3. Strategije izbora selidbi u cilju povećanja paralelizma

Selidbe treba praviti u vreme prevođenja, kada se povećava paralelizam na dinamičkim tragovima. Prva naivna primena pravila o selidbama operacija se zasnivala na sledećim koracima:

i. Uradi se lokalna optimizacija (paralelizacija) svih bazičnih blokova i na taj način dobije paralelizovan kôd pojedinačnih bazičnih blokova.
ii. Zatim se pokušavaju selidbe operacija između bazičnih blokova, za sve operacije koje su na početku i kraju rasporeda za bazični blok.
iii. Ako se izbacivanjem operacija iz rasporeda u jednom bazičnom bloku skraćuje raspored, a umetanjem tih operacija u drugi(e) bazični blok u već postojeći raspored ne produžava trajanje bazičnog bloka u koji su premeštene operacije, obavi se selidba operacija.

Ovakav postupak je vrlo retko dovodio do selidbe operacija kojom se ubrzava izvršavanje na paralelnim mašinama. Izložen je samo da bi se istakle njegove osnovne mane.

Te mane su sledeće:

a. Lokalna optimizacija se radila bez ikakve informacije o potrebnim selidbama, a time su ograničavane kasnije selidbe operacija između bazičnih blokova.
b. Lokalnom optimizacijom su se popunjavali resursi mašine u ciklusima, pa je postojala mala verovatnoća da će se ispuniti prethodno navedeni uslov iii.
c. Posmatraju se samo susedni bazični blokovi, a ne ceo kôd, tako da se operacije mogu seliti samo između susednih blokova.

Potrebno je posebno prodiskutovati stav c. Jednom preseljena operacija može u novom bazičnom bloku da postane kandidat za dalja seljenja, tako da operacija može da se preseli preko većeg broja grananja i spojeva, dokle god su zadovoljena pravila za semantički ispravnu selidbu kôda. Zbog male prosečne veličine bazičnih blokova, očigledno je da dobra globalna optimizacija mora da uključuje selidbe preko većeg broja grananja i spojeva – što podrazumeva agresivnu optimizaciju. Opisani naivni početni pokušaji globalne optimizacije pomoću selidbi operacija dali su loše rezultate. Idealno bi bilo da selidba poveća paralelizam na svakom dinamičkom tragu kroz bazične blokove koji su učestvovali u selidbi, ali je to retko moguće.

Nešto bolji način da se odlučuje o selidbi je da se prvo lokalno posmatraju kritični putevi u dva susedna bloka i da se selidba obavi ako:

a. Se skraćuje kritičan put u bloku iz koga se seli operacija.
b. U svim ostalim blokovima se ne produžava kritičan put.

Primenom ovako jednostavnog pravila dobija se paralelniji kôd, ako se nakon toga radi lokalna optimizacija – npr. raspoređivanje po listi u svim bazičnim blokovima. Ovo je sigurno bolji postupak od prethodnog naivnog, jer lokalna optimizacija bazičnih blokova bar nije prethodila selidbama i zauzimanjem resursa ograničavala selidbe operacija između susednih bazičnih blokova. Međutim, i dalje ostaje mana da se posmatraju samo susedni bazični blokovi, a ne ceo kôd.

Potrebno je pronaći postupak kojim će se, globalnim posmatranjem, definisati kriterijumi višeg nivoa koji će omogućiti da se pronađu selidbe koje dovode do optimizacije svih bazičnih blokova istovremeno. Da bi se to postiglo, moraju se nekako posmatrati svi dinamički tragovi i za njih definisati profitabilne transformacije (u smislu optimizacije jednog ili više tragova). Zbog velikog broja mogućih dinamičkih tragova, potrebno je po nekom kriterijumu definisati prioritetne i od njih početi optimizaciju. Definisanje profitabilnih transformacija je još kompleksnije, jer transformacija profitabilna za jedan dinamički trag ne mora istovremeno da bude profitabilna za neki drugi trag. Tri osnovne kategorija globalne optimizacije se zasnivaju na:

a. Primarnoj orijentaciji na optimizaciju najverovatnijih tragova (najpoznatiji je Trace scheduling) {FI 81}{ELL 86}
b. Definisanju hijerarhije transformacija u kojima su najniži nivo – bazične transformacije slične osnovnim selidbama operacija, a pravila višeg nivoa definišu kada i koje transformacije nižeg nivoa je potrebno sprovoditi tako da se dođe do najbolje paralelizacije kôda (najpoznatiji je Percolation scheduling) {NI 85}
c. Pokušaju izjednačavanja paralelizma u bazičnim blokovima sa paralelizmom mašine (najpoznatiji je Region scheduling) {GS 90}.

Algoritam prve kategorije je detaljno razmotren u knjizi – Trace scheduling, a elementi algoritma druge kategorije samo delom naznačeni kod pravila selidbi preko grananja. Treća kategorija algoritama danas ima samo istorijski značaj, a razlog za to je visok stepen integracije u integrisanim kolima, tako da procesori koji se zasnivaju na paralelizaciju tokom prevođenja gotovo uvek mogu da imaju dovoljno paralelizma u hardveru, za postojeći paralelizam na instrukcijskom nivou. Cilj današnjih algoritama za paralelizaciju u vreme prevođenja je da se transformacijama ostvari najviši nivo paralelizma, uz ograničeno zauzeće resursa koje potiče od ograničenja u resursima mašine., jer se pretpostavlja da se hardver može učiniti dovoljno paralelnim.

3.1. Trace scheduling

Trace scheduling, ili algoritam raspoređivanja po tragu, zasnovan je na tome da se iz grafa toka uzima ceo jedan mogući dinamički trag u kome se nalazi više bazičnih blokova, obično najverovatniji trag, i taj izabrani trag se optimizuje kao da je u pitanju jedan jedini bazični blok. Jasno je da ovako formirani trag (koji privremeno posmatramo kao bazični blok) sadrži mnogo više paralelizma nego pojedinačni bazični blokovi koji su ušli u trag. Za ceo trag se koristi algoritam List Scheduling, tako da se bez uzimanja u obzir grananja sa traga i uskakanja u trag dobija veoma paralelan kôd samo za taj najverovatniji trag. Naravno da se nakon paralelizacije traga mora rešiti pitanje obnavljanja grananja i spojeva vezanih za taj trag, da bi se zadržala semantička korektnost programa. To obnavljanje nije nimalo trivijalno, jer se mora sprovesti kada je već optimizovan trag u kome su izmešane operacije koje su pripadale različitim bazičnim blokovima na tragu. Ne samo da su samo izmešane operacije, već su one i preklopljene i raspoređene po ciklusima, a traju d(Opi) ciklusa, kao što je opisano u algoritmu List Scheduling. Rešenje se naravno mora zasnivati na opisanim pravilima za osnovne selidbe operacija preko grananja i spojeva, uključujući selidbe operacija u ciklusima, kao što je opisano u poglavlju 2. Nakon što se izvrši optimizacija jednog traga, bira se sledeći najverovatniji trag od preostalih bazičnih blokova. Za sada, razmatraćemo slučaj kada trag ne uključuje kôd unutar petlji.

