7. Optimizacija petlji - softverska protočnost

7.1. Fiksne iteracione distance

Diofantska jednačina oblika ax-by = n kod jednodimenzionog niza i jedne programske petlje najčešće ima oblik u kome su koeficijenti a i b jednaki (označićemo nadalje sa a). Kako se radi o Diofantskoj jednačini, a x i y kao vrednosti indeksa petlje su takođe celobrojne, tada 6.1.2. postaje:

$x-y = n/a$,

pri čemu n mora da bude deljivo sa a da bi postojalo rešenje jednačine. Tada, na osnovu izvođenja iz prethodnog poglavlja, za svaki par indeksa x (odnosno i_1) i y (odnosno i_2) postoji zavisnost od operacije koja koristi elemenat niza $X(g(i_2))$ od operacije kojom se definiše $X(f(i_1))$. Kako je razmak x-y (odnosno $i_2 - i_1$) ceo broj n/a (pretpostavljamo samo prave zavisnosti pošto će kasnije biti pokazana eliminacija antizavisnosti i izlaznih zavisnosti u petljama), a n i a su konstante poznate u vreme prevođenja. Dakle, razmak u iteracijama u kome postoji zavisnost po podacima je konstanta!

Primer 6.1. iz prethodnog poglavlja u svim slučajevima ispunjava uslov da je razmak $i_2 - i_1$ konstantan. Za takve zavisnosti po podacima operacija iz različitih iteracija se koristi termin petljama prenete zavisnosti po podacima sa fiksnom iteracionom distancom $i_2 - i_1$.

Za opštiji slučaj ugnježdenih petlji uvodi se obeležavanje iteracije preko vektora čiji je broj elemenata jednak broju idealno ugnježdenih petlji. Umesto označavanja indeksa ugnježdenih petlji i, j, k, … kao u prethodnom poglavlju, uvodi se označavanje i_1, i_2, i_3, … i_d za indekse ugnježdenih petlji. Tada se iteracija može jednoznačno odrediti preko vektora

$I = (i_1, i_2, \ldots i_d)$,

gde prvi indeks označava spoljnu petlju.

Pretpostavimo da postoje neka kasnija iteracija J i neka ranija iteracija I. U inicijalnoj netransformisanoj petlji mora biti ispunjen generalni uslov da nijedna operacija iz iteracije I ne može da bude zavisna po podacima od bilo koje operacije iz iteracije J. Zato se uvodi relacija prethođenja iteracija koja u najopštijem obliku ima formu:

$I < J \quad akko \quad \exists p: (i_p < j_p \wedge \forall\, q < p : i_q = j_q)$

Tada se uslov za prave zavisnost i antizavisnosti po podacima može generalizovati i označiti na sledeći način:

$I < J \quad f(I) = g(J)$ (7.1.)

Gde su f(I) i g(J) reference u iteracijama I i J i bar jedna referenca je vezana za upis. Ako se ograničimo samo na prave zavisnosti, tada, uz uslov I < J, referenca f(I) predstavlja upis, a referenca g(J) čitanje. Da bi se tvrdilo da ne postoji zavisnost po podacima između dve reference vektora u dve operacije petlje, za sve iteracije i za sve vrednosti I i J ne sme da važi skup uslova iz izraza (7.1.). Pretpostavimo da su X i Y dve proizvoljne vrednosti vektora iteracija, pri čemu važi X < Y. Tada se može uvesti distanca zavisnosti kao razlika vektora za dve reference u kojima se javlja zavisnost po podacima.

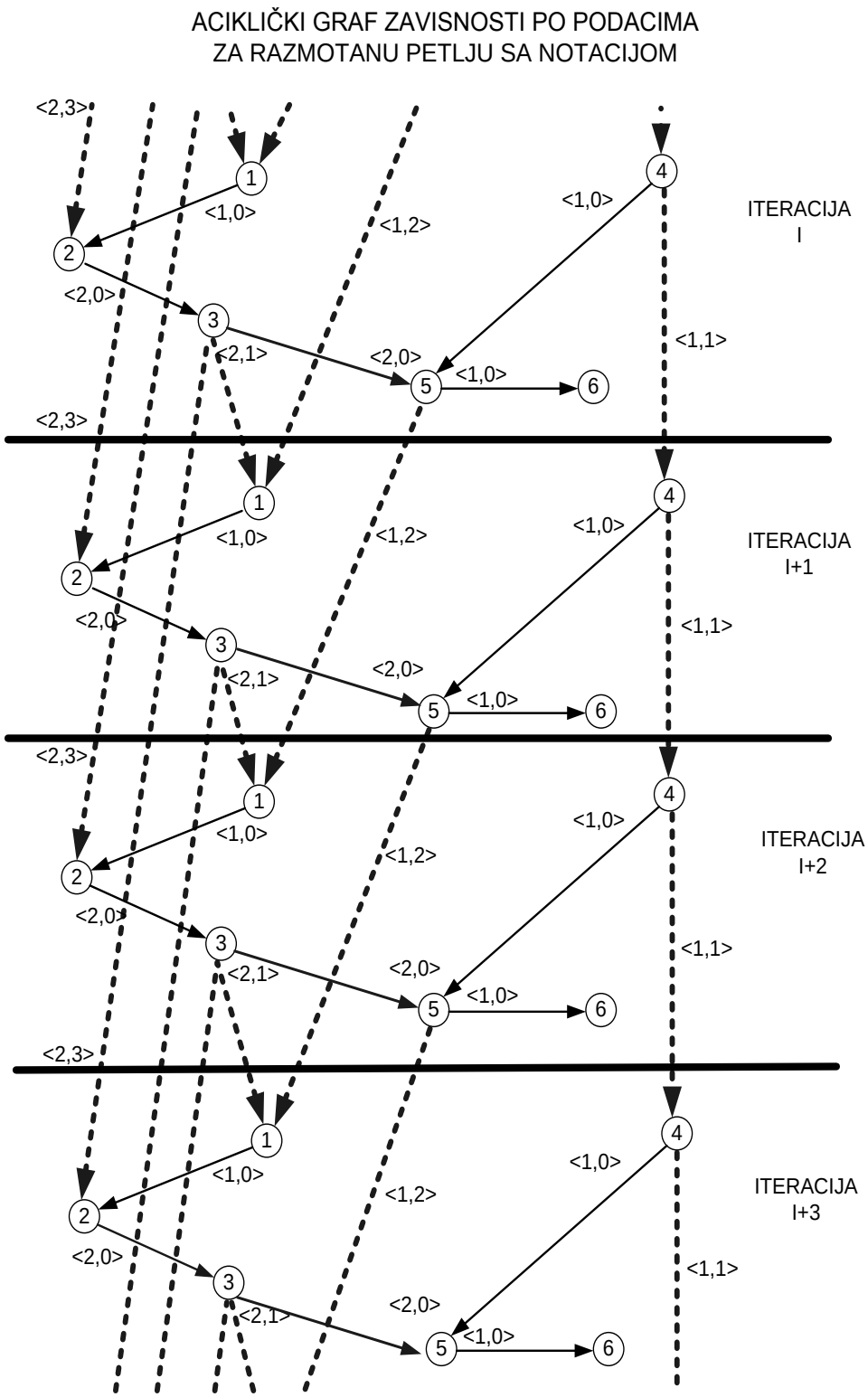
$V = Y - X = (y_1 - x_1, y_2 - x_2, \ldots, y_d - x_d)$.

Vrednost ove distance u opštem slučaju je zavisna od vrednosti X i Y. Ta zavisnost se može ogledati u činjenici da neka od razlika $y_i - x_i$ može da bude proizvoljna celobrojna funkcija. Međutim, često je razlika navedenih vektora konstanta, za sve iteracije ugnježdenih petlji, jer su svi elementi funkcije $(y_i - x_i)$ konstante. Tada se uvodi konstantan vektor distance zavisnosti, kojim su preko konstanti označene sve zavisnosti za par referenci. Kako za originalnu neparalelizovanu petlju važi da redosled izvršavanja iteracija zadovoljava zavisnosti po podacima, tako i za vektor V mora da važi $V>0$. Drugačiji način predstavljanja ovog uslova je definisanje legalnog vektora V kao leksikografski pozitivnog, što označava da prvi element vektora V koji nema vrednost 0 mora da ima pozitivnu vrednost. Vektor iteracione distance mora se pridružiti relaciji između operacija koje su u zavisnosti, da bi se kompletno prikazale zavisnosti po podacima.

7.1.2. Obeležavanje grana grafa zavisnosti po podacima programske petlje za fiksne iteracione distance

Kako je iteraciona distanca u daleko najvećem broju slučajeva konstantna, uvedena je notacija u kojoj se svakoj grani grafa zavisnosti po podacima pridružuje uređeni par u kome prvi element predstavlja trajanje operacije ili funkciju kašnjenja zavisnosti, a drugi element iteracionu distancu zavisnosti po podacima. Kako se ovde razmatraju samo prave zavisnosti po podacima, a nećemo detaljnije analizirati ugnježdene petlje, drugi element uređenog para je nenegativan ceo broj. Ukoliko je zavisnost po podacima unutar iste iteracije, tada će iteraciona distanca biti jednaka 0. Takvom predstavom se u potpunosti definiše zavisnost po podacima određena nekom granom, za slučaj kada je iteraciona distanca konstantna i kada nema ugnježdavanja petlji. U cilju pojednostavljivanja objašnjenja, u daljem tekstu će biti pretpostavljeno da su trajanje operacije ili funkcija kašnjenja zavisnosti jednaki (npr. zanemarene su faze dohvatanja i dekodovanja instrukcija (operacija) RISC-a).

Za raniji primer 6.1. u kome je 4 puta bila razmotana petlja (deo grafa beskonačno razmotane petlje), graf sa ovako uvedenom notacijom izgleda kao na slici 7.1. Operacije koje se periodično ponavljaju tokom izvršavanja kao posledica pojave te operacije u kodu iteracije će se u daljem tekstu nazivati **operacije ekvivalentne po telu petlje**. U grafu razmotane petlje, operacije ekvivalentne po telu petlje označene su istim brojem.



Sl. 7.1. Deo grafa razmotane petlje sa uvedenom notacijom za grane

7.1.3. Ugnježdene i nadgnježdene petlje

Petlja B je ugnježdena u petlju A ako se telo petlje B celo nalazi u petlji A. Tada važi da je petlja A nadgnježdena petlji B. Ovako definisane relacije ugnježdenosti i nadgnježdenosti su tranzitivne i antisimetrične i ujedno međusobno komplementarne.

Pretpostavimo da postoji *d* ugnježdenih petlji.

Definicija 7.1.: Graf ugnježdene petlje se predstavlja čvorovima ekvivalentnim po telu petlje i granama obeleženim uređenim parom p,V u kome p predstavlja prirodan broj koji pokazuje kašnjenje zavisnosti u ciklusima, a V vektor iteracionih distanci po svim petljama (ugnježdenim) te zavisnosti po podacima. Vektor V je dimenzije *d*.

Iteracione distance dobijaju se u opštem slučaju rešavanjem sistema Diofantskih jednačina. U slučaju ovde pretpostavljenih sistema linearnih Diofantskih jednačina, iteracione distance ne moraju da budu konstantne, već su linearne funkcije za linearne Diofantske jednačine. U najjednostavnijem slučaju, iteracione distance po svim ugnježdenim iteracijama su konstantne.

7.2. Grafovi zavisnosti po podacima za programske petlje

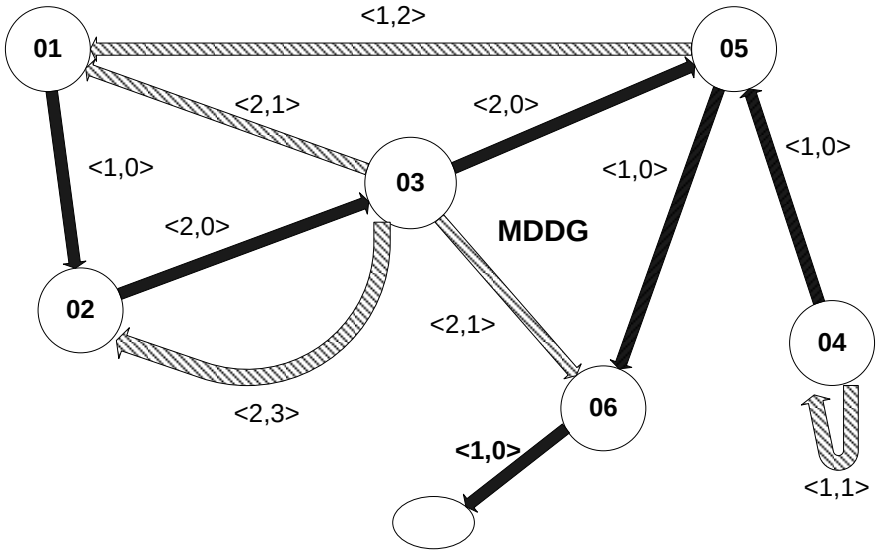
Periodične komponente u grafu beskonačno razmotane petlje otežavaju analizu paralelizma u programskim petljama. Zato se tražila jednostavnija predstava u kojoj se identične operacije iz različitih iteracija (operacije ekvivalentne po telu petlje) preslikavaju u jedan čvor grafa.

Definicija 7.2. Operacije su ekvivalentne po telu petlje ako su u originalnom kôdu jedne iteracije bile predstavljene jednom elementarnom instrukcijom (izvršavanje te instrukcija se ponavlja kako se izvršavaju iteracije petlje).

Predstavljanjem operacija ekvivalentnih po telu petlje sa jednim čvorom, sve grane koje se periodično javljaju između istih ekvivalentnih operacija stapaju se u jednu granu koja ih reprezentuje. Ovo se odnosi, kako na grane između operacija iz iste iteracije, tako i na grane između operacija iz različitih iteracija. Pritom se kao posledica periodičnosti programske petlje i fiksne iteracione distance javlja da sve grane stopljene u jednu granu imaju isto kašnjenje zavisnosti po podacima i, uzimajući u obzir uvedeno ograničenje u analizi, istu iteracionu distancu. Na ovaj način postignuto je da se dobije graf u kome nije izgubljena nijedna informacija u odnosu na graf beskonačno razmotane petlje. Model je ujedno kompaktan, tako da se mnogo lakše rade analize zavisnosti i definišu pravila za transformaciju kôda programskih petlji. Takav graf će se nazivati još i modifikovani graf zavisnosti po podacima ili MDDG. Njegova osnovna karakteristika je da može da bude ciklički graf, upravo zbog stapanja operacija ekvivalentnih po telu petlje u jedan čvor i odgovarajućeg stapanja grana grafa. Za petlju predstavljenu grafom na Sl. 7.1., on je prikazan na Sl. 7.2.

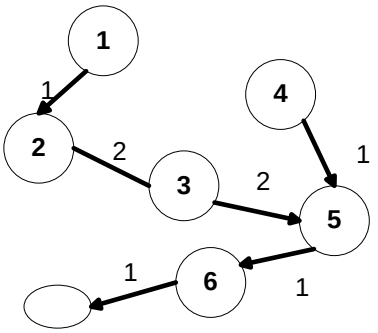
MODIFIKOVANI GRAF ZAVISNOSTI PO PODACIMA ZA PETLJU SA FIKSNIM ITERACIONIM DISTANCAMA (MDDG)

```
        DO I = 3,100
O1:             A(I) = C(I-1) + E(I-2)
O2:             B(I) = A(I) * C(I-3)
O3:             C(I) = B(I) * K1
O4:             D(I) = D(I-1) + K2
O5:             E(I) = C(I) + D(I)
O6:             F(I) = C(I-1) + E(I)
        END
```



1	O1, O4
2	O2
3	
4	O3
5	
6	O5
7	O6

NAJRANIJE IZVRŠAVANJE PO CIKLUSIMA



ADDG JEDNE ITERACIJE

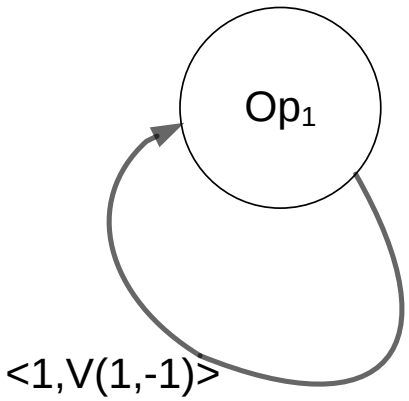
Slika 7.2. Ciklički (modifikovani) graf zavisnosti po podacima za programsku petlju sa slike 7.1.

Kako današnji procesori dominantno koriste sinhronu logiku, modeliranje zavisnosti po podacima pomoću kašnjenja u broju ciklusa podjednako tretira funkcionalne jedinice u hardveru koje generišu rezultat posle više ciklusa, bez obzira da li postoji protočnost ili ne. Razlika između te dve varijante javlja se samo u trenutku kada se uvode mašinske zavisnosti, odnosno kada se definišu vektori zauzeća resursa po modelu iz poglavlja 1.4.

Kako je prvi deo analize mašinski nezavisan, model preko grafova kojim se definiše samo kašnjenje pokriva oba slučaja. Navedimo još primer za slučaj ugnježdenih petlji, pre nego što se celokupna analiza sprovede samo za neugnježdene petlje. Za taj slučaj mora se generalizovati predstava zavisnosti po podacima i uvesti vektor zavisnosti.

Primer 7.1.: Pretpostavimo petlju kao na slici 7.3.

```
do i = 2, n
        do j = 1, n-1
Op1:            a[i, j] = a[i, j] + a[i-1, j+1]
        end do
end do
```



Sl. 7.3. Graf ugnježdene petlje sa dvodimenzionim vektorom iteracione distance

U ovom slučaju, vektor distance zavisnosti V je dvodimenzioni i potiče od drugog člana sa desne strane jednakosti. Prava zavisnost postoji zato što je izračunavanje rezultata u nekoj ranijoj iteraciji a[i-1, j+1] neophodno da bi se izračunalo a[i, j]. Vektor V po svakom indeksu predstavlja konstantu, pa prema tome ispunjava uslov da je konstantan vektor (1,-1). Vektor V ispunjava osnovni uslov da je leksikografski pozitivan jer je prvi nenulti član veći od 0.

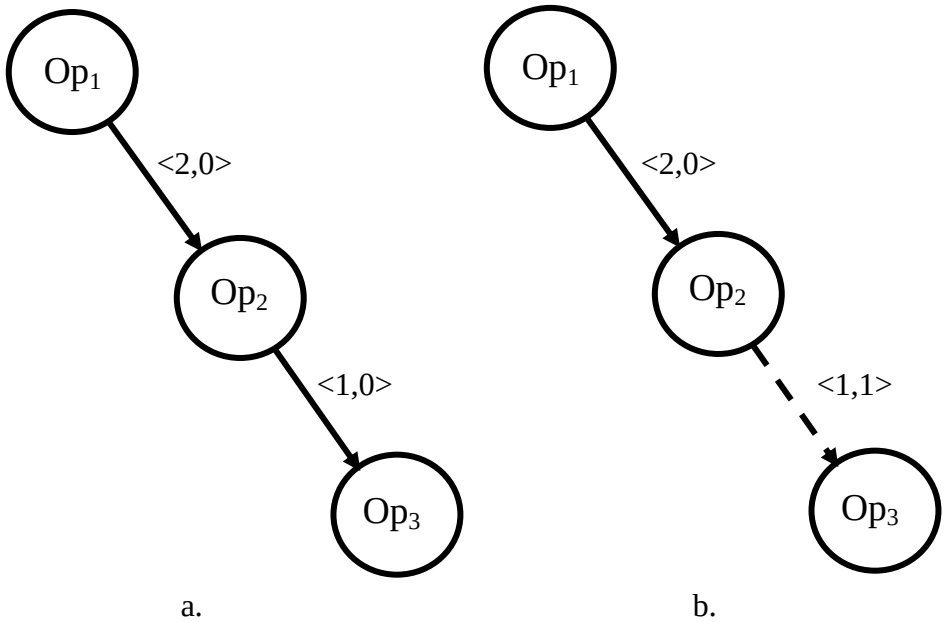
Za prvi argument operacije OP_1 postoji samo antizavisnost, ali kako je izračunavanje desne strane jednakosti neophodno pre upisa nove vrednosti u lokaciju, zavisnost se ne prikazuje.

7.2.1. DOALL i DoAcross petlje

Veoma često se u radovima potpuno pogrešno definišu termini DoAll i DoAcross programskih petlji. Ovde je prvo iznesena pogrešna definicija koja se javlja u nekim radovima.

Po pogrešnoj definiciji, DoAcross petlje su one kod kojih je potreban neki vid sinhronizacije između operacija iz različitih iteracija. Dakle, to bi značilo da su DoAll samo petlje kod kojih ne postoje zavisnosti po podacima za operacije iz različitih iteracija. DoAll petlje su dobile naziv zbog toga što se sve iteracije mogu obaviti paralelno. Međutim, programska petlja, na koju se može primeniti jednostavna transformacija može po pogrešnoj definiciji da bude istovremeno i DoAll i DoAcross. To je demonstrirano primerom 7.2.

Primer 7.2.: Pretpostavimo da je za programsku petlju $\{Op_1, Op_2, Op_3\}^n$ graf zavisnosti po podacima dat na Slici 7.4. a. Pritom nema zavisnosti po podacima za operacije iz različitih iteracija. Kako se potpuno ekvivalentan kôd dobija sa $Op_1, Op_2, \{Op_3\ Op_1, Op_2\}^{n-1}$, Op_3, graf zavisnosti po podacima nove iteracije ograničene sa {} dobija oblik dat na Slici 7.4. b. Po pogrešnoj definiciji, ova programska petlja bi sada bila DoAcross, jer je zavisnost operacije Op_3 od Op_2 sada petljom preneta sa iteracionom distancom 1. Ako se posmatra graf beskonačno razmotane petlje, postaje jasno da se i dalje mogu uraditi sve iteracije u paraleli, jer će i dalje slobodne na vrhu grafa bazičnog bloka svih iteracija biti Op_1, pa će prvo u paraleli moći da se urade operacije OP_1 svih iteracija, pa zatim Op_2 i na kraju Op_3 svih iteracija. Ta petlja se uostalom inverznom transformacijom može vratiti u prvobitan oblik, u kome nema zavisnosti po podacima za operacije iz različitih iteracija.



Sl. 7.4. Različiti grafovi DOALL petlje iz primera 7.2. pre (a) i nakon transformacije (b)

Programske petlje kod kojih ne postoje zavisnosti po podacima za operacije iz različitih iteracija uvek su DoAll tipa i sigurno se sve iteracije mogu obaviti u paraleli. Međutim, programske petlje kod kojih postoje zavisnosti po podacima za operacije iz različitih iteracija nisu uvek DoAcross. Prethodni primer je to ilustrovao.

