12 - Optimizacija softvera

Saša Malkov

Odlomak iz knjige Razvoj softvera

(u pripremi)

12.1 Performanse softvera

Softverski projekti se odlikuju velikim brojem funkcionalnih i nefunkcionalnih zahteva, koji opisuju ciljne karakteristike proizvoda. U nefunkcionalne zahteve se svrstavaju, između ostalog, pretpostavke o efikasnosti rada softvera.

Efikasnost rada softvera obično se izražava kroz *performanse softvera*. Performanse softvera predstavljaju ocenu odnosa obima obavljenog posla i zauzeća ili utroška različitih resursa. Obično o performansama govorimo u množini, zato što posmatramo različite mere obima obavljenog posla, kao i različite vrste resursa. Razumevanje performansi softvera nam omogućava da procenimo za koje specifične poslove možemo da koristimo softver, do kog obima podataka je on upotrebljiv, kakav računar nam je potreban da bi softver radio dobro u predviđenom domenu i sa predviđenim obimom podataka i slično.

Obim obavljenog posla se obično meri brojem obrađenih slučajeva ili obimom obrađenih podataka, ali može da se meri i dimenzijama pojedinačnih problema i na druge načine. Na primer, možemo da merimo količinu obrađenog teksta u

pojedinačnim znacima ili u bajtovima, ili broj obrađenih transakcija, ili dužinu obrađenog video ili zvučnog zapisa u sekundama i drugo.

Resursi čije zauzeće nas zanima mogu da budu svi resursi koji su prisutni u savremenim računarskim sistemima:

- opterećenje procesora:
  - utrošeno procesorsko vreme;
  - broj paralelnih tokova;
  - broj izvršenih instrukcija;
  - broj izvršenih naredbi grananja;
  - broj pogrešno predviđenih uslovnih grananja;
- opterećenje radne memorije:
  - obim podataka koji se čuvaju u radnoj memoriji;
  - broj čitanja ili pisanja iz radne memorije;
  - obim podataka koji se pišu ili čitaju iz radne memorije;
- opterećenje keš memorije...;
- opterećenje virtualne memorije...;
- opterećenje diska...;
- opterećenje magistrale:
  - broj operacija čitanja ili pisanja;
  - obim podataka koji se čitaju ili pišu;
- opterećenje mreže...;
- utrošak električne energije;
- zagrevanje računarskog sistema;
- i drugo.

Odnos obima obavljenog posla i zauzeća nekog resursa često izražavamo i kao složenost programa[64] (ili algoritma), a u odnosu na posmatrani resurs. Pri analiziranju algoritama najčešće se posmatra utrošak procesorskog vremena, ali se vrlo često posmatra i opterećenje radne memorije. Međutim, kada imamo pred sobom neki konkretan softver, koji radi na nekom konkretnom računarskom sistemu, onda nam često nije dovoljno da se ograničimo na posmatranje složenosti

[64] Pretpostavljamo da su se čitaoci već susretali sa pojmom složenosti algoritma kao uobičajenim vidom približnog opisivanja zavisnosti utroška procesorskog vremena od veličine problema. Na primer, kažemo da algoritam ima linearnu složenost ako je za rešavanje 10 puta većeg problema potrebno približno 10 puta više procesorskog vremena, a kvadratnu složenost ako je potrebno približno 10^2, tj. 100 puta više procesorskog vremena.

pojedinačnih algoritama, već nas zanima kako softver radi *kao celina* i kakvo je ukupno zauzeće resursa.

Različiti načini merenja obima obavljenog posla i raznovrsnost resursa koji nam mogu biti od interesa, primoravaju nas da pratimo veliki broj različitih parametara koji opisuju performanse softvera. Radi olakšavanja staranja o performansama, obično se fokusiramo na resurs koji trpi najveće opterećenje. Takav resurs nazivamo *usko grlo*. Često možemo da izdvojimo tačno jedno usko grlo, ali se dešava i da ih bude više.

Najčešći slučaj je da je glavno usko grlo procesorsko vreme. Zbog toga što povećavanje brzine rada programa često može da se ostvari tako što se algoritam modifikuje da pamti prethodne rezultate ili međurezultate, usko grlo relativno često može da predstavlja i radna memorija. U slučaju manjih samostalnih programa, čiji je zadatak da nešto izračunaju ili urade, ostali računarski resursi su najčešće raspoloživi u dovoljnoj meri.