Način na koji se biraju blokovi koji će ući u najverovatniji trag je izuzetno bitan. Potrebno je da taj trag bude verovatniji za izvršavanje od svih drugih, a na osnovu podataka poznatih u vreme prevođenja. Prevodilac može da odredi najverovatniji od svih tragova na osnovu ulaznih informacija koje daje programer. Programer može da navede očekivanu verovatnoću odlaska programa u neku granu za svako grananje, a da prevodilac na osnovu tih informacija i grafa kontrole toka izvršavanja programa pronalazi najverovatniji bazični blok i odatle počne da gradi najverovatniji trag. Kako se ne može uvek unapred pretpostaviti sa kakvim će sve ulaznim podacima raditi program, greške u proceni verovatnoća grananja mogu da naruše efikasnost Trace Scheduling algoritma. Pokazalo se da programeri mogu sa uspehom predvideti ove verovatnoće kada su u pitanju numerički orijentisani programi, ali za sistemske programe nisu bili tako uspešni. Drugi postupak otkrivanja najverovatnijeg traga je da se izvršavanjem neoptimizovanog programa za veći broj reprezentativnih uzoraka ulaznih podataka dobiju verovatnoće za grananja, za dati skup uzoraka. One zatim postaju ulaz za kasniju optimizaciju kôda, sa nadom da reprezentativni uzorak odgovara ulaznim podacima koji će se javljati u korišćenju prevedenog programa.

Prevodioci gotovo svih mašina do 1986. godine optimizovali su samo po jedan bazični blok u jednom trenutku. Na osnovu definicije bazičnog bloka, svi skokovi prema bazičnom bloku idu ka prvoj operaciji u bloku, a grananje iz tog bloka prema spoljašnjim blokovima mora biti na kraju bloka. Primenom Trace schedulinga, kada se najverovatniji trag posmatra kao da je jedan bazični blok, zanemaruju se granice bazičnih blokova na tragu. Kao rezultat List Scheduling-a mešaju se operacije iz različitih bazičnih blokova na tragu. Ne samo da se mešaju operacije iz različitih bazičnih blokova koji ulaze u trag, već su operacije preklopljene po ciklusima kao rezultat List schedulinga. Dakle, skokovi u trag ne moraju i ne mogu više biti ka početnoj operaciji bazičnog bloka kojem su prethodili, jer te operacije čak uopšte ne moraju biti prve koje se javljaju u paralelizovanom kôdu! Slično je i sa grananjima kojima se mora naći novo mesto na tragu. Ovim je znatno kompleksnije očuvanje semantičke ispravnosti programa i to se mora postići selidbama operacija preko grananja i spojeva **u ciklusima**. Sve grane u grafu kontrole toka kojima se ulazi u trag (spojevi) i sve grane kojima se izlazi iz traga (grananja) nazivaju se bočne grane traga. Proces kopiranja operacija u bočne grane traga, prilikom optimizacije, zove se **bookkeeping.**

Trace scheduling algoritam se dakle sastoji od tri osnovna sastavna algoritma:

- Izbor najverovatnijeg neraspoređenog traga (*trace picking*)
- List scheduling-a, koji optimizuje trag kao da je u pitanju bazični blok i
- Obnavljanja granica i spojeva na optimizovanom tragu (*bookkeeping algoritam*)

3.1.1. Izbor najverovatnijeg neraspoređenog traga (Profiling)

Pri izboru sledećeg traga iz grafa toka za koji će se vršiti paralelizacija, pokušava se pronaći onaj koji ima najveću verovatnoću izvršavanja od preostalog neoptimizovanog dela kôda. Njegov dobar izbor je bitan, pošto će biti optimizovan bez ikakvih ograničenja u slučaju prvog traga i sa manje ograničenja ukoliko je trag ranije izabran. Optimizacija prvih tragova se obavlja samo donekle na račun tragova koji će kasnije biti optimizovani. Varijanta izbora najverovatnijeg traga na osnovu reprezentativnih uzoraka ulaznih podataka koje obično generiše programski prevodilac je opstala, jer programeri nisu želeli da daju svoje procene verovatnoća grananja.

Prilikom izbora traga, koriste se procenjene verovatnoće izvršavanja bazičnih blokova. Svaki bazični blok *B* mora imati pridruženu vrednost *count* (*B*) kao očekivani broj izvršavanja tog bloka tokom izvršavanja celokupnog programa za različite skupove ulaznih podataka iz reprezentativnih skupova. Takođe, svaka grana *e* operacije uslovnog grananja mora imati pridruženu vrednost *prob*(*e*) kao verovatnoću da će se izvršavanje programa nastaviti tom granom kada se dođe do grananja. Ove očekivane vrednosti može ostaviti u pogodnom obliku sam programer ili se one mogu dobiti pomoću specijalnih programa – *profiler*-a. Jasno je da nisu u pitanju nezavisne veličine, jer očekivani broj izvršavanja za bazični blok C na grani *e* iza bloka *B* koji sadrži operaciju uslovnog grananja kao zadnju operaciju jednak je:

$$count\ (C) = count(B)\ prob(e)$$

U slučaju spoja, jasno je da se sabiraju count vrednosti svih ulaznih grana. Zato su ranije verzije prevodilaca koje su tražile informaciju od programera tipično tražile samo procenjene verovatnoće grananja.

