1. Uvod u instrukcijski nivo paralelizma

1.1. Paralelizam na instrukcijskom nivou i mrežno planiranje

U nizu oblasti ljudskog delovanja, potrebno je planirati redosled i dinamiku obavljanja pojedinih poslova – podzadataka potrebnih da se ostvari željeni cilj. Da bi se u vremenu i prostoru pravilno raspodelili raspoloživi resursi prilikom realizacije podzadataka i postiglo najkraće vreme realizacije ukupnog posla (zadatka), razvijena je oblast koja se naziva mrežno planiranje. Kod mrežnog planiranja, između pojedinih podzadataka definišu se relacije zavisnosti. U suštini, relacija zavisnosti znači da sprovođenje nekog podzadatka zahteva da se obezbede potrebni ulazi, a ti ulazi su ujedno rezultat(i) završetka nekog(ih) prethodnog(ih) podzadat(a)ka. Za svaki podzadatak se definiše vreme trajanja (očekivano, najduže moguće, izvedeno iz funkcije gustine verovatnoće trajanja, i sl.).

Kako je ta relacija zavisnosti binarna, pogodno je kod mrežnog planiranja koristiti grafove za grafičko predstavljanje. Naravno, grafovi ne služe samo za prikazivanje, već se grafovski algoritmi mogu koristiti za optimizaciju korišćenja resursa i postizanje minimalnog vremena završetka ukupnog posla. Zbog jasne orijentacije zavisnosti, jer je jedan podzadatak zavisan od završetka drugog, binarna relacija je antisimetrična i koriste se orijentisani grafovi. Jedan od načina predstavljanja plana kod mrežnog planiranja je da se čvorovima grafa predstave podzadaci, a granama grafa zavisnosti podzadataka. Trajanje podzadataka može se vezati bilo za čvor grafa, bilo za granu grafa.

U slučaju vezivanja trajanja za čvor grafa, smatramo da podzadatak ne može, pre sopstvenog završetka, da generiše izlaze potrebne za druge podzadatke. Drugačije rečeno, izlazi podzadataka za druge podzadatke ne mogu da budu vremenski distribuirani. Ukoliko postoji deo podzadatka koji po prirodi problema može da generiše podskup rezultata potrebnih drugim podzadacima pre sopstvenog završetka, a želi se vezivanje njegovog trajanja za čvor, podzadatak se kod takve predstave mora dalje dekomponovati na pod-podzadatke sa različitim trenucima završetka.

Drugi način predstavljanja je da se trajanje podzadatka veže za grane koje izlaze iz čvora. U ovom slučaju se trajanje može vezati za deo rezultata podzadataka i može biti različito za različite grane grafa koje izlaze iz čvora.

Osnovi cilj mrežnog planiranja je da se uz raspoložive resurse i ograničenja redosleda izvršavanja operacija postigne minimalno vreme trajanja do završetka ukupnog posla. Ovaj kratak uvod vezan za mrežno planiranje napravljen je da bi se uočile sličnosti između paralelizacije kôda na instrukcijskom nivou i mrežnog planiranja, jer je cilj isti – minimalno vreme završetka programa, odnosno posla.

1.1.1. Mrežno planiranje i paralelizacija kôda na nivou instrukcija.

Prve decenije razvoja računarstva su bile obeležene pokušajima da se izgrade procesori što manjih dimenzija, jer su dimenzije prvih računara bile prevelike za širu primenu. Kraj te prve faze obeležen je pojavom mikroprocesora. Dalja minijaturizacija u proizvodnji integrisanih kola je odjednom otvorila tehnološku mogućnost da se koriste svi oblici

paralelizma u obradi podataka. Dotadašnji programski prevodioci su generisali sekvencijalan kôd, pa je i u oblasti programskih prevodilaca morala da se napravi revolucija u paralelizaciji (optimizaciji) kôda. Treba napomenuti da je pojam optimizacije širi od pojma paralelizacije kôda (npr. optimizacija iskorišćenja registara je deo optimizacija u programskim prevodiocima), ali autori koji se bave paralelizacijom kôda su usvojili taj termin u svojim radovima za postupke traženja najkraćeg vremena izvršavanja programa. U tom smislu će se termin optimizacija kôda koristiti i u ovoj knjizi.

Prvi algoritmi optimizacije kôda su se vezivali samo za celine programskog kôda između uslovnih skokova i spojeva u kôdu. Definišimo prvo bazični blok. Bazični blok je deo programskog kôda koji nema uskakanja u kôd izuzev pre prve instrukcije i na kraju bazičnog bloka i nema u sebi grananja, sem u zadnjoj instrukciji. Pre prve instrukcije bazičnog bloka nalaze se instrukcija(e) skoka i/ili uskakanje(a) u kôd. Množine u prethodnoj rečenici su uvedene iz dva razloga:

1. Kod nekih paralelnih mašina je moguće skakanje na više adresa u istom ciklusu na osnovu više boolean promenljivih (sekvencijalno posmatrano, to su uzastopne instrukcije skoka kod kojih se samo zadnja može posmatrati kao ona koja prethodi bazičnom bloku, a paralelno posmatrano - paralelna instrukcija skoka na više od dve moguće sledeće adrese)
2. Uskakanje na isto mesto u kôdu iz većeg broja bazičnih blokova je moguće (višestruka uskakanja na isto mesto u kodu)

Poslednja instrukcija bazičnog bloka je ili instrukcija uslovnog skoka ili se iza te instrukcije nalazi uskakanje(a) u kôd. Po samoj definiciji je bazični blok vezan za sekvencijalni kôd, ali će se u daljem tekstu uvoda posmatrati kao celina kôda nad kojom se mogu definisati transformacije kojima se narušava prvobitno definisan redosled izvršavanja instrukcija. Primenjene transformacije će omogućavati da se instrukcije izvršavaju u izmenjenom redosledu ili paralelno (engl. - out of order execution). U slučaju klasičnih procesora, bazični blok je deo kôda u kome se kao generator adrese naredne sekvencijalne instrukcije koristi programski brojač , a promena (povećanje) njegove vrednosti se vezuje isključivo za broj bajtova ili reči koje je zauzimala prethodno učitana instrukcija. Da bi se generalizovao pristup i izbeglo vezivanje za instrukcije, uvedena je apstrakcija operacija. Ona se definiše – kao minimalna aktivnost koja: generiše rezultat, a posledica je unarne ili binarne operacije nad podacima ili kontroliše tok izvršavanja programa. U slučaju klasičnih procesora, pojam operacije uveden u ovoj knjizi odgovara atomskim operacijama RISC procesora, odnosno mašinskim instrukcijama CISC procesora. Kod nekih kategorija paralelnih mašina instrukcija sadrži veliki broj operacija (Very Large Instruction Word procesori – VLIW, odnosno Explicitly Parallel Instruction Computing - EPIC procesori), što će kasnije detaljno biti opisano u ovoj knjizi.

Definicija: **Dinamički trag** izvršavanja programa predstavlja informaciju o redosledu izvršavanja svih operacija u toku izvršavanja.

Kod sekvencijalnih mašina, redosled izvršavanja operacija definisan u vreme prevođenja unutar bazičnog bloka je zadržan i u toku izvođenja. Na granicama bazičnih blokova se u toku izvršavanja mogu ispitivati uslovi za skok i time će se određivati dinamički tragovi izvršavanja, za dati skup ulaznih podataka programa. Kompaktan način predstavljanja dinamičkih tragova je preko zapisa o redosledu izvršavanja bazičnih blokova, jer, zbog

definicije bazičnog bloka, izvršavanje njegove prve operacije dovodi do izvršavanja svih operacija bazičnog bloka. Dinamički tragovi mogu biti (i najčešće jesu) za različite skupove ulaznih podataka različiti, a ukupan broj mogućih dinamičkih tragova može u opštem slučaju biti eksponencijalno zavisan od broja grananja. Međutim, redosled izvršavanja nije samo uslovljen redosledom operacija unutar bazičnih blokova. On je uslovljen i time što završetak poslednje instrukcije nekog bazičnog bloka na sekvencijalnoj mašini, ako se bazični blok završava grananjem, uslovljava započinjanje prve instrukcije nekog narednog bazičnog bloka na dinamičkom tragu. U kasnijim poglavljima će detaljnije biti opisana ova relacija koju zasada uvodimo samo kao pojam – zavisnost po kontroli ili kontrolna zavisnost.

