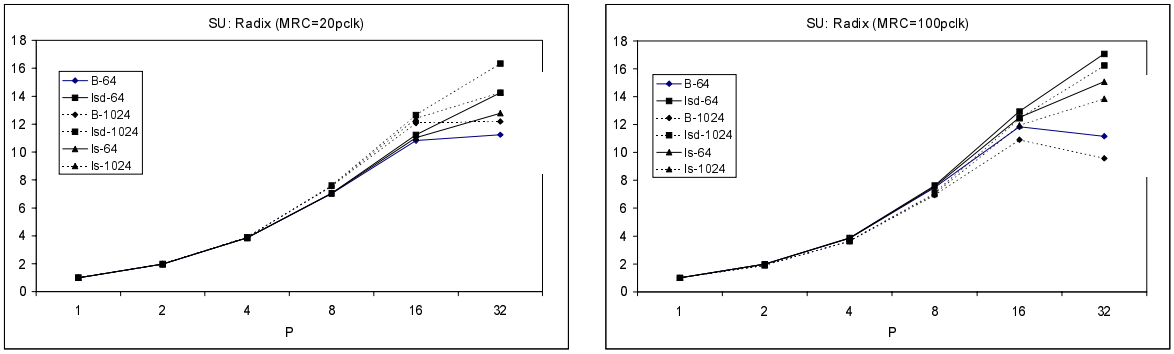
Na Sl. 5-26 prikazano je ubrzanje SU za posmatrane verzije aplikacije Radix, (B, Is i Isd) u sistemu sa keš memorijom kapaciteta 64KB i 1024KB i vremenom pristupa memoriji prilikom čitanja MRC=20pclk i MRC=100pclk. Na osnovu izgleda krivih mogu se izvesti sledeći zaključci:

(a) rešenja Is i Isd pokazuju poboljšanje performanse za sve analizirane parametre memorijskog podsistema.

(b) procenat poboljšanja performanse za rešenja Is i Isd raste sa brojem procesora; pri tom rešenje Isd uvek pokazuje bolje rezultate. Tako, u slučaju memorijskog sistema sa CacheSize=1024KB i MRC=20pclk, rešenje Is poboljšava performanse za 2,6% kada je P=16, odnosno za 14,5% kada je P=32, a rešenje Isd poboljšava performanse za 4,5% kada je P=16, odnosno za 25,4% kada je P=32.

(c) procenat poboljšanja performanse za rešenja Is i Isd raste sa porastom kapaciteta keš memorije. Na primer, u sistemu sa P=32 procesora i vremenom pristupa memoriji MRC=20pclk, poboljšanje performanse za rešenje Is iznosi 12% za sistem za keš memorijom kapaciteta 64KB, odnosno 14,5% za sistem za keš memorijom kapaciteta 1024KB, a poboljšanje performanse za rešenje Isd iznosi 21,1% za sistem sa 64KB, odnosno 25,4% za sistem sa 1024KB.

(d) efikasnost mehanizma injektiranja raste sa porastom vremena pristupa memoriji. Na primer, u sistemu sa P=32 procesora i kapacitetom keš memorije od 1024KB, poboljšanje performanse za rešenje Is iznosi 14,5% kada je MRC=20pclk, odnosno 30,9% kada je MRC=100pclk, a za rešenje Isd iznosi 25,4% kada je MRC=20pclk, odnosno 41,1% kada je MRC=100pclk.



(a) (b)

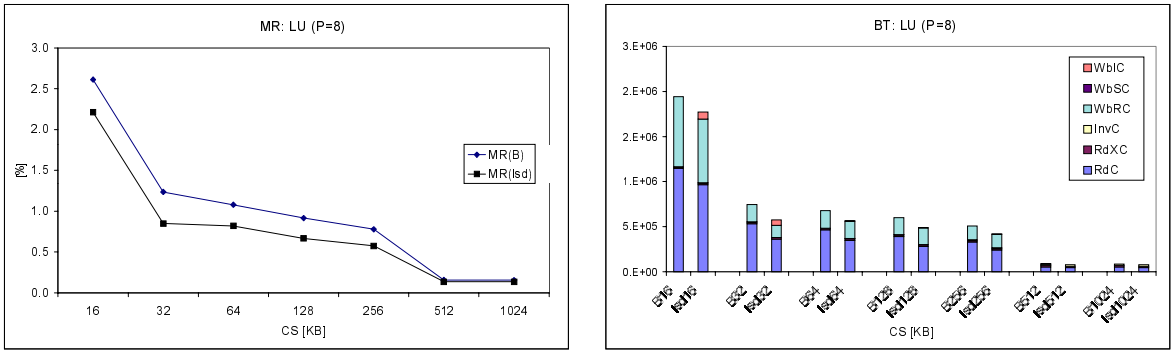
Sl. 5-26. Radix: SU.

5.2.5 LU

U ovom odeljku prikazani su rezultati simulacione analize za aplikaciju LU koja je opisana u odeljku 4.4.7. U eksperimentima koji slede posmatra se matrica dimenzija 256×256, dok je veličina bloka b=8. U odeljku 5.2.5.1 prikazani su rezultati preliminarne analize efikasnosti mehanizma injektiranja bazirane na PRAM-MESI simulatoru. Posmatra se originalna aplikacija B i aplikacija koja podržava injektiranje za sinhronizacione varijable i deljene podatke Isd. U odeljku 5.2.5.2 dati su rezultati simulacione analize bazirane na MESI-SPLIT simulatoru; posmatraju se polazna verzija programa B, verzija koja uključuje podršku injektiranju samo za sinhronizacione varijable Is, i verzija koja uključuje podršku injektiranju kako sinhronizacionih varijabli tako i prvih podataka Isd.

5.2.5.1 LU: PRAM-MESI

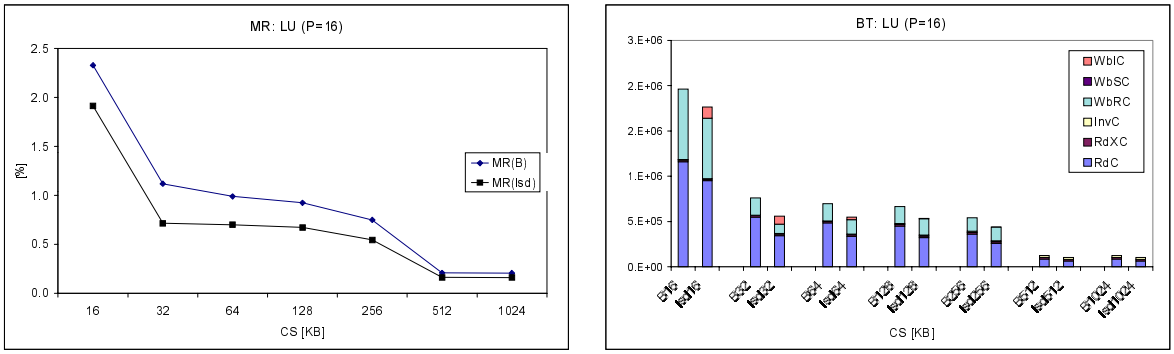
Na Sl. 5-27a i Sl. 5-27b prikazani su procenat promašaja u keš memoriji i saobraćaj na magistrali, redom, u zavisnosti od kapaciteta keš memorije kada je broj procesora u sistemu P=8. Mehanizam injektiranja redukuje procenat promašaja u keš memoriji od 13,5% od 31% u zavisnosti od kapaciteta keš memorije. Pri tom, u ovom slučaju ne važi pravilo da efikasnost mehanizma injektiranja raste sa porastom kapaciteta keš memorije. Naime, kôd za inicijalizaciju tabela injektiranja je napisan tako da podržava višestruka injektiranja, što posebno utiče na povećavanje efikasnosti mehanizma injektiranja u slučaju keš memorija malog kapaciteta. Tako, mehanizam injektiranja smanjuje procenat promašaja u keš memoriji za apsolutni iznos (MR(B)-MR(Isd)) od 0,4, ili za preko 15% u relativnom iznosu za keš memoriju kapaciteta 16KB; relativno poboljšanje za keš memorije kapaciteta od 32KB do 256KB je u opsegu od 31% do 26%, a za keš memoriju kapaciteta 512KB 14%. Mehanizam injektiranja redukuje broj RdC transakcija na magistrali, dok broj zahteva za invalidacijom InvC i ekskluzivnim čitanjem RdXC ostaje približno isti. Kako se u rešenju Isd ne koristi injektiranje tokom softverski iniciranih ciklusa ažuriranja, to je WbSC=0. Za sisteme sa keš memorijom malog kapaciteta važi da je ukupan broj izbačenih modifikovanih keš blokova u slučaju sa injektiranjem približno jednak broju izbačenih modifikovanih keš blokova u polaznom slučaju, tj. važi sledeća jednačina RWbC(B)≈RWbC(Isd)+IWbC(Isd)}.



(a) (b)

Sl. 5-27. LU: MR i BT; P=8.

Na Sl. 5-28 prikazani su rezultati kada je broj procesora u sistemu P=16. Procenat poboljšanja je veći i iznosi od 18% do 36% u zavisnosti od kapaciteta keš memorije. Najveća poboljšanja se dobijaju za sistem sa keš memorijom kapaciteta 32KB i 64KB i iznose 36% i 29%, redom. Napomene u vezi saobraćaja na magistrali date na primeru sistema sa P=8 procesora važe i u ovom eksperimentu. Procenat povećanja broja instrukcija dodatih da podrže injektiranje je oko 1.5%. Već je napomenuto da se taj broj instrukcija može dalje redukovati realokacijom, tako da blokovi zauzimaju kontinualni adresni prostor.



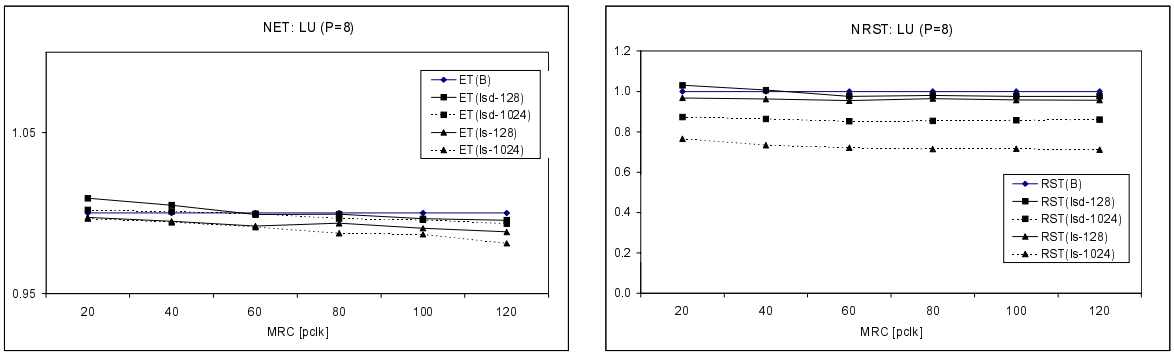
(a) (b)

Sl. 5-28. LU: MR i BT; P=16.