Da bi se izabrao najverovatniji trag, najpre se pronađe bazični blok sa najvećim procenjenim brojem izvršavanja i inicijalni trag, koji se još ne optimizuje, se sastoji samo od tog bloka (koji se naziva "seme"). Zatim se taj trag proširuje naniže i naviše. Proširivanje naniže obavlja se tako što se posmatraju blokovi koji slede blok ili blokovi koji su trenutno na kraju traga i u slučaju grananja, od blokova sledbenika se bira samo jedan "dobar" prema već definisanim pravilima i stavlja na kraj traga. Ovo širenje na dole se ponavlja sve dok se ne desi jedna od sledećih situacija:

1. Ne postoji nijedan "dobar" sledbenik po nekom kriterijumu (verovatnoće su približno podjednake za sledbenike na kraju traga)
2. Izabrani sledbenik je već optimizovan u nekom od ranijih tragova
3. Sledbenik koji bi bio izabran je granica iteracije.

Kada se desi jedna od ove tri situacije, završava se sa formiranjem traga naniže i na sličan način se nastavlja sa njegovim formiranjem naviše, počev od bloka koji je uzet za "seme". U širenju naviše, širenje traga se može završiti ako se javi spoj sa približno podjednakim verovatnoćama da će se iz grana prethodnika uskakati u trag, ili analognim slučajevima sa 2. i 3. Iz širenja traga na dole.

Šta znači termin "dobar" sledbenik pri formiranju traga naniže, odnosno "dobar" prethodnik pri formiranju naviše? U oba slučaja se iz skupa grana - kandidata bira ona po koja će najverovatnije biti deo dinamičkog traga. Pri formiranju traga naniže, grane - kandidati su grane koje polaze iz poslednjeg bloka koji je trenutno na kraju traga, a pri formiranju traga naviše, grane - kandidati su grane koje dolaze ka prvom bloku koji je trenutno na početku traga.

Neka je Op_k operacija koja je trenutni kraj traga u originalnom redosledu operacija pri formiranju traga naniže. Ako nije u pitanju grananje, tada je nedvosmisleno kako se trag produžava naniže jedinim mogućim putem. Ako je na kraju grananje, grana koja ima veću verovatnoću se pridružuje tragu. Ako paralelna mašina u hardveru podržava

višestrane skokove u istom ciklusu (multiway jump), što liči na hardverski ekvivalent select odnosno case kontrolne strukture u programskim jezicima, bira se grana koja ima najveću verovatnoću od svih mogućih destinacija skoka.

Neka je Op_j trenutno prva operacija na tragu na osnovu originalnog redosleda. Ako je granica bloka nastala kao posledica grananja na kraju prethodnog bloka, tada je nedvosmisleno kojim putem se produžava trag naviše. Ako operaciji Op_j prethodio spoj, tada se bira grana koja ima najveći procenjeni broj izvršavanja na osnovu *profiling-a* od svih grana koje dolaze ka spoju ispred Op_j.

Ova osnovna pravila za izbor najverovatnijeg traga vrlo često se modifikuju heuristikama koje uglavnom definišu granice širenja tragova ili imaju kompleksniju analizu koja uključuje i dalje blokove pri povećavanju traga.

3.1.2. List scheduling

List scheduling algoritam za optimizaciju kôda na nivou bazičnog bloka se neizmenjen primenjuje na izabrani trag kod Trace scheduling-a. Jedina bitna razlika je da su tragovi po broju operacija znatno veći od bazičnih blokova i prosečan nivo paralelizma, se približava 10. Zavisno od kriterijuma za prekid rasta traga, prosečna veličina „najverovatnijih“ tragova je od nekoliko puta do nekoliko desetina puta veća od veličine prosečnog bazičnog bloka. Takođe je jasno da se u ovom slučaju sigurno nisu primenjivi algoritmi za sigurno traženje optimalnog rešenja zbog njihovog eksponencijalnog karaktera (vreme optimizacije je eksponencijalno zavisno od broja operacija). Dakle, List Scheduling algoritam prikazan u poglavlju 1.3. se u potpunosti primenjuje za tragove.

3.1.3. Obnavljanje granica i spojeva na optimizovanom tragu

Nakon što se generiše tabela rasporeda operacija za konkretnu mašinu za dati trag, isti se uklanja iz grafa toka i zamenjuje tabelom koja predstavlja raspored početaka operacija na tragu po ciklusima. Kako se razmatraju mašine koje mogu da za svaki tip resursa imaju više instanci, u jednom ciklusu mogu istovremeno da se započinju operacije iz nekoliko različitih bazičnih blokova sa prilično udaljenih tačaka traga. Samim tim, mnoge od tih operacija primenom List scheduling algoritma su se izmešale i više praktično ne postoji mesto gde se iskače sa traga ili uskače na trag. Ako bi se trag jednostavno zamenio rasporedom po ciklusima dobijenim optimizacijom traga, bez pronalaženja novih tačaka (ciklusa) grananja i spojeva iz traga, dobio bi se neispravan program.
Tačke grananja iz traga i tačke spojeva kod uskakanja u trag se moraju generisati na novim mestima na tragu – tako da se dobije semantički ispravan kôd. Potrebno je pronaći ta nova mesta tako da se uopšte mogu primeniti pravila za selidbe operacija između bazičnih blokova. Kada se pronađu ta mesta, potrebno je, radi očuvanja ispravnosti, na svim ulaznim i izlaznim tačkama traga ubaciti deo kôda koji čine kopije preseljenih operacija. Za ovaj proces zamene traga tabelom i kopiranja potrebnih operacija van traga, koji je originalno nazvan ***bookkeeping***, ponekad se koristi termin sa nešto užim značenjem: obnavljanje grananja i spojeva.

Zbog lakšeg objašnjavanja, definisano je nekoliko pojmova. Uslovni skok sa traga zove se **Tgrananje**, a skok na trag zove se **Tspajanje**. Kod Tgrananja razlikujemo dve grane,

jednu koja vodi ka sledećem bloku, odnosno ka sledećoj operaciji, na tragu, i drugu, koja vodi van traga. Prva grana se zove grana na tragu (on-trace), a druga se zove grana van traga (off-trace). Samo u specijalnim slučajevima, obe grane mogu da vode na trag, ali je samo jedna na izabranom najverovatnijem tragu i ona će se smatrati granom na tragu (on-trace), a druga se preko spoja ponovo vraća kasnije na trag (off-trace). Slično je i kod Tspajanja: grana koja dolazi sa prethodne operacije na tragu naziva se grana na tragu (on-trace), a ona (ili one) koje dolaze sa blokova van traga zovu se grane van traga (off-trace).