Paralelizacija kôda na instrukcijskom nivou se u osnovi ne razlikuje od osnovnih metoda mrežnog planiranja, ako se vežemo za bazični blok. Da bi se generalizovala paralelizacija kôda, operacije će se tretirati kao najmanja celina u programu koja daje smisleni izlaz. Operacija kao ulaz dobija podatke generisane od neke druge operacije i generiše rezultat na osnovu zadatog kôda operacije. Kôd operacije je poznat pre izvršavanja i pretpostavlja se da je generisan ranijim postupkom prevođenja izvornog programa. U slučaju klasičnih sekvencijalnih procesora, operacija je najčešće izjednačena sa mašinskom instrukcijom. Operacija u paralelizaciji instrukcija je ekvivalent podzadatka u mrežnom planiranju. Kako operacija koja ne definiše kontrolu tola generiše rezultatu obliku podatka, zavisnost početka neke operacije od rezultata neke druge operacije će biti nazvana zavisnost po podacima. Izuzev retkih izuzetaka, u postojećim procesorima postoji samo jedan rezultat operacije. Jedan primer izuzetka je da operacija može da generiše istovremeno potreban rezultat na izlazu aritmetičke jedinice i potreban signal (flag) u procesorskoj statusnoj reči na osnovu koga se obavlja grananje. Trajanje operacija kod računara se može definisati na dva načina:

1. Vezuje se za ukupno trajanje operacije.
2. Vezuje se za broj ciklusa koliko je potrebno operaciji od trenutka kada je dobila argumente do trenutka kada je generisala rezultat. Ovo trajanje uzima u obzir samo deo ukupnog trajanja operacije – deo svih ciklusa izvršavanja. Svrha uvođenja ovako okrnjenog trajanja operacije je izdvajanja dela ukupnog trajanja koje dovodi do kašnjenja drugih operacija zbog postojanja zavisnosti od posmatrane operacije. Ovako definisano trajanje naziva se efektivno trajanje operacije ili samo trajanje zavisnosti. U nekim predstavama se uvodi i pojam kašnjenje zavisnosti, gde se posmatra minimalan pomeraj početka zavisne operacije u odnosu na početak operacije od koje je zavisna, izražen u ciklusima.

Kasnije će biti pokazano da se, zavisno od primene, koriste prethodne definicije (ponekad i dve) sa jasno razdvojenim značenjem. Trajanje operacije (ili zavisnosti ili kašnjenja zavisnosti), za razliku od mrežnog planiranja, izražava se prilikom paralelizacije (optimizacije) kôda brojem ciklusa procesora i diskretnog je karaktera.

1.2. Grafovi zavisnosti po podacima i tipovi zavisnosti unutar bazičnog bloka

1.2.1. Prave zavisnosti po podacima

Prava zavisnost po podacima je relacija kod koje podatak, koji jedna operacija generiše kao rezultat, predstavlja argument druge operacije. Ako takva relacija postoji, ne sme se menjati redosled izvršavanja tih operacija. Kako se razmena podataka između operacija

obavlja najčešće posredstvom memorije (registarska, ili bilo koji deo u hijerarhiji interne memorije), jedan od korišćenih naziva za definisanje prave zavisnosti po podacima potiče od redosleda obavljanja operacija koje bi narušile ispravnost programa. Taj termin koji označava narušavanje prave zavisnosti po podacima je ***hazard čitanje pre upisa***. Reč hazard označava da je memorijska lokacija očitana u cilju dohvatanja argumenta, pre nego što je odgovarajući rezultat (predviđen da bude argument) upisan u tu memorijsku lokaciju od strane operacije koja treba da upiše rezultat, pa se time narušava ispravnost izvršavanja programa,

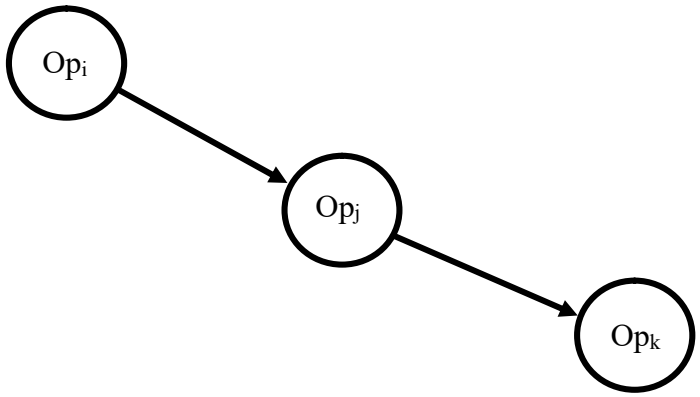
Najšire se ipak za ovu vrstu zavisnosti koristi već uvedeni termin prava zavisnost po podacima. Prava zavisnost po podacima očigledno predstavlja suštinsko ograničenje u paralelizaciji kôda. Kada govorimo o rezultatu prethodne operacije kod prave zavisnosti po podacima, najčešće podrazumevamo generisani izlaz aritmetičko logičke jedinice koji se negde upamti. U nekim slučajevima je to i/ili signal (flag) u procesorskoj statusnoj reči koji nam određuje uslov kod uslovnih skokova. Load operacija zavisi po podatku preko memorije (mora potreban podatak da bude prethodno upisan u memoriju), ali i zavisna od izračunate adrese, ako sama adresa nije deo instrukcije. Store operacija zavisi po podatku upisanom u registar i od izračunate adrese, ako sama adresa nije deo instrukcije.

Ako pod operacijom podrazumevamo RISC instrukciju, u tipičnom slučaju se u prva dva ciklusa takve operacije obavljaju *fetch* i *decode* faze instrukcije. U tim fazama dakle nije neophodno da instrukcija ima na raspolaganju argumente, a tek treća faza podrazumeva preuzimanje rezultata neke od prethodnih operacija. Dakle, može postojati delimično preklapanje operacija između kojih postoji prava zavisnost po podacima. Ta činjenica uvodi razliku između trenutka početka operacije i trenutka u kome zaista prava zavisnost po podacima ograničava preklapanje operacija. Ovakvo delimično preklapanje operacija između kojih postoji zavisnost po podacima se u RISC terminologiji naziva i prosleđivanje (forwarding). U većini analiza u kojima se ne razmatra problem uslovnih grananja, faze fetch i decode će biti zanemarene kod instrukcija - operacija, jer suštinski ne ograničavaju paralelizam. Njihovo izvođenje nije zavisno po podacima, tako da se teorijski mogu obaviti u bilo kom trenutku, jer postoji mogućnost čuvanja dekodovane instrukcije! Pod dekodovanom instrukcijom se podrazumevaju instrukcije u takvoj formi, da ciklusi iz faze izvršavanja mogu da počnu bez ijednog ciklusa kašnjenja zbog potrebe tumačenja instrukcije. Samo najnoviji procesori zaista dozvoljavaju odvajanje ove dve faze instrukcije od ostatka izvršavanja instrukcije, što podrazumeva privremeno čuvanje dekodovane instrukcije. Jedan od načina čuvanja dekodovane instrukcije je preko horizontalnog mikroprogramskog kôda instrukcije, tako što se pamte svi kontrolni signali neophodni za fazu izvršavanja.

Naravno da unapred urađene *fetch* i *decode* faze operacije mogu da budu nepotrebne, ako se urade za operaciju nakon nekog grananja, a ispostavi se da su urađene za operaciju za koju se naknadno utvrdi da se neće izvršiti. Tada su one i nepoželjne, jer zauzimaju resurse mašine u delu za fetch i decode, kao i deo memorije potreban za privremeno čuvanje dekodovanih instrukcija. Bitan parametar zavisnosti po podacima je, dakle, trajanje operacije od trenutka kada preuzme argumente - rezultat prethodne operacije, do trenutka generisanja sopstvenog rezultata. Trenutak generisanja sopstvenog rezultata je obično u zadnjem ciklusu operacije.