5.2.5.2 LU: MESI-SPLIT

Na Sl. 5-29a i Sl. 5-29b prikazani su normalizovano vreme izvršavanja NET i normalizovano vreme blokiranja prilikom čitanja NRST, redom, za sistem sa P=8 procesora. Posmatra se izvršavanje polazne verzije aplikacije (B), verzije koja uključuje podršku mehanizmu injektiranja samo za sinhronizacione varijable (Is) i verzije koja uključuje podršku injektiranju za sinhronizacione varijable i deljene podatke (Isd). Posmatraju se sistemi sa keš memorijom kapaciteta 128KB (B-128, Is-128 i Isd-128) i 1024KB (B-1024, Is-1024 i Isd-1024). Za sistem sa keš memorijom kapaciteta 128KB rešenje Is redukuje vreme blokiranja RST(Is-128) za 2,2% do 4,3% u zavisnosti od vremena pristupa memoriji prilikom čitanja, a rešenje Isd od 2,4% za MRC=60pclk do 3,5% za MRC=120pclk (RST(Isd-128)); rešenje Isd za MRC=20pclk i 40pclk neznatno povećava vreme blokiranja, usled kontencije na internim resursima zbog izvršavanja dodatnih instrukcija za podršku injektiranju. Ovakvi rezultati utiču i na vreme izvršavanja, tako da rešenje Is redukuje vreme izvršavanja od 0,3% (MRC=20pclk) od 1,2% (MRC=120pclk). Rešenje Isd praktično ne redukuje vreme izvršavanja, odnosno za sisteme kada je MRC=20pclk i 40pclk dolazi do povećavanja vremena izvršavanja; pri tom, vreme izvršavanja se povećava za manje od 1%. Za sistem sa keš memorijom kapaciteta

1024KB rešenje Is redukuje vreme blokiranja od 23% (MRC=20pclk) do 29% (MRC=120pclk), a rešenje Isd oko 13%. Vreme izvršavanja se redukuje od 0,4% (MRC=20pclk) do 1,9% (MRC=120pclk) za rešenje Is, odnosno od 0% do 0,7% za rešenje Isd.

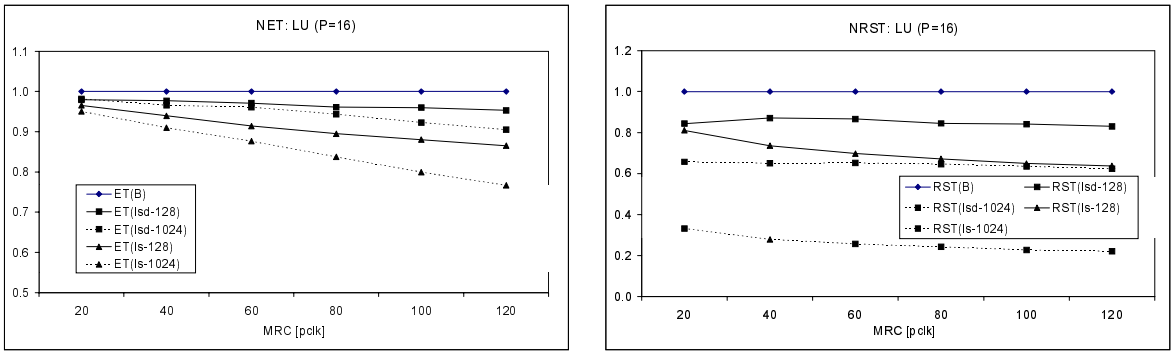


(a) (b)

Sl. 5-29. LU: NET i NRST; P=8.

Na Sl. 5-30 prikazani su rezultati simulacione analize kada je broj procesora u sistemu P=16. Za sistem sa keš memorijom kapaciteta 128KB rešenje Is redukuje vreme blokiranja od 19% (MRC=20pclk) do 36% (MRC=120pclk), a rešenje Isd od 16% (MRC=20pclk) do 18% (MRC=120pclk). Rešenje Is redukuje vreme izvršavanja od 3,5% (MRC=20pclk) do 13,5% (MRC=120pclk), a rešenje Isd od 2% (MRC=20pclk) do 4,7% (MRC=120pclk). Za sistem sa keš memorijom kapaciteta 1024KB rešenje Is redukuje vreme blokiranja od 67% (MRC=20pclk) do 78% (MRC=120pclk), a rešenje Isd od 34% (MRC=20pclk) do 38% (MRC=120pclk). Rešenje Is redukuje vreme izvršavanja od 5% (MRC=20pclk) do 21% (MRC=120pclk), a rešenje Isd od 2% (MRC=20pclk) do 10% (MRC=120pclk).

Dobijeni rezultati pokazuju da rešenje Is koje je sasvim jednostavno za implementaciju nadmašuje performanse rešenja Isd, što je sasvim neočekivan rezultat. Parametri koje se ovde ne razmatraju, kao što su procenat promašaja u keš memoriji (kod MESI-SPLIT simulatora) i saobraćaj na magistrali, pokazuju da rešenje Isd ipak smanjuje procenat promašaja u keš memoriji i saobraćaj na magistrali (Sl. 5-31) u odnosu na rešenje Is. Međutim, osnovni razlog zbog koga se taj potencijal nije pretvorio u dobitak u performansi je pre svega negativan uticaj na kontenciju na internim resursima keš kontrolera usled izvršavanja umetnutih instrukcija za podršku mehanizmu injektiranja.



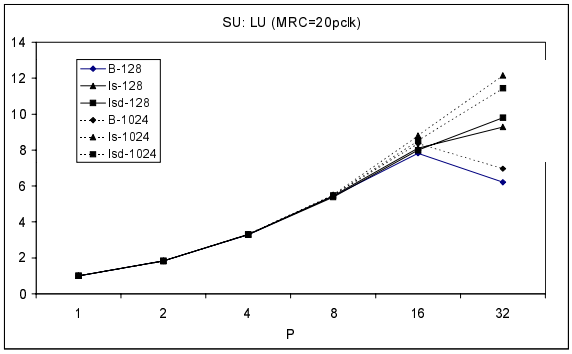
(a) (b)

Sl. 5-30. LU: NET i NRST; P=16.

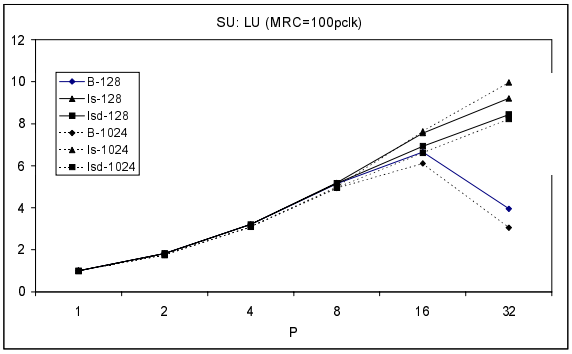
BusReq	RdC	RdXC	InvC	WbRC	WbSC	WbIC
B	463840	16475	7390	187699	0	0
Is	430813	16479	6989	187349	0	7
Isd	318818	16480	7136	183550	0	3808

Sl. 5-31. Saobraćaj na magistrali za različite verzije aplikacije LU; CacheSize=128KB, P=16.

Na Sl. 5-32a i Sl. 5-32b prikazano je ubrzanje SU za rešenja B, Is i Isd, kada je kašnjenje u pristupu memoriji usled čitanja MRC=20pclk i MRC=100pclk, redom. Na osnovu dobijenih rezultata mogu se izvesti sledeći zaključci:
(a) bez obzira na kapacitet keš memorije i vreme pristupa memoriji pokazuje se da usled zagušenja magistrale rešenje B pokazuje degradaciju performanse u sistemima sa P=32 procesora (*trashing*).
(b) rešenje Is poboljšava performanse, nezavisno od parametara memorijskog podsistema.
(c) rešenje Isd neznatno degradira vreme izvršavanja u sistemu sa P=2, 4 i 8 procesora kada je MRC=20pclk, i u sistemu sa P=2 i 4 procesora kada je MRC=100pclk. Međutim, degradacija performanse je zanemarljiva i kreće se u opsegu od 0,1% do 0,3%, zavisno od parametara memorijskog podsistema. U svim ostalim slučajevima rešenje Isd povećava performanse.
(d) rešenja Is i Isd pokazuju trend rasta performanse sa porastom broja procesora za sve parametre memorijskog podsistema. Pri tom, taj porast je veći u sistemima sa keš memorijom većeg kapaciteta i u sistemima sa većim kašnjenjem u pristupu memoriji prilikom čitanja. Tako, za sistem sa 128KB keš memorije SU(Is-128, P=16)=8,11, a SU(Is-128, P=32)=9,29, dok je za sistem sa 1024KB SU(Is-128, P=16)=8,8, a SU(Is-1024, P=32)=12,16.
(e) efikasnost mehanizma injektiranja raste sa povećavanjem vremena pristupa memoriji prilikom čitanja. Tako, rešenje Is poboljšava performanse u sistemu sa P=32 procesora i keš memorijom kapaciteta 1024KB za 42,7%, kada je MRC=20pclk, odnosno za 69,4% kada je MRC=100pclk; pod istim uslovima, rešenje Isd poboljšava performanse za 39,1% kada je MRC=20pclk, odnosno za 62,9% kada je MRC=100pclk.
(f) povećavanje kapaciteta keš memorije povećava efikasnost tehnike injektiranja. Tako, u sistemu sa P=32 procesora i vremenom pristupa memoriji prilikom čitanja MRC=20pclk rešenje Is poboljšava performanse za 33,1% u sistemu sa keš memorijom kapaciteta 128KB, odnosno 42,7% u sistemu sa keš memorijom kapaciteta 1024KB. Pod istim uslovima rešenje Isd poboljšava performanse za 36,6% kada je CacheSize=128KB, odnosno za 39,1% kada je CacheSize=1024KB.
(g) povećavanje broja procesora povećava efikasnost tehnike injektiranja. Tako, rešenje Is poboljšava performanse u sistemu sa 1024KB i MRC=20 pclk za 4,9% kada je broj procesora u sistemu P=16, odnosno za 42,7% u sistemu sa P=32 procesora. Pod istim uslovima rešenje Isd povećava performanse za 1,8% u sistemu sa P=16 procesora, odnosno za 39,1% u sistemu sa P=32 procesora.
(h) rešenje Isd nadmašuje performanse rešenja Is u sistemu sa P=32, MRC=20pclk, a CacheSize=128KB.