Značajno ograničenje u paralelizmu petlje postoji samo ako u grafu beskonačno razmotane petlje postoje putevi čija je dužina srazmerna broju iteracija. To se događa ako zavisnosti po podacima za operacije iz različitih iteracija povezuju operacije iz različitih iteracija tako da se u grafu beskonačno razmotane petlje javljaju put ili putevi koji se

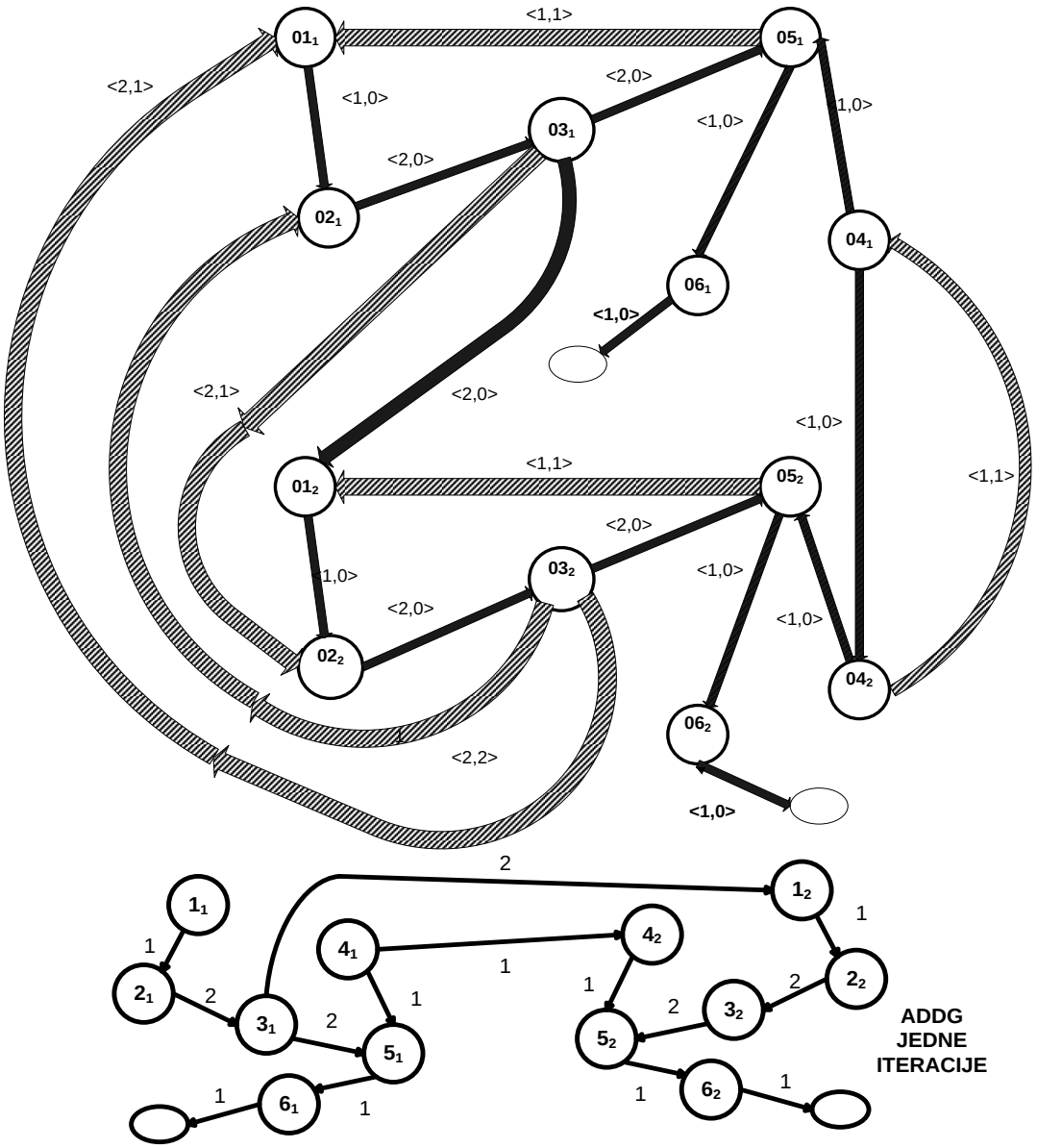
prostiru kroz sve iteracije. Takav put se periodično ponavlja, tako da se na nekom razmaku od malog pozitivnog broja iteracija počnu periodično javljati operacije ekvivalentne po telu petlje. Kako se prilikom optimizacije pretpostavlja da se obavlja veliki broj iteracija, takav put je uvek duži od najdužeg puta u acikličkom grafu zavisnosti po podacima jedne iteracije, jer je njegova dužina srazmerna broju iteracija. Takav put onemogućava paralelizaciju svih iteracija programske petlje i nikakvom transformacijom granica iteracije ili razmotavanjem se takva petlja ne može pretvoriti u programsku petlju kod koje nema zavisnosti po podacima za operacije iz različitih iteracija. Dakle, to su DoAcross petlje i ne mogu se u paraleli raditi sve iteracije.

7.2.2. Grafovi razmotane petlje za DoAll i DoAcross

Razmotavanjem programske petlje, paralelizam kod DoAll petlji povećava se srazmerno broju starih iteracija u novoj razmotanoj iteraciji. Samim povećavanjem broja starih iteracija u novoj razmotanoj iteraciji može se postići proizvoljno veliki stepen paralelizma, naravno ako se petlja izvršava dovoljan broj puta. Dakle, kod DoAll petlji, paralelizam koji se dobija prilikom izvršavanja limitiran je samo raspoloživim resursima, ako se izvršava dovoljan broj iteracija. One će uvek inspirisati projektante da ugrade još više paralelizma u hardver procesora.

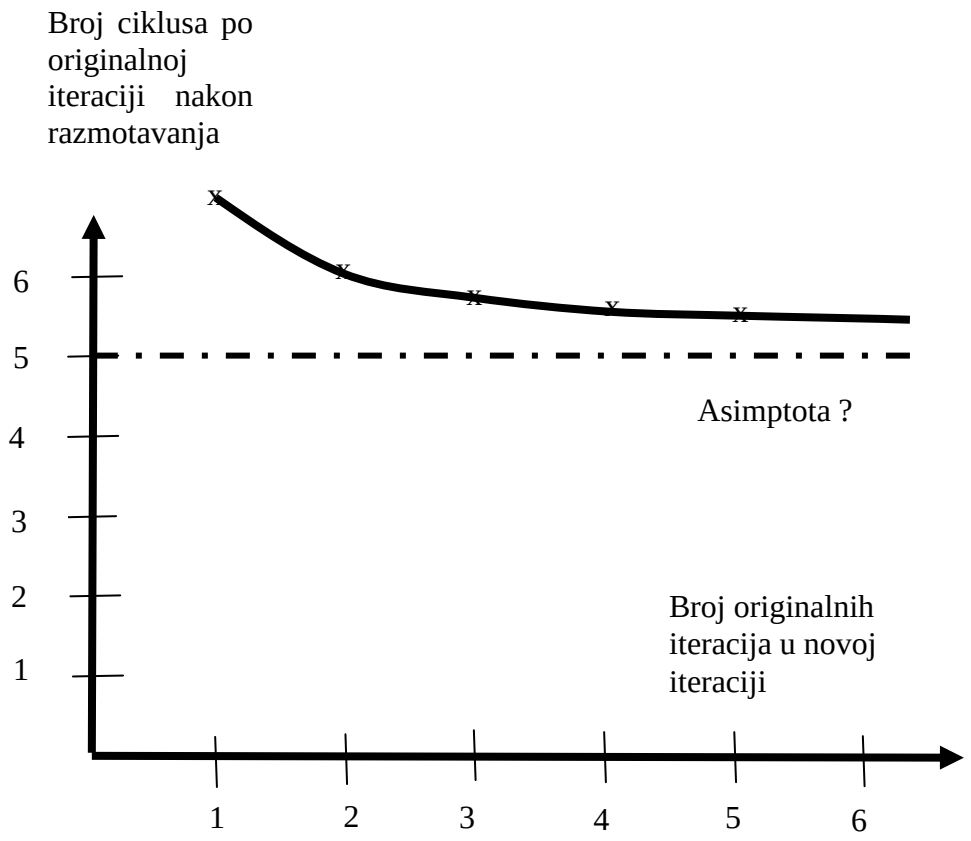
Kod DoAcross petlji, razmotavanjem se najčešće postiže viši nivo paralelizma, ali zbog limita u paralelizmu obično se razmotavanjem za relativno mali broj iteracija približavamo suštinskom ograničenju u paralelizmu petlje. Posmatrajmo primer sa Sl. 7.1. Prvo razmotrimo razmotavanje petlje kod koga će nova iteracija sadržati dve stare iteracije. Kako se na osnovu optimizacije Trace Scheduling-om nova iteracija posmatra kao bazični blok, interesantno je posmatrati kritičan put traga. Kod originalne (jedne) iteracije, njegova dužina je ukupno 7 ciklusa, jer se na osnovu trajanja rasporeda u poglavlju 1.4. mora uključiti i trajanje operacije Op_6. Međutim, kada se uradi razmotavanje u kome se u trag nove iteracije stapaju dve stare iteracije, MDDG graf razmotane petlje izgleda kao na slici 7.5. Operacije su obeležene tako da prvi indeks označava originalnu oznaku operacije iz nerazmotane petlje, a drugi indeks - kojoj po redu originalnoj iteraciji u novoj razmotanoj iteraciji je pripadala operacija. Ponovo je izostavljena tranzitivna grana od Op_3 prethodne iteracije ka Op_6 naredne iteracije.

Kada se posmatra novi ciklički graf dobijen razmotavanjem, uočava se dvostruko veći broj čvorova i takođe dvostruko veći broj grana, jer se trag (nova iteracija) sastoji iz dve stare iteracije. Najznačajnija je promena iteracionih distanci. Ako se posmatra kritičan put u acikličkom grafu dva puta razmotane petlje, uočava se da je on sada dužine 12 ciklusa. Taj put je: $Op_{11}, Op_{21}, Op_{31}, Op_{12}, Op_{22}, Op_{32}, Op_{52}, Op_{62}$. Kako se on odnosi na dve originalne iteracije, kritičan put sveden na originalnu iteraciju iznosi 6 ciklusa. Dakle razmotavanjem se postiglo kao da se uštedi jedan ciklusu na kritičnom putu po originalnoj iteraciji.



Slika 7.5. Graf zavisnosti po podacima dvostruko razmotane petlje

Daljim razmotavanjem programske petlje iz primera, (tri stare iteracije čine novu) put Op_{11},Op_{21},Op_{31}, Op_{12},Op_{22},Op_{32}, Op_{13},Op_{23},Op_{33}, Op_{53},Op_{63} postaje kritičan i ukupna dužina je 17 ciklusa. Svođenjem na originalnu iteraciju, kritičan put je 5 2/3 ciklusa po iteraciji. Daljim razmotavanjem na 4 iteracije dobija se 22 ciklusa za novu iteraciju, odnosno 5 1/2 ciklusa po originalnoj iteraciji. Vrednosti broja ciklusa po originalnoj iteraciji u funkciji broja originalnih petlji u razmotanoj petlji prikazane su na grafiku 7.6.



Slika 7.6. Grafik broja ciklusa po originalnoj iteraciji u funkciji broja razmotavanja.

Očigledno je da postoji asimptotska vrednost za kritičan put razmotane petlje kada se on izrazi u broju ciklusa po originalnoj iteraciji. Ta vrednost potiče od periodičnog ponavljanja $Op_{1i,}Op_{2i,}Op_{3i}$ i može se naslutiti da vrednost asimptote iznosi 5 ciklusa po iteraciji, zbog toga što je dužina periodičnog puta $Op_{1i,}Op_{2i,}Op_{3i}$ upravo 5 ciklusa. Takođe se uočava da petlju treba razmotati svega nekoliko puta, jer dalje razmotavanje vrlo malo doprinosi povećanju paralelizma.

Zanimljivo je posmatrati i ciklički graf dva puta razmotane petlje. On je dat na slici 7.5. On ima dvostruko više čvorova i svaka od ranijih grana se transformiše u dve nove grane. Najzanimljivije je posmatrati transformaciju iteracionih distanci grana koje su bile petljom prenesene. Grane čije su iteracione distance bile 1 u nerazmotanoj petlji su se transformisale u dve grane, od kojih jedna i dalje ima iteracionu distancu 1, ali druga postaje grana sa iteracionom distancom 0.

Grane čije su iteracione distance bile 2 su razmotavanjem transformisane u dve grane koje obe imaju iteracionu distancu 1 (što se moglo i očekivati). Grane sa iteracionom distancom 3 u inicijalnoj petlji su se transformisale u dve grane od kojih jedna ima iteracionu distancu 1, a druga iteracionu distancu 2.

Generalno pravilo je sledeće:

a. Ako razmotamo petlju k puta i postoji grana sa iteracionom distancom n, a n je deljivo sa k, ona se transformiše u k grana, a sve generisane grane imaju novu iteracionu distancu n/k.
b. Ako razmotamo petlju k puta i n nije deljivo sa k, tada se javljaju grane sa iteracionim distancama $\lfloor n/k \rfloor$ i $\lceil n/k \rceil$. Ovo označava prvu manju i prvu veću celobrojnu vrednost ovih količnika.

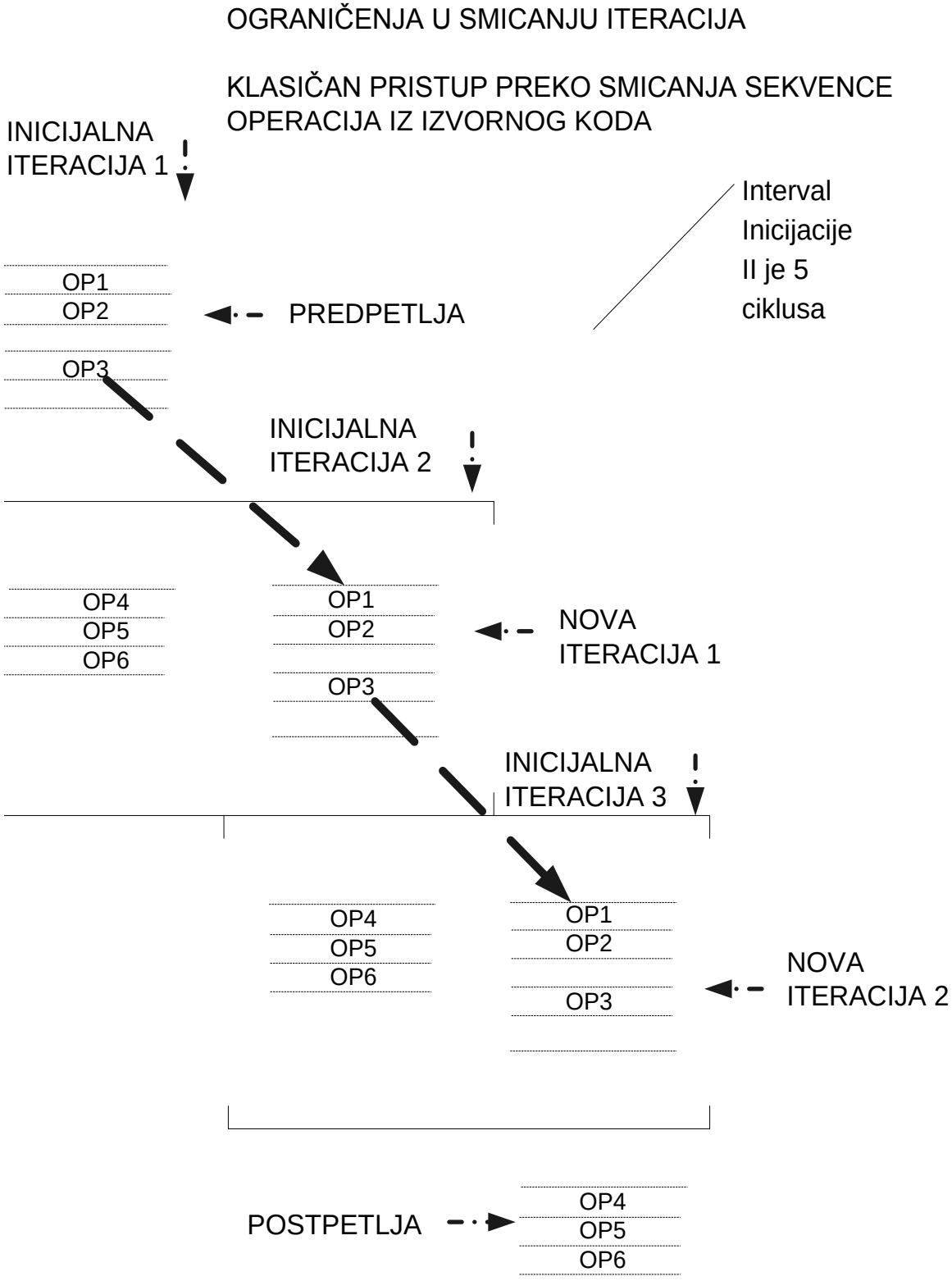
7.3. Softverska protočnost preko smicanja susednih iteracija

Softverska protočnost petlji u početku je definisana kao pokušaj imitiranja hardverske protočnosti. Analogija se sastojala u tome da programska petlja obavlja posao koji se ponavlja – slično nizu protočnih stepeni u hardveru. Preklapanje bi se postiglo tako što se relaksira zahtev iz sekvencijalnih mašina da se jedna iteracija mora završiti pre nego što započne sledeća iteracija iste petlje. Pretpostavka je da se po analogiji sa hardverskom protočnošću mogu delimično preklopiti iteracije i na taj način dobiti paralelniji kôd. U modelu softverske protočnosti, pokušaćemo da zadržimo mašinsku nezavisnost do poslednjih faza optimizacije, dok je kod hardverske protočnosti preklapanje ugrađeno u hardver. Naravno da je taj stepen preklapanja ograničen zavisnostima po podacima za operacije iz različitih iteracija i raspoloživim resursima mašine. Da bi se pojednostavio problem, pretpostavljeno je da iteraciju čini bazični blok, a problem softverske protočnosti u slučaju grananja unutar iteracije je detaljno diskutovan u {MJ 00}{MJ 01}{MJ 02}.

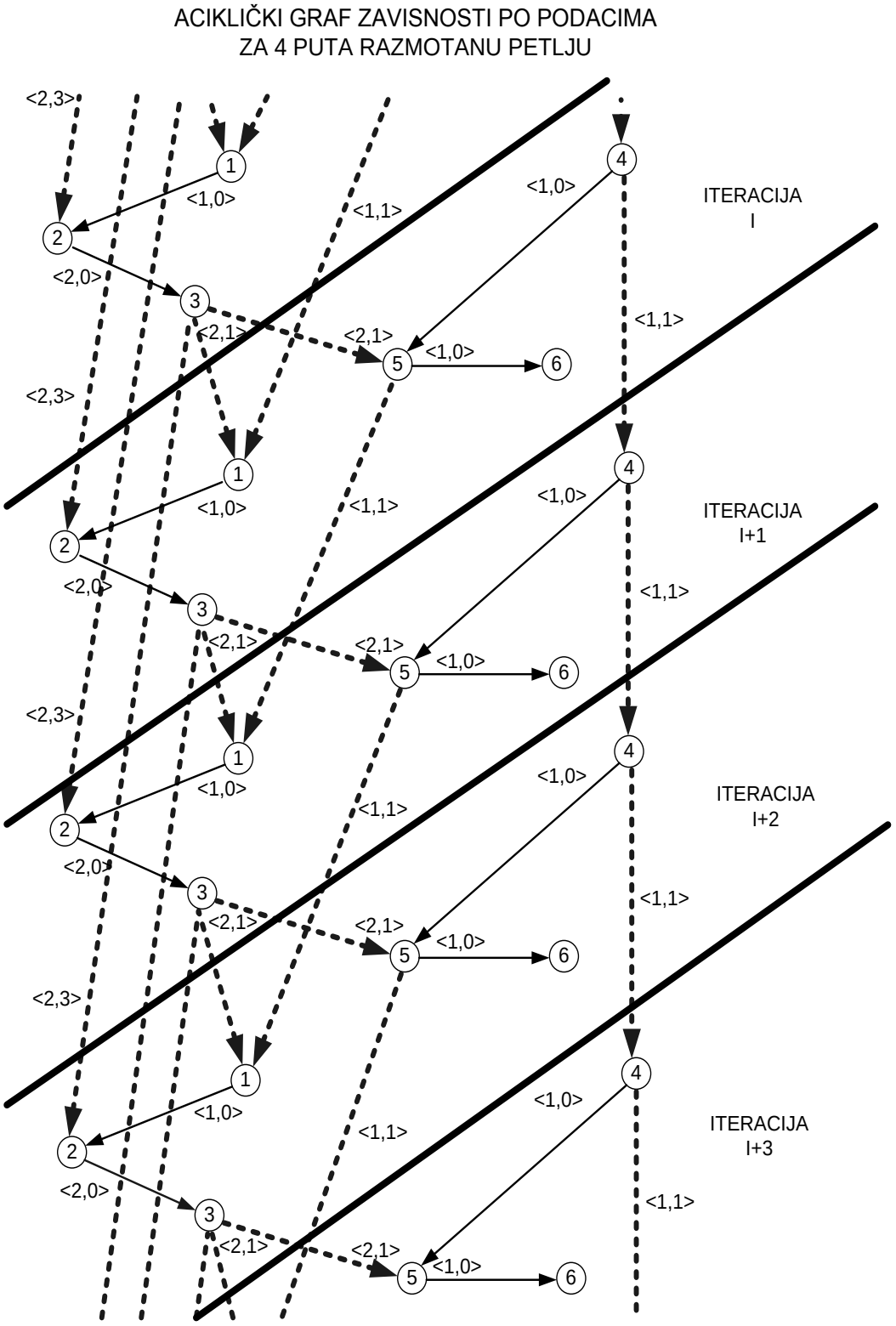
Primer 7.3.: Preklapanje i smicanje iteracija za programsku petlju iz Primera 6.1.

```
        DO I = 3,100
Op1:        A(I) := C(I-1) + E(I-2)
Op2:        B(I) := A(I) * C(I-3)
Op3:        C(I) := B(I) * K1
Op4:        D(I) := D(I-1) + K2
Op5:        E(I) := C(I) + D(I)
Op6:        F(I) := C(I-1) + E(I)
        END
```

Posmatrajmo preklapanje iteracija pri smicanju susednih iteracija. Za operacije sabiranja je pretpostavljeno da traju 1 ciklus, a za operacije množenja 2 ciklusa. Polazeći od prve 3 iteracije, raspoređenih po ciklusima prema trajanju operacija, na Sl. 7.7. je urađeno smicanje iteracija. Zanemarena su ograničenja mašine, tako da se kao osnovno ograničenje pojavila zavisnost po podacima između operacija Op_3 iz bilo koje iteracije i operacije Op_1 naredne iteracije. Ostale zavisnosti za operacije iz različitih iteracija se nisu pojavile kao kritične, jer nivo preklapanja još nije doveo do toga da one ograničavaju dalju paralelizaciju. Na osnovu takvog preklapanja je dobijeno periodično ponavljanje preklopljenih delova, za primer sa Sl. 7.7. označeno kao nova iteracija 1 i nova iteracija 2.



Sl. 7.7. Smicanje iteracija kod softverske protočnosti uz poštovanje petljom prenetih zavisnosti po podacima (zavisnost Op1 u narednoj iteraciji od Op3 prethodne iteracije ograničava smicanje)

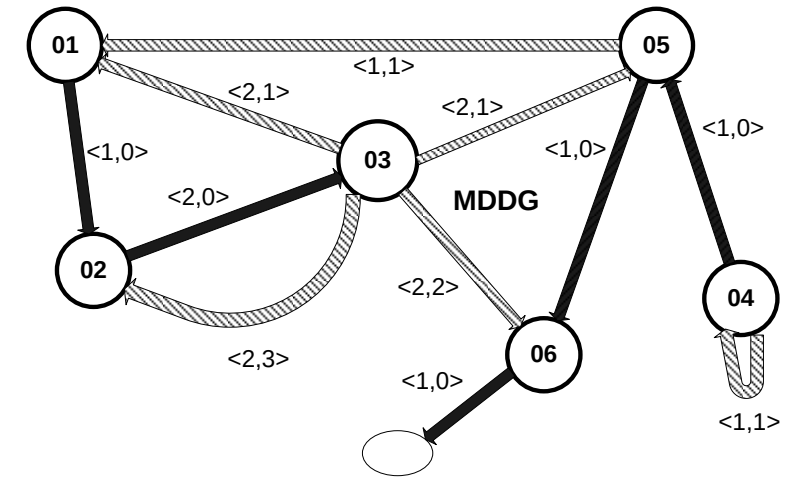


Slika 7.8. Nove granice iteracije i nove iteracione distance grana nakon smicanja iteracija

U ovom slučaju odmah je dobijena perioda ponavljanja koja obuhvata cele operacije jedne originalne (stare) iteracije. U okviru periode se nalaze delimično preklopljene operacije iz susednih iteracija. Taj deo kôda naziva se nova iteracija i ima više paralelizma od polazne iteracije. Pored termina nova iteracija, u literaturi se koristi još i termin kernel. Zanimljivo je da su se u acikličkom grafu jedne iteracije pojavile dve nepovezane komponente. Nepovezana komponenta na kojoj se nalazi kritičan put završava se granom dužine 2, jer operacija 3 traje 2 ciklusa.