Analogno mrežnom planiranju, uvedimo grafove zavisnosti po podacima. U početku će se koristiti pojednostavljena predstava u kojoj se ne prikazuje trajanje bilo operacije, bilo kašnjenja zavisnosti po podacima. Tada čvorovi grafa predstavljaju operacije, a orijentisane grane relaciju prethodno definisane prave zavisnosti po podacima. Orijentacija grane je od operacije koja generiše rezultat, ka operaciji koja koristi rezultat. Ako je u originalnom kôdu operacija Op_i prethodila operaciji Op_j, a rezultat Op_i se direktno koristi kao argument operacije Op_j, postoji prava zavisnost po podacima operacije Op_j od operacije Op_i. Ako postoji još i operacija Op_k koja je zavisna po podacima od Op_j, tada možemo da nacrtamo deo grafa zavisnosti po podacima koji uključuje ove tri operacije.

Očigledno je da se ne može menjati ni redosled operacija Op_i i Op_k, kao posledica pravih zavisnosti po podacima između Op_i i Op_j, kao i između Op_j i Op_k. Operacija Op_k je posredno zavisna po podacima od Op_i. Dakle, tu postoji zavisnost po podacima, ali je ona ostvarena indirektno, preko operacije Op_j. Zato se uvodi pojam ***direktne prave zavisnosti po podacima*** koji precizira da zavisna operacija neposredno koristi rezultat operacije od koje je zavisna. Indirektno ostvarena zavisnost po podacima je posledica tranzitivnosti relacije zavisnosti po podacima.

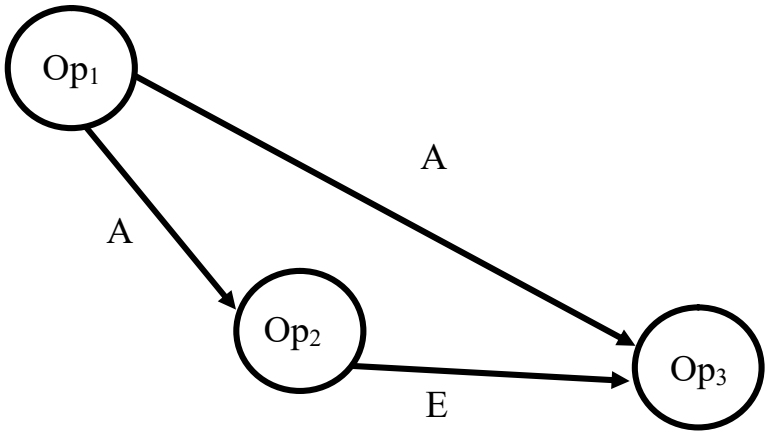


Sl. 1.1 Graf sa prikazanim direktnim pravim zavisnostima po podacima

U daljem tekstu će se prilikom korišćenja termina prava zavisnost po podacima podrazumevati slučaj direktne prave zavisnosti po podacima, jer su ostale (indirektne) zavisnosti posledica postojanja direktnih pravih zavisnosti po podacima. Iz ograničenja definisanih direktnim pravim zavisnostima po podacima, očigledno se mogu izvesti i sve ostale prave zavisnosti po podacima. Međutim, može se postaviti i obrnuto pitanje: Da li je neophodno poznavanje svih direktnih pravih zavisnosti po podacima, da bi se prikazala sva ograničenja u redosledu izvršavanja operacija, ili mogu postojati redundantne direktne prave zavisnosti po podacima (grane grafa)?

Primer: Sledeće tri operacije se nalaze u sekvencijalnom kôdu bazičnog bloka:
Op_1: A:=B+C;
Op_2: E:=A-D;
Op_3: F:=E*A:

U tom slučaju, graf zavisnosti po podacima prikazan je na Sl. 1.2, a uz grane su dodati nazivi promenljivih (registarskih ili memorijskih lokacija) preko kojih se razmenjuju rezultati i argumenti operacija.



Sl. 1.2. Grana grafa Op_1,Op_3 je tranzitivna, iako je direktna prava zavisnost

Taj graf sadrži granu od Op_1 do Op_3 , jer je rezultat operacije Op_1 ujedno argument operacije Op_3. Uprkos činjenici da ta grana označava direktnu pravu zavisnost po podacima, ona je redundantna zbog toga što Op_3 već zavisi od Op_2, a Op_2 već zavisi od Op_1. Grana je redundantna jer ne nosi nikakvu novu informaciju o dozvoljenim redosledima izvršavanja operacija, koju ostale grane nisu već definisale. Takva grana se naziva tranzitivnom, jer je zavisnost po podacima tranzitivna relacija. U ovom pojednostavljenom slučaju u kome ne razmatramo trajanje operacija, važi stav o tranzitivnosti: Ako operacija Op_2 ne može da se izvrši pre operacije Op_1, a operacija Op_3 ne može da se izvrši pre operacije Op_2, tada ni operacija Op_3 ne može da se izvrši pre operacije Op_1. Poželjno je prilikom paralelizacije kôda ***eliminisati tranzitivne grane***, jer su redundantne, kako bi se grafom zavisnosti po podacima sa minimalnim brojem grana definisala sva ograničenja u paralelizaciji kôda.

1.2.2. Antizavisnosti po podacima

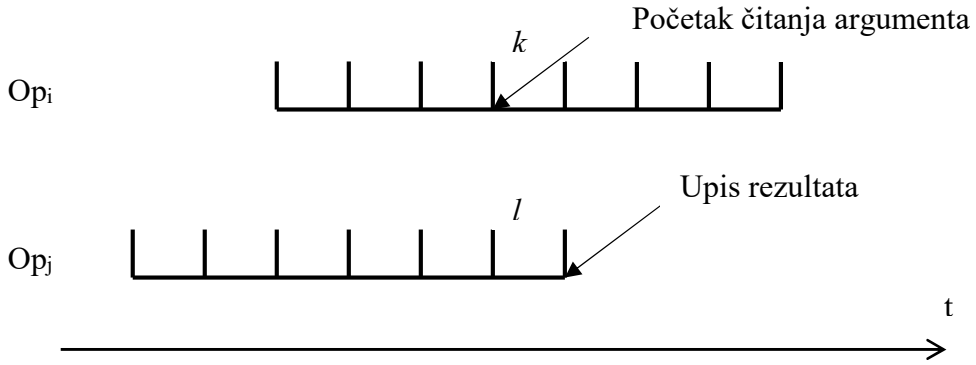
Pored pravih zavisnosti po podacima, javljaju se zavisnosti po podacima koje potiču od ponovnog korišćenja iste memorijske lokacije od strane operacija. Osnovna ideja ponovnog korišćenja lokacija je minimizacija zauzeća memorije. Ako neka operacija Op_i, koja u originalnom kôdu prethodi operaciji Op_j, čita iz lokacije u koju operacija Op_j upisuje, redosled operacija se ne sme menjati. Razlog za to je činjenica da bi operacija Op_j upisala novu vrednost u lokaciju, pa bi operacija Op_i pročitala tu novoupisanu vrednost umesto stare vrednosti predviđene originalnim sekvencijalnim kodom. Za ovakvu pogrešnu paralelizaciju koristi se termin - ***hazard upis pre čitanja***. Termin koji se mnogo češće koristi i koji će se koristiti u daljem tekstu je ***antizavisnost po podacima***.

Jedini razlog za postojanje antizavisnosti je ušteda u korišćenju memorijskih lokacija. Za razliku od prave zavisnosti, ova zavisnost se može eliminisati postupkom preimenovanja, odnosno alociranjem dvostruko većeg prostora u memoriji. Na ovaj način čuvamo staru vrednost promenljive u jednoj lokaciji, a novoupisanu vrednost čuvamo u drugoj lokaciji. Promenljivu sa starom vrednošću je neophodno sačuvati samo do trenutka dok i poslednja operacija koja je trebalo da pročita tu staru vrednost ne preuzme argument. Preimenovanje se koristi kao termin, jer na nivou izvornog kôda bi to značilo uvođenje nove promenljive i samim tim statičko alociranje dodatnih memorijskih lokacija potrebnih za pamćenje rezultata operacija. Danas mnogi procesori podržavaju dinamičko

preimenovanje, koje se često naziva preimenovanje registara, kako je opisano u poglavlju o superskalarnim procesorima.