(a)



(b)

Sl. 5-32. LU: SU.

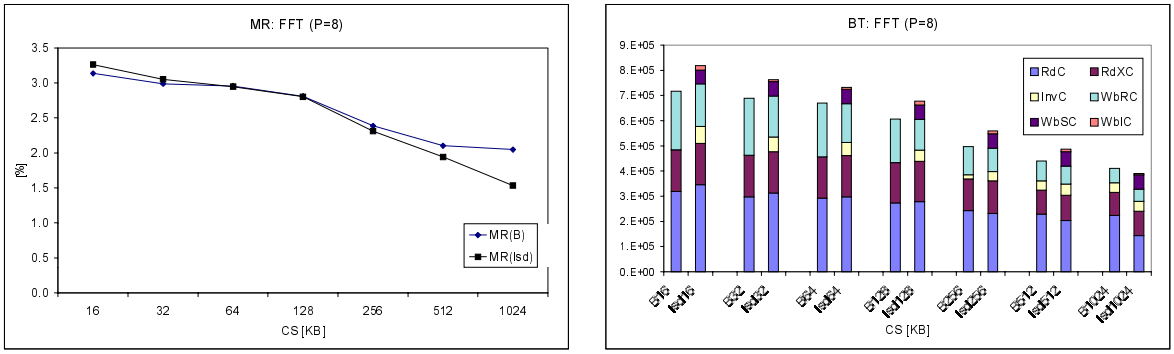
5.2.6 FFT

U ovom odeljku prikazani su rezultati simulacione analize za aplikaciju FFT koja je opisana u odeljku 4.4.8. U odeljku 5.2.6.1 prikazani su rezultati preliminarne simulacione analize bazirane na PRAM-MESI simulatoru. Posmatraju se originalna aplikacija B i aplikacija koja podržava injektiranje za sinhronizacione varijable i deljene podatke Isd. U odeljku 5.2.6.2 dati su rezultati simulacione analize bazirane na MESI-SPLIT simulatoru. Posmatraju se polazna verzija programa B, verzija koja uključuje podršku injektiranju samo za sinhronizacione varijable Is, i verzija koja uključuje podršku injektiranju kako sinhronizacionih varijabli tako i pravih deljenih podataka Isd. U svim eksperimentima posmatra se Furijeova analiza 2^{16} kompleksnih tačaka.

5.2.6.1 FFT: PRAM-MESI

Na Sl. 5-33 prikazani su procenat promašaja u keš memoriji i saobraćaj na magistrali u zavisnosti od kapaciteta keš memorije, kada je broj procesora u sistemu P=8. Rezultati pokazuju da za keš memorije malog kapaciteta od 16KB i 32KB mehanizam injektiranja povećava procenat promašaja u keš memoriji. Pri tom, relativni iznos pogoršanja iznosi oko 4% za sistem sa keš memorijom kapaciteta 16KB (MR(Isd)). Za multiprocesore sa keš memorijama većeg kapaciteta relativni iznos smanjenja procenta promašaja u keš memoriji je od 0,3% za keš memoriju kapaciteta 64KB do 25% za keš memoriju kapaciteta 1024KB.

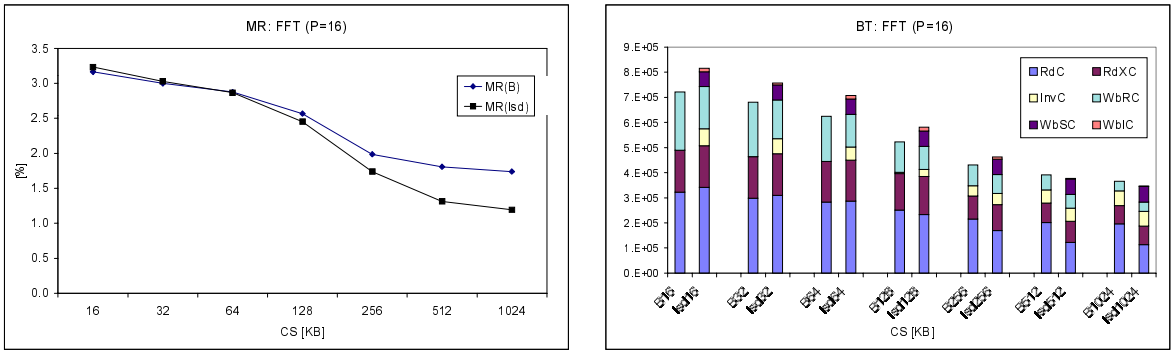
Rezultati sa Sl. 5-33b prikazuju saobraćaj na magistrali u zavisnosti od kapaciteta keš memorije. Kod keš memorija malog kapaciteta mehanizam injektiranja dovodi do povećavanja saobraćaja na magistrali. Povećanje saobraćaja je posledica nešto značajnijeg povećanja broja RdC transakcija, pojave invalidacionih transakcija InvC i softverski iniciranih ažuriranja SWbC. Međutim, sa povećavanjem kapaciteta keš memorije porast saobraćaja usled injektiranja postaje sve manji. Kod keš memorije kapaciteta 1024KB približno važi sledeća jednakost: SWbC(Isd)≈RdC(B)-RdC(Isd), što znači da su RdC transakcije koje blokiraju izvršavanje programske niti zamenjene transakcijama softverski iniciranog ažuriranja SWbC. Rezultati pokazuju da broj modifikovanih keš blokova koji se izbacuje iz keš memorije usled injektiranja nije veliki, tj. važi sledeća približna jednačina: RWbC(B) ≈RWbC(Isd)+IWbC(Isd).



(a) (b)

Sl. 5-33. FFT: MR i BT; P=8.

Na Sl. 5-34 prikazani su rezultati eksperimenata za slučaj kada je broj procesora u sistemu P=16. U slučaju keš memorije kapaciteta 16KB i 32KB, procenat promašaja u keš memoriji je veći nego za polazno rešenje B. Za keš memorije većeg kapaciteta procenat promašaja se redukuje od 0,3% za keš memoriju kapaciteta 64KB do 31% za keš memoriju kapaciteta 1024KB. Povećavanje broja procesora ne menja značajnije odnos saobraćaja na magistrali za posmatrana rešenja, pa važe isti zaključci izneti u diskusiji o saobraćaju kada je broj procesora u sistemu P=8.



(a) (b)

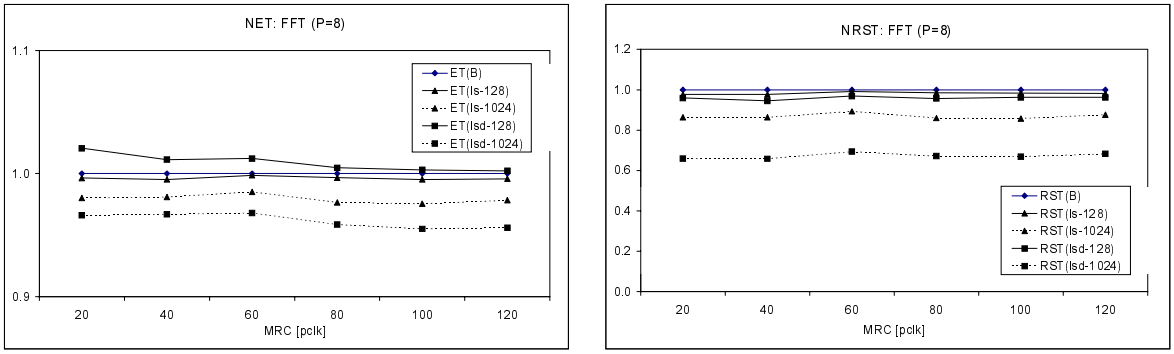
Sl. 5-34. FFT: MR i BT; P=16.

5.2.6.2 FFT: MESI-SPLIT

Na Sl. 5-35a i Sl. 5-35b prikazani su normalizovano vreme izvršavanja NET i normalizovano vreme blokiranja prilikom čitanja NRST, redom, za sistem sa P=8 procesora. Posmatra se izvršavanje polazne verzije aplikacije FFT (B), verzije koja uključuje podršku mehanizmu injektiranja samo za sinhronizacione varijable (Is) i verzije koja uključuje podršku injektiranju za sinhronizacione varijable i deljene podatke (Isd). Posmatraju se sistemi sa keš memorijom kapaciteta 128KB (B-128, Is-128 i Isd-128) i 1024KB (B-1024, Is-1024 i Is-1024).

Za sistem sa keš memorijom kapaciteta 128KB rešenje Is praktično ne doprinosi smanjivanju vremena izvršavanja jer je poboljšanje od 0,2% do 0,5% u zavisnosti od kašnjenja u pristupu memoriji tokom čitanja (ET(Is-128)), dok rešenje Isd čak doprinosi povećavanju vremena izvršavanja od maksimalnih 2% do 0,02% (ET(Isd-128)). Sa druge strane, pokazuje se da mehanizam injektiranja doprinosi smanjivanju vremena blokiranja od 1,5% do 2,5% za rešenje Is (RST(Is-128)), odnosno od 4% do 4,5% za rešenje (RST(Isd-128)). Međutim, i pored minimalnog smanjivanja vremena blokiranja to nije doprinelo smanjivanju vremena izvršavanja. Osnovni razlozi za to su sledeći (a) kompleksnost i dužina koda koji se umeće za

podršku injektiranju i (b) kolizija u keš memoriji između injektiranih podataka i podataka koji se trenutno obrađuju. Za sistem sa keš memorijom kapaciteta 1024KB mehanizam injektiranja redukuje vreme blokiranja RST za oko 14% za rešenje Is (RST(Is-1024)), odnosno oko 35% za rešenje Isd (RST(Isd-1024)). Značajnije redukovanje vremena blokiranja se odražava i na vreme izvršavanja: rešenje Is redukuje vreme izvršavanja od 2% do 2,5% (ET(Is-1024)), a rešenje Isd od 3,5% do 4,5% (ET(Isd-1024)). Kod ove aplikacije vreme blokiranja RST i vreme izvršavanja ne zavise značajno od vremena pristupa memoriji prilikom čitanja MRC.