Za razliku od selidbe operacija preko grananja na dole i inverzne operacije selidbe identičnih kopija preko grananja naviše, gde je pretpostavljeno da se operacija želi preseliti preko grananja, u ovom slučaju je optimizacija traga nametnula drugi problem. Potrebno je odrediti novo mesto Tgrananja za koje je moguće selidbama operacija dobiti semantički ispravan program. To mesto Tgrananja u pojednostavljenoj varijanti mora da bude iznad najranijeg početka svake operacije na tragu koja je u originalnom kôdu bile posle tačke grananja. Dakle, Tgrananja će se zbog izmešanosti operacija iz različitih bazičnih blokova na tragu kao posledice List schedulinga pomeriti pre velikog broja operacija koje su prethodile Tgrananju u originalnom kôdu. Ovo se može smatrati pomeranjem grananja naviše. Na sve operacije koje su u svim raspoređenim operacijama na tragu ispod tačke novopostavljenog Tgrananja, a originalno su bile iznad Tgrananja, mora se primeniti pravilo selidbi operacija preko grananja na dole.

Redosled koraka je dakle sledeći:

1. optimizuje se najverovatniji trag primenom List scheduling-a bez ograničenja i operacije iz svih bazičnih blokova na tragu su se pomešale u ciklusima na tragu

2. neophodno je pronaći novo mesto Tgrananja za koje se može napraviti semantički ispravan kôd i to dovodi do seljenja grananja naviše

3. seljenjem Tgrananja naviše se postiže ekvivalentan efekat kao da su sve operacije koje su u originalnom kôdu bile pre tog Tgrananja preseljene preko Tgrananja na dole. Kako kopija operacije već postoji u jednoj od grana (on–trace) kao posledica List Schedulinga, zbog primene ***Selidbe preko grananja na dole (poglavlje 2.2.2)*** je neophodno generisati dodatnu kopiju na off-trace grani.

Kao posledica toga se van traga generiše kôd od svih kopiranih operacija nastalih selidbama operacija na dole preko tako postavljenog Tgrananja. Praktično, selidba Tgrananja naviše nameće da se veliki broj operacija sa traga smatra preseljenim na dole preko grananja. Primenom selidbe operacija na dole uz uvedena pravila, generiše se dodatan kôd van traga. Količina dodatno generisanog kôda može biti veoma velika i cilj je da se minimizira. Zato se za novo mesto grananja bira najkasnija tačka na tragu koja zadovoljava uslov da bude pre najranije operacije na tragu koja je originalno bila posle grananja: mesto Tgrananja mora da bude neposredno pre početka najranije operacije na tragu koja sadrži operacije sa traga koje su u originalnom kôdu bile posle tačke grananja. Ovim pojednostavljenim pravilom se naizgled minimizira količina dodatno generisanog kôda.

U prethodnim pasusima je navedeno da je izbor mesta grananja opisano kao pojednostavljena varijanta. Razlog za to je činjenica da se u retkim slučajevima događa da prve operacije inicijalno postavljene u ciklusu posle tako pomerenog grananja, sve istovremeno ispunjavaju sledeći uslov: rezultat operacije je mrtav u grani van traga (off-trace). U tom slučaju, grananje se može pomeriti naniže za jedan ili nekoliko (mali broj) ciklusa. Ovakvom modifikacijom može se još više smanjiti eksplozija kôda. Ovo pokušavanje pomeranja Tgrananja u ciklusima na dole po tragu zaustavlja se kada se naiđe na operaciju (ili operacije) koja je bila originalno iza grananja i generiše rezultat koji je živ na grani van traga. Tada se ciklus grananja može pomeriti do ciklusa te (ili tih) operacija neposredno pre upisa rezultata, jer bi se rezultat upisivao na on trace grani, a ne bi se upisivao na off-trace grani. Tek na ovaj način je minimiziran dodatni kôd prilikom izbora mesta Tgrananja u Trace scheduling algoritmu.

Razmotrimo sada Tspajanja. Potrebno je kao i u slučaju Tgrananja pronaći tačku novog spoja na optimizovanom tragu. Ta tačka mora da bude iza najkasnijeg kraja operacije koja je originalno bila iznad Tspajanja. Ovaj uslov proizilazi iz činjenice da pretpostavljamo da u kôdu grane van traga ne postoji identična operacija slobodna na dnu toj najkasnijoj operaciji. Isto tako se pretpostavlja da se upisi rezultata kod operacija obavlja u poslednjem ciklusu operacije. Kako su se operacije na tragu izmešale tokom List scheduling optimizacije, veliki broj operacija koje su u originalnom kôdu bile ispod Tspajanja će se ovakvim izborom novog mesta Tspajanja naći iznad nove tačke Tspoja na tragu, pa se moraju primeniti pravila selidbe operacije preko spoja na gore. Kao posledica primene tih pravila, generisaće se dodatni kôd u grani van traga. To opet izaziva znatan porast kôda.

Na ovaj način se izaziva rast svih bazičnih blokova u koje se skokom moglo prelaziti sa traga i blokova koji su se spajali na trag. Pritom se izaziva porast paralelizma i u tim off-trace blokovima, samim rastom veličine blokova!

Primer 3.1.:
Pretpostavimo najjednostavniji slučaj traga i grana u okolini traga. Ukupno postoje četiri bazična bloka koja se optimizuju: bazični blok ispred IF-THEN-ELSE strukture, THEN i ELSE grane i grane iza IF-THEN-ELSE strukture. Pretpostavljeno je da se ispred prvog bloka nalazi granica (kraj) programske petlje i da se iza zadnjeg bloka takođe nalazi granica (početak) programske petlje. Ukupan kôd je dat sledećim instrukcijama:

```
Petlja 1
OP1 :  j : = i + k;
OP2 :  i := i + l;
OP3 :  IF BOOL THEN
       DO
OP4 :     j := j * 2;
OP5 :     c: = a / b;
       END
       ELSE
       DO
OP6 :     d := d * 2;
OP7 :     a := a +8;
OP8 :     c := j / 2;
OP9 :     i := i * k;
       END
OP10 :  i: = i + 5;
OP11 : d: = c - 2;
OP12 : m: = i * 2;
Petlja 2
```

Sl. 3.1. originalan kôd sa 4 bazična bloka i jednom IF-THEN-ELSE strukturom

Pretpostavka je da je na raspolaganju mašina koja ima jednu jedinicu koja obavlja sve operacije izuzev množenja i deljenja i jednu jedinicu množač/delitelj (ograničenje u resursima iskazano preko vektora zauzeća resursa). Pretpostavimo radi jednostavnosti (kako se ne bi razmatrali ciklusi operacija) da se sve operacije izvršavaju u jednom ciklusu. Određivanje uslova za skok kod grananja obavlja se u paraleli sa navedenim operacijama. Na osnovu rada profiler-a dobilo se da je verovatnoća *prob(BOOL-THEN)* da BOOL bude true jednaka 80%, pa je verovatnije izvršavanje THEN grane.