U jednom trenutku razvoja računarstva, predviđalo se da će osnovni pravac evolucije biti u pravcu programskih jezika koji će na nivou koncepta jezika dozvoljavati upis u svaku memorijsku lokaciju samo jedanput. Takvi jezici se nazivaju jezicima za funkcionalno programiranje, a kao njihova osnovna osobina, ističe se da nemaju bočnih efekata (side effects). Primer takvog programskog jezika je Sisal (Single assignment language) kod koga se pravilima na nivou višeg programskog jezika onemogućava ponovno upisivanje u istu lokaciju i time na nivou jezika eliminiše antizavisnost. Ovaj pravac razvoja paralelizacije kôda danas nije prevladao, jer se neracionalno koristi memorija zbog eliminacije **svih** antizavisnosti, čak i kada one ne ograničavaju paralelizaciju kôda.

Sledi detaljnija analiza antizavisnosti za slučaja u kome operacije traju više ciklusa. Kada postoji antizavisnost, uočimo odnose po ciklusima između operacija kod kojih postoji relacija antizavisnost, ako se ne radi preimenovanje, a podaci se pamte u registrima koji su deo protočnih stepeni procesora. Ako operacija Op_i prethodi operaciji Op_j u originalnom kôdu i Op_i čita lokaciju, a operacija Op_j upisuje, tada ciklus čitanja argumenta za operaciju Op_i mora da bude pre ciklusa ili u istom ciklusu na čijem kraju operacija Op_j radi upis svog rezultata. Naravno, ovo važi ako ne radimo preimenovanje. Ako operacija Op_j upisuje na kraju svog ciklusa l, a Op_i čita na početku svog ciklusa k (tipično je k manje od l, jer se čita argument pre generisanja i upisa rezultata), Op_j sme da započne najranije l-k ciklusa pre Op_i. Dakle, u tipičnom slučaju, Op_j sme da započne nekoliko ciklusa pre Op_i, čak i kada postoji antizavisnost koja nije eliminisana preimenovanjem.

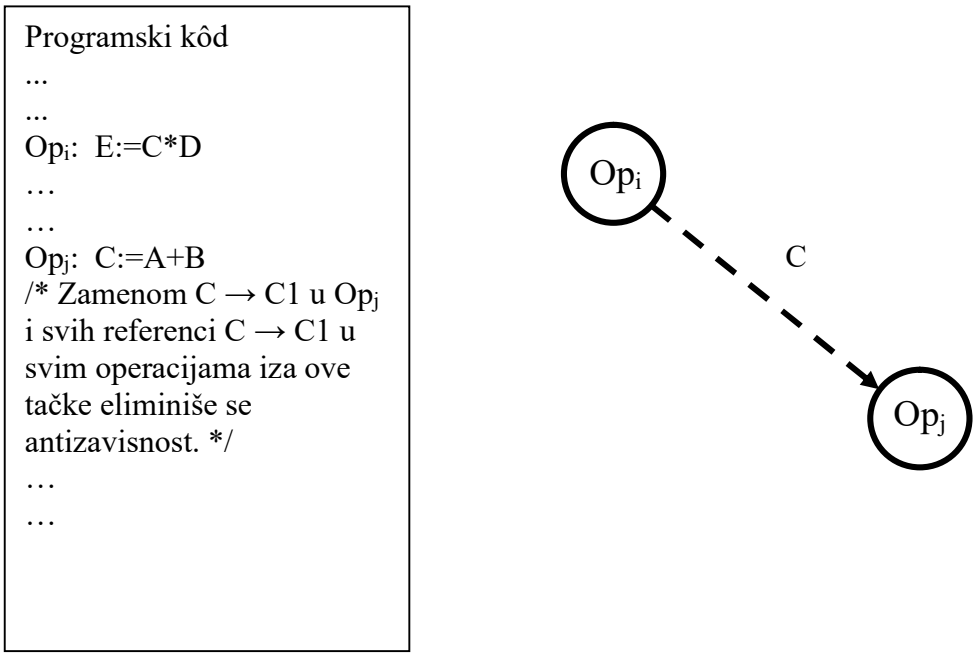


$l = 6$, $k = 4$, Op_j može da započne najranije 2 ciklusa pre Op_i

Sl. 1.3. Najranije izvršavanje zavisne operacije kod antizavisnosti po podacima

Primer preimenovanja

Pretpostavimo da su operacije Op_i i Op_j kao na slici 1.4. Tada zbog ponovnog korišćenja lokacije C nastaje antizavisnost. Uvođenjem promenljive C1 – preimenovanjem se menja operacija Op_j, a sve operacije koje su u kôdu bazičnog bloka sledile iza Op_j nakon preimenovanja koriste promenljivu C1. Time je eliminisana antizavisnost i međusobni redosled operacija Op_i i Op_j je proizvoljan.



Sl. 1.4. Eliminacija antizavisnosti preimenovanjem

1.2.3. Izlazne zavisnosti po podacima

Izlazna zavisnost se javlja kada dve operacije upisuju u istu lokaciju, pa pokušajem izmene njihovog međusobnog redosleda menjamo zaostali sadržaj lokacije. Kako taj sadržaj lokacije koriste neke kasnije operacije, ovim se narušava korektnost izvršavanja kôda. U tom slučaju se zavisnost javlja opet zbog ponovnog korišćenja lokacije. Drugi naziv je ***hazard upis pre upisa***.

Ako se u analizi zavisnosti po podacima u vreme prevođenja (Static dependency analysis) ograničimo na bazični blok, izlazna zavisnost se može javiti u dva slučaja:

1. Između ta dva upisa u lokaciju, nijedna druga operacija ne koristi sadržaj lokacije. Tada je prvi upis nepotreban kôd i kompajler bi morao da ga eliminiše kao mrtav kôd.
2. Između ta dva upisa se javlja bar jedno čitanje te lokacije. U tom slučaju se javlja par zavisnosti: prava zavisnost – antizavisnost, pa izlazna zavisnost postaje tranzitivna. U ovom slučaju eliminacija antizavisnosti preimenovanjem dovodi i do eliminacije izlazne zavisnosti.

Uočavamo da je izlaznu zavisnost na nivou bazičnog bloka nepotrebno uvoditi, jer bi dobro napravljen prevodilac u slučaju 1. eliminisao prvu operaciju upisa kao mrtav kôd, a u slučaju 2. izlazna zavisnost daje tranzitivnu granu, koja ne donosi novu informaciju o ograničenjima u redosledu izvršavanja operacija. Izlazna zavisnost se ipak javlja kao ograničenje u paralelizaciji kôda, što će u kasnijim poglavljima biti pokazano, prilikom razmatranja paralelizacije izvan granica bazičnih blokova.

1.2.4. Grafovi zavisnosti po podacima u optimizaciji kôda u vreme prevođenja

Pretpostavimo da programski jezik ne pripada kategoriji jezika za funkcionalno programiranje. Da bi se ograničenja u redosledu izvršavanja operacija na nivou bazičnih blokova predstavila jedinstveno, bez obzira na vrstu zavisnosti po podacima, i dobila prava predstava neophodna za optimizacione algoritme, uvedeni su sledeći grafovi (ujedno i koraci u optimizaciji):

a. **Polazni graf** zavisnosti po podacima na kome su prikazane sve vrste zavisnosti (direktne prave, antizavisnosti i izlazne zavisnosti).
b. Graf formiran isključivo **od direktnih pravih zavisnosti po podacima**. Ovaj graf sadrži sve čvorove i samo deo grana prethodnog polaznog grafa. Dobija se pod pretpostavkom da su svugde urađena preimenovanja, pa su zbog toga nestale grane koje su posledica antizavisnosti i izlaznih zavisnosti.
c. Graf dobijen eliminacijom svih grana proisteklih iz tranzitivnih direktnih pravih zavisnosti po podacima u grafu dobijenom pod b. Tako se dobija **graf koji je osnovni ulaz u optimizacione algoritme**, koji nema nijednu redundantnu granu.
d. Nakon optimizacije (paralelizacije) kôda u vreme prevođenja na osnovu grafa formiranog korakom iz tačke c, ako je poznat raspored izvršavanja u vreme izvršavanja, utvrđuje se da li je neophodno preimenovanje na mestima gde su u polaznom grafu dobijenom tačkom a. postojale antizavisnosti ili izlazne zavisnosti. **Samo ako se utvrdi da je neophodno, uvodi se preimenovanje**.