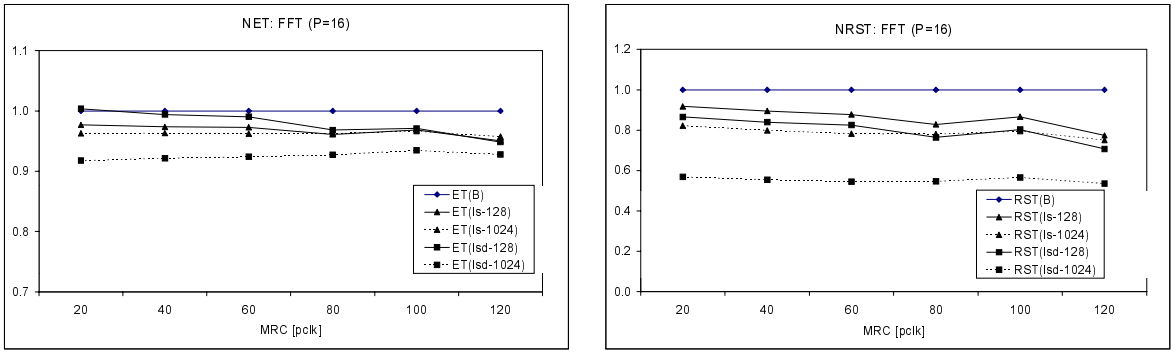


(a) (b)

Sl. 5-35. FFT: NET i NRST; P=8.

Na Sl. 5-36 prikazani su rezultati kada je broj procesora u sistemu P=16. Za sistem za keš memorijom kapaciteta 128KB rešenje Is redukuje vreme blokiranja od 8% do 22% u zavisnosti od vremena pristupa memoriji (RST(Is-128)), a rešenje Isd od 13% do 29% (RST(Isd-128)). Vreme izvršavanja se redukuje od 2,3% do 5% za rešenje Is (ET(Is-128)), odnosno od 0% do 5,2% za rešenje Isd (ET(Isd-128)). Za sistem sa keš memorijom kapaciteta 1024KB rešenje Is redukuje vreme blokiranja RST od 18% do 25% u zavisnosti od vremena pristupa memoriji prilikom čitanja, a rešenje Isd oko 44% (RST(Isd-1024)). Vreme izvršavanja se redukuje za oko 4% za rešenje Is (ET(Is-1024)), odnosno za oko 8% za rešenje Isd (ET(Isd-1024)). Rezultati pokazuju da su posmatrane veličine slabo zavisne od vremena pristupa memoriji prilikom čitanja.

Značajno redukovanje u vremenu blokiranja RST kod rešenja Isd ukazuje da je potencijal ovog pristupa značajan ali da je u konkretnom slučaju njegova efikasnost u velikoj meri ograničena kontencijom na internim resursima keš kontrolera usled izvršavanja dodatnih instrukcija za podršku mehanizmu injektiranja. Međutim, realokacija matrice kompleksnih tačaka, tako da svaka submatrica zauzima kontinualni adresni prostor može omogućiti redukciju umetnutog koda za podršku injektiranju, a time bi se značajno povećala i efikasnost tehnike injektiranja kod rešenja Isd.



(a) (b)

Sl. 5-36. FFT: NET i NRST; P=16.

Na Sl. 5-37a i Sl. 5-37b prikazano je ubrzanje SU za rešenja B, Is i Isd, kada je kašnjenje u pristupu memoriji usled čitanja MRC=20pclk i MRC=100pclk, redom. Na osnovu dobijenih rezultata mogu se izvesti sledeći zaključci:

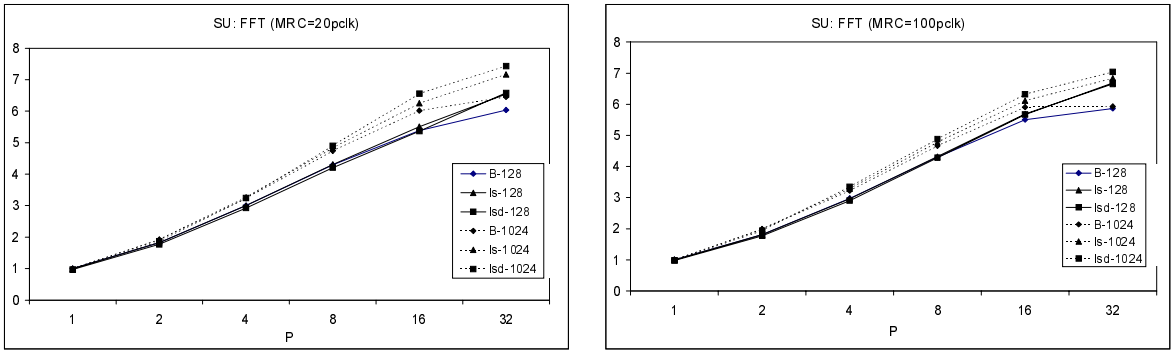
(a) rešenje Is praktično uvek pokazuje poboljšanje, izuzev u slučaju kada je broj procesora u sistemu P=2, a kapacitet keš memorije 128KB.

(b) rešenje Isd degradira vreme izvršavanja u sistemu sa keš memorijom kapaciteta 128KB i vremenom pristupa od MRC=20pclk od 3% za sistem sa P=2 procesora do 0,4% za sistem sa P=16 procesora; za sistem sa keš memorijom kapaciteta 1024KB vreme izvršavanja se degradira samo u sistemu sa P=2 procesora. U svim ostalim slučajevima rešenje Isd povećava performanse.

(c) efikasnost mehanizma injektiranja raste sa povećavanjem vremena pristupa memoriji prilikom čitanja. Tako, rešenje Is poboljšava performanse u sistemu sa P=32 procesora i keš memorijom kapaciteta 1024KB za 10%, kada je MRC=20pclk, odnosno za 13,3% kada je MRC=100pclk; pod istim uslovima rešenje Isd poboljšava performanse za 13,2% kada je MRC=20pclk, odnosno za 15,8% kada je MRC=100pclk.

(d) povećavanje kapaciteta keš memorije povećava efikasnost tehnike injektiranja. Tako, u sistemu sa P=32 procesora i vremenom pristupa memoriji prilikom čitanja MRC=20pclk rešenje Is poboljšava performanse za 7,7% u sistemu sa keš memorijom kapaciteta 128KB, odnosno 10% u sistemu sa keš memorijom kapaciteta 1024KB. Pod istim uslovima rešenje Isd poboljšava performanse za 8,2% kada je CacheSize=128KB, odnosno za 13,2% kada je CacheSize=1024KB. Prema očekivanju, kapacitet keš memorije značajnije utiče na efikasnost rešenja Isd kod koga postoji injektiranje tokom ciklusa ažuriranja memorije.

(e) povećavanje broja procesora povećava efikasnost tehnike injektiranja. Tako, rešenje Is poboljšava performanse u sistemu sa 1024KB i MRC=20pclk za 3,7% kada je broj procesora u sistemu P=16, odnosno za 10% u sistemu sa P=32 procesora. Pod istim uslovima rešenje Isd povećava performanse za 8,2% u sistemu sa P=16 procesora, odnosno za 13,2% u sistemu sa P=32 procesora.



(a) (b)

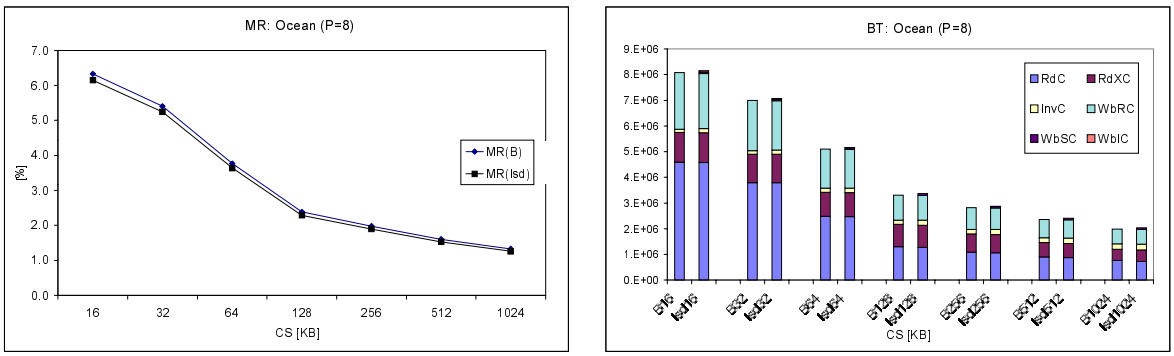
Sl. 5-37. FFT: SU.

5.2.7 Ocean

U ovom odeljku prikazani su rezultati simulacione analize za aplikaciju Ocean koja je opisana u odeljku 4.4.9. U eksperimentima koji slede posmatra se simulacija okeana sa 128×128 negraničnih tačaka. U odeljku 5.2.7.1 dati su rezultati preliminarne analize efikasnosti mehanizma injektiranja na bazi PRAM-MESI simulatora. Posmatra se originalna aplikacija B i aplikacija koja podržava injektiranje sinhronizacionih varijabli i pravih deljenih podataka Isd. U odeljku 5.2.7.2 dati su rezultati analize bazirane na MESI-SPLIT simulatoru; posmatraju se polazna verzija programa B, verzija koja uključuje podršku injektiranju samo za sinhronizacione varijable Is, i verzija koja uključuje podršku injektiranju kako sinhronizacionih varijabli tako i pravih podataka Isd.

5.2.7.1 Ocean: PRAM-MESI

Na Sl. 5-38a i Sl. 5-38b prikazani su procenat promašaja u keš memoriji MR i saobraćaj na magistrali BT, redom, u zavisnosti od kapaciteta keš memorije, kada je broj procesora u sistemu P=8. Procenat promašaja u keš memoriji se redukuje od 3% za sistem sa keš memorijom malog kapaciteta do 5% za sistem sa keš memorijom većeg kapaciteta (MR(Isd)). Saobraćaj na magistrali se neznatno povećava zbog dodatnih SWbC ciklusa, naročito za sisteme za keš memorijom malog kapaciteta.

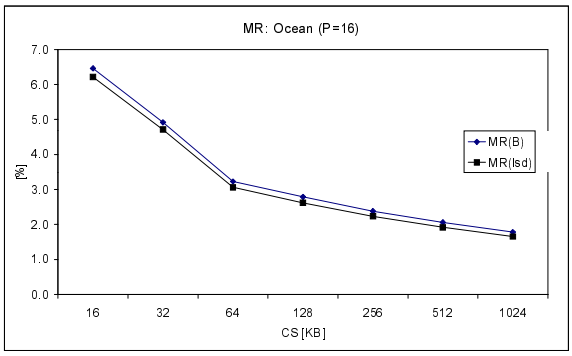


(a) (b)

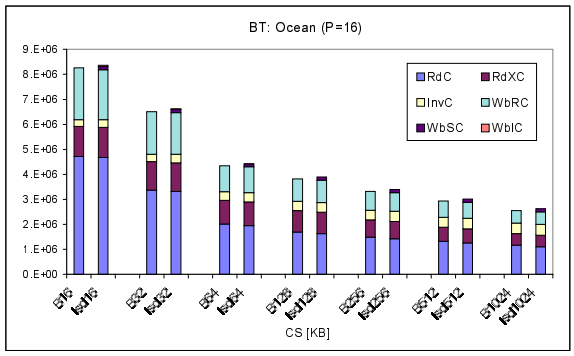
Sl. 5-38. Ocean: MR i BT; P=8.