Kod složenih sistema, kao što su distribuirani sistemi, ili sistemi koji su zaduženi za složenu i obimnu transakcionu obradu, ili sistemi sa jedinicama za masivnu paralelnu obradu (GPU), povećava se opterećenje drugih resursa i dešava se da usko grlo predstavlja procesorska ili sistemska magistrala, mreža, diskovi ili nešto drugo.

12.2 Pojam optimizacije softvera

Performanse softvera su veoma važna karakteristika softvera. Iako se uobičajeno ne svrstavaju u funkcionalne karakteristike, one mogu da budu od presudnog značaja za procenjivanje dovršenosti ili prihvatljivosti softvera. Ako softver pravimo da radi u nekom domenu, a u tom domenu postoje slučajevi za koje su ostvarene performanse neprihvatljivo niske, onda takav softver verovatno nije prihvatljiv i klijent će smatrati da nije dovršen.

Niske ili nedovoljne performanse imaju mnogo zajedničkih karakteristika sa bagovima – mogu da se uoče rano ili kasno, kada se uoče onda moraju da se popravljaju, možemo da se tokom razvoja staramo da do njih ne dođe, postoje uslovi koji pogoduju njihovom nastajanju ili predstavljaju dobru prevenciju i slično. U skladu sa tim i rad na unapređenju performansi često možemo da posmatramo kao vid debagovanja – ako debagovanje posmatramo kao skup aktivnosti koje preduzimamo da bismo popravili neispravno ponašanja programa, onda ono svakako obuhvata i aktivnosti usmerene na podizanje performansi softvera, zato što je i to vid popravljanja ponašanja.

Međutim, staranje o performansama ima i neke specifične elemente zbog kojih ga razlikujemo od debagovanja i posmatramo kao posebnu aktivnost u okviru razvoja softvera. Ključna razlika u odnosu na debagovanje je da pri popravljanju performansi mi zapravo ne želimo da promenimo ponašanje u smislu ispravnosti

rezultata, već samo da spustimo cenu izračunavanja. U tom kontekstu možemo da uočimo da podizanje performansi ima sličnosti sa refaktorisanjem – u oba slučaja menjamo implementaciju, a ne želimo da promenimo rezultat izračunavanja. Razlika je u tome što je cilj refaktorisanja pravljenje *lepšeg* programa, a cilj podizanja performansi je pravljenje *efikasnijeg* programa. Kao što ćemo videti u nastavku ovog poglavlja, insistiranje na visokoj efikasnosti se često ne slaže najbolje sa dobrim dizajnom i lepotom programskog koda.

Staranje o performansama se uobičajeno naziva *optimizacijom softvera*. Obuhvata strategije optimizacije softvera, razmatranje ciljeva, planiranje vremena, mesta i količine optimizovanja, kao i određivanje mesta i uloge optimizovanja u okviru celovitog procesa razoja softvera. Kažemo da je to šire značenje pojma optimizacije softvera i da obuhvata sve aktivnosti koje preduzimamo da bi softver imao odgovarajuće performanse.

Sa druge strane, kada govorimo o konkretnim tehnikama optimizacije, onda se tu obično ne radi o optimizaciji u širem smislu, već o jednom mnogo užem i konkretnijem značenju ovog pojma. Optimizacija softvera, u užem smislu, je postupak menjanja strukture i implementacije programa radi postizanja boljih performansi. Bolje performanse najčešće postižemo rasterećivanjem resursa koji predstavlja usko grlo. Pri tome, rasterećivanje jednog resursa često (ali ne uvek) ima za posledicu povećavanje opterećenja nekog drugog resursa. Zbog toga se često ističe da optimizacija softvera predstavlja preraspoređivanje opterećenja različitih računarskih resursa, u skladu sa potrebama i mogućnostima.

Optimizacija softvera se često pogrešno povezuje sa pojmom optimalnog rešenja. Kažemo da je neko rešenje *optimalno* ako predstavlja *najbolje moguće* rešenje[65]. U nekom idealnom slučaju, možda bismo mogli da kažemo da je cilj optimizacije softvera dostizanje tog optimalnog rešenja, ali to najčešće ne stoji. Umesto toga, optimizacija softvera skoro uvek ima za cilj dostizanje *dovoljno efikasnog rešenja*, tj. rešenja koje je dovoljno blizu nekog idealizovanog optimuma. Osnovni razlozi za to su cena razvoja i ograničenost raspoloživog vremena za završetak projekta. Da bismo znali da je neko rešenje optimalno, to moramo da dokažemo, što zahteva i odgovarajući složen matematički aparat i dovoljno dobru teorijsku podršku za rešavanje odgovarajućeg problema. Algoritmika i arhitektura računara su složene oblasti i veoma je teško dokazati da ne postoji baš nikakav način da se performanse nekog iole složenog posla još malo poprave. Sa druge strane, pronalaženje načina da se neki posao uradi efikasnije obično zahteva mnogo rada (analiziranje, eksperimentisanje, implementiranje, testiranje,...), pa razvojni tim nema uvek