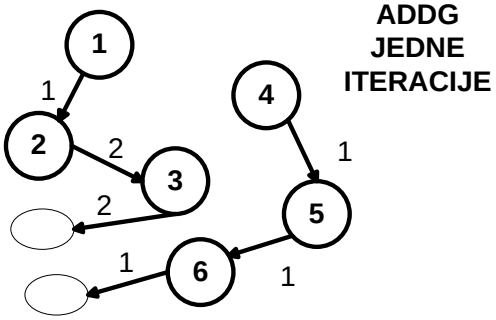
```
O1:     A(3) = C(2) + E(1)
O2:     B(3) = A(3) * C(0)
O3:     C(3) = B(3) * K1

        DO I = 3,99
O1:         A(I+1) = C(I) + E(I-1)
O2:         B(I+1) = A(I+1) * C(I-2)
O3:         C(I+1) = B(I+1) * K1
O4:         D(I) = D(I-1) + K2
O5:         E(I) = C(I) + D(I)
O6:         F(I) = C(I-1) + E(I)
        END
O4:         D(100) = D(99) + K2
O5:         E(100) = C(100) + D(100)
O6:         F(100) = C(99) + E(100)
```



1	O1, O4
2	O2, O5
3	O6
4	O3
5	

NAJRANIJE IZVRŠAVANJE PO CIKLUSIMA NA OSNOVU GRAFA JEDNE ITERACIJE



ADDG JEDNE ITERACIJE

Slika 7.9. Graf petlje za dobijenu iteraciju sa Slika 7.7 i 7.8.

Interval u kome se javlja perioda ponavljanja izražen u ciklusima mašine naziva se interval inicijacije i biće u daljem tekstu označen sa II. On je istovremeno broj ciklusa nakon kojeg započinje izvršavanje kôda druge originalne iteracije, broj ciklusa od početka originalne iteracije *I* nakon koga počinje izvršavanje iteracije *I+1*, odnosno perioda u ciklusima u kojoj treba obezbediti resurse za paralelno izvršavanje operacija preklopljenih originalnih iteracija. Dakle, II je ustvari i trajanje nove iteracije, ali operacije nove iteracije mogu biti presavijene preko granica iteracije (započinje na kraju nove iteracije, a završava na početku sledeće nove iteracije). Odnos između vremena trajanja izvršavanja iteracije na sekvencijalnoj mašini u ciklusima i II daje ubrzanje izvršavanja iteracije petlje uvođenjem nove iteracije. U konkretnom primeru ubrzanje iznosi 7/5.

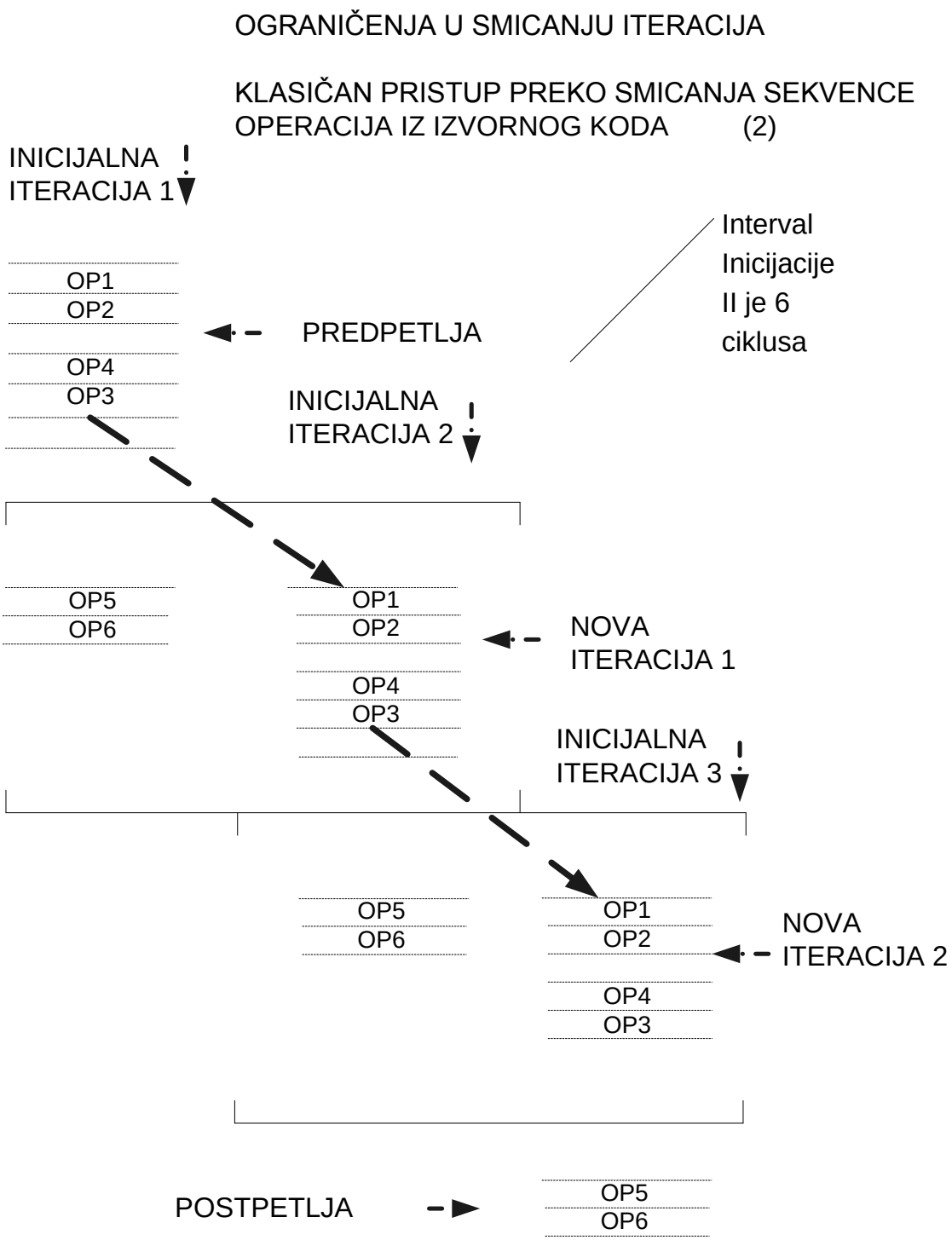
Ako se posmatraju pretpostavke koje su dovele do rešenja, mogu se uočiti sledeći nedostatci navedenog postupka:

a. Iteracija koja se smiče definisana je **isključivo za redosled operacija unutar iteracije koji je odabrao programer**. Taj, kao i svaki drugi sekvencijalni redosled naravno mora da zadovolji ograničenja koja postoje zbog zavisnosti po podacima unutar iteracije. Od zavisnosti po podacima između iteracija, smicanje uključuje samo onu koja ograničava dalje preklapanje susednih iteracija za izabrani sekvencijalni redosled.
b. **Izmene redosleda operacija** u sekvencijalnom kôdu dozvoljena grafom zavisnosti po podacima jedne iteracije **dovode do drugačijeg rešenja** i može da dovede do drugih vrednosti II
c. Mogućnosti preklapanja iteracija zasnivaju se samo na paralelizaciji preklopljenog kôda nekoliko susednih originalnih sekvencijalnih iteracija. Pritom **ne postoji mogućnost** da se koristi preklapanje u kome se preklapaju već **paralelizovane originalne iteracije**.
d. Kada je dobijen raspored, od **mašine se traži da ima dovoljno resursa u svakom ciklusu** – pretpostavlja se idealna mašina.

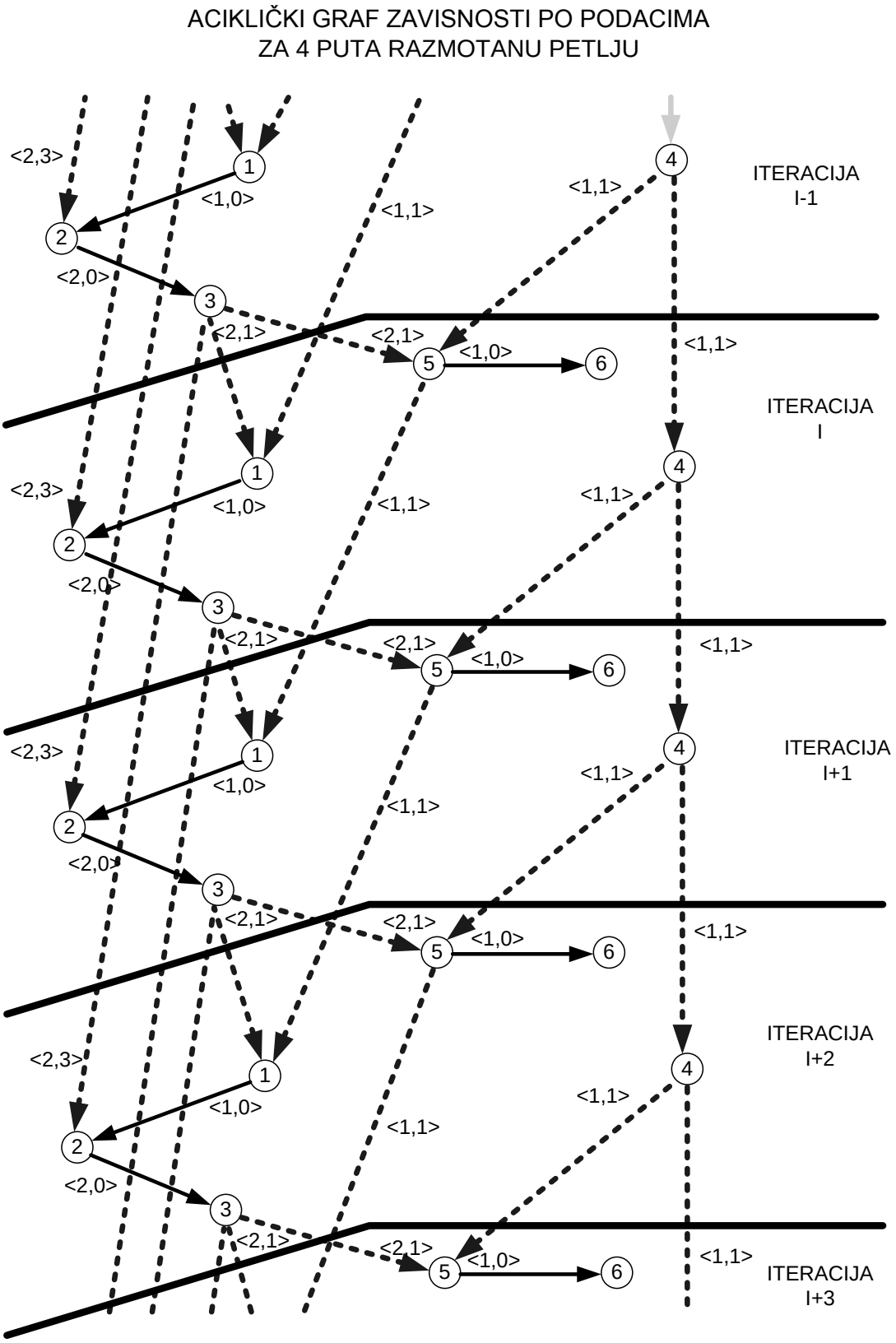
Primer kojim se ističu neke mane postupka dat je na Sl. 7.10. Ista iteracija je napisana kao sekvencijalni kôd u kome je promenjen redosled operacija Op_3 i Op_4. Ova promena redosleda dozvoljena je na osnovu grafa zavisnosti po podacima jedne iteracije. U novodobijenom kôdu, preklapanjem iteracija dobijeno je da II sada iznosi 6 ciklusa. Ovaj primer ilustruje pre svega nedostatak dat pod b, ali i pod a i c.

Međutim, kada se posmatra graf jedne iteracije, tada uopšte nema razloga za produžavanje II. To je očigledno kada se posmatra graf razmotane petlje na Slici 7.11. Jedini razlog zašto je nova iteracija manje paralelna potiče od načina formiranja nove iteracije! Pritom graf dobijene iteracije sadrži čak 3 nepovezane komponente i na osnovu grafa zavisnosti po podacima za iteraciju ne mora da traje duže od 5 ciklusa. Jedini razlog što traje duže su redosled koji je definisao programer i postupak smicanja kojim se dobija nova iteracija.

Analogija sa hardverskom protočnošću postoji u preklapanju, ali se gubi u drugim elementima. Pre svega, kod softverske protočnosti primarno se ne vodi računa o zauzeću resursa, dok je kod hardverske protočnosti ravnomerno zauzeće resursa i podizanje takta osnovni motiv. Osim toga, kod softverske protočnosti postoji velika sloboda izbora načina preklapanja, dok su kod hardverske protočnosti, protočni stepeni velikim delom fiksirani u fazi proizvodnje integrisanih kola.



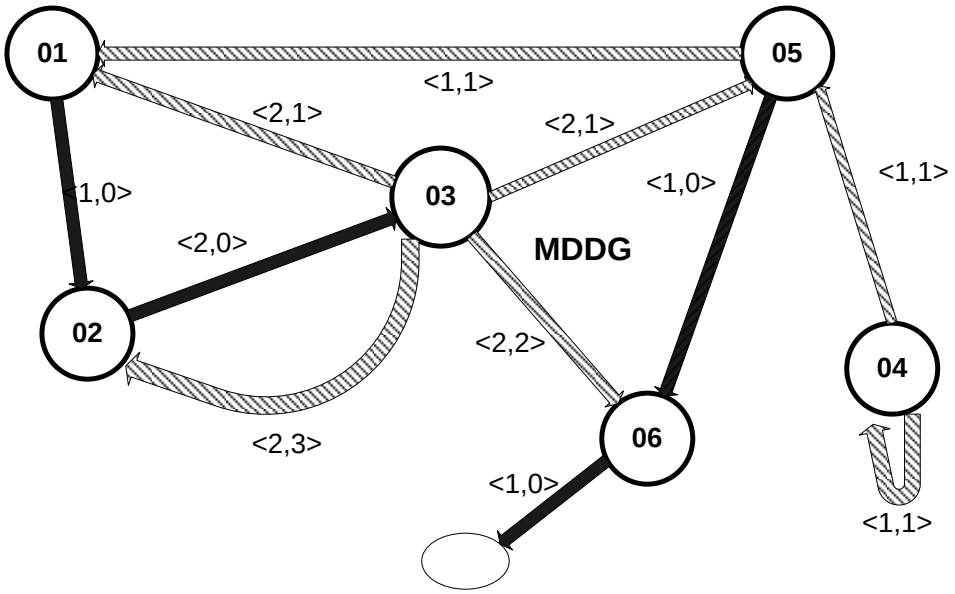
Sl. 7.10. Smanjena mogućnost preklapanja iteracija kao posledica izmene redosleda operacija unutar sekvencijalne originalne iteracije



Sl. 7.11. Nove granice iteracija za smaknute iteracije sa Sl. 7.10.

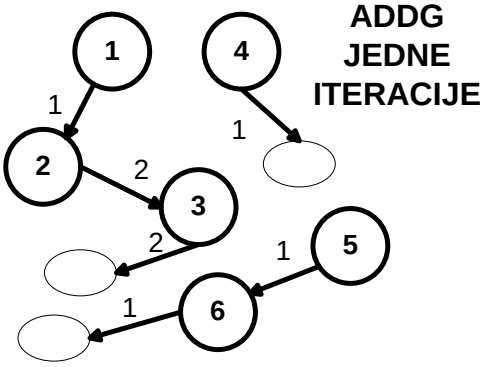
```
O1:        A(3) = C(2) + E(1)
O2:        B(3) = A(3) * C(0)
O3:        C(3) = B(3) * K1
O4:        D(3) = D(2) + K2

           DO I = 3,99
O1:            A(I+1) = C(I) + E(I-1)
O2:            B(I+1) = A(I+1) * C(I-2)
O3:            C(I+1) = B(I+1) * K1
O4:            D(I+1) = D(I) + K2
O5:            E(I) = C(I) + D(I)
O6:            F(I) = C(I-1) + E(I)
           END
O5:        E(100) = C(100) + D(100)
O6:        F(100) = C(99) + E(100)
```



1	O1, O4, O5
2	O2, O6
3	
4	O3
5	

NAJRANIJE IZVRŠAVANJE PO CIKLUSIMA NA OSNOVU GRAFA JEDNE ITERACIJE



Sl. 7.12. Graf petlje sa kodom i ADDG za novu iteraciju sa Sl. 7.10 i 7.11.

Predpetlja i postpetlja

Na primeru se uočava da se do uspostavljanja periodičnosti ponavljanja iteracija, javlja kôd koji potiče od nekoliko prvih iteracija. Taj kôd mora da bude izvan nove iteracije, jer

se ne javlja sav periodičan kôd iz nove iteracije. Taj deo kôda naziva se predpetlja (*preloop* ili *prolog* na engleskom) {LAM 88}{Al 95}.

Sličan dodatni kôd generiše se kada treba da se izvrše delovi poslednjih iteracija koje se ne mogu obuhvatiti novom iteracijom petlje. Ti delovi kôda se nazivaju postpetlja (*postloop* ili *postlog*). Ova dva dela kôda koja se ne nalaze unutar nove iteracije imaju svoj pandan kod hardverske protočnosti u popunjavanju i pražnjenju protočnog niza.

7.4. Opšti model ulaska u novu iteraciju kod softverske protočnosti

U najopštijem slučaju može doći do preklapanja operacija iz n iteracija (opseg zahvata nove iteracije) da bi se dobila nova iteracija. Tada predpetlja mora da sadrži delove n-1 iteracije, jer se delovi n-te iteracije uključuju u novu kompletnu iteraciju. Uvodi se pojam nivoa softverske protočnosti (LSP od engleskih reči Level of Software Pipelining) kojim je označen opseg zahvata nove iteracije, a njegova vrednost je n-1. Razlog za uvođenje ovakve definicije je što nivo softverske protočnosti u slučaju kada nova iteracija zahvata dve stare iteracije LSP treba da ima vrednost 1 (to je najmanja vrednost kod koje se uopšte javlja softverska protočnost).

Kao što predpetlja ima delove prvih n-1 originalnih iteracija, tako i postpetlja sadrži delove n-1 poslednjih iteracija. Analogija je potpuna sa klasičnim punjenjem i pražnjenjem hardverskog niza protočnih stepeni. Pretpostavimo da se petlja L izvršava k puta u originalnoj iteraciji. Obeležimo sa α i Ω predpetlju i postpetlju, i sa K kernel – novu iteraciju. Tada se izvršavanje petlje i transformacija kod softverske protočnosti mogu predstaviti sa

$$L^k = \alpha K^m \Omega.$$

U ovom izrazu stepen ^{m} označava broj novih iteracija koje će se izvršiti. Stepen ^{k} označava ukupan broj iteracija netransformisane (originalne) petlje. Između stepena k i m tada važi relacija m = k - n +1 za k > n – 1, odnosno za k > LSP. Naravno, ovakav izraz važi pod uvedenom pretpostavkom da nova iteracija ima po jednu operaciju reprezenta iz originalne iteracije (nije došlo još i do razmotavanja petlje).

U slučaju da će se sigurno izvršiti više od n-1 iteracija (važi k > LSP), u predpetlji nije potrebno da postoje nikakva grananja kojima se obnavljaju grananja na kraju prvih iteracija originalne petlje. Kôd nastao od delova prvih LSP iteracija je tada veliki bazični blok koji se stapa sa bazičnim blokom koji je neposredno prethodio petlji. Tako i predpetlja ima visok nivo paralelizma, a transformacija petlje softverskom protočnošću takođe treba da dovede do vrlo paralelnog kôda nove iteracije. Postpetlja su delovi poslednjih LSP iteracija i ona se uvek javlja kao veliki bazični blok ili deo velikog bazičnog bloka. Softverska protočnost petlji je veoma dobra metoda ako je sigurno ispunjen uslov k > LSP, jer su i predpetlja i postpetlja veoma paralelne i mogu se smatrati generalizovanom varijantom ljuštenja petlji (loop peeling). Generalizacija je da se ne radi ljuštenje iteracija, već ljuštenje delova prvih i zadnjih iteracija, ali sa primarnim ciljem da se optimizuje nova iteracija.

Veoma je zanimljivo analizirati šta se događa ako ne važi k > n – 1. U slučaju da je k poznato u vreme prevođenja, problem se trivijalno rešava razmotavanjem cele petlje u taj mali broj iteracija. Ako vrednost k nije poznata u vreme prevođenja, znači da je urađena transformacija petlje pomoću softverske protočnosti, a da se u vreme izvršavanja dogodilo da se izvršava manje od n iteracija koliko predstavlja neophodan broj iteracija da se bar jednom izvrši nova iteracija. Praktično, dobili bi delimični kôd razmotane petlje za tih k iteracija u predpetlji i morao bi se obnavljanjem grananja i spojeva kao kod Trace Scheduling-a izgraditi komplement od tih k iteracija, kao novi kôd kojim se zaobilazi kôd nove iteracije.

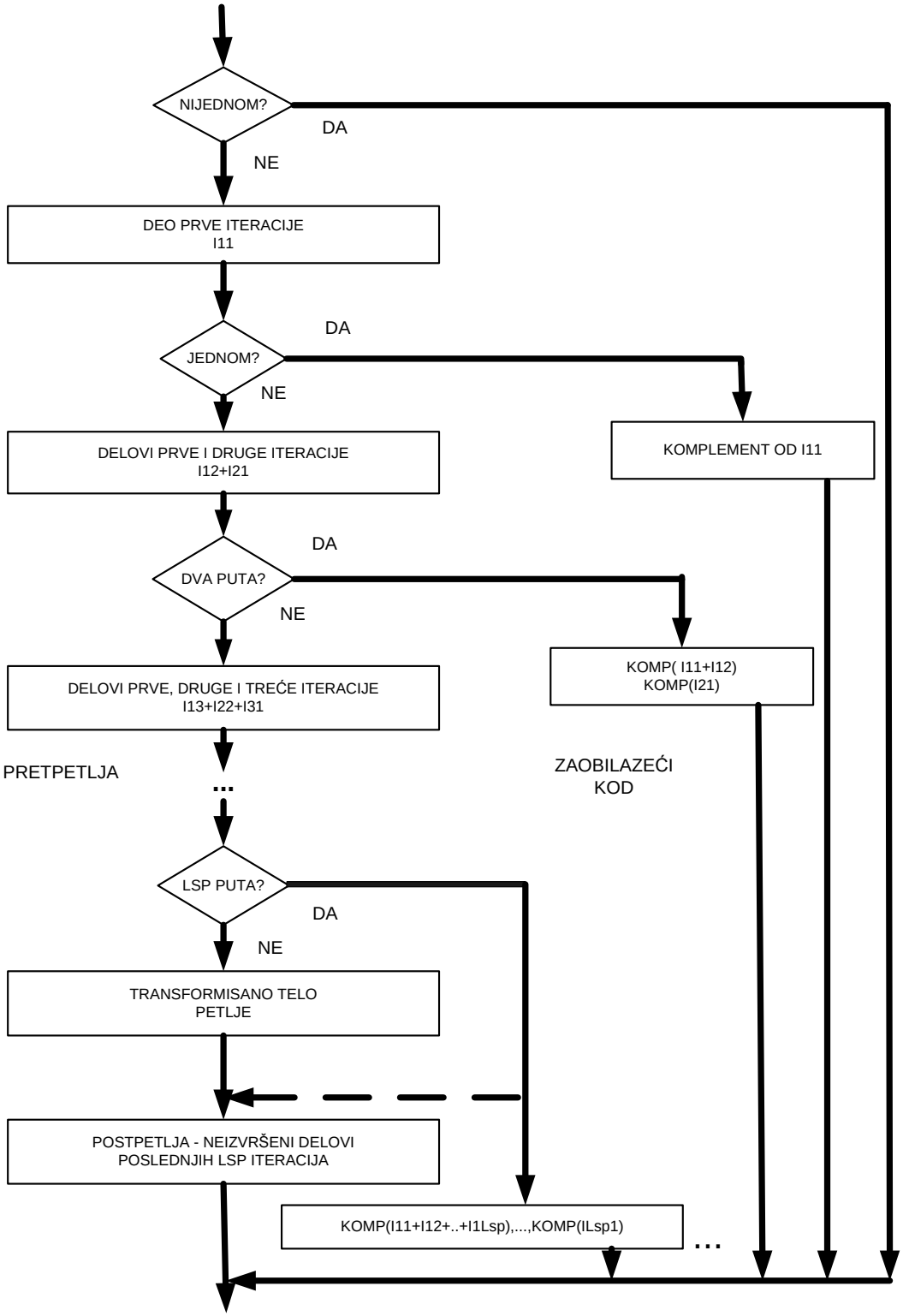
Za prikaz tog slučaja korišćena je sledeća notacija: I_{xz} označava da je u pitanju podskup operacija originalne iteracije x, a indeks z pokazuje koji je po redu podskup te iteracije x u pitanju. Neki od ovih podskupova u opštem slučaju mogu da budu i prazan skup.