Tačka d. je moguća samo ako se tačno zna paralelni redosled izvršavanja operacija po ciklusima već nakon prevođenja. Preimenovanje se može uraditi u fazi prevođenja (statičko preimenovanje) i tada se postupak sprovodi u optimizacionom delu prevodioca paralelne mašine. Međutim, preimenovanje se može raditi i u toku izvršavanja, ako je mašina sposobna da dinamički, u toku izvršavanja, odredi sve vrste zavisnosti po podacima i ima rezervne lokacije za dinamička preimenovanja (pre svega registre procesora).

1.3. Lokalna optimizacija kôda za idealnu i realnu mašinu

1.3.1. Idealna mašina

Prvi korak da se od potpuno apstraktne predstave preko acikličkih grafova zavisnosti po podacima približimo realnoj mašini je da se pretpostavi da je na raspolaganju idealna mašina. **Idealna mašina** se definiše na različite načine, a najjednostavnija predstava je: mašina ima neograničen broj resursa, svaka operacija se obavlja za jedan ciklus i rezultat svake operacije može bez kašnjenja da se dostavi tako da bude argument bilo koje druge operacije. Ovakva predstava idealne mašine je suviše pojednostavljena, pa se u ovom tekstu model približava realnoj mašini prvo po tome što operacije imaju u vreme prevođenja definisan veći broj ciklusa izvršavanja (zavisno od tipa operacije i osobina same mašine) i što operacije mogu da budu takve da zahtevaju argument tek na početku sopstvenog ciklusa *k*. Razmatraće se samo prave zavisnosti po podacima.

Na početku optimizacije kôda bazičnog bloka za idealne mašine, uočićemo skup operacija koje nisu zavisne po podacima ni od jedne druge operacije. Taj skup naziva se

slobodan skup operacija na vrhu, ***data ready*** skup operacija ili samo slobodan skup operacija. Zbog neograničenih resursa idealne mašine, sve operacije iz *data ready* skupa mogu istovremeno da započnu izvršavanje u prvom ciklusu.

1.3.2. Najranije izvršavanje

Nakon istovremenog početka svih *data ready* operacija u prvom ciklusu, na početku svakog narednog ciklusa mora se ispitati da li može da se započne neka druga operacija (da li ima *data ready* operacija u tom ciklusu). Postupak sa otkrivenim *data ready* operacijama je drugačiji, zavisno od toga da li se otkrivanje zavisnosti radi u vreme prevođenja ili u vreme izvršavanja. Ako se to ispitivanje radi u vreme prevođenja, tada se sve neraspoređene *data ready* operacije raspoređuju na izvršavanje u paralelnu instrukciju tačno u ciklusu u kome postaje *data ready*. Dinamičko definisanje rasporeda na osnovu dinamički određenih zavisnosti po podacima će biti opisano u poslednjem poglavlju.

Kriterijum za ispitivanje da li u nekom ciklusu može da započne neka neraspoređena operacija je da se ispita da li postaje *data ready* pre ili u datom ciklusu. Jedino operacije kojima se argumenti izračunavaju u već raspoređenim operacijama su kandidati da postanu *data ready*. Za te operacije se prvo analizira u kom ciklusu će biti raspoloživi rezultati raspoređenih operacija generatora argumenata te neraspoređene operacije. Zatim se za svaki argument neraspoređene operacije ispita da li je trenutni analizirani ciklus za njeno raspoređivanje takav da će, u izvršavanju, argumenti neraspoređene operacije biti raspoloživi u ciklusu *k* u kome je to potrebno. Ako je to ispunjeno, operacija postaje *data ready* u tom ciklusu koji se analizira i može da se rasporedi u tom ciklusu ili bilo kom kasnijem ciklusu.

Do sada smo birali da se operacija rasporedi u najranijem ciklusu u kome postaje *data ready*, a takav postupak naziva se **najranije izvršavanje**. Raspoređivanjem svih operacija bazičnog bloka po ovom postupku, dobijamo na kraju raspored za najbrže paralelno izvršavanje na idealnoj mašini. Ovaj raspored je napravljen po kriterijumu da se operacije izvršavaju u najranijem mogućem trenutku, ali vodi do samo jednog od mogućih rasporeda za najbrže izvršavanje na idealnoj mašini.

Izuzev za operacije iz inicijalnog slobodnog skupa, za sve ostale operacije je relevantno kašnjenje od trenutka kada su na raspolaganju argumenti, do trenutka generisanja rezultata koji će biti na raspolaganju operacijama koje su zavisne po podacima od te operacije. Ovo kašnjenje će se u daljem tekstu zvati **efektivno trajanje operacije**.

Minimalno trajanje izvršavanja bazičnog bloka može se dobiti tako što se odredi broj ciklusa u kojima se završe sve operacije, po kriterijumu najranijeg mogućeg izvršavanja svih operacija. Jedina ograničenja su zavisnosti po podacima i trajanja operacija i delova operacija, jer smo razmatrali idealnu mašinu. To optimalno trajanje bazičnog bloka može se odrediti kao najduži put u grafu zavisnosti po podacima. Taj najduži put

Da bi se odredio najduži put, potrebno je odrediti postupak za određivanje dužine puta. Karakteristično je za realne mašine da se rezultat operacije generiše u poslednjem ciklusu operacije. Zato se put određuje kao zbir trajanja svih operacija na putu, pri čemu se inicijalnom slobodnom skupu operacija dodeljuje ukupno trajanje, a svim ostalim operacijama efektivno trajanje (zbog pretpostavke da se u zadnjem ciklusu generiše

rezultat). Najduži put je put najveće dužine određen na prethodan način i naziva se kritičan put, a on određuje donju granicu (optimum) trajanja izvršavanja bazičnog bloka, izražen brojem ciklusa, prilikom paralelizacije kôda bazičnih blokova u vreme prevođenja za idealnu mašinu.

1.3.3. Ostali optimalni rasporedi na idealnoj mašini

Redosled izvršavanja na idealnoj mašini dobijen najranijim izvršavanjem nije jedini redosled za optimalno izvršavanje. Moguće je napraviti redosled izvršavanja tako da se sve operacije izvršavaju u najkasnijem mogućem trenutku, uz uslov da se zadrži najkraće vreme izvršavanja. Najčešće za datu idealnu mašinu i dati graf zavisnosti po podacima operacija u bazičnom bloku postoji još mnogo drugih optimalnih rasporeda, koji se razlikuju od optimalnih rasporeda sa najranijim i najkasnijim mogućim izvršavanjem operacija.

Posebno su značajne operacije kod kojih su najranije i najkasnije moguće izvršavanje moraju da budu u istom ciklusu. To su operacije na kritičnom putu (ili kritičnim putevima). Za operacije izvan kritičnog puta, ukoliko je razlika između najranijeg i najkasnijeg ciklusa operacije veća, imamo veću slobodu u izboru trenutka započinjanja izvršavanja operacija. Po kriterijumu slobode izbora ciklusa u kojem se izvršava operacija, idealno je da operacija bude takva da u bazičnom bloku ona nije zavisna od bilo koje operacije, niti je bilo koja operacija zavisna od nje. Za takve operacije se kaže da su istovremeno slobodne i na vrhu i na dnu bazičnog bloka. Kod njih postoji najveća sloboda raspoređivanja, jer je razmak između najranijeg i najkasnijeg izvršavanja najveći.

1.3.4. Eksponencijalnost algoritma za optimalan kôd realnih mašina

Kod realnih mašina, postoji ograničenje u broju raspoloživih resursa. Da bi se precizno opisao problem, uveden je formalni model kojim se modeliraju ograničenja u paralelizaciji.