Najverovatniji trag nam je dakle blok pre IF-THEN-ELSE, THEN grana i na kraju blok iza IF-THEN-ELSE.

Optimizacijom tog traga, kao da je u pitanju bazični blok, primenom List Scheduling-a dobija se:

```
Petlja 1
c: = a / b;  j : = i + k;
i := i + l;    j := j * 2;
i: = i + 5;
d: = c - 2;  m: = i * 2;
Petlja 2
```

pri čemu pojava operacija u istom redu označava paralelno izvršavanje u istom ciklusu. Ovde je List scheduling faza realizovana tako što nisu rađena preimenovanja zbog antizavisnosti, jer je dobijeno minimalno trajanje traga, uzimajući u obzir ograničenja u pogledu broja resursa, od 4 ciklusa.

Kada se sada primeni **pojednostavljena varijanta** traženja mesta Tgrananja, dobija se mesto Tgrananja iznad prve paralelne instrukcije na tragu. Tspoj bi najranije mogao da bude iza druge paralelne instrukcije (obnavljanje spoja) i u ovom specijalnom slučaju ne

bi došlo do seljenja operacija preko grananja na gore. U tom slučaju bi ukupan kôd bio sledeći:

```
OP3 :  IF BOOL THEN
DO
        c: = a / b;  j : = i + k;
        i := i + l;  j := j * 2;  /* optimizovan - raspoređen deo kôda */
END
ELSE
    DO
OP1 :    j : = i + k;
OP2 :    i := i + l;
OP6 :    d := d * 2;
OP7 :    a := a +8;  /* neoptimizovan deo kôda */
OP8 :    c := j / 2;
OP9 :    i := i * k;
    END
        i: = i + 5;        /* optimizovan - raspoređen ostatak traga */
        d: = c - 2; m: = i * 2;
```

Sl. 3.2. Optimizovan verovatniji trag i preseljeni kôd u ELSE grani za kôd sa 4 bazična bloka i jednom IF-THEN-ELSE strukturom

Ovakva selidba dovodi do potrebe da se u paraleli obavlja odluka o izlasku iz programske petlje koja je prethodila navedenom kôdu i o izboru između THEN i ELSE grane u istom ciklusu ako se želi veći stepen paralelizma. Upravo ovakvi zahtevi proizašli iz optimizacije zahtevaju da se prave mašine koje omogućavaju višestruke skokove u istom ciklusu (multiway jump). Paralelizacijom ELSE traga koji se sada sveo na bazični blok dobija se rešenje:

```
OP3 :  IF BOOL THEN
DO
c: = a / b;  j : = i + k;
i := i + l;  j := j * 2;
END
ELSE
    DO
j : = i + k;   d := d * 2;
i := i + l;    c := j / 2;
a := a +8;     i := i * k;
    END
i: = i + 5;
d: = c - 2; m: = i * 2;
```

Sl. 3.3. Finalno optimizovan kôd za 4 bazična bloka sa jednom IF-THEN-ELSE strukturom

U paralelizaciji ELSE grane takođe nije striktno poštovan kriterijum kritičnog puta za prave zavisnosti kod List scheduling algoritma, već su zadržane i antizavisnosti da se ne

bi radila nepotrebna preimenovanja za promenljivu i. Dakle, kôd koji bi se bez paralelizacije izvršavao za 7 ciklusa (ne računajući grananje) bi na navedenoj paralelnoj mašini trajao samo 4 ciklusa pri dinamičkom tragu kroz THEN granu. Na dinamičkom tragu kroz ELSE granu (ne računajući grananje) dobija se trajanje od 5, umesto 9 ciklusa.

IF se našao ispred prvog ciklusa zbog OP_5 : c: = a / b, koja je pripadala kôdu posle grananja. U ELSE grani se, međutim, promenljiva c prvo upisuje, pa je taj rezultat mrtav u ELSE grani. Zato se grananje može pomeriti za jedan ciklus na dole. Tako se dobija drugačiji kôd sa manje dodatnih operacija zbog obnavljanja grananja. Na ovaj način je pokazana na primeru razlika između potpuno pojednostavljene varijante traženja mesta grananja i varijante u kojoj se analizira da li je rezultat mrtav u drugoj grani. Takođe je pokazano kako dodatna operacija OP_5 : c: = a / b, koja se izvršava na paralelnoj mašini i u slučaju prolaska kroz ELSE granu ne produžava izvršavanje. U ovom slučaju mora da bude razrešen problem obrade izuzetaka generisanih hardverom, jer bi se npr. u slučaju da je b=0 dogodio izuzetak prilikom izvršavanja operacije OP_5. Pritom, ta operacija uopšte nije trebalo da se izvrši u dinamičkom tragu sa ELSE granom po originalnom kôdu, a može da dovede do zaustavljanja izvršavanja programa. Zato najnoviji procesori imaju poseban hardver za ispitivanje efekata izuzetaka i odlaganje posledica izuzetaka do trenutka u kome se konačno u vreme izvršavanja ispita da li je operacija uopšte trebala da se izvrši. Finalni paralelizovani kôd dobijen Trace schedulingom, uz pomeranje spoja na dole zbog iskorišćenja resursa mašine je:

```
c: = a / b;  j : = i + k;  IF BOOL THEN
DO
i := i + l;  j := j * 2;
i: = i + 5;
END
ELSE
      DO
i := i + l;     c := j / 2;
a := a +8;    i := i * k;
i: = i + 5;    d := d * 2;
      END
d: = c - 2; m: = i * 2;
```

Sl. 3.4. Finalno optimizovan kôd sa minimalnim porastom kôda za 4 bazična bloka sa jednom IF-THEN-ELSE strukturom

U ovom rešenju je uočljivo i da ne mora da postoji multiway jump osobina mašine za minimalno vreme izvršavanja, jer je završetak programske petlje pre analiziranog traga za jedan ciklus iznad novog mesta Tgrananja.