Da bi se pojednostavljeno prikazala predpetlja kao postupak ulaska u novu iteraciju, na Sl. 7.13. je grafički prikazan ulazak u novu petlju. I_{11} je podskup operacija prve originalne iteracije koji nije preklopljen ni sa jednim podskupom bilo koje druge iteracije. On ne sme da bude prazan podskup. Ukoliko utvrdimo u predpetlji da treba da se uradi samo jedna stara iteracija, tada se iza podskupa I_{11} mora javiti grananje za izlaz iz predpetlje i kôd koji je komplementaran u odnosu na I_{11}, u odnosu na prvu originalnu iteraciju kako bi se kompletirala prva iteracija.

I_{12} je podskup prve iteracije preklopljen sa podskupom druge iteracije I_{21}, a pritom nije preklopljen sa podskupovima bilo koje kasnije iteracije. Ovi podskupovi takođe nisu preklopljeni sa podskupom I_{11}, ali za drugu iteraciju je I_{21} ekvivalent kôda I_{11} prve iteracije. Ako se u toku izvršavanja pokaže da treba da se urade samo dve iteracije, tada se iza unije podskupova I_{11}, I_{12} i I_{21} mora javiti grananje. Kôd iza grane, u slučaju da se iskače iz predpetlje, tada je: unija komplementa unije skupova I_{11} i I_{12} u odnosu na prvu originalnu iteraciju i komplementa podskupa I_{21} u odnosu na drugu originalnu iteraciju.

I_{13}, I_{22}, i I_{31} su preklopljeni delovi prve, druge i treće iteracije koji nisu preklopljeni sa operacijama bilo koje kasnije iteracije, a nisu preklopljeni ni sa delovima prve dve iteracije I_{11}, I_{12} i I_{21}. U opštem slučaju, preklopljeni su u fazi n–1=LSP pre uspostavljanja nove iteracije: $I_{1\ n-1}$, $I_{2\ n-2}$, $I_{3\ n-3}$,..., $I_{n-1\ 1}$, kao podskupovi originalnih iteracija od 1 do LSP koji nisu bili preklopljeni sa svim prethodno formiranim već preklopljenim podskupovima delova prvih LSP-1 iteracija.

Ovakvo formiranje preklopljenih delova ranijih iteracija i skokova kojim se obezbeđuje korektan kôd, ako se ne izvrši više od LSP iteracija prikazan je na slici 7.13. Kôd formiran od komplemenata delova prvih iteracija nazvan je zaobilazećim kôdom. Zanimljiv je komplement koji se javlja nakon grananja neposredno pre ulaska u transformisanu petlju. Taj komplement ekvivalentan je sa postpetljom i odgovarajućim podešavanjima indeksa može se najčešće ostvariti skok na početak postpetlje. Eksplozija kôda, ako je n veliko u ovom slučaju je veoma velika. Pre svega, zbog same predpetlje i postpetlje se stvara ekvivalent od LSP iteracija.



Sl. 7.13. Ulazak u transformisanu petlju kod softverske protočnosti, kada nije unapred poznat broj iteracija

Korisno je proceniti ukupnu eksploziju kôda nastalu u ovom slučaju kada se generišu grananja, ako se pretpostavi da su predpetlja i postpetlja jednake veličine. Takođe se mora pretpostaviti da je za svaku inicijalnu iteraciju x, broj operacija u I_{xz} za svako z jednako. Naravno da je uvedena pretpostavka veštačka, pre svega zato što broj operacija

u iteraciji (označen sa NOp) ne može u praktičnim primenama da bude deljiv istovremeno sa n, n-1, … , 2. Uz ove pretpostavke se ipak može dobiti najbliža procena eksplozije kôda i ona iznosi:

$$AC = NOp * [(n - 1) + n^2 / 2] \qquad (7.2.)$$

Gde je AC broj dodatnih operacija, a NOp broj operacija u inicijalnoj iteraciji. Prvi član u izrazu unutar srednje zagrade posledica je predpetlje i postpetlje, bez zaobilazećeg kôda. Drugi član je posledica zaobilazećeg kôda, pri čemu je pretpostavljeno da se kôd iza zadnjeg grananja pre nove iteracije mora obaviti nezavisno od postpetlje.

Ovo je veliko povećanje srazmerno sa kvadratom od n, iako je broj n relativno mali prirodan broj veći od 1. Srazmernost sa kvadratom od n potiče od zaobilazećeg kôda. U tipičnoj globalnoj optimizaciji kôda van petlji, sva grananja u predpetlji će se seliti naviše i doći će do kopiranja operacija zbog selidbe velikog broja operacija preko grananja nadole.

Ako se primeni Trace Scheduling, najverovatniji trag je onaj koji prolazi kroz predpetlju, a ne kroz zaobilazeći kôd. Samim tim se optimizacija predpetlje (bar još sa bazičnim blokom koji mu prethodi) prva obavlja tako da se mešaju operacije na tom tragu. Samim tim se grananja zbog zaobilazećeg kôda sele vrlo visoko. U ovom slučaju broj kopiranja, zbog ekvivalentne selidbe operacija preko grananja nadole postaje srazmeran broju n zbog broja grananja. Kako se selidbe primenjuju na deo predpetlje srazmeran sa n*NOp kao u izrazu 7.2., ukupna eksplozija broja operacija i dalje ostaje srazmerna sa drugim stepenom broja n. Naravno, koeficijenti uz drugi stepen povećavaju se nakon selidbe operacija u Trace Scheduling-u. Na osnovu navedenih rezultata, prilikom primene algoritma softverske protočnosti, stepen softverske protočnosti treba da bude što manji, naročito ako u vreme prevođenja nije poznat broj iteracija koje treba izvršiti. U slučaju kada važi k > n – 1 i k je poznato u vreme prevođenja, povećanje kôda srazmerno je sa n. Ako k nije poznato unapred, nastaje povećanje kôda $O(n^2)$.

7.5. Generalizacija softverske protočnosti u grafu beskonačno razmotane petlje

Sve mane predstavljanja nove iteracije preko smicanja već su naznačene u poglavlju 7.3. Drugačija generalizovana predstava softverske protočnosti može se dobiti ako se nova iteracija dobijena smicanjem preslika u graf beskonačno razmotane petlje. Ako se Primer 7.3. preslika, dobija se nova iteracija kao nov prozor koji definiše granice iteracije. Sada se menjaju i iteracione distance nekih grana. Pre svega, grane koje su kod originalne (stare) iteracije povezivale operacije iz različitih iteracija u slučaju nove iteracije mogu da povezuju operacije iz iste nove iteracije. Isto tako grane koje su povezivale operacije iz iste iteracije mogu nakon transformacije da povezuju operacije iz različitih iteracija. Nova iteracija podseća na promenjeni prozor u grafu beskonačno razmotane petlje, uz uslov da je novim prozorom obuhvaćen po jedan reprezent svake operacije iz originalne petlje (ekvivalentne po telu petlje). Kako se aciklički graf jedne iteracije dobija od grana čija je iteraciona distanca 0, promene iteracionih distanci nekih grana menjaju graf jedne iteracije. Mašinski nezavisan parametar koji pokazuje koliko se paralelizovala iteracija upravo je kritičan put u acikličkom grafu jedne iteracije. Intuitivno je jasno da se

postupak optimizacije svodi na traženje nove iteracije koja ima aciklički graf jedne iteracije sa minimalnim kritičnim putem. Najvažniji sekundarni parametar je smanjenje stepena softverske protočnosti, kako bi se smanjila eksplozija kôda.

Na Primeru 7.3. može se lako uočiti da je stepen softverske protočnosti 1 (n=2). Predpetlju u ovom slučaju čine operacije Op_1, Op_2 i Op_3 iz prve iteracije. One čine podskup I_{11} sa Slike 7.13. Ako nije u vreme prevođenja sigurno da će se petlja izvršiti bar dve iteracije, mora se napraviti skok pre ulaska u novu iteraciju, kojim se ne ulazi u novu iteraciju, već u zaobilazeći kôd. Očigledno je da postpetlju predstavljaju operacije Op_4, Op_5 i Op_6 iz poslednje iteracije. To se može najlakše uočiti na Sl. 7.6., ako se pretpostavi da su se izvršavale samo 4 stare iteracije, odnosno 3 nove iteracije. Operacije u postpetlji su tada upravo komplement operacija podskupa I_{11}.

Za ovaj primer prikazan je i ciklički (modifikovani) graf petlje u slučaju nove iteracije. Jedine promene u odnosu na ciklički graf originalne iteracije na Sl. 7.7. su promene iteracionih distanci! Sve grane, a samim tim i zatvoreni putevi ostali su isti, što se i očekivalo, jer su zavisnosti po podacima morale ostati. Uvođenjem apstrakcije prozora kao reprezenta iteracije u grafu beskonačno razmotane petlje očigledno se mnogo generalnije predstavlja nova iteracija softverske protočnosti u odnosu na postupak smicanja iteracija originalne petlje. Prozor u grafu razmotane petlje bio je predstavljen zadebljanim linijama. Zato je u daljem tekstu softverska protočnost definisana generalno, a zatim je korišćenjem apstrakcije prozora sproveden postupak traženja optimalne iteracije.

Prozor u grafu beskonačno razmotane petlje uklanja ranije navedene mane smicanja iteracija na sledeći način:

a. Oblik prozora potpuno je nezavistan od redosleda operacija koje je definisao operater, jer se odmah vezujemo za graf zavisnosti po podacima.
b. Ranije smicanje iteracija ograničeno jednom zavisnosti po podacima prevodi se sada u ograničenja grafa zavisnosti po podacima, a ograničenja u zavisnosti po podacima za operacije iz različitih iteracija prirodno se uklapaju u takav model
c. Traženje nove iteracije i dobijanje II je nezavisno od redosleda operacija koji je u sekvencijalnom kôdu definisao programer.
d. Apstrakcijom prozora uključene su i sve moguće varijante paralelizacije kôda unutar jedne iteracije i razmatraju se jedinstveno kod preklapanja operacija iz različitih iteracija. Tako se mogu pronaći sve postojeće nove iteracije koje se mogu dobiti softverskom protočnošću.

U slučaju apstrakcije sa prozorom, dobija se nova iteracija u obliku acikličkog grafa jedne iteracije koji se može optimizovati, čime se može postići mašinska nezavisnost novog optimizovanog grafa (iteracije). Nakon toga se za realnu mašinu može, na bazi optimizovanog grafa jedne iteracije, tražiti raspored kao da se radi o bazičnom bloku (npr. List scheduling).

7.5.1. Ograničenja nove iteracije kod softverske protočnosti

Uvedimo prvo definiciju softverske protočnosti.

Definicija 7.2.: Softverska protočnost je semantički ispravna transformacija programske petlje koja primenjuje identične intervale inicijacije za svaku novu iteraciju i ima identične interne rasporede za svaku novu iteraciju.

Ovakva definicija izbegava spominjanje smicanja originalnih iteracija da bi se dobila nova iteracija. Ona takođe ne razmatra petlje sa promenama intervala inicijacije. Ako se želi izbeći spominjanje intervala inicijacije, dobija se još generalnija definicija.

Definicija 7.3.: Softverska protočnost je semantički ispravna transformacija programske petlje kod koje se operacije iz različitih inicijalnih iteracija petlje preslikavaju u novu iteraciju. Postoji preslikavanje 1:1 između operacija ekvivalentnih po telu petlje iz stare i nove iteracije.

Usmereni ciklički putevi (ciklusi ili usmerena kola) u grafu petlje odgovaraju putevima koji se periodično ponavljaju u acikličkom grafu beskonačno razmotane petlje. Na slici 7.2. to su putevi koji prolaze kroz operacije 1231, operacije 12351, operacije 232 i operaciju 4. Period ponavljanja izražen u iteracijama razmaka može se dobiti kao zbir iteracionih distanci pojedinih grana na usmerenom kolu. Za ovu sumu će se u daljem tekstu koristiti termin *Raspon Usmerenog Kola u Iteracijama (RUKI)*.

Lema 7.1.: Svako usmereno kolo proizvodi RUKI puteva čija je dužina srazmerna broju iteracija.

Dokaz: Dve operacije ekvivalentne po telu petlje su na osnovu MDDG-a povezane putem sa razmakom od RUKI iteracija, kao posledica postojanja usmerenog kola u grafu petlje. Između tih dveju operacija se nalazi još RUKI – 1 operacija, njima ekvivalentnih po telu petlje, koje ne mogu da budu na tom istom putu, zbog ograničenja vezanih za periodičnost. Svaka od tih operacija ima put sa razmakom od RUKI iteracija do operacije ekvivalentne po telu petlje. Dakle, postoji ukupno RUKI puteva čija je dužina srazmerna broju iteracija za svako usmereno kolo. □

Pomeranje prozora iteracije u inkrementima iteracije kroz graf beskonačno razmotane petlje i postojanje samo jedne operacije ekvivalentne po telu petlje u novoj iteraciji su osnova na kojoj će se tražiti koje su transformacije semantički korektne.

Lema 7.2.: Ako su dve operacije ekvivalentne po telu petlje pripadale susednim iteracijama za polaznu petlju i pripadaju bar jednom usmerenom kolu sa RUKI jednako 1 koje ih povezuje, one takođe moraju pripadati susednim iteracijama u istom redosledu u transformisanom telu petlje.

Dokaz: Dve bilo koje operacije koje pripadaju bar jednom usmerenom kolu sa RUKI = 1 ne mogu da promene redosled. To važi i za dve operacije ekvivalentne po telu petlje na usmerenom kolu u susednim iteracijama. Kako taj odnos važi za sve susedne iteracije, redosled operacija ekvivalentnih po telu petlje na ciklusu sa RUKI = 1 mora se očuvati. Dakle susednost iteracija i redosled izvršavanja za takve operacije ekvivalentne po telu petlje moraju biti očuvani. □
Generalizacija Leme 7.2. je jedna od osnovnih osobina softverske protočnosti iskazana u Lemi 7.3.

Lema 7.3.: Kod softverske protočnosti, dve operacije ekvivalentne po telu petlje koje pripadaju zatvorenim putevima (ciklusima) u grafu petlje moraju da imaju iste iteracione distance i redosled u originalnoj i transformisanoj petlji.

Dokaz: Direktno iz dokaza Leme 7.2. proizilazi analogan dokaz, jedino je relacija pomeranja puteva generalizovana. Ako je RUKI veće od 1, redosled sada razmaknutih operacija se opet ne može promeniti zbog zavisnosti po podacima. Pokušaj izmene iteracionog razmaka dve operacije ekvivalentne po telu petlje na istom zatvorenom putu može da bude samo pokušaj povećavanja ili smanjivanja RUKI. Ako se pokuša povećati RUKI, tada se preklapanjem svih RUKI puteva događa da se javljaju nove iteracije u kojima nedostaju neke operacije. Kako to narušava definiciju softverske protočnosti, to nije moguće. Ako se pokuša smanjiti RUKI, preklapanjem RUKI postojećih puteva se moraju javiti iteracije koje imaju po dve ili više operacija ekvivalentnih po telu petlje (istih). To je opet u kontradikciji sa definicijom softverske protočnosti, pa se RUKI ne može ni smanjiti. Dakle, iteracione distance operacija ekvivalentnih po telu petlje moraju se očuvati. □

Dakle, redosled izvršavanja operacija ekvivalentnih po telu petlje mora biti očuvana kao rezultat transformacije.

Lema 7.4.: Kod Doall petlji može se izabrati proizvoljno mali interval inicijacije, tako da on bude i manji od 1 ciklusa, ekvivalentne veličine 1/k, ako se pretpostavlja da se petlja izvršava neograničen broj puta.

Dokaz: Dokažimo prvo da se DoAll petlja može uvek transformisati u oblik u kome nema zavisnosti po podacima za operacije iz različitih iteracija. Kako po definiciji DoAll petlji nema puteva čija je dužina srazmerna broju iteracija, nema nijednog puta između operacija ekvivalentnih po telu petlje. Sledi da se uvek može izdvojiti skup acikličkih grafova koji povezuju po jednog reprezenta operacije ekvivalentne po telu petlje, a da pritom taj skup acikličkih grafova nije povezan ni sa jednom drugom operacijom u grafu beskonačno razmotane petlje. Sledi da transformacijom softverske protočnosti može da se napravi iteracija koja neće imati zavisnosti po podacima za operacije iz različitih iteracija. Dakle, DoAll petlja se transformacijom pomoću softverske protočnosti može uvek svesti na petlju u kojoj nema zavisnosti po podacima između operacija iz različitih novih iteracija. Kako za petlje kod kojih nema zavisnosti po podacima između različitih iteracija nema nikakvih ograničenja za međusobno preklapanje iteracija, svaka transformacija kojom se poštuju zavisnosti po podacima unutar iteracije je korektna. Ekstremum je da II bude 0, kada su sve iteracije petlje preklopljene bez smicanja. Takođe, može se preklopiti po k iteracija da budu nesmaknute, pa da se zatim svaka naredna grupa od k iteracija smiče za jedan ciklus. U tom slučaju je interval inicijacije 1/k, ako se posmatra iz ugla originalne (stare) iteracije. Kako je interval inicijacije u principu prirodan broj, tada se prethodni slučaj može posmatrati kao k puta razmotana DoAll petlja sa intervalom inicijacije od jednog ciklusa. □

7.5.2. Uslovi da nova iteracija bude semantički ispravna.

Primenom transformacije $L^k = \alpha K^m \Omega$□□ dobijamo novu transformisanu iteraciju. Da bi transformacija bila semantički ispravna, ne sme se narušiti nijedna zavisnost po podacima za beskonačno razmotanu petlju. U transformaciji je jasno da predpetlja mora da prethodi bilo kojoj novoj iteraciji, a postpetlja mora da sledi iza bilo koje nove

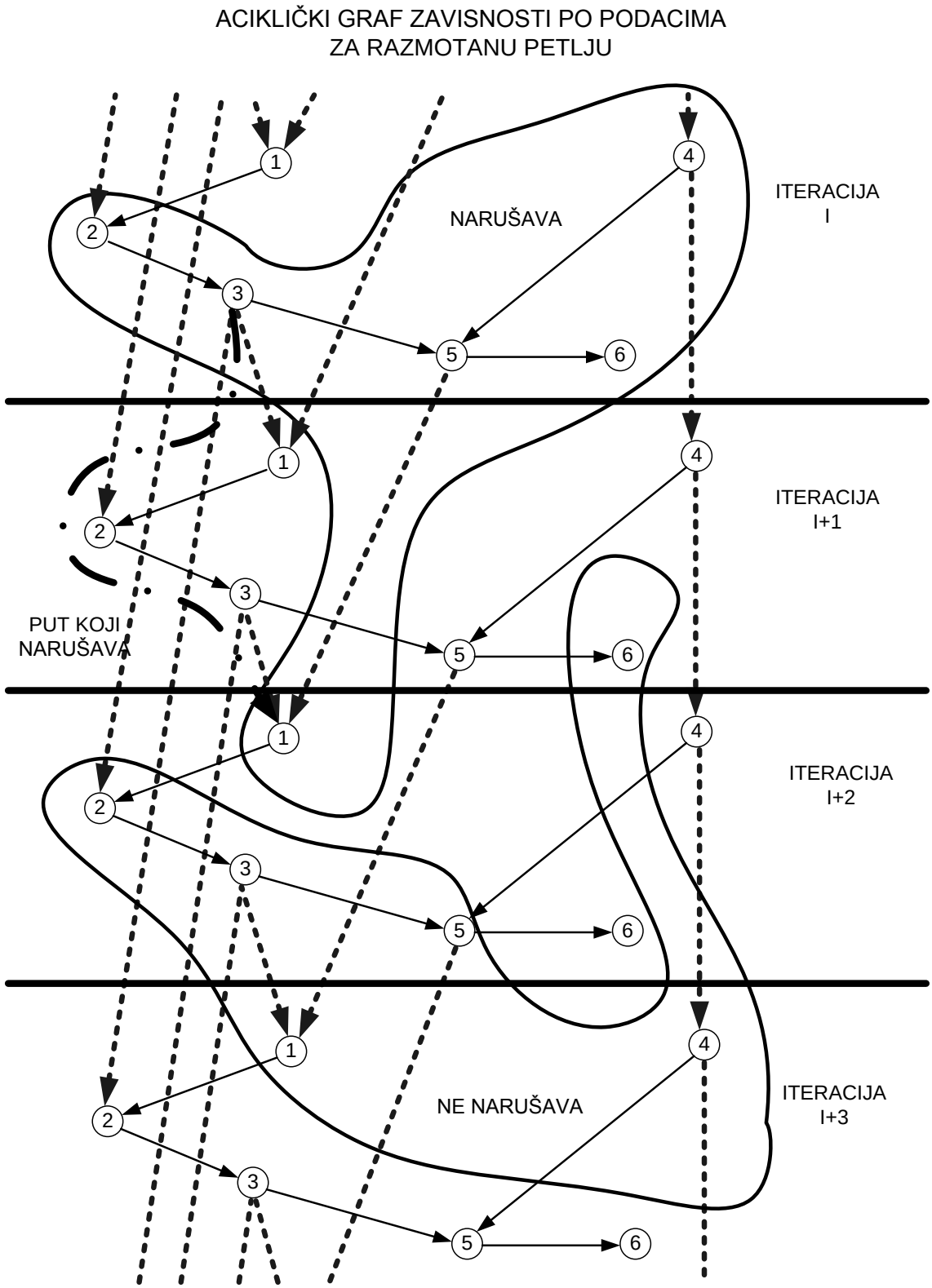
iteracije i naravno predpetlje. Na osnovu toga, ne sme da postoji nijedna operacija u predpetlji koja je zavisna od bilo koje operacije iz bilo koje iteracije nove petlje. Očigledno je takođe da nijedna operacija iz predpetlje ne sme da bude zavisna od bilo koje operacije iz postpetlje. Na kraju, nijedna operacija iz bilo koje nove iteracije ne sme da bude zavisna od bilo koje operacije iz postpetlje. Sva ova ograničenja moraju da važe za bilo koju vrednost `m`, n i `k` iz izraza
`m = k` - n +1 i za bilo koje iteracije. Navedena ograničenja su u daljem tekstu nazvana Predpetlja – Transformisana iteracija – Postpetlja ograničenja ili skraćeno PTP ograničenja. Ona predstavljaju rezultat kombinovanja zadržanih, ali transformisanih ograničenja zbog kontrolnih zavisnosti (granice iteracije) i zavisnosti po podacima u grafu beskonačno razmotane petlje.