Na Sl. 5-39 prikazani su rezultati preliminarne analize na bazi PRAM-MESI simulatora, kada je broj procesora u sistemu P=16. Procenat promašaja u keš memoriji se redukuje od 3,8% za sistem sa keš memorijom kapaciteta 16KB do 7,2% za sistem sa keš memorijom kapaciteta 1024KB. Kod sistema sa malom keš memorijom saobraćaj na magistrali neznatno raste zbog

dodatnih SWbC transakcija u uslovima kada se broj RdC transakcija ne redukuje ili se redukuje vrlo malo. Kod sistema sa velikom keš memorijom saobraćaj na magistrali približno odgovara saobraćaju u slučaju polaznog rešenja, jer važe sledeće približne jednakosti SWbC(Isd)≈RdC(B)-RdC(Isd) i RWbC(B)≈RWbC(Isd)+IWbC(Isd).



(a)

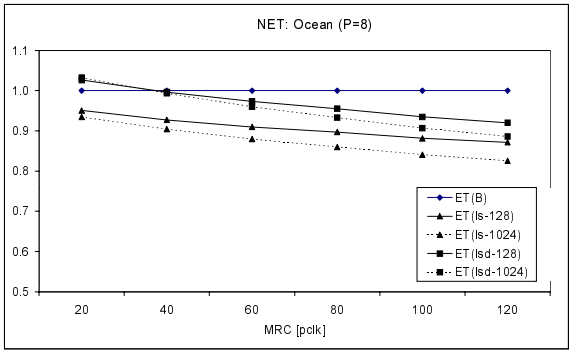


(b)

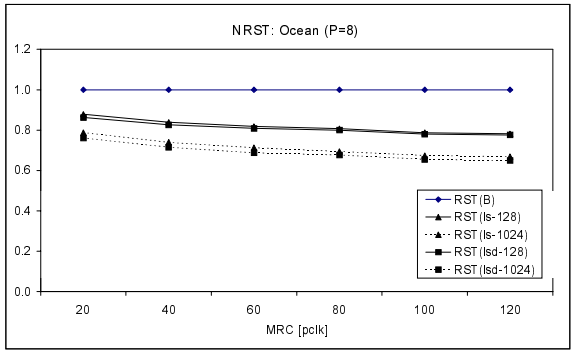
Sl. 5-39. Ocean: MR i BT; P=16.

5.2.7.2 Ocean: MESI-SPLIT

Na Sl. 5-40a i Sl. 5-40b prikazani su normalizovano vreme izvršavanja NET i normalizovano vreme blokiranja RST za posmatrana rešenja B, Is i Isd u zavisnosti od vremena pristupa memoriji prilikom čitanja MRC, kada je broj procesora u sistemu P=8. Posmatraju se dva sistema sa keš memorijom kapaciteta 128KB i 1024KB. Za sistem sa keš memorijom kapaciteta 128KB rešenje Is redukuje vreme blokiranja (RST(Is-128)) od 12% do 22% u zavisnosti od vremena MRC, a rešenje Isd od 14% do 22,5% (RST(Isd-128)). Rešenje Is redukuje vreme izvršavanja od 4% za MRC=20pclk do 13% za MRC=100pclk (ET(Is-128)). Rešenje Isd neznatno povećava vreme izvršavanja kada je MRC=20pclk; međutim, kada je MRC=100pclk rešenje Isd redukuje vreme izvršavanja za 8% (ET(Isd-128)). Za sistem sa keš memorijom kapaciteta 1024KB rešenje Is redukuje vreme blokiranja od 21% za MRC=20pclk do 33% za MRC=100pclk (RST(Is-1024)), a rešenje Isd od 24% za MRC=20pclk do 35% za MRC=100pclk (RST(Isd-1024)). Rešenje Is redukuje vreme izvršavanja od 6,5% za MRC=20pclk do 17,5% za MRC=100pclk (ET(Is-1024)), a rešenje Isd od 0,7% za MRC=40pclk do 11,5% za MRC=100pclk (ET(Isd-1024)); pri tom, rešenje Isd degradira vreme izvršavanja za MRC=20pclk za oko 3%.



(a)



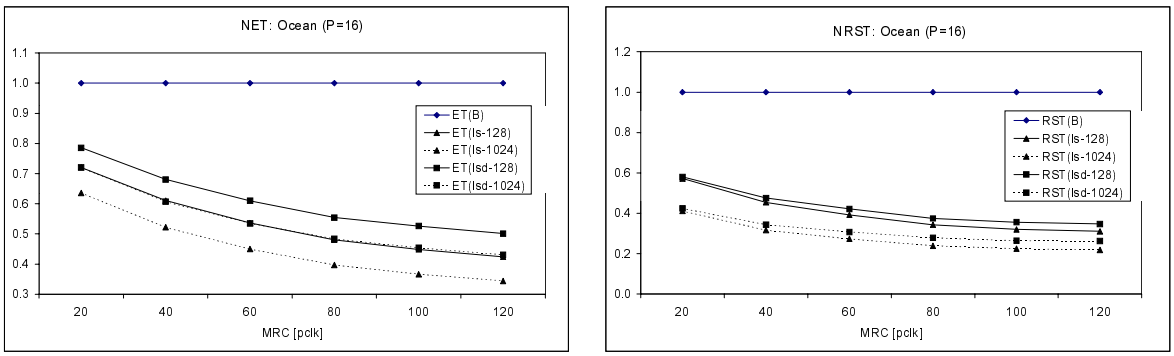
(b)

Sl. 5-40. Ocean: NET i NRST: P=8.

Na Sl. 5-41 prikazani su rezultati eksperimenata kada je broj procesora u sistemu P=16. U sistemu sa keš memorijom kapaciteta 128KB, rešenje Is redukuje vreme blokiranja od 43% za

MRC=20pclk do 69% za MRC=100pclk (RST(Is-128)), a rešenje Isd od 42% za MRC=20pclk do 65% za MRC=100pclk (RST(Isd-128)). Rešenje Is redukuje vreme izvršavanja od 28% za MRC=20pclk do 58% za MRC=100pclk (ET(Is-128)), a rešenje Isd od 21,5% od 50% (ET(Isd-128)). U sistemu sa keš memorijom kapaciteta 1024KB rešenje Is redukuje vreme blokiranja od 59% za MRC=20pclk do 79% za MRC=100pclk (RST(Is-1024)), a rešenje Isd od 57,5% za MRC=20pclk do 74% za MRC=100pclk (RST(Isd-1024)). Rešenje Is redukuje vreme izvršavanja od 36,5% za MRC=20pclk do 65,5% za MRC=100pclk (ET(Is-1024)), a rešenje Isd od 28% za MRC=20pclk do 57% za MRC=100pclk (ET(Isd-1024)).

Dobijeni rezultati pokazuju da rešenje Isd ne doprinosi poboljšanju performanse u odnosu na rešenje Is koje je jednostavnije za primenu. I pored toga što rešenje Isd doprinosi smanjivanju broja blokirajućih promašaja u keš memoriji u odnosu na rešenje Is, kompleksnost koda koji je neophodan za podršku injektiranju, kao i kontencija na internim resursima su razlog da ovo rešenje ne opravdava svoju primenu.



(a) (b)

Sl. 5-41. Ocean: NET i NRST; P=16.

Na Sl. 5-42a i Sl. 5-42b prikazano je ubrzanje SU za rešenja B, Is i Isd za sistem sa keš memorijom kapaciteta 128KB i 1024KB i vremenom pristupa memoriji prilikom čitanja od MRC=20pclk i MRC=100pclk. Na osnovu dobijenih rezultata mogu se izvesti sledeći zaključci:

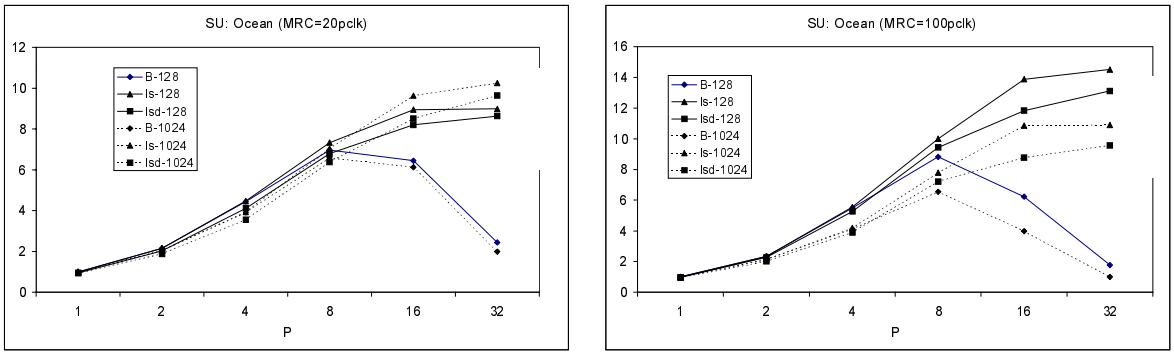
(a) bez obzira na kapacitet keš memorije i vreme pristupa memoriji pokazuje se da usled zagušenja magistrale rešenje B pokazuje degradaciju performanse u sistemima sa P=16 i P=32 procesora.

(b) rešenja Is i Isd pokazuju trend rasta performanse sa porastom broja procesora za sve parametre memorijskog podsistema. Pri tom, taj porast je veći u sistemima sa keš memorijom većeg kapaciteta i u sistemima sa većim kašnjenjem u pristupu memoriji prilikom čitanja. Tako, za sistem sa 128KB keš memorije SU(Is-128, P=16)=8,94, a SU(Is-128, P=32)=8,98, dok je za sistem sa 1024KB SU(Is-128, P=16)=9,63, a SU(Is-1024, P=32)=10,24.

(c) rešenje Is uvek poboljšava performanse i uvek pokazuje bolju performanse od rešenja Isd. Procenat poboljšanja je najveći u sistemima sa velikim kašnjenjem u pristupu memoriji i velikim brojem procesora. Tako, u sistemu sa P=32, CacheSize=1024KB i MRC=100pclk rešenje Is poboljšava performanse za 87,7%.