Primer je značajno pojednostavljen u odnosu na realan po dva osnova:

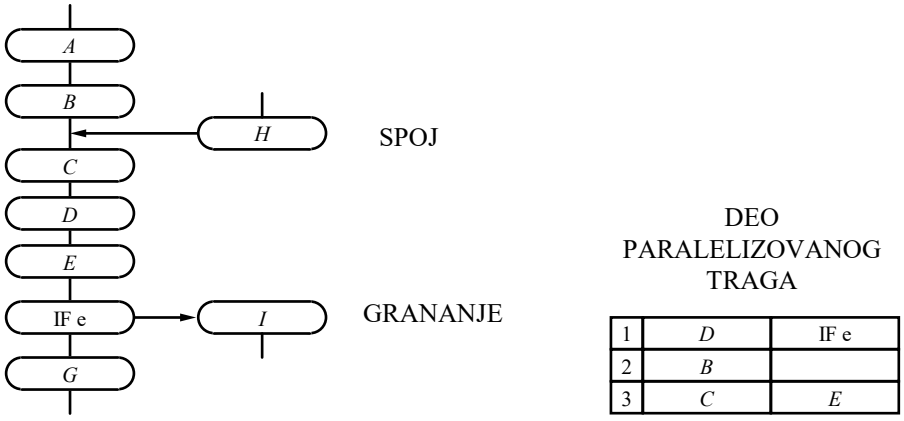
1. Mašina je krajnje uprošćena, jer današnje mašine imaju mnogo više resursa koji se mogu iskoristiti u paraleli.
2. Pretpostavka da sve operacije traju jedan ciklus je potpuno nerealna, jer je jedan od glavnih izvora paralelizma protočnost (pipelining).

Ipak, primer ilustruje osnovne korake Trace scheduling-a.

Celokupan dosadašnji opis Trace scheduling-a je zbog jednostavnijeg objašnjenja potpuno zanemario kontrolne zavisnosti, jer se pretpostavljalo da je promenljiva BOOL tipa boolean bila izračunata ranije. Naravno, radi se o kontrolnoj zavisnosti. Da je kojim slučajem Boolean uslov bio i != 0, kao rezultat operacije OP_2 : i := i + l;, nijedan od navedenih rasporeda nije korektan. Za eliminaciju kontrolnih zavisnosti postoji više rešenja, a jedno jednostavno, originalno predloženo od autora Trace Scheduling-a, je da se grafu zavisnosti po podacima dodaju kontrolne zavisnosti koje polaze od operacije koja izračunava uslov grananja (generišu flag koji je uslov za grananje). Tako se uz dodatna ograničenja – dodatne grane zavisnosti, dodate zbog kontrolnih zavisnosti - prilikom optimizacije traga List scheduling-om obezbeđuje da postoji korektno rešenje. Ostala rešenja za eliminaciju kontrolnih zavisnosti – pre svega spekulativno i predikatsko izvršavanje opisani su u poglavlju 4.

3.1.4. Selidba Skokova

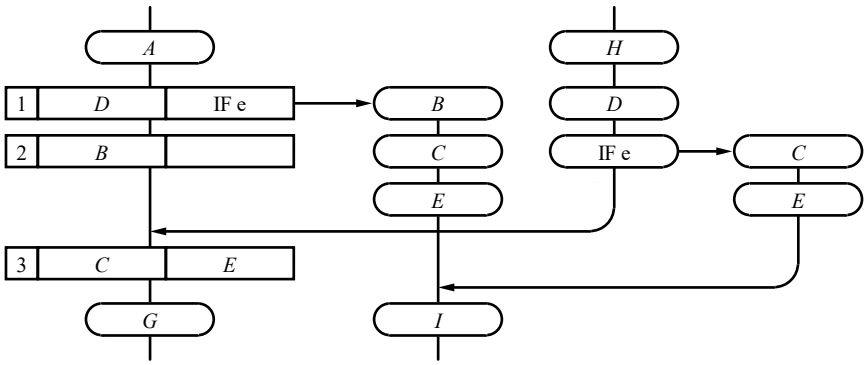
Na osnovu prethodnog prikaza obnavljanja grananja i spojeva na optimizovanom tragu, pokazalo se da se selidbe grananja obavljaju naviše, a selidbe spojeva naniže, gledano u odnosu na originalni kôd. Kao posledica toga, može se pojaviti potreba da se grananje pri selidbi prebaci preko spoja na gore. To je ilustrovano primerom sa Sl. 3.5. Pretpostavljena je paralelizacija predstavljena tabelom koja prikazuje 3 ciklusa. Takođe je pretpostavljeno da operacije obeležene slovima traju jedan ciklus. Pogledajmo sledeći trag i za njega formiranu tabelu po ciklusima za deo optimizovanog traga:



Sl. 3.5. Dijagram toka izvršavanja i optimizovan trag

Pretpostavimo da je već određeno mesto grananja i operacija u delu paralelizovanog kôda, uz pretpostavku da operacije traju jedan ciklus. Spoj se mora postaviti nakon drugog ciklusa zbog operacije B, na osnovu pravila za postavljanje novog mesta Tspoja u bookkeeping fazi Trace Scheduling algoritma. Istovremeno, operacije *B*, *C* i *E* su preseljene ispod grananja, i moraju se kopirati u granu koja nije na tragu. Operacije *D* i IF e preseljene iznad spajanja, pa se i one moraju seliti u granu van traga. Da bi se

sačuvala korektnost programa, mora se realizovati selidba grananja preko spoja na gore. To se realizuje tako da se grananja javljaju u obe grane iznad spoja i radi se dodatno kopiranje operacija. Primer za to je kopiranje operacija *C* i *E* u off-trace granu kopiranog IF-a:



Sl. 3.6. Seljenje operacija grananja preko spoja na gore

Generalno, sve operacije koje su na tragu bile ispod spajanja i iznad grananja moraju biti kopirane u off-trace grane kopiranog IF-a. Ako se među tim operacijama nalazi neka operacija skoka, ona biva kopirana kao i svi drugi skokovi, ali se ovo pravilo na nju *ne* primenjuje rekurzivno. Ovim je definisano i pravilo selidbe grananja preko spoja na gore. Jednostavnije pravilo je da se grananja kopiraju u obe grane tako da za sva četiri moguća dinamička traga koja se jave nakon ovog kopiranja, moraju da se zadrže sve operacije na tragovima kao i u originalnom kôdu.