Uslov da nova iteracija bude semantički ispravna može se vezati za kontrolne zavisnosti iteracije. Za izabrani prozor iteracije u grafu beskonačno razmotane petlje, ne sme postojati put koji polazi iz nove iteracije i završava u istoj novoj iteraciji, a da pritom uključuje bar jednu operaciju koja ne pripada toj iteraciji. Tada bi operacija izvan iteracije istovremeno morala da se izvršava i pre i posle nove iteracije, što je kontradikcija. Drugi ugao posmatranja je: kada bi takav put postojao, to bi značilo da se nova iteracija ne može izvršiti kao celina. Zato se ovo ograničenje naziva ograničenjem integriteta transformisane iteracije. Primeri dve nove iteracije, od kojih jedna narušava ograničenje integriteta, a druga ne narušava, prikazani su u grafu beskonačno razmotane petlje na Slici 7.14.

Lema 7.5.: Ako je PTP restrikcija zadovoljena, tada je i restrikcija integriteta takođe zadovoljena.
Dokaz: Prvo će biti pokazano da, ako je restrikcija integriteta narušena, onda je i PTP restrikcija narušena.
Ako restrikcija integriteta nije zadovoljena za prvu iteraciju, postoje dva slučaja:

1. Postoji operacija (čvor grafa zavisnosti po podacima) u nekoj operaciji u prvoj novoj iteraciji od koje polazi zavisnost (grana grafa zavisnosti po podacima) i završava u predpetlji.
2. Postoji operacija (čvor grafa) u nekoj kasnijoj iteraciji zavisna od operacije iz prve nove iteracije od koje istovremeno polazi zavisnost (grana grafa) i završava u nekoj operaciji u prvoj novoj iteraciji.

U slučaju 1., u isto vreme ta operacija iz predpetlje izvan prve nove iteracije je zavisna od neke operacije u prvoj novoj iteraciji. To znači da je operacija u predpetlji zavisna od neke operacije iz novih iteracija. Sledi da PTP restrikcija ne može da važi, čime je pokazano da u slučaju 1., ako se naruši restrikcija integriteta, tada su narušene i PTP restrikcije.



Slika 7.14. Iteracije (prozori iteracija) koje(i) narušavaju i ne narušavaju ograničenje integriteta

Dokažimo sada to i za slučaj 2. Zbog periodičnosti petlji važi periodičnost zavisnosti po podacima. Tada će morati da postoji put koji polazi iz iteracije θ, prolazi kroz operaciju u iteraciji θ + ω i ponovo završava u novoj iteraciji θ. To će važiti za sve vrednosti iteracije θ, zbog periodičnog ponavljanja zavisnosti. Ako posmatramo slučaj da je θ neka od poslednjih iteracija, tada jedino u slučaju da je θ poslednja iteracija i ω = 1, a operacija izvan iteracije pripada operacijama ekvivalentnim po telu petlje iz skupa I_{11} sa Slike 7.13., neće biti narušena PTP restrikcija u postpetlji. U svim ostalim slučajevima, bi bila narušena i PTP restrikcija. Međutim, proverimo taj izuzetak u slučaju da važi θ = 1. Tada postoji iz operacije ekvivalentne po telu petlje operaciji iz skupa I_{11} koja pripada prvoj iteraciji zavisnost (grana) prema operaciji iz predpetlje. Ova zavisnost je posledica periodičnosti zavisnosti. Sledi da je i u drugom slučaju narušena PTP restrikcija.
Dakle, uvek kada je narušen integritet iteracije, narušena je i PTP restrikcija. Dakle, PTP restrikcije su u opštem slučaju strožije od restrikcije integriteta iteracije.

Jedan od slučajeva narušavanja PTP restrikcija koji ne dovodi do narušavanja restrikcije integriteta je naveden kao primer. Pretpostavimo da iz neke kasnije iteracije
θ + ω
postoji zavisnost od operaciji $Op_{\theta+\omega\, y}$ prema operaciji $Op_{\theta\, z}$ u iteraciji θ u neispravnoj iteraciji. Ako pritom ne postoji nijedan put u grafu od operacije $Op_{\theta\, z}$ prema bilo kojoj operaciji u iteraciji θ + ω u grafu beskonačno razmotane petlje, neće biti narušena restrikcija integriteta iteracije. Pritom može biti narušena PTP restrikcija, jer će zbog periodičnosti moći da postoji grana iz neke od prvih ω iteracija prema operacijama u predpetlji, zbog operacija Op_y ekvivalentnih po telu petlje. Time je dokazano da može da postoji narušavanje PTP restrikcija, a da ne bude narušena restrikcija integriteta.

Teorema 1: Skup svih transformacija softverske protočnosti koje zadovoljavaju PTP restrikcije jednak je skupu svih semantički ispravnih transformacija softverske protočnosti.

Dokaz: Na osnovu definicije PTP restrikcija, skup svih semantički ispravnih transformacija kod softverske protočnosti može samo da bude podskup svih transformacija koje zadovoljavaju tu restrikciju. Pretpostavimo da postoji transformisana iteracija koja zadovoljava PTP restrikciju, ali nije semantički ispravna. Jedini način na koji bi još bila narušena semantička ispravnost transformacije je narušavanje zavisnosti po podacima za operacije iz različitih iteracija. To bi značilo da formiranjem novih iteracija narušavamo zavisnosti po podacima tako što u nekoj iteraciji θ postoji operacija zavisna po podacima od operacije iz neke kasnije iteracije θ + ω, ω > 0. Za □ω > n - 1, takva zavisnost ne može da postoji, jer je nova iteracija formirana od n delova n starih iteracija i početna iteracija je bila semantički ispravna. Dakle, samo za ω < n, pretpostavka da postoji takva transformacija može da postoji. Ako je operacija u iteraciji θ zavisna po podacima od operacije u iteraciji θ + ω, tada ista zavisnost postoji i između operacija u iteracijama θ + ω + ϕ i θ + ϕ, gde je ϕ proizvoljan celi broj. Ako ϕ ima takvu vrednost da važi θ + ω + ϕ + 1 – n > 0 i θ + ϕ + 1 – n < 0, tada postoje zavisnosti po podacima koje polaze iz prvih iteracija petlje ili postpetlje i završavaju u predpetlji.

U daljem razmatranju će se koristiti simboli sa Sl. 7.13. Jedini izuzetak za prethodni stav, za sve vrednosti ϕ koje zadovoljavaju prethodne relacije i sve vrednosti θ, postoji ako operacija zavisna po podacima iz θ + ϕ + 1 – n pripada skupu operacija ekvivalentnih po telu petlje operacijama iz skupa I_{1n} (odnosno komplementu od $I_{11}+I_{12} + \ldots + I_{1\,n-1}$). U

tom slučaju ne postoje operacije ekvivalentne po telu petlje sa I_{1n} u predpetlji pa bar za predpetlju ne postoji narušavanje PTP restrikcija. Sve takve operacije I_{xn} javljaju se za n-1 originalnih iteracija u postpetlji. Posmatrajmo za taj slučaj šta se događa u postpetlji. Zavisnosti po podacima ka tim operacijama moraju postojati od operacije iz postpetlje, izuzev ako zavisnost polazi iz operacija ekvivalentnih po telu petlje sa I_{11}, jer se te operacije ne postoje u postpetlji. Kako izvršavanje I_{x1} prethodi izvršavanju I_{xn} za n-1 novu iteraciju, a za prvu (a samim tim i ostale) iteracije je bilo moguće izvršiti I_{1n} posle I_{11}, sledi da takva zavisnost može da postoji samo za veličinu $\omega > n - 1$. Kako smo već pokazali da samo za $\omega < n$ može da postoji zavisnost, pokazano je da ne može da postoji transformacija koja nije semantički ispravna, a da zadovoljava PTP restrikciju. Sledi da su sve semantički ispravne iteracije one koje ne narušavaju PTP restrikcije.

7.6. Ograničenja u brzini izvršavanja za DoAcross petlje

Prvo će se posmatrati jednostavniji slučaj ciklusa u grafu petlje, kada postoji na ciklusu samo jedna grana sa iteracionom distancom različitom od 0, i to sa iteracionom distancom 1. Podsetimo se da tada važi Lema 7.1.

Teorema 2: Za graf petlje u kome sve petljom prenesene zavisnosti (zavisnosti koje prelaze granicu iteracije) imaju iteracionu distancu 1 i na svakom ciklusu postoji samo jedna takva grana, vreme izvršavanja ne može da bude kraće od najdužeg puta duž ciklusa izraženog u mašinskim ciklusima.

Dokaz: Na osnovu Leme 1, ako dve operacije ekvivalentne po telu petlje pripadaju susednim iteracijama za polaznu petlju i istovremeno pripadaju bar jednom usmerenom kolu sa RUKI jednako 1 koje ih povezuje, one takođe moraju pripadati susednim iteracijama u istom redosledu u transformisanom telu petlje. Kako zatvoreni put u grafu petlje označava put koji povezuje dve po telu ekvivalentne operacije iz susednih iteracija, jedna i samo jedna iteraciona distanca na tom putu ima zavisnost sa iteracionom distancom jednakom 1. Ukoliko to ne bi bilo tačno, narušili bi restrikciju integriteta iteracije. Transformacijom iteracije primenom softverske protočnosti, iteraciona distanca tog puta mora ostati ista, a samo se menja grana (zavisnost po podacima) na kojoj se javlja iteraciona distanca jednaka 1. Samim tim, podgraf acikličkog grafa transformisane iteracije koji odgovara ciklusu u grafu petlje sa RUKI = 1 imaće istu dužinu kao i ciklus grafa u broju mašinskih ciklusa. Kritičan put u acikličkom grafu zavisnosti po podacima može biti jednak ili veći od najdužeg od svih ciklusa u grafu DoAcross petlje, za cikluse koje imaju RUKI = 1. Znači da vreme izvršavanja transformisane petlje ne može da bude kraće od najdužeg ciklusa sa RUKI = 1. □

Teorema 2 analizira samo specijalnu kategoriju programskih petlji i uvodi za njih restrikciju koja proizilazi iz ciklusa u grafu petlje. U daljem tekstu je Teorema 2 generalizovana na opšti slučaj.

Teorema 3: Suma iteracionih distanci svih zavisnosti po podacima na bilo kom ciklusu grafa petlje mora ostati konstantna i jednaka sumi iteracionih distanci grafa originalne (stare) petlje. Najveći broj grana na svakom ciklusu koje imaju iteracione distance različite od 0 jednak je iteracionoj distanci ciklusa.

Dokaz: Na osnovu Leme 2 sledi da dve operacije ekvivalentne po telu petlje koje pripadaju ciklusima u grafu petlje moraju da imaju iste iteracione distance u originalnoj i transformisanoj petlji. Dakle, prvi deo teoreme direktno sledi iz Leme 2. Pretpostavimo da postoji ciklus sa iteracionom distancom dužine RUKI1. Kako ne sme da bude narušen integritet bilo koje iteracije između dve po telu ekvivalentne operacije na razmaku RUKI1, sledi da put može da ima operacije u svakoj novoj iteraciji između krajnjih iteracija tog puta. U tom slučaju će postojati tačno RUKI1 zavisnosti po podacima koje imaju iteracione distance jednake 1. Ukoliko u bilo kojoj od usputnih iteracija ne postoji operacija na tom putu, bar neka od zavisnosti po podacima će imati iteracionu distancu veću od 1. Zbog pojave takvih grana i Leme 2, ukupan broj grana sa iteracionom distancom tada je manji od RUKI1. Dakle maksimalan broj grana sa iteracionom distancom različitom od nule je upravo RUKI1. Kako nije uvedena nikakva pretpostavka u postupku dokazivanja za RUKI1, ovo važi za svaku vrednost RUKI1. Time je dokazan i drugi deo Teoreme 3. □

Iz rezultata Teoreme 3 proizilazi generalno ograničenje za brzinu izvršavanja iteracije DoAcross petlje.

Definicija: Efektivna Dužina Ciklusa r u grafu zavisnosti po podacima petlje (EDCr) je funkcija plafona količnika CLr i SIDr, pri čemu je CLr dužina ciklusa r u mašinskim ciklusima, a SIDr suma iteracionih distanci svih zavisnosti po podacima duž ciklusa r.

Funkcija plafona nekog prirodnog broja jednaka je tom prirodnom broj. Ako je broj racionalan pozitivan broj i nije ceo, funkcija plafona je prvi veći prirodni broj (zaokruživanje uvek na gore). Ova funkcija se prikazuje pomoću oznaka $\lceil\ \rceil$. Odatle sledi i izraz:

$$EDCr = \lceil CLr/SIDr \rceil$$

Teorema 4: Vreme izvršavanja iteracije DoAcross petlje ne može da bude kraće od najveće vrednosti EDCr za sve cikluse u grafu zavisnosti po podacima petlje.

Dokaz: Iz Teoreme 3, svaki ciklus r u grafu zavisnosti po podacima petlje mora da ima sumu iteracionih distanci svih grana SIDr jednaku sumi iteracionih distanci na tom ciklusu za početnu petlju, ako je semantički ispravna iteracija dobijena softverskom protočnošću. Ako je ta suma za neki ciklus SIDr, taj ciklus može da se u acikličkom grafu jedne iteracije pojavi kroz najviše SIDr proizašlih nepovezanih puteva. Proizašli putevi iz ciklusa r su obeleženi sa $CPPr_t$. Ako se razmatraju samo zavisnosti po podacima u ciklusu, tada bi ti putevi $CPPr_t$ bili nepovezani u acikličkom grafu jedne iteracije. Minimalno vreme izvršavanja iteracije, kao posledica posmatranja tog jednog ciklusa je ograničeno dužinom najvećeg $CPPr_t$, jer to postaje kritičan put proizašao iz tog ciklusa. Vrednosti $CPPr_t$ mogu da budu najmanje, ako se na ciklusu javi upravo SIDr grana sa iteracionom distancom 1. Na taj način je fiksna dužina ciklusa u mašinskim ciklusima podeljena na najveći broj (tačno SIDr) puteva $CPPr_t$. Uvedimo KPCPPr oznaku za najduži (kritičan) put među $CPPr_t$ putevima.

$$KPCPPr = \max \text{ put } (CPPr_t)$$

Cilj je da KPCPPr bude minimalno da bi ograničenje brzine izvršavanja iteracije zbog ciklusa r bilo minimalno. Ako postoji tačno SIDr puteva i oni su svi međusobno jednaki, KPCPPr ima minimalnu vrednost za ciklus r. Međusobnu jednakost nije moguće postići zato što dužine svih $CPPr_t$ puteva moraju da budu celobrojne vrednosti. Zato se uslov o međusobnoj jednakosti svih puteva $CPPr_t$ može sprovesti samo ako je CLr deljivo sa SIDr. Ako CLr nije deljivo, tada težimo da KPCPPr dobije minimalnu vrednost tako što će najduži $CPPr_t$ biti prva veća celobrojna vrednost od vrednosti količnika CLr/SIDr. Oba slučaja su upravo predstavljena sa EDCr. U slučaju kada količnik nije prirodan broj, postojaće bar jedan put $CPPr_t$ koji će za jedan ciklus biti kraći od najdužeg $CPPr_t$. Dakle EDCr definiše minimalan KPCPPr za neku iteraciju. Ako se odredi maksimalan EDCr za sve cikluse u grafu DoAcross petlje, on ograničava najbrže izvršavanje acikličkog grafa zavisnosti po podacima jedne iteracije. Ovim je dokazana Teorema 4. □

Teorema 2 je samo specijalan slučaj Teoreme 4, kada je broj grana sa iteracionim distancama različitim od 0 na svim ciklusima jednaka 1 i kada su sve iteracione distance petljama prenesenih zavisnosti po podacima jednake 1.

Posmatrajmo primere kojim je ilustrovano važenje Teoreme 4.

Primer 7.4.:
Kôd petlje je isti kao u primeru 6.1., a aciklički graf razmotane petlje i graf petlje dati su na Slici 7.1. i 7.2. Na ovom mestu ponovljen je graf petlje i aciklički graf jedne iteracije. Graf petlje ima četiri ciklusa. Označimo kao prvi ciklus r=1 onaj kojim se zatvara put Op1, Op2, Op3, Op1 (nebitno je koja je prva navedena operacija). Za taj ciklus je SID1 = 1, a CL1 je jednak 5. Kao posledica toga EDC1 = CL1/SID1 = 5. Dakle taj ciklus generiše jedan put $CPP1_1$ dužine 5 mašinskih ciklusa. Drugi ciklus r=2 se zatvara operacijama Op1, Op2, Op3, Op5, Op1. Za taj ciklus je SID2 = 2, a ciklus CL2 = 6. Kao posledica toga EDC2 = CL2/SID2 = 3. Dakle taj ciklus može da generiše dva puta $CPP2_1$ i $CPP2_2$, ali u kôdu koji je programer napisao, postoji samo jedan put $CPP2_t$, jer je iteraciona distanca zavisnosti od operacije Op5 do operacije Op1 dve iteracije. Taj jedini put proizašao iz ciklusa 2 zato ima dužinu 6 u acikličkom grafu jedne iteracije. Na takav put se nadovezuje grana zbog zavisnosti operacije 6 od operacije 5, pa je kritičan put polazne iteracije upravo 7 ciklusa. Kada bi se iteracija transformisala tako da ovaj ciklus dobije dva puta $CPP2_t$, tada bi se ukupna dužina CL2 = 6 raspodelila na $CPP2_1$ i $CPP2_2$ i moglo bi u idealnom slučaju da se podjednako raspodeli. U tom slučaju bi se dobilo EDC2 = CL2/SID2 = 3 kao minimalna dužina kritičnih puteva proizašlih iz ciklusa 2.

Ciklus 3 čine samo dve operacije Op2, Op3 i ponovo Op2. Za taj ciklus je SID3 = 3, a ciklus CL3 = 4. Kao posledica toga EDC3 = $\lceil CL3/SID3 \rceil$ = $\lceil 4/3 \rceil$ = 2. Ovde se po prvi put radi zaokruživanje na prvi veći prirodan broj, kao posledica činjenice da CL3 nije deljivo sa SID3.

Dakle taj ciklus može da generiše tri puta $CPP3_1$,$CPP3_2$, i $CPP3_3$, ali u kôdu koji je programer napisao, postoji samo jedan put $CPP3_t$, jer je iteraciona distanca zavisnosti od operacije Op2 do operacije Op3 tri iteracije. Taj jedini put proizašao iz ciklusa 3 zato ima dužinu 4 u acikličkom grafu jedne iteracije. Kada bi se iteracija transformisala tako da ovaj ciklus dobije tri puta $CPP3_t$, tada bi se ukupna dužina CL3 = 4 raspodelila na $CPP3_1$, $CPP3_2$, i $CPP3_3$. Tada se u cilju postizanja približno podjednakih dužina proizašlih puteva mora napraviti raspodela tako da dva od $CPP3_t$ budu dužine 1, a jedan

bi morao da bude dužine 2. Ovde se na primeru i vidi funkcija plafona kod određivanja EDC, kao posledica celobrojnosti trajanja puteva unutar iteracije u ciklusima.

Poslednji ciklus označićemo kao ciklus 4 i on sadrži samo operaciju Op4, jer je ona zavisna sama od sebe. U njenom slučaju je SID4 = 1, a ciklus CL4 = 1. Kao posledica toga EDC4 = CL4/SID4 = 1. U ovom slučaju u svakoj iteraciji se generiše samo jedan put $CPP4_1$ koji je ustvari samo operacija Op4.

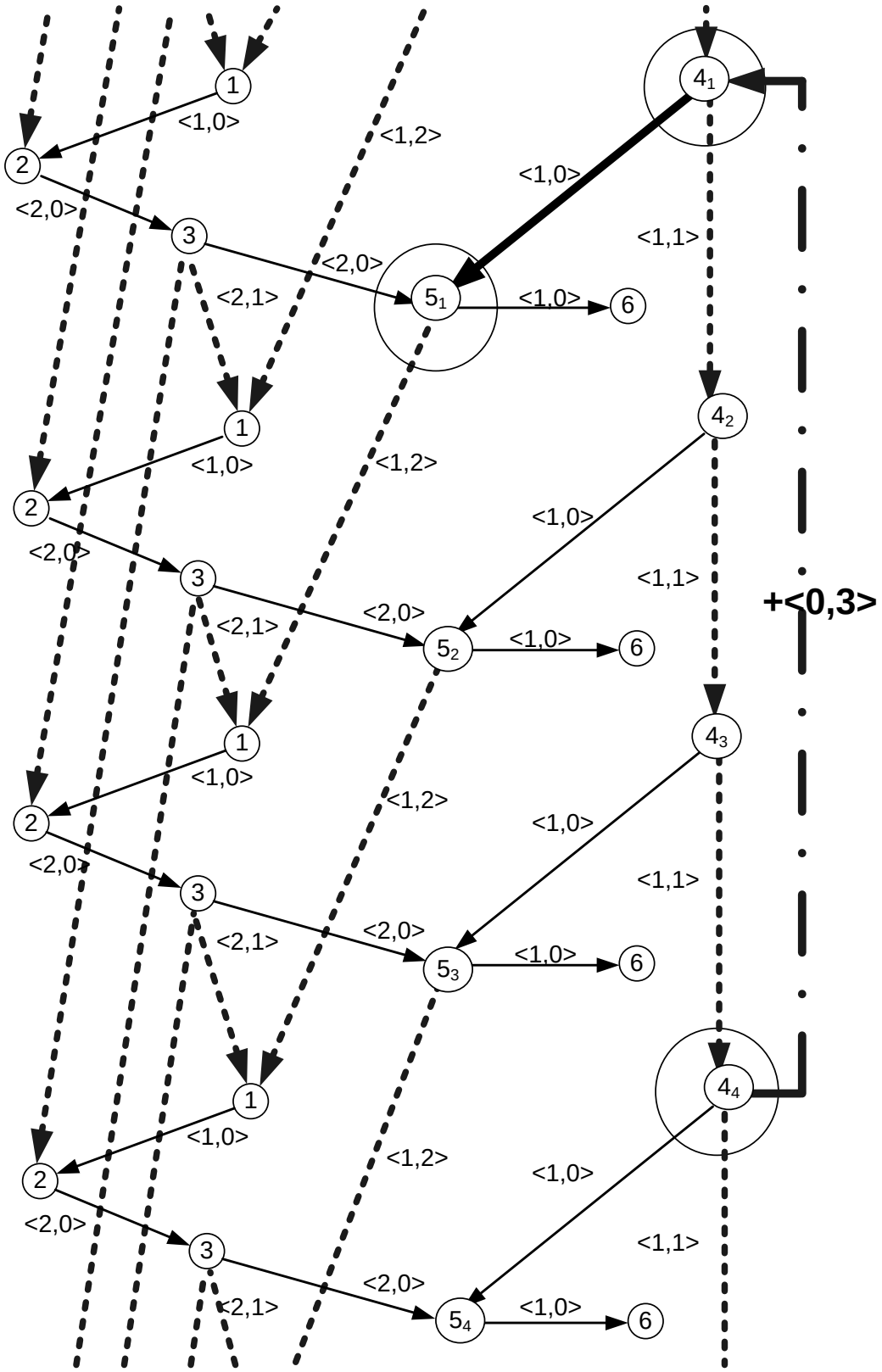
Maksimalan EDCr za sve cikluse je EDC1, pa on predstavlja donji limit za trajanje iteracije u ciklusima. Njegova vrednost od 5 ciklusa upravo se poklapa sa asimptotskom vrednošću dobijenom kod razmotavanja petlje i prikazanom na Slici 7.6.

Kada se sada prikaže graf petlje za transformisano telo petlje kao na Slikama 7.8. i 7.11., može se uočiti da je ciklus 2 upravo podeljen na dva $CPP2_t$ puta u grafu jedne iteracije, pa shodno tome se postiže veći paralelizam iteracije (jedna grana iz ciklusa 2 je uklonjena iz acikličkog grafa iteracije i dužina puteva $CPP2_t$ je sigurno manja nego kada je dužina celog ciklusa pretvorena u $CPP2_1$. U tim slučajevima je kritičan put u acikličkom grafu jedne iteracije upravo bio jednak dužini puta $CPP1_1$, čime je dobijeno minimalno vreme izvršavanja iteracije.