1. Realna mašina poseduje ***skup tipova resursa R*** koji označavamo sa R = {r_1, … , r_m} (m različitih tipova resursa)
2. ***Konfiguracioni vektor resursa*** R_m je vektor dimenzije m, sa elementima koji su prirodni brojevi, a k-ti element vektora R_m (k) označava broj jedinica resursa tipa r_k koji su na raspolaganju u konfiguraciji mašine.
3. ***Skup svih operacija u bazičnom bloku*** $O = \{Op_1, \dots, Op_i, \dots, Op_l\}$
4. ***Funkcija trajanja operacija d***: O -> N, gde je d(Op_i) prirodan broj kojim se operacija preslikava u broj ciklusa koliko ukupno traje i predstavlja prirodan broj. Dakle, za operaciju Op_i, d(Op_i) označava ukupan broj ciklusa potrebnih da se operacija završi. Smatra se da se nakon započinjanja operacije ne može menjati trajanje operacije, što odgovara funkcionisanju protočnih stepeni mašine.
5. ***Funkcija zauzeća resursa za operaciju*** koja se dobija kao vektor dimenzije m u svakom ciklusu izvršavanja instrukcije. R_O: O x Z -> N^m

   Vrednost tog vektora u ciklusu x je

R_O (Op_i,x) = vektor zauzeća resursa operacije Op_i u njenom ciklusu x ako je $0 \leq x < d(Op_i)$

U svim ostalim slučajevima R_O (Op_i,x) je nulti vektor (izvan ciklusa u kojima traje operacija $Op_{i\ ,\ ona\ ne\ zauzima\ resurse}$).

6. ***Graf zavisnosti po podacima*** DDG = (O,E) je orijentisani graf koji uključuje samo direktne prave zavisnosti po podacima bez tranzitivnih grana. Njime se uvodi parcijalno uređenje operacija O preko skupa grana grafa (pravih zavisnosti po podacima) ***E***. Pojedine grane se označavaju sa ε.
7. ***Funkcija kašnjenja zavisnosti*** δ: E -> N, (prirodan broj) definisana za grane grafa zavisnosti po podacima DDG, gde je $\varepsilon = (Op_i, Op_j)$, a $\delta(\varepsilon)$ je minimalno kašnjenje koje se mora poštovati između početaka operacija Op_i i Op_j, da se ne naruše zavisnosti po podacima. Do sada smo razliku između *Funkcije trajanja operacija d* i *Funkcije kašnjenja zavisnosti* δ uveli preko pojma efektivnog trajanja operacije. Potrebno je uočiti razliku - *Funkcija kašnjenja zavisnosti* δ se definiše preko minimalnog pomeraja početaka dve operacije i vezana je za grane grafa, dok se efektivno trajanje operacije vezuje samo za jednu operaciju i predstavlja razliku između trenutka primanja argumenata i predaje rezultata te operacije.

Cilj je dobiti minimalno trajanje kôda. Proizvoljan raspored koji zadovoljava navedena ograničenja će u daljem tekstu biti označavan sa σ. Da bi se definisalo minimalno trajanje kôda uvodi se pojam dužine rasporeda. Pretpostavimo da je inicijalni ciklus definisan kao nulti ciklus. Pod dužinom rasporeda podrazumevamo maksimalnu vrednost zbira početnog trenutka operacije i trajanja operacije u ciklusima, za sve operacije iz skupa O. Time se dobija najkasniji trenutak završetka bilo koje operacije iz rasporeda, kao dužina rasporeda. Pritom se naravno moraju poštovati ograničenja vezana za zavisnosti po podacima i ograničenja vezana za ukupan broj raspoloživih resursa mašine.

Ograničenja vezana za zavisnosti po podacima podrazumevaju da, ukoliko postoji zavisnost po podacima, razlika između početnih trenutaka započinjanja operacija između kojih postoji zavisnost mora da bude najmanje jednaka funkciji kašnjenja zavisnosti δ (ε). Ograničenja vezana za resurse se svode na činjenicu da bilo koji raspored mora da zadovolji nejednačinu da za bilo koji tip resursa u bilo kom ciklusu rasporeda, broj potrebnih resursa svakog tipa mora da bude manji ili jednak broju raspoloživih resursa u mašini. Uvedimo još i pojam rasporeda operacije Op – to je početni ciklus neke operacije u nekom rasporedu σ i označava se sa $\sigma(Op)$.

Ove konstatacije se mogu i formalno definisati:

1. Minimalnost: σ je minimalne dužine. Dužina rasporeda σ, $len(\sigma) = \max(\sigma(Op) + d(Op))$, za $\forall\ Op \in O$. Ovo praktično znači najkasniji trenutak završavanja operacije iz skupa svih raspoređenih operacija, a celokupni raspored se planira u vreme prevođenja.
2. Ograničenja zavisnosti po podacima zbog kašnjenja zavisnosti:
   $\forall \varepsilon = (Op_i, Op_j) \in E, (\sigma(Op_j) - \sigma(Op_i)) \geq \delta(\varepsilon)$
3. Ograničenje u raspoloživim resursima: Pod sabiranjem vektora podrazumevamo da se sabiraju sve njegove komponente – normalno sabiranje u svakoj od

dimenzija m za tipove resursa. Uvedimo specifičnu komparaciju vektora definisanu na sledeći način – $V_1 \leq V_2$ ako i samo ako za $\forall\ 0 \leq k \leq m \quad V_1(k) \leq V_2(k)$. Tada se ograničenje u raspoloživim resursima vezano za konfiguracioni vektor resursa $\boldsymbol{R_m}$ može definisati na sledeći način:

$\forall\ 0 \leq t \leq \text{len}(\sigma)$ suma po svim operacijama iz skupa O vektora zauzeća resursa operacije R_O: $\Sigma R_O\ (Op_i, t - \sigma(Op_i)) \leq \boldsymbol{R_m}$

Optimalan kôd realne mašine - Metod kojim bi sigurno došli do optimalnog rasporeda za realnu mašinu je sledeći:

1. Pretpostavićemo da je tekući skup slobodnih – *data ready* operacija dat sa:

   $\text{Data\_Ready} := \{Op \in O \mid \text{Pred}(Op) = \varnothing\}$,

   a da je skup raspoređenih operacija označen sa Scheduled i nazivaćemo ga još i parcijalni raspored, dokle god ne obuhvati sve operacije.

   Nakon raspoređivanja bilo koje operacije Op_i, mora se obavljati ažuriranje skupova Scheduled i Data_Ready prema izrazima:

   $\text{Scheduled} = \text{Scheduled} \cup Op_i$

   $\text{Data\_Ready} := (\text{Data\_Ready} - \{Op_i\}) \cup \{Op_{succ} \in \text{Succ}(Op_i) \mid \text{Pred}(Op_{succ}) \subseteq \text{Scheduled}\}$

   $\text{Succ}(Op_i)$ je skup svih operacija zavisnih po podacima od Op_i, a $\text{Pred}(Op_{succ})$ je skup svih operacija od kojih je operacija Op_{succ} zavisna po podacima. Kada je skup $\text{Pred}(Op_{succ})$ raspoređen, operacija se pridružuje skupu Data Ready. Pripadnost tom skupu posmatra kao stanje operacije u raspoređivanju, ali se još ne raspoređuje u određeni ciklus rasporeda.

Generisaćemo sve različite početne rasporede tako što rasporedimo po jednu operaciju iz početnog slobodnog skupa. Na početku, broj potencijalno različitih parcijalnih rasporeda će biti jednak broju operacija u početnom slobodnom skupu. Za sve takve parcijalne rasporede ćemo ažurirati slobodne skupove. Za svaki različit dobijeni parcijalni raspored ponovo formiramo onoliko novih parcijalnih rasporeda koji se dobijaju najranijim raspoređivanjem po jedne različite operacije iz slobodnog skupa tog parcijalnog rasporeda. Neka je τ srednja veličina skupa slobodnih operacija parcijalnih rasporeda. Već nakon drugog koraka dobijamo red veličine τ^2 različitih rasporeda. Naravno, u nekim slučajevima će doći do preklapanja rešenja do kojih se došlo na osnovu različitih parcijalnih rasporeda. Da bi se izbegla preklopljena rešenja, uvodi se srednji broj operacija koje mogu započeti u vremenu srednjeg trajanja operacija υ i označava se sa *PARO* i srednja dužina dobijenih rasporeda ℓ. Ukoliko je paralelizam u algoritmu daleko veći od paralelizma mašine, celokupan broj potencijalno minimalnih rasporeda je reda $PARO^{\ell\text{-}\upsilon}$, čime je jednostavno dokazana eksponencijalnost broja dobijenih rasporeda. Na kraju bi se između tih reda - veličine $PARO^{\ell\text{-}\upsilon}$ rasporeda morao pronaći onaj koji najmanje traje (određuje se prema $\text{len}(\sigma) = \max(\sigma(Op) + d(Op))$, za $\forall Op \in O$ za svaku operaciju svakog rasporeda). To bi bio optimalan raspored.