3.1.5. Generalnija pravila selidbi

Generalnija pravila osnovnih selidbi je definisao Nicolau u svojoj tehnici optimizacije prilikom prevođenja nazvanoj Percolation scheduling. Ta pravila sadrže generalizacije pravila za selidbe operacija opisanih detaljnije u ovoj knjizi. Te generalizacije su relativno trivijalne i logički direktno proizilaze iz već navedenih transformacija. One obuhvataju slučajeve: višestrukih grananja u jednom ciklusu, višestrukih spojeva i uskakanja spoja u granu, neposredno posle grananja. Generalizovana pravila nisu detaljnije razmatrana u ovom radu, a mogu se naći u {NIC 85}.

3.1.6. Pomoćni blokovi – Pojavljivanje i eliminacija blokova

U dosadašnjim primerima podrazumevano je da uslovno grananje sa traga uvek vodi ka nekoj operaciji koja je van traga. To nije uvek slučaj, a očigledan je slučaj IF THEN strukture sa jednim bazičnim blokom u THEN grani u kojoj je verovatnije da će se THEN grana izvršiti. Tada van traga ne postoji nijedna operacija, kada se ne obavi THEN bazični blok, ali postoji potreba da se obnove i grananje i spoj. Pritom, nijedna grana tog grananja ne vodi ka nekoj fiksnoj (u tom momentu fiksnoj) operaciji. Obnavljanjem grananja i spoja će se, međutim, izgenerisati operacije koje će formirati novi bazični blok. Postoje u različitim fazama Trace scheduling-a trenutci u kojima je potrebno između traga i ostatka grafa toka vršiti ubacivanje pomoćnih blokova, nazvanih **baferi** traga. Osim na off-trace grane grananja, baferi se ubacuju i na off-trace grane spajanja, kao i na početak i na kraj traga. Bazični blokovi koji tokom optimizacije ostaju prazni uklanjaju se iz grafa toka. Na taj način se optimizacijom istovremeno radi restrukturiranje grafa toka kontrole.

3.1.7. Eksplozija kôda

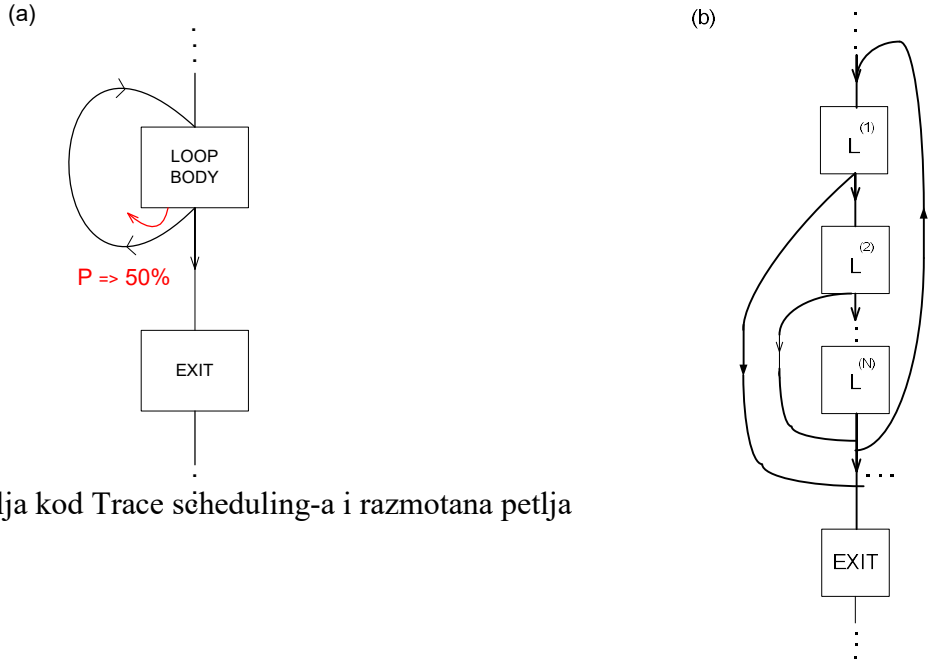
Operacije nastale kopiranjem u okviru procesa obnavljanja Tgrananja i Tspajanja povećavaju kôd van traga. Svaka optimizacija novog traga dovodi do novog povećavanja kôda. Čak se postavilo pitanje da li proces Trace schedulinga dovodi do konačne veličine kôda, jer se tokom algoritma javljaju stalna nova kopiranja i pravljenja velikog broja novih operacija. Dokazano je da proces ipak konvergira, mada generisanje velikog broja kopija operacija predstavlja istinski problem i naziva se eksplozijom kôda. Da bi se donekle smanjila eksplozija kôda, uvedeno je više heuristika koje umanjuju broj kopiranih operacija. Paralelizam u kôdu izvan petlji dobijen Trace scheduling-om je reda 5-10 puta, ali uz tipično povećanje veličine kôda (zauzeća programske memorije) od preko 10 puta.

3.2. Optimizacija kôda petlji primenom Trace scheduling-a – razmotavanje petlji

Programske petlje imaju na kraju svog među-kôda uslovni skok kod koga ishod-skok (true) označava da će se izvršiti nova iteracija programske petlje, a ukoliko se ne dogodi skok (false), izlazi se iz petlje. Na to grananje se takođe primenjuju pravila selidbi operacija uvedena u ovom poglavlju, bez obzira što se radi o skoku unazad. Osim toga, uslovni skok na kraju iteracije je očigledan primer uslovnog skoka kod koga postoji izrazita asimetrija u verovatnoći izvršavanja True i False grane. Statistike pokazuju da je prosečna verovatnoća oko 90% da će doći do skoka kod uslovnih skokova na kraju petlje. Zato su ovi skokovi idealni za direktnu primenu Trace scheduling algoritma.