7.7. Optimizacija petlje kod softverske protočnosti

Navedeni primeri i niz teorema i lema pokazali su da se preko grafa petlje mogu prikazati sva ograničenja i transformacije prilikom formiranja nove iteracije. Neke od slučajno izabranih transformacija dovele su do optimalne iteracije po trajanju, čak i uz nivo softverske protočnosti jednak 1. Teoremom 4 definisan je ekvivalent kritičnog puta za DoAcross petlje, kao maksimalan EDCr. Za taj EDCr, svi putevi proizašli iz ciklusa r $CPPr_t$, t = 1, … , SIDr potencijalno su kritični, s tim da su neki od njih tačno veličine kritičnog puta, dok neki mogu biti kraći za jedan ciklus mašine. Sve navedene teoreme i ograničenja predstavljaju osnovu za traženje algoritma kojim će se naći optimalna iteracija. Posao traženja optimalne iteracije može se podeliti na dva podzadatka. Prvi je traženje iteracije koja generiše optimalan graf za aciklički graf jedne iteracije. Drugi se svodi na traženje optimalnog rasporeda za realnu mašinu za bazični blok koji predstavlja jednu optimalnu iteraciju (aciklički graf).

Prethodnim teoremama definisana su ograničenja vezana za transformaciju iteracije koja su poticala od ciklusa u grafu zavisnosti po podacima za petlje. Utvrđeno je da postoje ozbiljne restrikcije za paralelizaciju koje potiču pre svega od puteva proizašlih iz ciklusa. Uopšte nisu pominjana ograničenja koja se mogu javiti zbog zavisnosti po podacima (grana grafa) koje ne mogu biti na ciklusima (zatvorenim putevima). Početak te analize vezan je za Primer 7.5. i za jednu takvu granu. Posmatrajmo u grafu beskonačno razmotane petlje zavisnost po podacima operacije Op_5 od operacije Op_4. Ta zavisnost ispunjava uslov da **grana ne pripada nijednom ciklusu**.



Slika 7.15. Izračunavanje Op_4 ranije za tri iteracije u odnosu na Op_5

Primer 7.5.: Pretpostavimo da se posmatra samo relativan odnos operacija Op_{41} i Op_{51} i njihova iteraciona distanca u novoj iteraciji. Izbor programera je bio da te dve operacije budu unutar iste iteracije, a samim tim ta zavisnost postoji u acikličkom grafu zavisnosti po podacima jedne iteracije. Pretpostavimo i da se nova iteracija izabere tako da zavisnost po podacima operacija Op_4 bude razmaknuta za 3 iteracije od operacija Op_5 u kojoj se završava. To može da se uradi samo tako što će se u predpetlji prvo izračunati Op_4 za prve iteracije, a tek zatim ući u novu iteraciju u kojoj će se u istoj novoj iteraciji izračunavati Op_{44} iz četvrte stare iteracije i Op_{51} iz prve stare iteracije. Rezultati operacije Op_4 iz prve tri stare iteracije, a eventualno i iz četvrte stare iteracije moraju se sačuvati do trenutka dok nisu potrebne za Op_5 iz prve iteracije. Efektivno, događa se da mi rezultat operacije Op_4 izračunavamo nekoliko iteracija ranije, čime ta zavisnost od Op_4 ka Op_5 postaje petljom prenesena zavisnost (zavisnost po podacima operacija iz različitih iteracija) sa iteracionom distancom jednakom 3.

Nameće se pitanje da li se u novu iteraciju može uvesti takav razmak između operacija Op_4 i Op_5 bez narušavanja PTP restrikcija. Kako je Op_4 zavisila samo od same sebe, prerano izračunavanje Op_4 ne može da dovede do narušavanja PTP restrikcija. Čak se izračunavanje Op_4 moglo uraditi kompletno pre izvršavanja programske petlje, nezavisno od ostatka petlje. Time bi pocepali petlju na dve nezavisne petlje, kojima je određen redosled izvršavanja.

Uspostavljanjem takvog odnosa između operacija Op_4 i Op_5 da transformacijom uvedemo iteracionu distancu, zavisnost po podacima između Op_4 i Op_5 u acikličkom grafu jedne iteracije nestaje. Zanimljivo je da smo mogli da izaberemo proizvoljnu iteracionu distancu za zavisnost između Op_4 i Op_5. Ukoliko se želeo postići isti efekat eliminacije grane u acikličkom grafu jedne iteracije, uz minimalan porast veličine kôda prema izrazu 7.2., izabrali bi iteracionu distancu jednaku 1, jer bi time proizašli stepen softverske protočnosti bio minimalan.

Očigledno je iz primera 7.5. da sa aspekta paralelizacije kôda postoji izrazita razlika između zavisnosti po podacima (grana) na zatvorenim putevima i grana koje se ne mogu naći na zatvorenim putevima. Zato je potrebno koristiti delove teorije grafova koji mogu da pomognu u razdvajanju ove dve kategorije grana grafa.

7.7.1. Komponente jake povezanosti u grafu petlje

Definicije 7.4.: U orijentisanom grafu, čvor B je dohvatljiv iz čvora A ako postoji orijentisan put od A do B. Čvorovi A i B su međusobno dohvatljivi ako je B dohvatljivo iz A i A dohvatljivo iz B. Skupovi svih međusobno dohvatljivih čvorova su ***Komponente jake povezanosti*** (engl. *Strongly Connected Components* - SCC).

Manje formalna definicija glasi da svi parovi čvorova koji mogu da se nalaze na orijentisanom ciklusu pripadaju jednoj komponenti jake povezanosti. Za komponente jake povezanosti nadalje će se koristiti skraćenica SCC. Grane orijentisanog grafa koje polaze iz čvora jedne komponente jake povezanosti i završavaju u čvoru druge komponente jake povezanosti će se u daljem tekstu nazivati inter SCC grane.

Inter SCC grane ne mogu, na osnovu definicije, da pripadaju orijentisanim ciklusima. Ako stopimo sve čvorove SCC u jedan čvor, a zadržimo inter SCC grane, dobija se novi graf koji se naziva ***Graf parcijalnog uređenja komponenti jake povezanosti***. Grane tog

grafa, u slučaju grafova zavisnosti po podacima, znače da postoji zavisnost po podacima operacije iz jedne SCC (gde zavisnost završava) od operacije iz druge SCC (iz koje zavisnost polazi). Ovaj graf određuje eventualne redoslede izvršavanja SCC, ako se SCC izvršavaju kao celine, na osnovu zavisnosti po podacima za operacije iz različitih SCC. Kod ovog grafa, često se može dogoditi da više od jedne grane povezuje dva SCC čvora.

Graf parcijalnog uređenja komponenti jake povezanosti je aciklički graf. On određuje zavisnosti po podacima operacije(a) iz jedne komponente jake povezanosti od završetka operacije(a) iz druge komponente jake povezanosti. Ako postoji zavisnost po podacima neke operacije iz SCCq od neke operacije iz SCCp, koristićemo termin da je SCCq zavisna po podacima od SCCp. Kako dosadašnjim pravilima za grane na zatvorenim putevima nisu jasno definisana ograničenja za ovu vrstu zavisnosti po podacima, nameće se pitanje da li se te grane mogu eliminisati iz acikličkog grafa zavisnosti po podacima nove iteracije?

7.7.2. Promene iteracionih distanci za inter SCC grane grafa

Prvo će se posmatrati slučaj kada postoje dve SCC, SCCp i SCCq, i postoji više operacija u SCCq koje su zavisne po podacima od operacija iz SCCp. Pretpostavka je takođe da su te zavisnosti po podacima sa različitim iteracionim distancama. Izaberimo prozor nove iteracije prilikom formiranja, tako da sve operacije u SCCp iz originalne (stare) iteracije i + j istovremeno uđu u prozor sa svim operacijama iz SCCq iz iteracije i. Tada se rezultati svih operacija iz SCCp izračunaju najmanje j novih iteracija pre nego što su potrebni kao argumenti za operacije iz SCCq, jer je potrebno j novih iteracija da prozor stigne do operacija SCCq iz iteracije i + j. Dakle, zavisnost po podacima između ovih SCC navedenom transformacijom dobija najmanje iteracionu distancu j. Za iteracione distance svih inter SCC grana između SCCp i SCCq originalne iteracije definisane od strane programera, ovakvo pomeranje komponenti jake povezanosti za j iteracija dovodi do povećanja iteracionih distanci svih inter SCC grana za j. Ovo razmatranje je generalizacija primera 7.5. U tom primeru su operacije Op_4 i Op_5 bile operacije iz dve komponente jake povezanosti i upravo zato se mogla povećati iteraciona distanca za zavisnost po podacima između tih operacija, razmicanjem operacija Op_4 i Op_5 po iteracijama.

Lema 7.6.: Ako je SCCq zavisno po podacima od SCCp, izvršavanje SCCp j iteracija ranije je semantički ispravna transformacija, a svim SCC granama između SCCp i SCCq se iteraciona distanca povećava za j.

Dokaz: Drugi deo Leme 7.6. je već dokazan, tako da je neophodno samo dokazati semantičku ispravnost izvršavanja SCCp za j iteracija ranije. Na osnovu teoreme 1, nova iteracija je semantički ispravna ako nisu narušene PTP restrikcije. Izvršavanjem SCCp za j iteracija ranije, inter SCC zavisnosti između SCCp i SCCq će se pojavljivati kao:

a. zavisnosti iz ranijeg dela predpetlje u kasniji deo predpetlje (ove zavisnosti se mogu javiti ako je $j < n - 1$)
b. zavisnosti iz predpetlje ka operacijama u novim iteracijama.
c. zavisnosti iz nove iteracije ka postpetlji
d. zavisnosti iz ranijeg dela postpetlje u kasniji deo postpetlje (ove zavisnosti se mogu javiti ako je $j < n - 1$)
e. zavisnost iz predpetlje ka postpetlji (ove zavisnosti se mogu javiti ako je broj novih iteracija k mali).

Kako nijedna od ovih zavisnosti ne narušava PTP restrikcije, dokazana je semantička ispravnost. □

Prilikom dokazivanja Leme 7.6. nisu bile uvedene nikakve pretpostavke o SCCp i SCCq, a j je bio proizvoljan prirodan broj. Kako je j bar 1, sve inter SCC zavisnosti po podacima između SCCp i SCCq će imati iteracionu distancu od najmanje 1. To znači da se neće pojaviti u acikličkom grafu zavisnosti po podacima nove iteracije. Ova eliminacija grane iz acikličkog grafa jedne iteracije potencijalno može da dovede do kraćeg kritičnog puta i bržeg izvršavanja nove iteracije. Izvršavanje SCCp za j iteracija pre komponente jake povezanosti SCCq je nazvano protočnost komponenti jake povezanosti {JO 91}.

Teorema 5: Primenom protočnosti komponenti jake povezanosti, sve inter SCC grane grafa petlje mogu da dobiju iteracionu distancu od najmanje 1 i da time nestanu iz acikličkog grafa zavisnosti po podacima nove iteracije.

Dokaz: Na osnovu Leme 3, protočnost proizvoljne dve SCC može se uraditi tako da minimalna iteraciona distanca inter SCC komponenti može da bude proizvoljno velika. Aciklički graf parcijalnog uređenja komponenti jake povezanosti sa druge strane ima najduži put konačne veličine i dužinu puta označimo sa L. Pritom se dužina najdužeg puta L definiše kao broj SCC čvorova na tom putu. Da bi sve inter SCC grane dobile iteracionu distancu od bar 1, za SCC na najdužem putu je potrebno pomeranje susednih SCC bar za jednu iteraciju. Tada se SCC sa najdužeg puta L u novoj iteraciji raspoređuju tako da prozor nove iteracije čine SCC raspoređene na L susednih iteracija originalne (stare) petlje. SCC na vrhu acikličkog grafa parcijalnog uređenja komponenti jake povezanosti se pritom raspoređuje u najkasniju inicijalnu iteraciju, jer je operacije iz te SCC potrebno izračunati najranije. Svakoj SCC sa najdužeg puta pridružuje se posebna inicijalna iteracija. Tako sve inter SCC grane na najdužem putu L dobijaju iteracione distance od bar 1, na osnovu Leme 3. Za SCC koje nisu na najdužem putu, uvek se može napraviti raspored da sve inter SCC grane između njih i ostalih SCC mogu da imaju iteracionu distancu od najmanje 1. Na osnovu Leme 3 ove transformacije petlje su semantički ispravne za svaki par SCC, pa su samim tim ispravne i za celu novu iteraciju. □

Moglo se dogoditi da je programer napisao kôd tako da sve grane između dve SCC budu inicijalno sa iteracionom distancom. Tada se ne mora raditi razmicanje izvršavanja tih komponenti jake povezanosti. U dokazu je pretpostavljeno da sve grane na kritičnom putu imaju iteracionu distancu 0 (najčešći slučaj), ali to nikako ne umanjuje generalnost dokaza.

7.7.3. Paralelizam u acikličkom grafu jedne iteracije kada je primenjena protočnost komponenti jake povezanosti

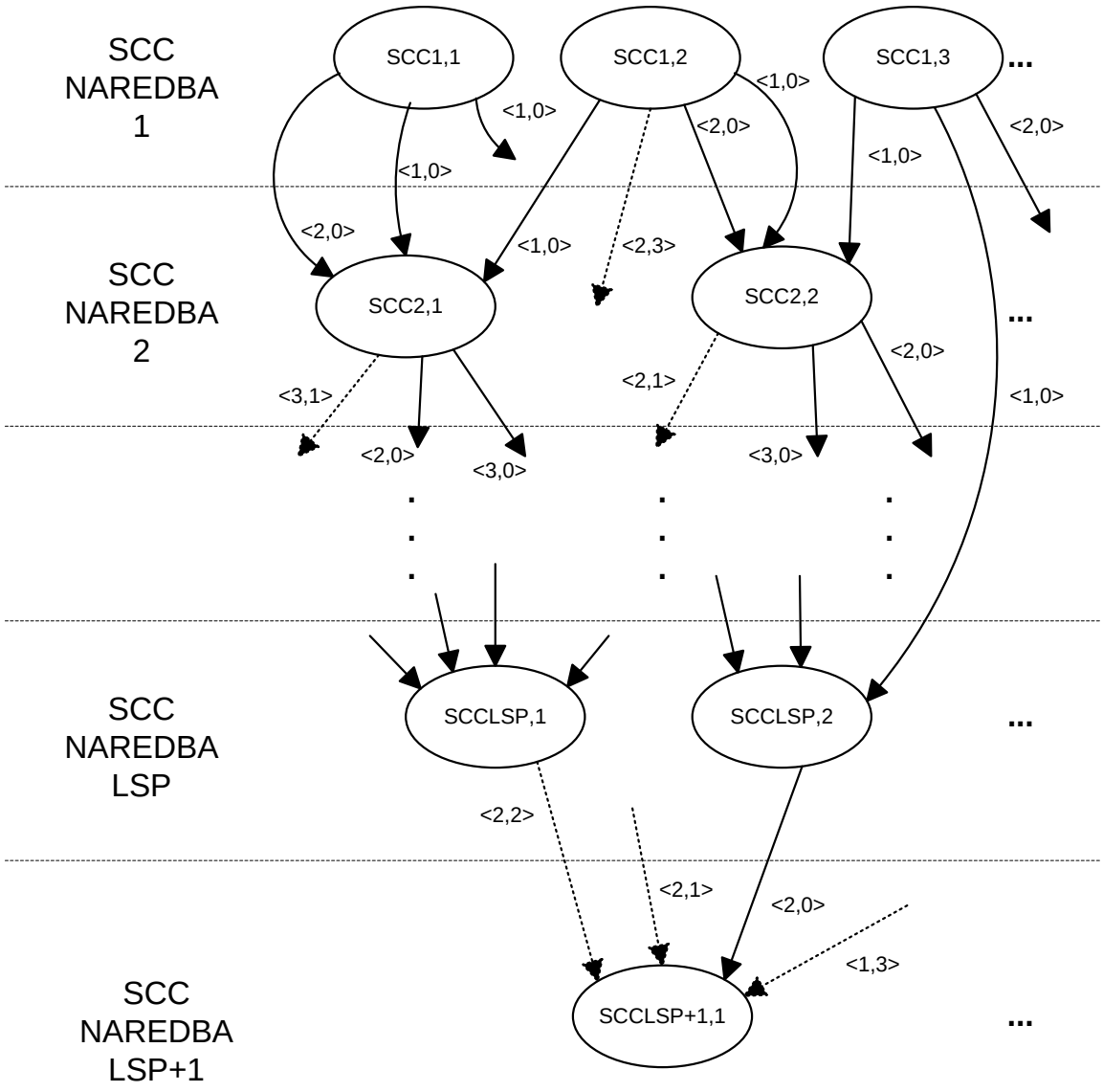
U acikličkom grafu jedne iteracije, ako se primeni protočnost komponenti jake povezanosti, minimalan paralelizam jednak je ili veći od ukupnog broja komponenti jake povezanosti. To je posledica činjenice da svi delovi grafa proizašli iz komponenti jake povezanosti, kao posledica povećanja iteracionih distanci grana između komponenti jake povezanosti moraju da budu međusobno nepovezani. Iteracione distance zavisnosti po podacima za sve grane između komponenti jake povezanosti mogu se učiniti proizvoljno velikim, mada suvišno udaljavanje dovodi do eksplozije kôda čak i u slučaju kada se

unapred zna da će se izvršiti bar LSP + 1 iteracija. Udaljavanje komponenti jake povezanosti za veći broj iteracija dovodi do povećanja predpetlje i postpetlje, a posebno mimoilazećeg kôda ako se broj iteracija petlje ne zna u vreme prevođenja.

Udaljavanjem komponenti jake povezanosti između kojih postoji zavisnost po podacima za iteracione distance veće od 1, potrebno je da se rezultat generisan u jednoj komponenti jake povezanosti mora sačuvati jednu ili više iteracija dok se ne iskoristi od strane operacije iz komponente jake povezanosti koja je bila zavisna od tog rezultata. Kako se u narednim iteracijama generišu novi rezultati, proizilazi da za takve rezultate treba predvideti FIFO u kome će se čuvati rezultati do trenutka kada će biti iskorišćeni. Ako grana polazi iz SCCp i završava u SCCq i ako se iteraciona distanca poveća sa 0 na m, tada FIFO mora da ima m-1 ili m lokacija za čuvanje rezultata iteracija za promenljivu zbog koje postoji zavisnost. Da li je neophodno m-1 ili m lokacija zavisi od krajnjeg rasporeda dobijenog optimizacijom kôda acikličkog grafa nove iteracije. Ako rezultat zbog kojeg nastaje zavisnost po podacima za operacije iz različitih iteracija predstavlja element niza, tako da se u svakoj novoj iteraciji upisuje nov element niza, ne mora da bude neophodno da se realizuje FIFO.

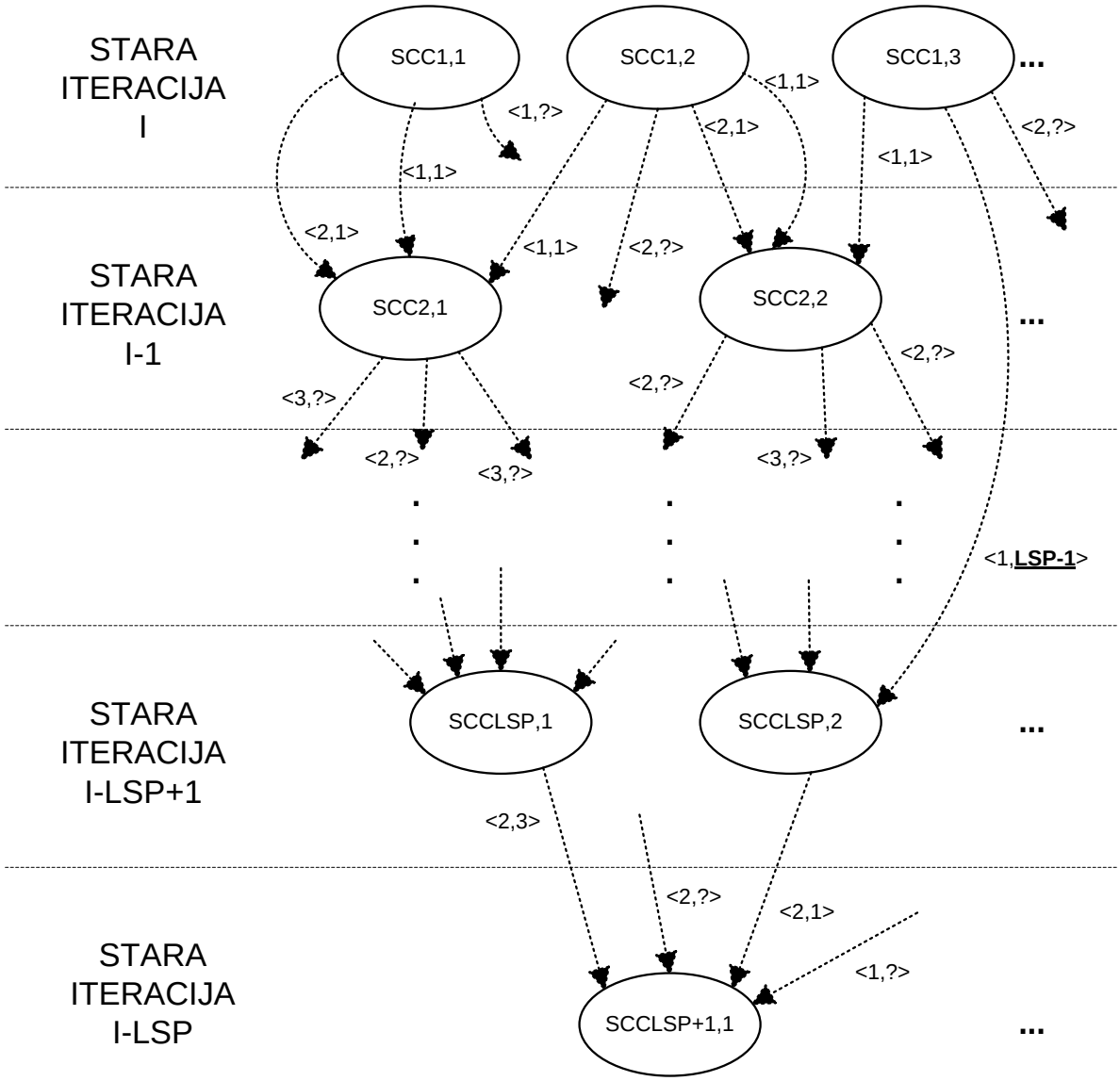
U cilju minimizacije eksplozije kôda i smanjenja broja potrebnih FIFO lokacija, u daljem tekstu je pretpostavljeno da se radi minimalno udaljavanje komponenti jake povezanosti i to samo kada je to potrebno. Neophodno je raditi udaljavanje kada postoji bar jedna grana između SCC koja ima iteracionu distancu 0. Tako će se u acikličkom grafu nove iteracije izgubiti sve grane koje su između komponenti jake povezanosti. U tom slučaju, broj originalnih iteracija od čijih operacija se formira nova iteracija mora da bude jednak najdužem putu u acikličkom grafu parcijalnog uređenja komponenti jake povezanosti. Pritom se dužina puta može izražavati preko broja komponenti jake povezanosti na njemu. Kada se koristi oznaka za stepen softverske protočnosti LSP (LSP jednak dužini kritičnog puta) tada je broj originalnih iteracija iz kojih formiramo novu iteraciju je LSP + 1. Ovo naravno važi pod uslovom da su iteracione distance svih grana na najdužem putu u acikličkom grafu parcijalnog uređenja komponenti jake povezanosti bile jednake 0.