Iz prethodnog proizilazi da garantovano nalaženje optimalnog rasporeda za bazični blok nije primenjivo u praksi, već da se moraju primenjivati heuristički algoritmi. Zbog toga što heuristički algoritmi ne daju uvek optimalan kôd, često je u radovima korišćen termin ***lokalna kompakcija (sabijanje) kôda,*** da bi se istakla ta osobina neoptimalnosti algoritama. Danas se češće koristi termin ***lokalna optimizacija kôda***, za optimizaciju na nivou bazičnog bloka.

Prosečan broj operacija u bazičnom bloku je približno 7 i zavisi od tipa kôda , tako da bi uprkos eksponencijalnom karakteru algoritma, za većinu bazičnih blokova čak bio prihvatljiv takav algoritam. Međutim, samo jedan veći bazični blok bio bi dovoljan da dovede do neprihvatljivo dugog trajanja optimizacije kôda bazičnih blokova. Osim toga, u kasnijem tekstu knjige je istaknuto da se algoritmi ***globalne optimizacije*** kôda oslanjaju na algoritme lokalne optimizacije. Kod njih je prosečan broj operacija na ulazu algoritma za lokalnu optimizaciju daleko veći od 7. Zato je neophodno uvesti heurističke algoritme koji ne pronalaze garantovano najkraći raspored, ali u najvećem broju slučajeva ipak daju najkraći raspored. Veoma malo odstupanje od najkraćeg rasporeda i to u malom broju slučajeva je sasvim prihvatljivo u praksi. Ta odstupanja, i kada se pojave, morala bi da budu u najvećem broju slučajeva veoma mali broj ciklusa (jedan ili najviše dva).

1.4. Heuristički algoritmi

Osnovna ideja heurističkih algoritama optimizacije kôda je da se po nekom kriterijumu izabere samo jedna operacija iz skupa *Data ready* operacija i da se ona rasporedi u najranijem mogućem trenutku uz poštovanje ograničenja proizašlih iz zavisnosti po podacima i ograničenja u pogledu raspoloživih resursa. Kada je ta izabrana operacija definisana, u principu se odbacuju rasporedi koji bi se jedino mogli dobiti raspoređivanjem prvo bilo koje druge operacije iz *Data Ready* (slobodnog) skupa. Kroz takvo odlučivanje se u svakom koraku odbacuju rešenja koja bi potencijalno mogla da dovedu do optimalnog rešenja.

Kriterijum za izbor operacije iz *Data Ready* skupa koja će se prva rasporediti definisan je heuristikom. Heuristika mora da bude takva da se vrlo retko u svakom raspoređivanju operacije može dogoditi da se odbace rešenja koja dovode do optimalnog rešenja. Dužina trajanja algoritma je linearno srazmerna sa brojem odluka o izboru *data ready* operacije i tako je dobijen algoritam koji je primenljiv u realnim prevodiocima. Ovakvi algoritmi su dobili naziv raspoređivanje po listi (***List Scheduling***) {GAS 89}.

Formalno se algoritam može napisati na sledeći način:

Function Raspoređivanje_po_listi(DDG = (O,E), R_m) RETURN Raspored σ
BEGIN
    Data_Ready := {$Op \in O$ | Pred(Op) = $\varnothing$};
    Raspoređeno := $\varnothing$;
    WHILE Data_Ready $\neq \varnothing$; DO
      Izaberi jednu operaciju Op_{np} kao operaciju sa najvišim prioritetom u Data_Ready po heurističkom kriterijumu;

Min_Ciklus := max(σ (*Op*) + δ(*Op, Op*$_{np}$)), za ∀ *Op* ∈ Raspoređeno i ako postoji bar jedna ε = *(Op,Op*$_{np}$*),* u suprotnom je prvi (odnosno nulti) ciklus;
Pronađi najraniji OK_Ciklus ≥ Min_Ciklus, uz ispunjen uslov:
∀ 0 ≤ t ≤ d(*Op*$_{np}$) suma po svim raspoređenim operacijama *Op*$_r$ iz skupa *O* (Raspoređeno) i *Op*$_{np}$ vektora zauzeća resursa zadovoljava:

$$R_O(Op_{np}, t) + \sum R_O(Op_r, (OK\_Ciklus - \sigma(Op_r) + t)) \leq R_m$$

/* OK_Ciklus - σ (*Op*$_r$) svodi vreme za sve *Op*$_r$ i *Op*$_{np}$ na isti početni trenutak u vremenu */
σ (*Op*$_{np}$) := OK_Ciklus;
Raspoređeno := Raspoređeno ∪ {*Op*$_{np}$};
Data_Ready := (Data_Ready – {*Op*$_{np}$}) ∪ {*Op*$_{succ}$ ∈ Succ({*Op*$_{np}$ } | Pred(*Op*$_{succ}$) ⊆ Raspoređeno};
ENDDO;
RETURN Raspoređeno (σ);
END Raspoređivanje_po_listi;

Svaka operacija, kao i ceo raspored se može prikazati formalno u trodimenzionom prostoru čije su dimenzije:

1. Tip resursa
2. Broj resursa istog tipa
3. Vreme u ciklusima

Za svaku od dimenzija postoje ograničenja. Pre svega, mašine ne moraju da imaju neki tip resursa (npr. hardverski delitelj), jer to zavisi od tipa mašine. Broj resursa istog tipa, za svaki tip resursa ukazuje na paralelizam ugrađen u mašinu. Instrukcija traje ograničen broj ciklusa, a broj ciklusa je zavisan od izbora onih koji su projektovali paralelnu mašinu kojoj se u vreme prevođenja paralelizuje kôd. U poglavlju o protočnosti, analizirano je šta sve može da utiče na broj ciklusa neke operacije u mašini. Za operacije je karakteristično da zauzimaju samo neke tipove resursa i po pravilu najviše jedan resurs određenog tipa u ciklusima u kojima traje operacija. Zato je broj resursa određenog tipa dimenzija kod koje se prilikom određivanja OK_Ciklus-a po pravilu ispituje samo da li još ima bar jedan raspoloživi resurs, a ne i koliko. U kontekstu celog već ostvarenog parcijalnog rasporeda na mašini, neophodno je da postoji dovoljan broj slobodnih resursa, za zahtevane tipove resursa, u svim zahtevanim ciklusima operacije, da bi se neka operacija mogla rasporediti. Naravno, traži se najraniji mogući raspored za koji su ispunjeni prethodni uslovi. Taj najraniji raspored je limitiran zavisnostima po podacima preko OK_Ciklus-a.