(d) rešenje Isd degradira performanse u sistemima sa P=2, 4 i 8 procesora kada je kašnjenje u pristupu memoriji MRC=20pclk, odnosno u sistemima sa P=2 i 4 procesora kada je kašnjenje u pristupu memoriji MRC=100pclk. Pri tom, stepen degradiranja performanse je najveći u sistemima sa vremenom pristupa memoriji prilikom čitanja od MRC=20pclk i iznosi maksimalno 9,9% kada je broj procesora u sistemu P=4. U svim ostalim slučajevima rešenje Isd poboljšava performanse.

(e) efikasnost mehanizma injektiranja raste sa povećavanjem vremena pristupa memoriji prilikom čitanja. Tako, rešenje Is poboljšava performanse u sistemu sa P=32 procesora i keš memorijom kapaciteta 1024KB za 80,6%, kada je MRC=20pclk, odnosno za 90,9% kada je MRC=100pclk; pod istim uslovima rešenje Isd poboljšava performanse za 71,7% kada je MRC=20pclk, odnosno za 79,3% kada je MRC=100pclk.
(f) povećavanje kapaciteta keš memorije povećava efikasnost tehnike injektiranja. Tako, u sistemu sa P=32 procesora i vremenom pristupa memoriji prilikom čitanja MRC=20pclk rešenje Is poboljšava performanse za 72,8% u sistemu sa keš memorijom kapaciteta 128KB, odnosno 80,6% u sistemu sa keš memorijom kapaciteta 1024KB. Pod istim uslovima rešenje Isd poboljšava performanse za 71,7% kada je CacheSize=128KB, odnosno za 79,3% kada je CacheSize=1024KB.
(g) povećavanje broja procesora povećava efikasnost tehnike injektiranja. Tako, rešenje Is poboljšava performanse u sistemu sa 1024KB i MRC=20 pclk za 36,4% kada je broj procesora u sistemu P=16, odnosno za 80,6% u sistemu sa P=32 procesora. Pod istim uslovima rešenje Isd povećava performanse za 28% u sistemu sa P=16 procesora, odnosno za 79,3% u sistemu sa P=32 procesora.



(a) (b)

Sl. 5-42. Ocean: SU.

5.3 Rezime

U ovom odeljku dat je kratak pregled rezultata simulacione analize efikasnosti mehanizma injektiranja u slučaju: (a) sinhronizacionih jezgara LTEST i BTEST, (b) originalno razvijenih paralelnih aplikacija PC, MM i Jacobi, i (c) aplikacija Radix, LU, FFT i Ocean, preuzetih iz SPLASH-2 skupa paralelnih programa. Kao mera performanse uzeto je relativno skraćivanje vremena izvršavanja koje se izračunava na sledeći način: 100*(ET(B)-ET(I))/ET(B).

Analizom vremena potrebnog za dobijanje *lock*-a (LAT) i vremena izvršavanja (ET) sinhronizacionih jezgara LTEST i BTEST utvrđeno je da mehanizam injektiranja značajno poboljšava performanse redukujući vremena LAT i ET. Pri tom, procenat poboljšanja raste sa rastom broja procesora u sistemu i povećavanjem kašnjenja u pristupu memoriji. Tako, mehanizam injektiranja redukuje vreme izvršavanja jezgra LTEST, od 12% za sistem sa P=4 procesora, do 79% za sistem sa P=16 procesora, kada je vreme pristupa memoriji tokom ciklusa čitanja MRC=20pclk, odnosno od 48% (P=4) do 84% (P=16) kada je MRC=100pclk. Vreme izvršavanja jezgra BTEST se redukuje od 56% za sistem sa P=4 procesora do 94,5% za sistem sa P=32 procesora, kada je MRC=20pclk, odnosno od 63% (P=4) do 96% (P=32), kada je MRC=100pclk.

Na Sl. 5-43 prikazano je poboljšanje performanse usled mehanizma injektiranja u slučaju aplikacija PC, MM i Jacobi u sistemima sa P=16 i P=32 procesora. Rezultati pokazuju da mehanizam injektiranja poboljšava performanse, bez obzira na veličinu keš memorije i vreme pristupa memoriji prilikom čitanja MRC. Pri tom, po pravilu efikasnost mehanizma injektiranja raste sa porastom broja procesora, kapacitetom keš memorije i vremenom pristupa memoriji. Izuzetak je aplikacija MM kod koje je procenat poboljšanja veći u sistemima sa malom keš memorijom i malim vremenom pristupa. Naime kod ove aplikacije, usled ograničenog kapaciteta keš memorije i podrške mehanizmu injektiranja, dolazi do višestrukih injektiranja, pa je efikasnost veća nego u sistemima sa većom keš memorijom. Takođe, mala vremena pristupa memoriji čine da je vreme izvršavanja u potpunosti određeno saobraćajem na magistrali, pa relaksiranje saobraćaja u slučaju injektiranja dodatno utiče na poboljšanje performanse.

	PC		MM		Jacobi	
	P=16	P=32	P=16	P=32	P=16	P=32
Isd(csl, 20pclk)	7,4%	40,6%	74,6%	82,3%	3,8%	50,1%
Isd(csl, 100pclk)	14%	44,4%	42,2%	68,3%	33%	75,8%
Isd(1024KB, 20pclk)	9,7%	43,8%	18,8%	53,5%	6,2%	53,6%
Isd(1024KB, 100pclk)	22,5%	49,4%	17,3%	50,9%	35,9%	77,7%

Sl. 5-43. Poboljšanje performanse ostvareno mehanizmom injektiranja za aplikacije PC, MM i Jacobi. Opis: Csl=64KB za test primere PC i MM, odnosno Csl=128KB za test primer Jacobi.

Na Sl. 5-44 prikazano je poboljšanje performanse usled primene mehanizma injektiranja kod SPLASH-2 aplikacija Radix, LU, FFT i Ocean, u sistemima sa P=16 i P=32 procesora. Posmatraju se verzije aplikacija koje uključuju podršku injektiranju samo sinhronizacionih varijabli Is, i verzije koje podržavaju injektiranje sinhronizacionih varijabli i pravih deljenih podataka Isd. Rezultati potvrđuju ranije uočene osobine da efikasnost mehanizma injektiranja raste sa porastom broja procesora u sistemu, kapaciteta keš memorije i vremena pristupa memoriji tokom ciklusa čitanja. Mehanizam injektiranja uvek poboljšava performanse izuzev u slučaju Isd verzije aplikacije FFT u sistemu sa malim kapacitetom keš memorije i malim kašnjenjem u pristupu memoriji; međutim, degradacija performanse u tom slučaju nije značajna i iznosi oko 0,4%. Rešenja Isd uvek pokazuju bolje performanse od rešenja Is kod aplikacije Radix, i u većini slučajeva kod aplikacije FFT. Iznenađenje je da kod aplikacije LU rešenje Isd pokazuje bolje performanse od verzije Is samo u jednom slučaju (P=32, CacheSize=128KB, MRC=20pclk), mada rešenje Isd redukuje saobraćaj na magistrali i procenat promašaja u keš memoriji u većoj meri u odnosu na rešenje Is. Međutim, relaksiranjem interne arhitekture keš kontrolera može se povećati efikasnost verzije Isd. Kod aplikacije Ocean, kompleksnost umetnutog koda za podršku injektiranju pravih deljenih podataka i njena mala efikasnost utiču da rešenje Is pokazuje uglavnom bolje performanse u odnosu na rešenje Isd.

	Radix		LU		FFT		Ocean	
	P=16	P=32	P=16	P=32	P=16	P=32	P=16	P=32
Is(csl, 20pclk)	1,9%	12%	3,5%	33%	2,3%	7,7%	28%	73%
Isd(csl, 20pclk)	3,7%	21,1%	2%	36,6%	-0,4%	8,2%	21,5%	72%
Is(csl, 100pclk)	5,4%	26%	11,9%	57%	3,2%	11,9%	55,1%	87,7%
Isd(csl, 100pclk)	8,5%	34,6%	4%	53,1%	2,9%	12,3%	47,4%	86,4%
Is(1024KB, 20pclk)	2,6%	14,5%	4,9%	42,7%	3,7%	10%	36,4%	80,6%
Isd(1024KB, 20pclk)	4,5%	25,4%	1,8%	39,1%	8,2%	13,2%	28%	79,3%
Is(1024KB, 100pclk)	8,6%	30,9%	20%	69,4%	3,4%	13,3%	63,3%	90,9%
Isd(1024KB,100pclk)	12,3%	41,1%	7,7%	62,9%	6,5%	15,8%	54,6%	89,6%

Sl. 5-44. Poboljšanje performanse ostvareno mehanizmom injektiranja za aplikacije Radix, LU, FFT i Ocean.

Opis: Csl=64KB za aplikacije Radix, LU, FFT i Ocean. Csl=64KB za Radix, odnosno Csl=128KB za LU, FFT i Ocean.

Poglavlje 6

Zaključak

Jedna od najvažnijih klasa paralelnih računara su multiprocesori sa deljenom memorijom (*shared memory multiprocessors*). Ključna osobina ove klase paralelnih računara je da se komunikacija odvija implicitno preko klasičnih upisa i čitanja deljenih podataka. Tipična organizacija ovih računara uključuje više komercijalnih procesora koji preko sprežne mreže komuniciraju sa memorijom i ulazno/izlaznim uređajima. Najjednostavniji i najpopularniji tip sprežne mreže je zajednička magistrala. Kako su sve memorijske lokacije podjednako udaljene od svih procesora ovi sistemi se često zovu *Symmetric Multiprocessor* (SMP). Kod modernih SMP sistema jedan čvor povezan sa memorijom i I/O uređajima preko magistrale sadrži procesor sa jednim ili više nivoa keš memorije. Održavanje koherencije podataka u keš memoriji ostvaruje se protokolima baziranim na nadgledanju transakcija na magistrali. Najrasprostranjeniji protokol za održavanje koherencije je MESI *write-back invalidate*.

Međutim, i pored keširanja deljenih podataka, glavna prepreka postizanju veće performanse kako kod SMP sistema sa zajedničkom magistralom, tako i kod drugih tipova multiprocesora, predstavlja kašnjenje u pristupu memoriji. Naime, trend rasta brzine mikroprocesora (>60% godišnje) daleko premašuje trend rasta brzine memorije. Stoga, eliminisanje ili prikrivanje kašnjenja u pristupu memoriji predstavlja jedan od ključnih izazova arhitektama ovih sistema. U cilju rešavanja problema velikih kašnjenja u pristupu memoriji veliki broj istraživanja poslednjih godina je posvećen tehnikama za prikrivanje kašnjenja u pristupu memoriji. U ovoj tezi analiziraju se softverski-kontrolisane tehnike za prikrivanje kašnjenja u pristupu memoriji.