Ukoliko to nije tačno, dužina puta u inicijalnom grafu umanjuje se za sumu vrednosti minimalnih iteracionih distanci grana između SCC komponenti na putu i za tako modifikovane dužine puteva traži se najduži (kritičan) put. U opštem slučaju, u grafu parcijalnog uređenja komponenti jake povezanosti, prilikom traženja najdužeg puta, mora se dakle, na specifičan (modifikovan) način tražiti dužina puta. Razlog za uvođenje modifikacije je činjenica da je i sam programer mogao inicijalno da napravi iteraciju kod koje postoji iteraciona distanca veća od nule za grane između operacija iz različitih komponenti jake povezanosti. Modifikovana dužina puta se ovom slučaju dobija tako što se ukupna dužina puta umanjuje za sumu iteracionih distanci grana na tom putu (pretpostavka je da postoji samo jedna grana između SCC na putu). Naravno da se minimalna potrebna vrednost LSP koja u opštem slučaju dovodi do eliminacije grana između komponenti jake povezanosti mora da proizilazi iz modifikovane, a ne prave dužine puta u grafu. Zbog lakše analize, a bez gubitka generalnosti, u daljem tekstu je pretpostavljeno da grane na kritičnom putu između komponenti jake povezanosti imaju iteracionu distancu 0. Kritičan put pritom nije mogao biti nacrtan na Sl. 7.16., jer je dat najopštiji slučaj.



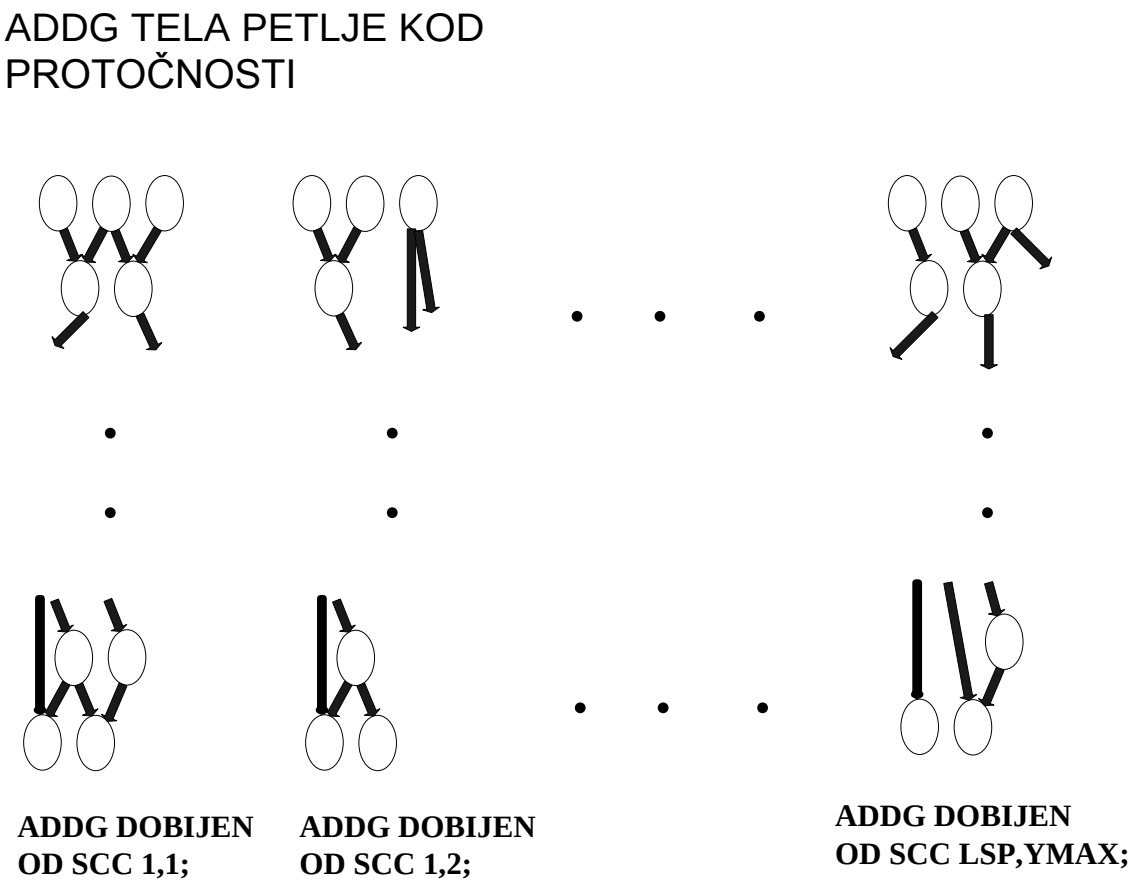
Slika 7.16. Graf parcijalnog uređenja komponenti jake povezanosti za originalnu (polaznu) iteraciju

Posmatrajmo sliku 7.16. kojom je predstavljen graf parcijalnog uređenja komponenti jake povezanosti. On je proizašao iz MDDG grafa petlje, pa su sve operacije ekvivalentne po telu petlje predstavljene samo jednim čvorom. U realnoj petlji bi tipično bilo mnogo manje komponenti jake povezanosti, a primer na Sl. 7.16. dat je za najgeneralniji slučaj. Najduži put u grafu ima dužinu LSP+1 i sve grane na tom putu imaju iteracionu distancu 0. Ovom prilikom se ne razmatra raspoređivanje operacija unutar komponenti jake povezanosti, već samo raspoređivanje celokupnih komponenti jake povezanosti u novoj iteraciji. Sve komponente jake povezanosti na najdužem putu u grafu moraju se rasporediti u odgovarajuće originalne (stare) iteracije, prilikom formiranja novog prozora iteracije. Ako se posmatra prvih LSP+1 originalnih (starih) iteracija, prilikom formiranja prozora, SCC na početku najdužeg puta (npr. SCC1,3 na slici 7.16.) se prilikom prvog ulaska u novu iteraciju bira tako da bude iz stare iteracije LSP+1. Analogno tome, komponenta jake povezanosti SCCLSP+1,1 će biti izabrana iz stare iteracije 1.



Slika 7.17. Protočnost komponenti jake povezanosti – razmicanje SCC naredbi za po jednu originalnu (polaznu, staru) iteraciju kod formiranja nove iteracije

Za sve ostale komponente jake povezanosti koje nisu na kritičnom putu postoji više inicijalnih iteracija iz kojih se može selektovati prozor za novu iteraciju. Najranije izvršavanje svih komponenti jake povezanosti se naziva najranijim SCC rasporedom. Prilikom predstavljanja komponenti jake povezanosti na slici 7.16. implicitno je pokazan upravo najraniji raspored (early SCC partition), povlačenjem horizontalnih linija. Ako se izabere takav raspored, predpetlja će biti najveća, a postpetlja najmanja za minimalnu veličinu softverske protočnosti koja eliminiše sve inter SCC grane iz acikličkog grafa jedne iteracije. Potrebno je uočiti da se izborom najranijeg rasporeda ne dobija minimalna veličina FIFO memorija potrebnih za čuvanje međurezultata zbog kojih postoje zavisnosti po podacima za operacije iz različitih komponenti jake povezanosti (ovde i iteracija). Nasuprot ovome mogao se izabrati drugi raspored – najkasnije izvršavanje. Tom rasporedu bi odgovarala najmanja predpetlja, ali najveća postpetlja, a samim tim i najmanja potrebna FIFO memorija za čuvanje međurezultata.



Slika 7.18. Aciklički graf jedne iteracije nakon razmicanja komponenti jake povezanosti – broj nepovezanih komponenti je veći ili jednak ukupnom broju SCC

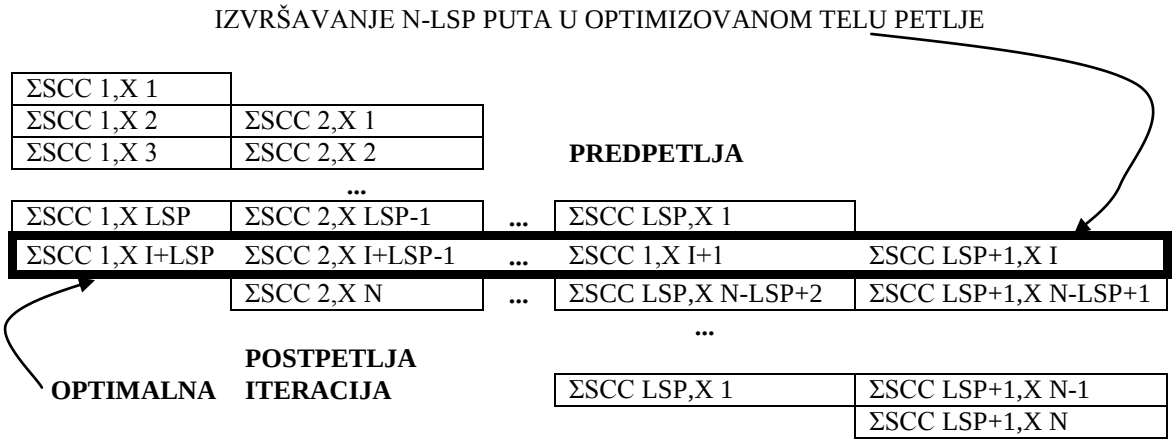
Ako koristimo analogiju sa slobodnim skupom operacija, kod acikličkih grafova zavisnosti po podacima za bazične blokove, tada su sve SCC1,i komponente jake povezanosti slobodne na vrhu za izvršavanje i one se prve i izvršavaju. Na idealnoj mašini, one se mogu sve izvršavati u paraleli i opet po analogiji, koristiće se termin kompletna SCC naredba. Grupa komponenti jake povezanosti koja se prilikom formiranja nove iteracije preslikava iz iste originalne (stare) iteracije naziva se SCC naredba. Dakle SCC naredbe na Slici 7.16 su ustvari SCC naredbe najranijeg rasporeda. Oznake na slici za komponente jake povezanosti su SCCx,y pri čemu:
x označava SCC naredbu kojoj pripada komponenta jake povezanosti pri najranijem rasporedu, a y označava redni broj komponente jake povezanosti unutar SCC naredbe.

Kada se na takav graf primeni protočnost komponenti jake povezanosti, tako da se duž najdužeg puta umeću najmanje iteracione distance – veličine 1, tada aciklički graf parcijalnog uređenja komponenti jake povezanosti izgleda kao na slici 7.18. Slika ukazuje na nepovezanost grafova proizašlih iz svih SCCx,y. To se može posmatrati i kao umetanje iteracione distance 1 između susednih SCC naredbi. Sve inter SCC grane sada dobijaju iteracionu distancu 1 ili više i ne mogu se pojaviti u acikličkom grafu jedne iteracije. Zanimljivo je posmatrati iteracionu distancu grane koja polazi iz SCC1,3 i završava u SCCLSP,2. Ta grana zbog udaljavanja SCC naredbi dobija iteracionu distancu LSP-1, pa se za tu promenljivu mora uspostaviti FIFO veličine LSP-1 ili LSP,

zavisno od konačnog rasporeda pojedinih operacija na mašini dobijenih raspoređivanjem acikličkog grafa jedne iteracije.

U skladu sa uobičajenim načinom predstavljanja protočnosti i softverske protočnosti, kao i načina predstavljanja punjenja protočnog niza kod protočnosti, na slici 7.19. je data predstava rasporeda komponenti jake povezanosti, pod pretpostavkom da raspolažemo idealnom mašinom i da se ne rade nikakve transformacije unutar komponenti jake povezanosti. U prvom koraku uspostavljanja protočnosti raspoređuju se sve operacije komponenti jake povezanosti koje pripadaju SCC naredbi 1 iz prve iteracije. To je u osnovi deo prve iteracije I11 sa Slike 7.13. Zatim se u drugom koraku raspoređuju operacije iz komponenti jake povezanosti koje pripadaju SCC naredbi 1 iz druge originalne (stare) iteracije i operacije iz komponenti jake povezanosti koje pripadaju SCC naredbi 2 iz prve originalne (stare) iteracije. To su komponente I_{12} i I_{21} sa slike 7.13. Posle izvršavanja LSP puta svih operacija iz komponenti jake povezanosti iz SCC naredbe 1, LSP-1 puta svih operacija iz komponenti jake povezanosti iz SCC naredbe 2, ... , i izvršavanja svih operacija iz SCCLSP za prvu inicijalnu iteraciju, nova iteracija počinje da se izvršava i uključuje sve komponente jake povezanosti.

DO I=1,N N>LSP



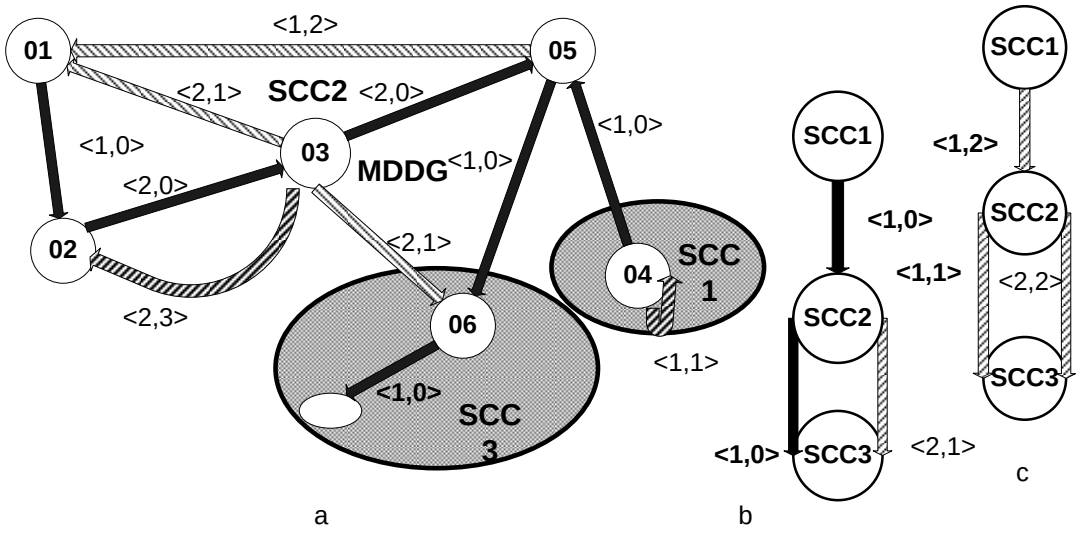
Sl. 7.19. Izvršavanje petlje kod protočnosti komponenti jake povezanosti – oznake kao na Sl. 7.17.

Izvršavanje svake komponente jake povezanost proizvodi jednu ili više nepovezanih delova u acikličkom grafu iteracije. Maksimalan broj tih nepovezanih komponenti proizašao iz jedne komponente jake povezanosti jednak je najvećoj sumi iteracionih distanci na zatvorenim putevima u toj komponenti jake povezanosti. Dakle aciklički graf iteracije ima najmanje onoliko nepovezanih delova koliko ima ukupno komponenti jake povezanosti. Taj paralelizam ilustrovan je slikom 7.18. Ukupan paralelizam naravno u tipičnom slučaju mnogo je veći, jer MDDG ima dva dodatna izvora paralelizma: nepovezane komponente kada postoje zatvoreni putevi sa sumom iteracionih distanci većom od 1 i paralelizam unutar svake nepovezane komponente proizašle iz grafa jedne komponente jake povezanosti.

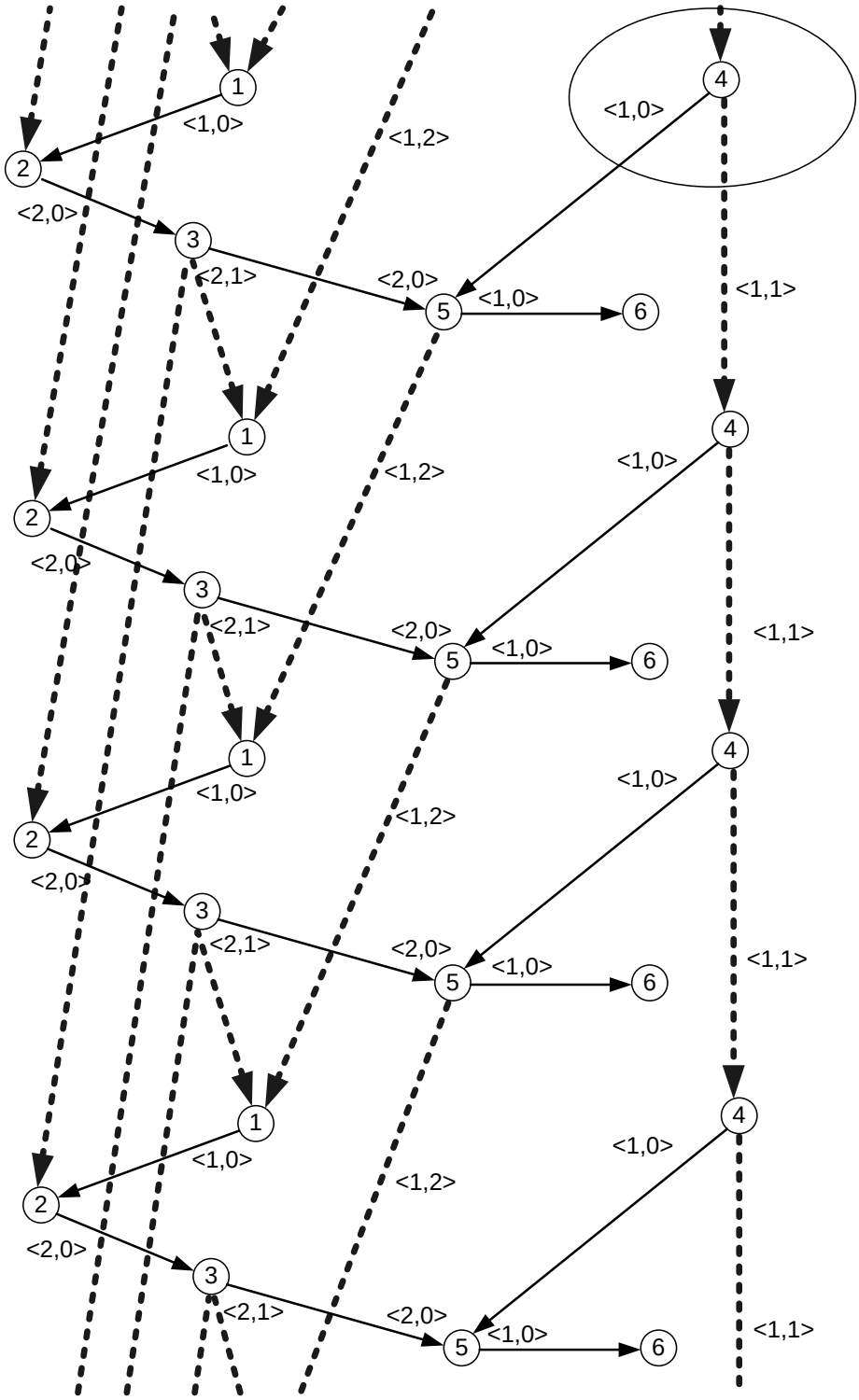
U teoriji grafova, svaki čvor orijentisanog grafa koji se ne nalazi na zatvorenom putu smatra se komponentom jake povezanosti. Prenoseći matematičku definiciju komponenti jake povezanosti na acikličke grafove zavisnosti po podacima za DoAll petlje, dobija se

zanimljiv rezultat primene protočnosti komponenti jake povezanosti. Praktično se ostvaruje paralelizacija svih operacija, kao najviši oblik paralelizacije kôda primenom softverske protočnosti, kada se istovremeno ne radi razmotavanje petlji.

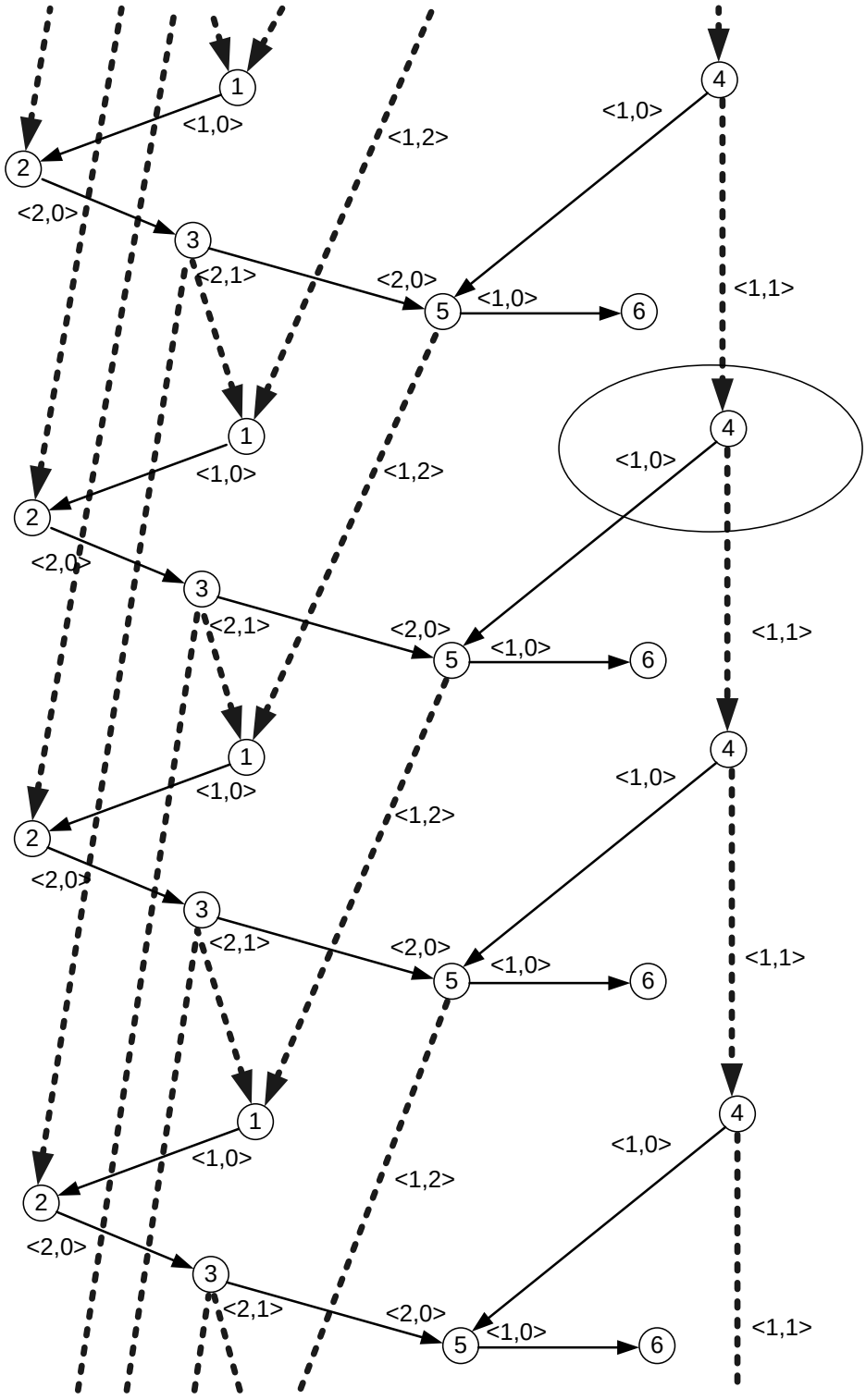
Primer 7.6.: Za programsku petlju čiji su kôd i graf zavisnosti po podacima dati na Slici 7.2. i na Slici 7.1. u formi razmotane petlje, primenjena je metoda protočnosti komponenti jake povezanosti. Primenom algoritama za određivanje komponenti dobija se da su komponente jake povezanosti SCC11 (operacija Op_4), SCC21 (operacije Op_1, Op_2, Op_3 i Op_5) i SCC31 (operacija Op_6). Graf parcijalnog uređenja komponenti jake povezanosti dat je na Slici 7.20b, ali su zbog jednostavnosti grafa parcijalnog uređenja komponenti jake povezanosti SCCxy zamenjene samo sa SCCx, jer y uvek ima vrednost 1. Izabrano je rešenje u kome se u novoj iteraciji uvode iteracione distance veće od 0 za sve inter SCC grane u grafu. Pritom je namerno izabrano rešenje u kome se komponente jake povezanosti ne razmiču minimalno, već je između SCC1 i SCC2 namerno napravljen razmak od dve iteracije. Novi graf parcijalnog uređenja komponenti jake povezanosti dat je na Sl. 7.20.c. Takvim primerom demonstrirano je da se mogu proizvoljno razmicati komponente jake povezanosti. Slikama 7.21., 7.22. i 7.23. su prikazane sve faze u predpetlji. Na Slici 7.24. prikazan je prozor nove iteracije koji se sastoji iz »ostrva« koja potiču od komponenti jake povezanosti. Zatim je na slici 7.25. prikazan kôd predpetlje, nove iteracije i postpetlje, zajedno sa novim acikličkim grafom jedne iteracije. Na kraju, na Slici 7.26. je prikazana transformacija grafa petlje, grafa parcijalnog uređenja komponenti jake povezanosti i acikličkog grafa iteracije posle transformacije protočnosti SCC.