Primenom logike Trace Scheduling algoritma, uvek bi verovatnoća da se dogodi skok bila veća, tako da bi se uvek iza kôda iteracije na tragu nalazio kôd nove iteracije. Nekritičkom primenom logike koja se primenjuje za ostala grananja, dobio bi se na kraju beskonačan trag sa iteracijama petlje. Zato se primenjuju sledeći koraci:

a. umesto da se pravi beskonačan trag, pravi se trag koji čini N polaznih iteracija
b. na kraju tog ograničenog traga se ponovo nalazi uslovni skok kojim se vraća kontrola na početak traga, čime se formira nova (veća) iteracija od N celokupnih starih iteracija
c. radi se optimizacija traga nove iteracije (tj. najverovatnijeg traga, razmotane petlje) privremeno zanemarujući ranije granice iteracija, osim na kraju traga



Sl. 3.7. Petlja kod Trace scheduling-a i razmotana petlja

d. određuju se nova mesta grananja za sva ranija iskakanja iz nove iteracije koja moraju da postoje kada je u vreme prevođenja ne znamo koliko će biti iteracija originalne petlje. Ta mesta se dobijaju seljenjem Tgrananja na gore (posledica bookkeeping faze), odnosno ekvivalentnim seljenjem operacija preko Tgrananja na dole.
e. na nova mesta grananja (posledica ranijih mesta izlaska iz petlje) se primenjuje obnavljanje grananja, tako da se dobijaju novi bazični blokovi nastali od operacija prebačenih preko T-grananja na dole u off-trace granu. Pritom se generišu novi bazični blokovi za iskakanje iz petlje.
f. Svi novodobijeni bazični blokovi se spajaju iza tačke završetka traga koji čini novu iteraciju

Zbog načina stapanja N iteracija, koristi se termin razmotavanje petlji N puta. Trag od N polaznih iteracija se naziva razmotana petlja ili samo nova iteracija.

Po kriterijumu mogućnosti pojednostavljivanja prethodno navedenih koraka u postupku razmotavanja, postoje dve osnovne kategorije programskih petlji. Prva kategorija su petlje kojima je unapred, u vreme prevođenja, poznat broj iteracija koji će se izvršiti. Druga kategorija su petlje kod kojih se izlazi ispitivanjem uslova izračunatog tokom izvršavanja, a ne može se odrediti u vreme prevođenja. Detaljnije će biti razmotrene obe kategorije.

3.2.1. Broj iteracija petlje poznat u vreme prevođenja

Ako je broj iteracija petlje poznat u vreme prevođenja i jednak je broju M, moguće je izbeći obnavljanje grananja unutar razmotane petlje. Trivijalno rešenje je da se petlja razmota M puta. U tom slučaju sve iteracije su uključene u trag, a dolazi i do stapanja bazičnih blokova koji su prethodili ili se nalazili iza petlje. Tada se, ako u originalnoj programskoj petlji nije bilo uslovnih grananja, najčešće formira ogroman bazični blok. Često je M suviše veliko, tako da se zauzima suviše veliki deo memorije, ako se uradi potpuno razmotavanje. Zato se bira N << M tako da se dobije dovoljno velika nova iteracija koja ima dovoljno paralelizma, a ne zauzima suviše programske memorije.

Izbor veličine N nije trivijalan. Kod jedne intuitivne vrste izbora se traži N koje daje dovoljno paralelizma, nije preterano veliki i istovremeno je činilac broja M. Koliko treba da bude N da bi bilo dovoljno paralelizma je predmet kasnije detaljnije diskusije paralelizma programskih petlji i vrsta zavisnosti po podacima kod petlji. Kada se odredi minimalno N sa stanovišta paralelizma, pristupa se određivanju prve veće ili jednake vrednosti N koja je činilac broja M.

3.2.1.2. Ljuštenje petlji

U petljama najčešće ima više paralelizma, nego u kôdu izvan petlji. Zato se u optimizaciji kôda često prva(e) iteracija(e) pridružuje kôdu koji prethodi petlji i time se uvećava bazični blok iz koga se ulazi u programsku petlju. Ovim povećavanjem se vrlo verovatno povećava i mogućnost preklapanja operacija, a samim tim i paralelizam. Ukoliko se izvršava dovoljno iteracija, ovim se ne smanjuje paralelizam u petlji, a povećava se u kôdu na ulasku u petlju.

Ista transformacija se može izvoditi i sa zadnjom(im) iteracijom(ama), tako da trag iza petlje dobije više paralelizma. Pritom se u tipičnom slučaju ne smanjuje mnogo paralelizam u razmotanoj petlji. Kako se ovom transformacijom sa krajeva petlje ljušte iteracije, naziv tehnike je ljuštenje petlji (*loop peeling*).

Često se pri izboru N kod razmotavanja prvo radi ljuštenje, a broj sljuštenih iteracija se bira po dva kriterijuma:

a. Da N ima minimalnu vrednost za koju nova iteracija ima dovoljno paralelizma da se zauzmu resursi mašine,
b. Da se ljuštenjem ukupno τ iteracija dobija broj M- τ koji je deljiv sa željenom vrednošću N.

3.2.2. Broj iteracija petlje nije poznat u vreme prevođenja

U ovom slučaju se mora primeniti obnova grananja unutar razmotane petlje proizašlih iz grananja na kraju originalne petlje. Svi koraci Trace scheduling algoritma od a. do f. iz 3.2. moraju se kompletno sprovesti. Ukoliko više razmotamo petlju, postoji veća eksplozija kôda u ovom slučaju. Zato se bira N tako da ima minimalnu vrednost za zahtevani nivo paralelizma.

3.2.3. Operacije invarijantne u odnosu na petlju

Operacija je invarijantna u odnosu na petlju ako višestruko izvođenje operacije ima isti efekat kao i da je operacija obavljena samo jedanput. Ovakve operacije pružaju dodatnu mogućnost skraćivanja vremena izvršavanja kôda. Optimizacija petlje razmotavanjem jasno definiše granice traga u Trace scheduling-u kao granice razmotane petlje. Svi ciklusi operacija na tragovima koji prethode petlji moraju se završiti pre petlje. Granice početka operacija slobodnih na vrhu svih tragova iza završetka petlje su ciklus odmah iza razmotane petlje. Izuzetak su operacije invarijantne u odnosu na petlju. Ako se primeni sledeći redosled optimizacije:

a. Procena paralelizma neposredno pre i neposredno posle petlje i ljuštenje petlje ako je potrebno.
b. Izbor N za razmotavanje (sljuštene?) petlje i njena optimizacija.
c. Traženje operacija koje su slobodne na dnu bazičnog bloka pre petlje i slobodne na vrhu bazičnog bloka posle petlje.
d. Ispitivanje da li je neka od tih operacija invarijantna u odnosu na petlju i da li se na osnovu zavisnosti po podacima može ubaciti u petlju.
e. Traženje mogućnosti da se u već realizovan raspored petlje ubaci još neka od tih operacija invarijantnih u odnosu na petlju, tako da se iskoriste resursi mašine koji ne bi inače bili iskorišćeni.

Razmotavanje petlji sa stanovišta paralelizma će biti detaljno analizirano u poglavlju o paralelizmu u programskim petljama.