6.1 Rezime teze

Istraživanje koje je rezultovalo ovom tezom započelo je detaljnim izučavanjem postojećih softverski kontrolisanih tehnika za prikrivanje kašnjenja u pristupu memoriji kod multiprocesora sa deljenom memorijom. U poslednjih 5 godina veliki broj istraživanja posvećen je tehnikama za dohvatanje podataka unapred (*data prefetching*) i tehnikama za prosleđivanje podataka budućim korisnicima (*data forwarding*). U većini istraživanja efikasnost ovih tehnika je analizirana na primeru skalabilnih CC-NUMA multiprocesora. Izuzetak su istraživanja sprovedena na Univerzitetu Washington, Seattle, koja su pokazala da

je efikasnost dohvatanja podataka unapred kod SMP sistema sa zajedničkom magistralom ispod očekivanja i u velikoj meri ograničena zauzećem magistrale i niskom efikasnošću za prave deljene podatke. Sa druge strane, implementacija tehnike prosleđivanja podataka budućim korisnicima u sistemima sa zajedničkom magistralom zahteva značajne izmene u organizaciji magistrale; do sada nisu poznata istraživanja koja bi analizirala tehniku prosleđivanja kod SMP sistema sa zajedničkom magistralom.

Polazeći od uočenih osobina postojećih tehnika za dohvatanje podataka unapred i prosleđivanje podataka budućim korisnicima, i specifičnosti SMP sistema baziranih na magistrali sa MESI protokolom za održavanje koherencije keš memorije, predložena je nova tehnika, nazvana injektiranje u keš memoriju (*cache injection*). Predložena tehnika ne isključuje primenu postojećih tehnika, a ima za cilj podizanje sveukupne efikasnosti tehnika za prikrivanje kašnjenja. Koristeći osobine zajedničke magistrale predložena tehnika eliminiše neke od nedostataka postojećih tehnika, kao što su: negativan uticaj na deljene podatke, kontencija na magistrali, kompleksnost algoritama programskog prevodioca za podršku postojećim tehnikama i cena koja se plaća usled umetanja novih instrukcija.

Kod tehnike injektiranja procesor korisnik podataka instrukcijom `OpenWindow` inicijalizuje specijalnu tabelu injektiranja koja je deo keš kontrolera u skladu sa očekivanim potrebama. Tokom *snooping* faze ciklusa čitanja ili pisanja na magistrali keš kontroler proverava sadržaj tabele injektiranja; ukoliko se adresa keš bloka tekuće transakcije na magistrali nalazi u tabeli injektiranja, procesor prihvata taj blok u svoju keš memoriju. Postoje dva osnovna scenarija kada dolazi do injektiranja: tokom ciklusa čitanja i tokom ciklusa upisa na magistrali.

Injektiranje tokom ciklusa čitanja: kada više procesora čita iste podatke (*read only data*), onda mehanizam injektiranja podrazumeva da oni inicijalizuju svoje tabele injektiranja pre trenutka stvarnog korišćenja podataka. Kada prvi od procesora čita podatak, imaće promašaj u keš memoriji i iniciraće dohvatanje deljenog keš bloka. Tokom ciklusa čitanja svi ostali procesori koji imaju validan ulaz u tabeli injektiranja sa adresom tog bloka, prihvataju podatak u svoju keš memoriju. Tako, tokom jednog ciklusa na magistrali svi procesori prihvataju blok u svoju keš memoriju, a samo jedan od njih će biti blokiran, tj. videće promašaj u keš memoriji.

Injektiranje tokom ciklusa upisa: kada se deljeni podaci modifikuju (*write shared data*), mehanizam injektiranja podrazumeva akcije na strani procesora proizvođača podataka i na strani procesora potrošača podataka. Kao u prethodnom slučaju, procesori potrošači podataka inicijalizuju svoje tabele injektiranja pre trenutka stvarnog korišćenja podataka. Sa druge strane, procesor proizvođač podataka po završetku obrade keš bloka inicira ciklus upisa u memoriju instrukcijom `Update`. Svi procesori koji imaju validan ulaz u svojim tabelama injektiranja prihvataju blok u svoju keš memoriju.

Efikasnost mehanizma injektiranja proizilazi iz redukcije promašaja usled čitanja i smanjivanja saobraćaja na zajedničkoj magistrali. Dodatna hardverska kompleksnost je minimalna i podrazumeva podršku instrukcijama `OpenWindow`, `CloseWindow` (poništavanje odgovarajućeg ulaza u tabeli injektiranja) i `Update` i implementaciju tabele injektiranja u keš kontroleru. Pored toga, preliminarne analize ukazuju da bi složenost algoritma za umetanje dodatnih instrukcija koji se realizuje u prevodiocu bila manja u odnosu na postojeće tehnike. Mehanizam injektiranja je posebno efikasan kada postoji deljenje podataka tipa 1-Proizvođač-N-Potrošača. Takav tip deljenja je naročito prisutan u implementaciji sinhronizacionih primitiva *lock*&*unlock* i *barrier* i kod pravih deljenih podataka.

U cilju verifikacije predložene tehnike koristi se simulaciona analiza bazirana na stvarnom izvršavanju paralelnih aplikacija. Simulacija se vrši korišćenjem programskog alata Limes. Kao radno opterećenje koriste se originalno razvijeni paralelni test primeri, originalno razvijene paralelne aplikacije koje demonstriraju situacije od interesa i realne paralelne aplikacije preuzete iz skupa paralelnih programa SPLASH-2. Za svaku aplikaciju razmatraju se performanse originalnog programa i jedne ili više verzija programa koje uključuju podršku mehanizmu injektiranja. Instrukcije za podršku injektiranju umeću se u kôd ručno, na osnovu statičke analize kôda aplikacije i tipa deljenja podataka. Preliminarna analiza efikasnosti mehanizma injektiranja ostvarena je korišćenjem originalnog PRAM-MESI simulatora idealnog memorijskog podsistema. Kao mera performanse koristi se procenat promašaja u keš memoriji (MR – *Miss Rate*) i saobraćaj na magistrali (BT – *Bus Traffic*). Stvarna analiza efikasnosti mehanizma injektiranja bazirana je na korišćenju originalno razvijenog simulatora MESI-SPLIT koji opisuje realni memorijski podsistem. Kao mera performanse posmatra se vreme izvršavanja (ET - *Execution Time*) i vreme blokiranja procesora tokom čitanja (RST - *Read Stall Time*) u zavisnosti od broja procesora u sistemu, kapaciteta keš memorije i kašnjenja u pristupu memoriji tokom ciklusa čitanja.

Dobijeni rezultati pokazuju značajnu efikasnost tehnike injektiranja u implementaciji sinhronizacionih primitiva. Tako, mehanizam injektiranja redukuje vreme izvršavanja od 12% do 84% za sinhronizaciono jezgro LTEST, odnosno od 56% do 94,5% za sinhronizaciono jezgro BTEST, zavisno od broja procesora i kašnjenja u pristupu memoriji. Za razmatrani skup aplikacija mehanizam injektiranja redukuje vreme izvršavanja od 1,9% do 90,9%, zavisno od broja procesora, kapaciteta keš memorije i kašnjenja u pristupu memoriji. Pri tom, efikasnost mehanizma injektiranja raste sa rastom broja procesora, kapaciteta keš memorije i kašnjenjem u pristupu memoriji.

6.2 Osnovni rezultati

Glavni rezultat ove teze je predlog nove tehnike za prikrivanje kašnjenja u pristupu memoriji kod SMP sistema sa zajedničkom magistralom, nazvane injektiranje u keš memoriju (*cache injection*). Predložene su instrukcije i hardverski resursi potrebni za podršku mehanizmu injektiranja. Takođe, data je kvalitativna analiza predloženog mehanizma injektiranja na jednostavnim test primerima koji ilustruju situacije od interesa. Efikasnost predložene tehnike je detaljno evaluirana korišćenjem simulacione analize bazirane na realnom izvršavanju paralelnih programa.

Sekundarni doprinosi uključuju:
(a) detaljan pregled relevantnih istraživanja iz oblasti softverski kontrolisanih tehnika za prikrivanje kašnjenja u pristupu memoriji kod multiprocesora sa deljenom memorijom;
(b) implementaciju PRAM-MESI simulatora idealnog memorijskog podsistema multiprocesora sa deljenom memorijom za brzu verifikaciju predloženih ideja;
(c) implementaciju MESI-SPLIT simulatora realnog memorijskog podsistema SMP sistema sa zajedničkom magistralom koja podržava razdvojene transakcije čitanja;
(d) razvoj originalnih test paralelnih programa LTEST, BTEST, PC, MM i Jacobi koji demonstriraju tipična radna opterećenja paralelnih računara;
(e) instrumentaciju kôda test primera i paralelnih aplikacija iz skupa SPLASH-2.

6.3 Pravci mogućih budućih istraživanja

Imajući u vidu trend rasta zahteva korisnika i buduće tehnološke trendove očekuje se da će značaj paralelnih računara stalno rasti. U uslovima kada pristup memoriji postaje glavna prepreka postizanju veće performanse može se očekivati da će nove tehnike za prikrivanje kašnjenja u pristupu memoriji dobijati na značaju. Predložena tehnika poseduje dobru osobinu da joj efikasnost raste sa rastom kašnjenja u pristupu memoriji i povećavanjem broja procesora u sistemu.

Postoje tri osnovna pravca daljih istraživanja. Jedan pravac je implementacija mehanizma injektiranja na drugim vrstama multiprocesora, pre svega CC-NUMA multiprocesorima. Za sada postoje ideje kako bi se ovaj mehanizam implementirao u multiprocesorima koji poseduju prsten (*ring*) kao sprežnu mrežu. Takođe, moguća je modifikacija mehanizma tako da bude implementirana u CC-NUMA sistemima koji poseduju sprežne mreže tipa rešetke (*mesh*). U tom slučaju, deo odgovornosti za implementaciju mehanizma bi se preselio u kontroler memorije.

Drugi pravac ne zavisi od konkretne arhitekture, a podrazumeva istraživanja koja imaju za cilj da se razviju algoritmi programskog prevodioca koji bi bili odgovorni za umetanje instrukcija za podršku injektiranju kod pravih deljenih podataka, budući da se mehanizam injektiranja kod sinhronizacionih primitiva uspešno može rešiti modifikovanjem bibliotečnih funkcija za sinhronizacione operacije.

Treći pravac podrazumeva razmatranje potpune implementacije mehanizma injektiranja u hardveru. Ovaj pristup podrazumeva projektovanje posebnog hardverskog resursa koji bi bio odgovoran za inicijalizaciju tabele injektiranja na osnovu analize komunikacije između procesora tokom izvršavanja paralelnog programa.