Primenom navedenog algoritma, jedna po jedna operacija popunjava navedeni trodimenzionalni prostor u vreme prevođenja i uvodi dodatna ograničenja za raspoređivanje novih operacija u svim ciklusima trajanja te operacije. Osnovno ograničenje u ovom prostoru predstavlja broj resursa određenog tipa, koji proizilazi iz konfiguracionog vektora resursa. Postupak u algoritmu se u kontekstu trodimenzionalnog prostora može definisati na sledeći način:

a. Odredi se operacija iz Data Ready skupa koja se prva raspoređuje po nekom kriterijumu.

b. Prvo se locira najraniji ciklus početka operacije u vremenu, na osnovu zavisnosti po podacima koje potiču od raspoređenih operacija, nazvan Min_Ciklus.
c. Nakon toga se, za izabrani ciklus početka operacije, proverava u svim ciklusima operacije da li ima raspoloživih resursa svih tipova resursa za izvršavanje operacije. To podrazumeva uklapanje trodimenzione predstave operacije u trodimenzionu predstavu broja zauzetih resursa po tipovima i to u svim ciklusima trajanja operacije.
d. Ako nedostaje makar jedan resurs bilo kog tipa u bilo kom ciklusu, mora se zakasniti potencijalni trenutak izvršavanja operacije za jedan ciklus i ponovo sprovesti test raspoloživosti resursa pod c. Ovo praktično znači da se dogodilo da je bar za jedan tip resursa u bar jednom ciklusu postojala potreba da se sa već raspoređenim operacijama i novododatom operacijom zauzme veći broj resursa nego što ih ima u konfiguracionom vektoru resursa mašine.
e. Kada je sprovođenjem koraka c. pronađen najraniji ciklus u kome se može rasporediti operacija, moraju se ažurirati: skup raspoređenih operacija, skup zauzetih resursa mašine za već raspoređene operacije i ukloniti iz skupa neraspoređenih operacija upravo raspoređena operacija. Takođe se ažurira skup operacija spremnih za raspoređivanje, jer se operacije koje su bile zavisne samo još od upravo raspoređene operacije mogu pridružiti skupu spremnih po podacima.
f. Ponavljaju se koraci a. – e., dokle god ima neraspoređenih operacija.

Kriterijumi za izbor prioritetne operacije Op_{np} su uglavnom vezani za davanje prednosti operacijama koje su na kritičnom putu u grafu zavisnosti po podacima preostalih neraspoređenih operacija. Na taj način se praktično uvek prvo raspoređuju operacije koje, ako se ne rasporede prve, potencijalno dovode do produžavanja rasporeda. Ovim se ujedno obezbeđuje uspešno dugoročno planiranje resursa kod formiranja rasporeda.

Pored ovog kriterijuma, operacije koje dovode do najvećeg povećanja skupa Data_Ready operacija su kandidati da budu prve raspoređene. Razlog za takav kriterijum je da se što bolje kratkoročno iskoriste resursi mašine. Postoje i kombinacije kratkoročnih i dugoročnih heuristika, ali u praktičnim primenama dominiraju heuristike zasnovane na kriterijumu kritičnog puta ili njegovim modifikacijama.

Heuristički algoritmi pokazali su se veoma uspešnim za praktičnu primenu i danas dominiraju u optimizacionim algoritmima na instrukcijskom nivou u prevodiocima koji dozvoljavaju paralelizaciju kôda. Drugi ravnopravan termin koji se koristi za agresivnu paralelizaciju je prekoredno izvršavanje kôda (out of order execution). U praktičnim primenama, model mašine je nešto komplikovaniji od generalnog modela koji je ovde opisan.

Osnovni nedostatak prikazanog modela je u tome što se resursi različitog tipa posmatraju izolovano, kada se posmatra zauzeće resursa od strane jedne operacije. U realnim situacijama, za obavljanje operacija se javljaju alternative u skupovima konkretnih resursa različitog tipa koji mogu da obave operaciju. Dakle, ponekad nije moguće modelirati realnu situaciju samo preko zauzimanja bilo kog resursa određenog tipa u određenom ciklusu od strane operacije. Tada se mora pribeći zauzimanju raznih alternativnih skupova instanci resursa različitog tipa u različitim ciklusima operacije. Pritom broj zauzeća instanci resursa određenog tipa mogu da budu različiti za različite

alternativne skupove instanci resursa. Tipičan primer je da iz neke registarske memorije podatak možemo da dovedemo do nekih aritmetičko logičkih jedinica preko lokalne magistrale, a do nekih drugih, po tipu identičnih aritmetičkih jedinica, samo korišćenjem i nekih globalnih magistrala, čime zauzimamo i dodatnu instancu resursa magistrala podataka. Drugi primer su množači/sabirači (množači/akumulatori), koji u jednom ciklusu mogu da obave samo množenje, samo sabiranje, ili množenje sa akumulacijom {JO 88}.

Ovakvi primeri alternativnog zauzeća resursa su bili ugrađeni u assembler paralelne VLIW mašine FRT-300 razvijene u Institutu Vinča i opisani su u {MU 84}{JO 89}{JM 86}. Broj alternativa vrlo lako može da poraste tako da se neprihvatljivo produži vreme samog prevođenja. Zbog kompleksnosti koju u optimizaciju uvode skupovi alternativnih resursa za izvršavanje operacija, u praktično realizovanim mašinama se, kroz organizaciju mašine, teži da se napravi mašina koja odgovara modelu korišćenom za funkciju Raspoređivanje_po_listi. Osnovna hardverska komponenta pomoću koje se to postiže je višeportna registarska memorija, sa velikim brojem ravnopravnih ulaza i izlaza.

Dodatni nedostatak navedenog modela je da se operacije posmatraju izolovano, tako da se ne mogu uočiti slučajevi u kojima se manje zauzimaju resursi kada se dve operacije izvršavaju pomerene za fiksan broj ciklusa. Primeri takvih operacija su sledeći:

1. Operacije koje koriste iste argumente, tako da sa iste magistrale mogu da pokupe argument ako se obavljaju zajedno.
2. Operacije kod kojih je rezultat jedne operacije argument druge. Tada se operacije ulančavaju, tako što se izlaz jedne jedinice direktno prosleđuje na ulaz druge jedinice, pa se može izbeći nepotrebno pamćenje međurezultata ili zauzeće magistrala u mašini. Optimizacija kôda za vektorsku jedinicu (procesor) računara CRAY se dominantno oslanjala na ulančavanje operacija i time minimizaciju zauzeća resursa (magistrala podataka i memorija) za potrebe čuvanja međurezultata. Na taj način je i ujedno povećavan broj protočnih stepeni u protočnom nizu, a preko toga i paralelizam.

1.5. Limiti u paralelizmu kada se optimizacija sprovodi samo na nivou bazičnih blokova.

Dosadašnja razmatranja paralelizacije operacija su se odnosila na bazične blokove – sekvence. Globalna razmatranja optimizacije moraju da obuhvate i paralelizaciju u prisustvu grananja, pre svega uslovnih. Kod uslovnih grananja postoje dve osnovne kategorije – grananja kojim se izlazi iz programskih petlji i grananja koja nisu vezana za povratni uslovni skok kod petlje. Prva kategorija će se uglavnom kasnije razmatrati u okviru optimizacije kôda petlji, mada i za njih naravno važe generalno uvedena pravila za uslovna grananja. Druga kategorija će pre svega poslužiti za razmatranje svih osnovnih metoda globalne optimizacije kôda.

Za paralelizam koji se može dobiti na nivou bazičnih blokova su važni statistički parametri vezani za grafove zavisnosti po podacima za bazične blokove, kod prosečnog programskog kôda. Osnovno ograničenje za paralelizam je prosečno mala veličina bazičnog bloka. Ona je direktna posledica činjenice da je, statistički posmatrano, približno svaka sedma instrukcija, u neparalelizovanom kôdu, instrukcija uslovnog

skoka. Kako u ovom slučaju postoji 1:1 korespondencija između instrukcija i operacija, grafovi zavisnosti po podacima bazičnog bloka će u proseku imati samo 6 operacija. Definišimo paralelizam grafa bazičnog bloka kao odnos vremena sekvencijalnog izvršavanja instrukcija (operacija) u ciklusima i najkraćeg (optimalnog) vremena izvršavanja rasporeda operacija na idealnoj mašini u ciklusima. Prosečan paralelizam grafova bazičnog bloka iznosi približno 2. Dakle, ukupan paralelizam je relativno mali i takva paralelizacija kôda na instrukcijskom nivou je korišćena u programskim prevodiocima još pre 1986 godine.

Kako bi se prevazišla ograničenja, neophodno je naći postupak za seljenje operacija između bazičnih blokova. Intuitivno se nameće da, ako bi bilo moguće seliti operacije između bazičnih blokova, onda bi se na osnovu nekih globalnih kriterijuma mogla definisati pravila za te selidbe. Tako bi se na kraju mogao dobiti znatno paralelniji kôd nego kada se optimizacija radi samo na nivou bazičnih blokova.