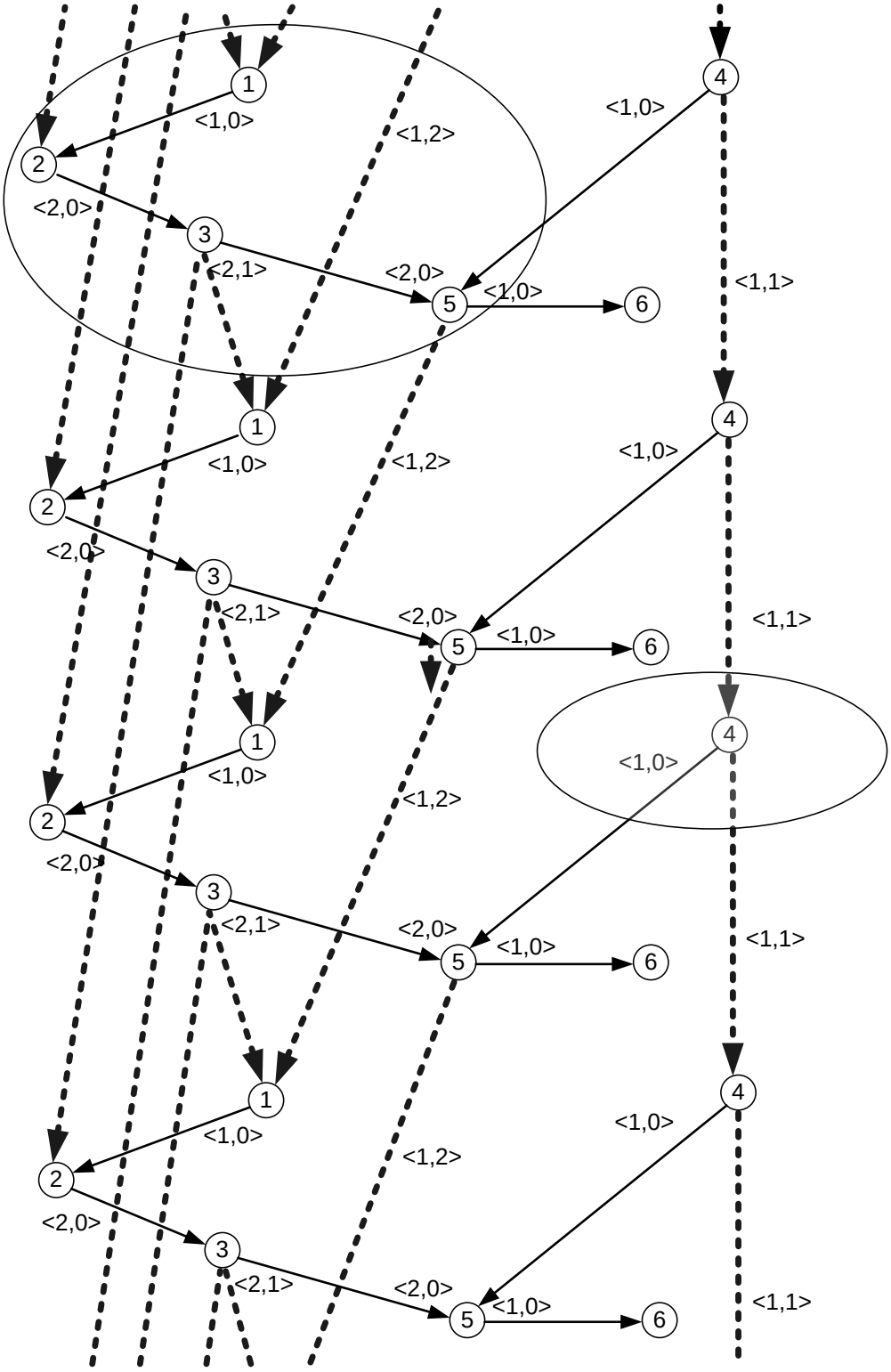
Slika 7.20. Graf petlje i parcijalnog uređenja komponenti jake povezanosti



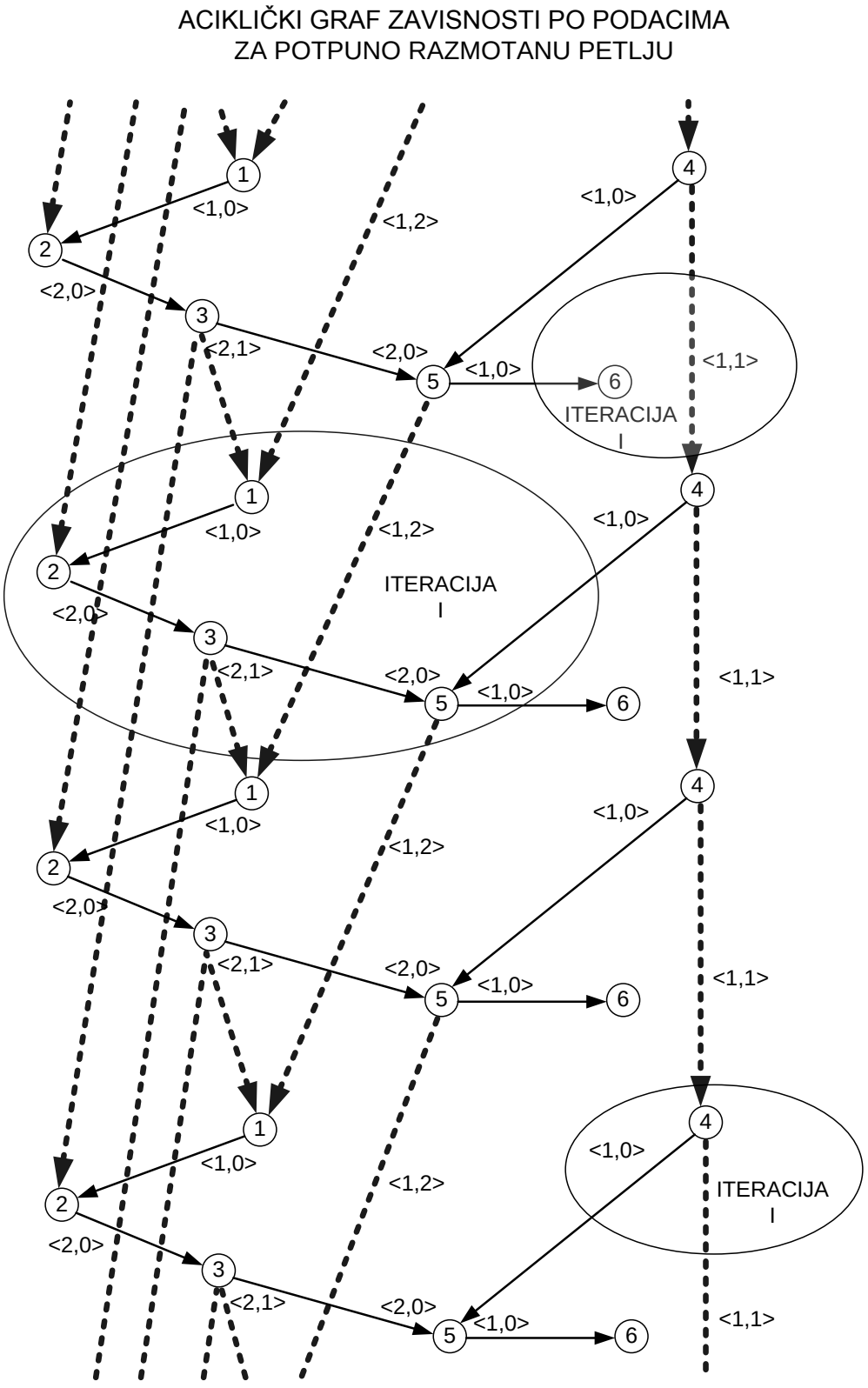
Slika 7.21. Početak predpetlje sa I_{11}, po notaciji sa Sl. 7.13.



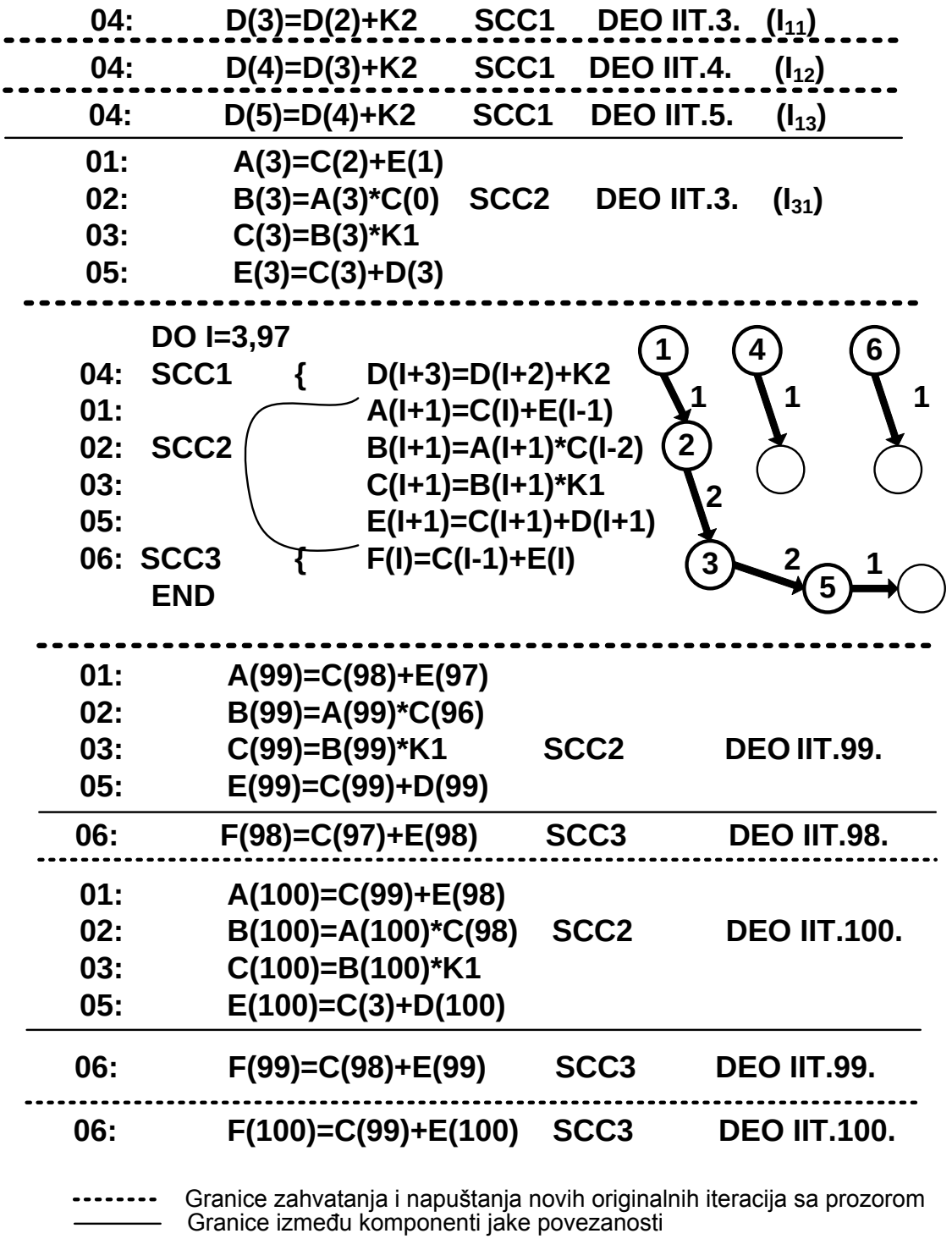
Slika 7.22. Deo predpetlje sa I_{12} – (I_{21} je prazan skup), po notaciji sa Sl. 7.13.



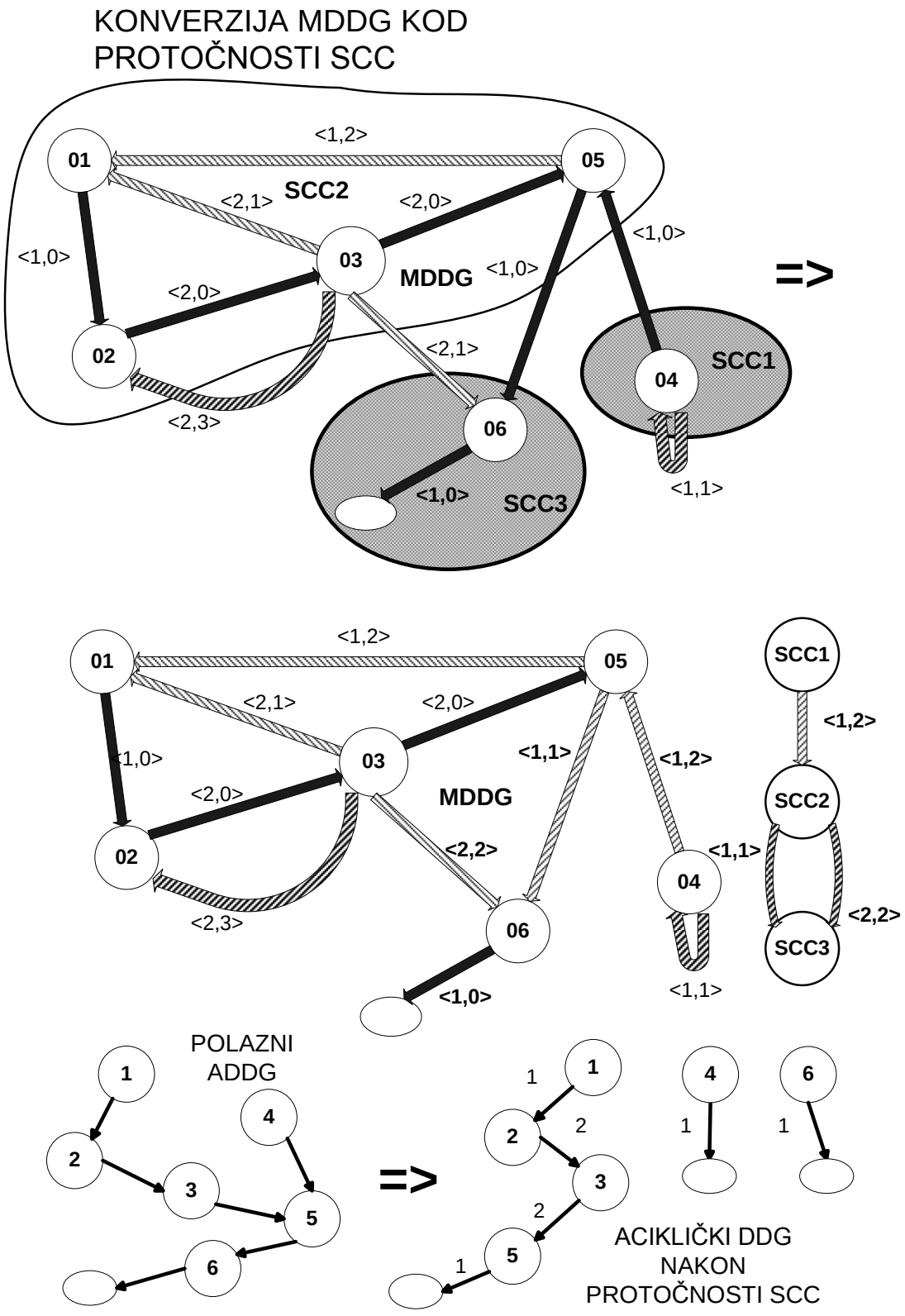
Slika 7.23. Delovi predpetlje I_{13} i I_{31} – (I_{22} je prazan skup), po notaciji sa Sl. 7.13.



Slika 7.24. Kompletna nova iteracija.



Slika 7.25. Transformisani kôd sa predpetljom i postpetljom u višem programskom jeziku i aciklički graf jedne iteracije kao posledica nove iteracije kao na Sl. 7.24.



Slika 7.26. Transformacija grafa petlje, grafa parcijalnog uređenja komponenti jake povezanosti i acikličkog grafa iteracije posle transformacije protočnosti SCC

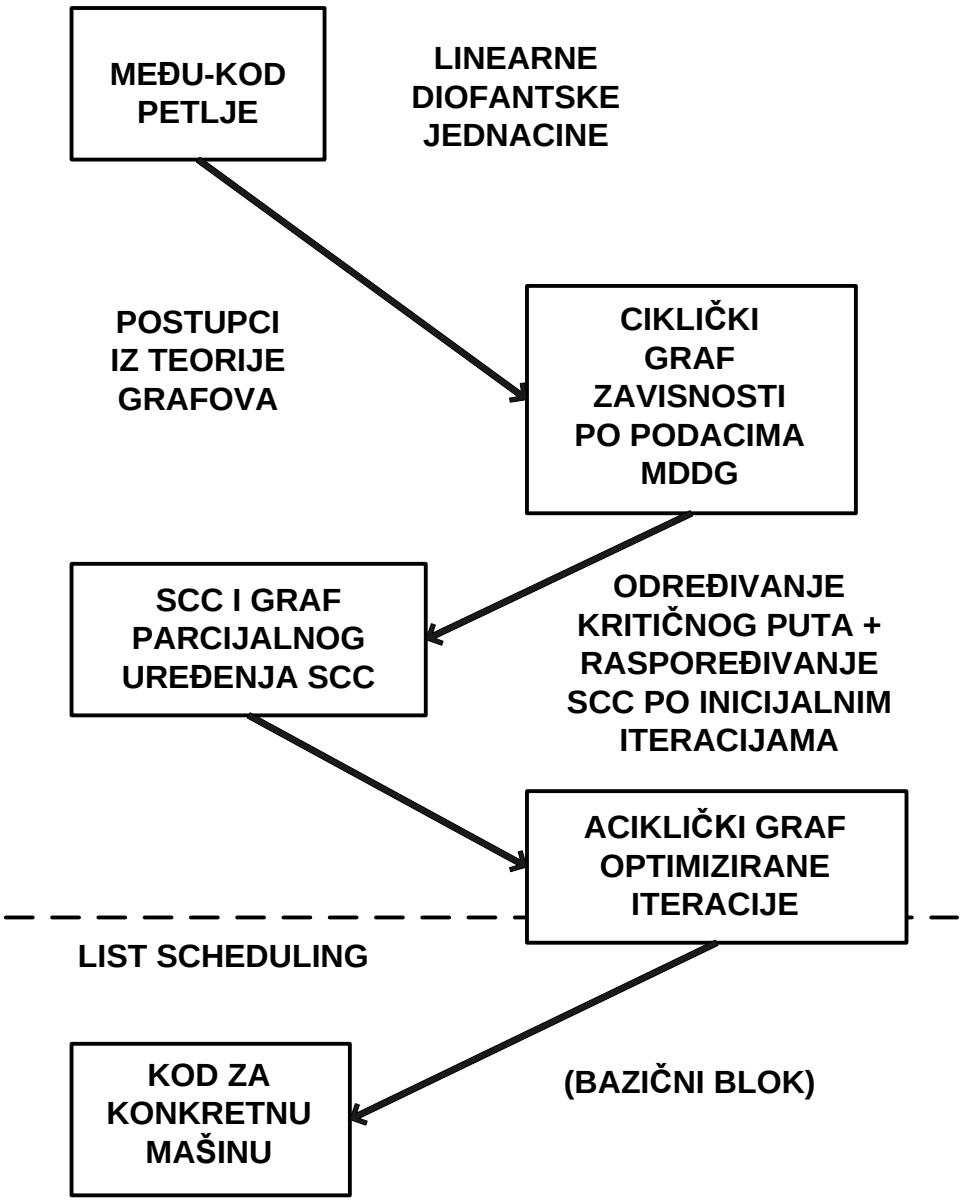
Dobijeno rešenje nije optimalno, jer i pored tri nepovezana dela u acikličkom grafu jedne iteracije, što dovodi do povećanog paralelizma, deo acikličkog grafa proizašao iz

komponente jake povezanosti SCC2 ima kritičan put koji je veći od minimalnog kritičnog puta. Minimalni kritični put proizilazi iz cikličkog grafa petlje i na osnovu Teoreme 4 iznosi 5 ciklusa po iteraciji. Razlog za to je činjenica da nije rađeno nikakvo raspoređivanje grana sa iteracionom distancom većom od 0 duž zatvorenih puteva, pa je samim tim raspored unutar komponente jake povezanosti SCC ostaje raspored koji je definisao programer. U ovom primeru, očigledno je da granu sa iteracionom distancom većom od 1 (između Op5 i Op1) treba svesti na iteracionu distancu 1 i "dobijenu" iteracionu distancu postaviti na neku drugu granu duž zatvorenog puta (Op_1, Op_2, Op_3 i Op_5). Transformacija rasporeda iteracionih distanci unutar komponente jake povezanosti, može dovesti do promene iteracionih distanci interSCC grana. Distance koje su postale jednake 1 primenom softverske protočnosti komponenti jake povezanosti mogu kao posledica rasporeda unutar komponenti jake povezanosti postati distance veće od 1 ili distance 0. U prvom slučaju to dovodi do rasporeda koji unosi nepotrebno velik stepen softverske protočnosti i samim tim nepotrebnu eksploziju kôda. Promena u drugom slučaju dovodi do pojave interSCC grana u acikličkom grafu jedne iteracije i samim tim potencijalnog smanjenja paralelizma. Zbog osobine metode protočnosti komponenti jake povezanosti da se iteraciona distanca interSCC grana može učiniti proizvoljno velikom, dodatnim razmicanjem komponenti jake povezanosti može se **uvek** opet postići eliminacija intraSCC grana iz acikličkog grafa jedne iteracije.

7.7.4. Ukupan postupak optimizacije kôda primenom protočnosti komponenti jake povezanosti

Postupak optimizacije kôda primenom protočnosti komponenti jake povezanosti može se razdvojiti u nekoliko koraka. U prvom, radi se analiza zavisnosti po podacima koja se svodi na rešavanje sistema Diofantskih jednačina u granicama indeksa petlji i leksikografske pozitivnosti. Primenom nekog od navedenih algoritama dobija se graf zavisnosti po podacima za programsku petlju. Ako je u pitanju DOALL petlja, paralelizacija se izvodi krajnje jednostavno. U slučaju DOACROSS petlje, primenom algoritama iz teorije grafova određuju se komponente jake povezanosti i formira graf parcijalnog uređenja komponenti jake povezanosti. Zatim se formira nova iteracija tako da iz komponenti jake povezanosti proizađu nepovezane komponente u acikličkom grafu jedne iteracije. Taj aciklički graf posmatra se kao mašinski nezavisan paralelan aciklički graf bazičnog bloka i na njega se može primeniti algoritam List Scheduling, kako je opisano u poglavlju 1.4. Ovaj postupak prikazan je i šematski na slici 7.27. U postupku postoji nekoliko pretpostavki koje treba istaći:

a. Pretpostavljeno je da su sve zavisnosti sa konstantnom iteracionom distancom.
b. Kôd unutar komponenti jake povezanosti nije modifikovan na bilo koji način u odnosu na polazni kôd
c. Unutar polazne iteracije ne postoje grananja, već samo jedan bazični blok



Slika 7.27. Optimizacija kôda metodom protočnosti komponenti jake povezanosti – celokupan postupak