Literatura

[Amza*96] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel, "ThreadMarks: Shared Memory Computing on Network of Workstations," *IEEE Computer*, Vol. 32, No. 2, February 1996, pp. 18-28.

[BaerC*91] J. L. Baer, T.F. Chen, "An Efficient On-chip Preloading Scheme to Reduce Data Access Penalty," *Proceedings of Supercomputing'91*, November 1991, pp. 176-186.

[Bail*91] D. Bailey, J. Barton, T. Lasinski, H. Simon, "The NAS Parallel Benchmarks," *Technical Report RNR-91-002*, NASA Ames Research Center, August 1991.

[Berry*89] M. Berry et al., "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *Internation Journal of Supercomputers Applications*, vol. 3, Fall, 1989, pp. 5-40.

[Brors*96] M. Brorsson, P. Stenstrom, "Modelling Accesses to Migratory and Producer-consumer Characterized Data in a Shared Memory Multiprocessor," Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society Press, October 1996, pp. 612-619.

[Byrd*99] G. T. Byrd, M. J. Flynn, "Producer-Consumer Communication in Distributed Shared Memory Multiprocessors," *Proceedings of the IEEE*, vol. 87, no. 3, March 1999, pp. 456-466.

[Call*91] D. Callahan, K. Kennedy, A. Porterfield, "Software prefetching," *Proceedings of the 4th International Conference on ASPLOS*, April 1991, pp. 40-52.

[ChenB*92] T.F. Chen, J. L. Baer, "Reducing Memory Latency via Non-Blocking and Prefetching Caches," *Proceedings of the 5th ASPLOS*, 1992, pp. 51-61.

[Crumm*91] J. Mellor-Crummey, M. Scott, "Algorithms for Scalable Synchronization on Shared Memory Multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21-65, February 1991.

[Culler*98] D. Culler, J. P. Singh, A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, San Francisco, CA, August 1998.

[Dahlg*95] F. Dahlgren, M. Dubois, P. Stenstrom, "Sequential Hardware Prefetching in Shared Memory Multiprocessors," *IEEE Transaction on Parallel and Distributed Technology*, Vol. 6, No. 7, July, 1995, pp. 733-746.

[Dahlg*95b] F. Dahlgren, P. Stenstrom, "Effectiveness of Hardware-based Stride and Sequential Prefetching in Shared Memory Multiprocessors," *Proceedings of the 3rd HPCA*, IEEE Computer Society Press, pp. 68-77, 1995.

[FuPat*92] J. Fu, J. Patel, B. Janssens, "Stride Directed Prefetching in Scalar Processors," *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992, pp. 102-110.

[FuPat91] J. Fu, J. Patel, "Data prefetching in Multiprocessor Vector Cache Memories," *Proceedings of the 18th Annual Symposium on Computer Architecture*, May 1991, pp. 54-63.

[Golds93] S. Goldschmidt, *Simulation of Multiprocessors: Accuracy and Performance*, Ph.D. Thesis, Stanford University, June 1993.

[Hammo*97] L. Hammond, B. Nayfeh, K. Olokotun, "A Single-Chip Multiprocessor," *IEEE Computer*, Vol. 30, No. 9, September 1997, pp. 79-85.

[Hwang93] K. Hwang, *Advanced Computer Architecture:Parallelism, Scalability, Programmability*, McGrow-Hill, Inc., 1993.

[Ikodi99] I. Ikodinović, *Trace-driven simulacija u programskom alatu Limes*, Diplomski rad, Elektrotehnički fakultet u Beogradu, Mart 1999.

[Jerem*95] T. Jeremiassen, S. Eggers, "Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations," *Proceedings of the ACM SIGPLAN Sysmposium on Principles & Practice of Parallel Programming*, July 1995, pp. 179-188.

[Jouppi90] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proceedings of the 17th Annual Symposium on Computer Architecture*, May 1990, pp. 364-373.

[Klaib*91] A. C. Klaiber, H. M. Levy, "Architecture for software-controlled data prefetching," *Proceedings of the 18th ISCA*, May 1991, pp. 43-63.

[Koufa*96] D. A. Koufaty, X. Chen, D. K. Poulsen, J. Torrellas, "Data Forwarding in Scaleable Shared Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Technology*, Vol. 7, No. 12, 1996, pp. 1250-1264.

[Lenos*92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, M. S. Lam, *The Stanford DASH Multiprocesor*, IEEE Computer, March 1992, pp. 63-79.

[Lenos*95] D. Lenoski, W. D. Weber, *Scalable Shared-Memory Multiprocessing*, Morgan Kaufmann Publishers, San Francisco, CA, 1995.

[LimAg94] B. H. Lim, A. Agarwal, "Reactive Synchronization Algorithms for Multiprocessors," *ASPLOS-VI*, 1994.

[Magdic97] D., Magdic, "Limes: A Multiprocessor Simulation Environment," *TCCA Newsletter*, March 1997, pp. 68-71.

[Magdic97a] D., Magdic, *Limes: An Execution-driven Multiprocessor Simulation Tool for the i486+ based PCs – USER's Guide*, http://SOLAIR.EUnet.yu/~dav0r/limes.

[Mari*98] D. Marinov, D. Magdić, A. Milenković, J. Protić, I. Tartalja, V. Milutinović, "An Approach to Characterization of Parallel Applications for DSM Systems," *Proceedings of the 31st HICSS*, IEEE Computer Society Press, Vol. 7, January 1998, pp. 782-784.

[Mile*98a] A. Milenković, V. Milutinović, "Lazy Prefetching," *Proceedings of the 31st HICSS*, IEEE Computer Society Press, Vol. 7, January 1998, pp. 780-782.

[Mile*98b] A. Milenković, V. Milutinović, "Cache Injection on Bus-based Multiprocessors," *Proceedings of the Workshop on Advances in Parallel and Distributed Systems*, (held in conjunction with 17-th IEEE Symposium on Reliable Distributed Systems), West Lafayette, Indiana, October 1998.

[Milut*97] V. Milutinović, A. Milenković, G. Sheaffer, "The Cache Injection Control Architecture: Initial Performance Analysis," *Proceedings of the MASCOTS'97*, Haifa, Israel, January 1997.

[Miluti97] V. Milutinović, "Some Solutions for Critical Problems in the Theory and Practice of Distributed Shared Memory," IEEE TCCA Newsletter, September 1997, pp. 7-12.

[Mowry*97] T. Mowry, C. Luk, "Predicting data Cache Misses in Non-Numeric Applications Through Correlation Profiling," *Proceedings of the 30th Micro*, December 1997, pp. 314-320.

[Mowry94] T. Mowry, Tolerating Latency Through Software-Controlled Data Prefetching, Phd Thesis, University of Stanford, 1994.

[Papa*84] M. Papamarcos, J. Patel, "A low overhead Coherece Solution for Multiprocessors with Private Cache Memories," *Proceedings of the 11th Annual International Symposium on Computer Architecture*, June 1984, pp. 348-354.

[Patte*96] D. Patterson, J. L. Hennessy, *Computer Architecture – A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[Pouls*94] D. K. Poulsen, P. C. Yew, "Data Prefetching and Data Forwarding in Shared Memory Multiprocessors," *Proceedings of the 1994 International Conference on Parallel Processing*, volume II, August 1994, pp. 276-280.

[Pouls*93] D. K. Poulsen, P. C. Yew, "Execution Driven Tools for Parallel Architectures and Applications," *Proceedings of Supercomputing'93*, November 1993, pp. 860-869.

[Rama*95] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, I. Yanasak, "Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors," *Proceedings of the Supercomputing'95*, vol. 2, December 1995, pp. 1737-1775.

[Ranga*97] P. Ranganathan, V. S. Pai, H. Abdel-Shafi, S. Adve, "The Interaction of Software Prefetching with ILP Processors in Shared Memory Systems," *Proceedings of the 24th ISCA*, June 1997, pp. 144-156.

[Shafi*97] H. A. Shafi, J. Hall, S. Adve, V. Adve, "An Evaluatiion of Fine-Grain Producer Initiated Communication in Cache-Coherent Multiprocessors," *Proceedings of the 3rd HPCA*, IEEE Computer Society Press, February 1997, pp. 204-215.

[Skepp*95] J. Skeppstedt, P. Stenstrom, "A Compiler Algorithm that Reduces Read Latency in Ownership-Based Cache Coherence Protocols," *Proceedings of the PACT'95*, IEEE Computer Society Press, June 1995, pp. 69-78.

[Tanen*90] A. Tanenbaum, Y. Langsam, M. Augenstein, *Data Structures Using C*, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1990.

[Tarta*96] I. Tartalja, Milutinović V., *Tutorial on the Cache Coherence Problem in Shared-Memory Multiprocessors: Software Solutions*, IEEE Computer Society Press, Los Alamitos, California, 1996.

[Tomaš*93] M. Tomašević, V. Milutinović, *Tutorial on the Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions*, IEEE Computer Society Press, Los Alamitos, California, 1993.

[Tomaš94] M. Tomašević, *Novi hardverski protokol za koherenciju keš memorija u multiprocesorskim sistemima sa zajedničkom memorijom*, Doktorska teza, Elektrotehnički fakultet, Univerzitet u Beogradu, Beograd, 1994.

[Tous*95] F. Mounes-Toussi, D. Ljilja, "The Potential of Compile-Time Analysis to Adapt the Cache Coherence Enforcement Strategy to the Data Sharing Characteristics," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 5, May 1995, pp. 470-481.

[Tranc*96] P. Trancoso, J. Torrellas, "The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding," *Proceeding of the 25th ICPP*, IEEE Computer Society Press, Vol. 3, August 1996, pp. 79-86.

[Tulls*93] D.M. Tullsen, S.J. Eggers, "Limitations of Cache Prefetching on a Bus-Based Multiprocessor," Proceedings of the 20th Annual International Symposium on Computer Architecture, June 1993, pp 278-288, 1993.

[Tulls*95] D. Tullsen, S. Eggers, "Effective cache prefetching on bus-based multiprocessors," *ACM Transactions on Computer Systems*, Vol. 13, No. 1, 1995, pp. 57-88.

[Singh*91] J. P. Singh, W. Weber, A. Gupta, "SPLASH: Stanford Parallel Applications for Shared Memory," *Technical Reports CSL-TR-91-469*, Computer Systems Laboratory, Stanford University, April 1991.

[WooO*95] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” *Proceedings of the 22nd ISCA*, June 1995, pp. 24-36.

[Zhang*95] Z. Zhang, J. Torrellas, “Speeding up Irregular Applications in Shared Memory Multiprocessors: Memory Binding and Group Prefetching,” *Proceedings of the 22nd ISCA*, June 1995, pp. 188-199.