[65] U zavisnosti od toga šta se i kako meri, nekada može da postoji više optimalnih rešenja.

dovoljno prostora da mu se posveti. U realnom razvoju softvera često smo u prilici da znamo da bi nešto moglo da se unapredi, ali da je u konkretnoj situaciji važnije da se projekat privede kraju i da radi ispravno, nego da se performanse dodatno podignu. U takvim slučajevima se pretpostavljene mogućnosti za unapređivanje evidentiraju u obliku budućih ili potencijalnih zadataka i ostavljaju za neko drugo vreme.

12.3 Nivoi optimizacije

Optimizacija može da se izvodi na više različitih nivoa, posmatrano u odnosu na nivo apstrakcije dela softvera koji se optimizuje. *Optimizacija visokog nivoa* obuhvata sve oblike optimizacije koji su *iznad* nivoa programskog koda, a *optimizacija niskog nivoa* se odnosi na one vidove optimizacije koji se tiču konkretnog programskog koda ili postupka izgradnje softvera od napisanog koda.

12.3.1 Optimizacija visokog nivoa

Optimizacija visokog nivoa se obično izvodi na nivou projekta ili na nivou algoritma. Optimizacija na nivou projekta se prvenstveno tiče arhitekture softvera. Cilj je da se arhitektura oblikuje uz puno razumevanje poznatih ili potencijalnih uskih grla. Optimizacija na nivou projekta proširuje skup alata za oblikovanja arhitekture dodatnom tehnikom – *dekomponovanjem prema resursima*. Ideja je da se najpre prepoznaju funkcionalnosti i njihov odnos prema uskim grlima i da se onda funkcionalnosti koje dele isto usko grlo razdvoje u različite komponente. U zavisnosti od procene, takve komponente mogu odmah da se projektuju tako da rade na različitim uređajima ili može da se ostavi prostor za njihovo eventualno kasnije distribuiranje. Na taj način se primenom modularizacije ostvaruje dodatna fleksibilnost u odnosu na uočen potencijalan problem preopterećenja konkretnog uskog grla.

Obično nije lako da se unapred proceni efikasnost neke arhitekture. U trenutku njenog oblikovanja još uvek ne postoji odgovarajući programski kod, ili postoje tek neki pojednostavljeni obrisi tog koda, pa nismo u prilici da izmerimo performanse. Ipak, pre nego što donesemo odluku da je neka arhitektura odgovarajuća, trebalo bi da procenimo i da li je dovoljno efikasna. Tu može da nam bude od pomoći da napravimo različite eksperimente ili prototipove. Kao i sa drugim vrstama prototipova, njihovom pravljenju mora da se pristupa pažljivo i bez preterivanja.

Optimizacija na nivou algoritma se odnosi na unapređivanje ili čak zamenjivanje konkretnog algoritma radi smanjivanja opterećenja uskog grla. Za rešavanje nekih problema imamo na raspolaganju više algoritama, koji imaju različite osobine. Najčešće će nam najviše odgovarati onaj algoritam koji ima najmanju složenost u odnosu na procesorsko vreme, ali to nije uvek tako. Na primer, algoritmi koji imaju

manju složenost, nekada mogu da imaju mnogo skuplji pojedinačan korak, tako da njihova efikasnost dolazi do izražaja tek na veoma velikim skupovima podataka. Slično tome, neki algoritmi imaju različitu složenost u odnosu na broj problema i veličinu pojedinačnih problema, pa u zavisnosti od toga da li je potrebno da naš softver radi sa velikim brojem jednostavnih problema ili sa malim brojem složenih problema, neće uvek najbolji izbor da predstavlja isti algoritam.

Optimizacija visokog nivoa može da bude veoma skupa. Bilo da se radi na nivou arhitekture ili na nivou algoritma, optimizacija visokog nivoa može da dovede u pitanje upotrebljivost postojeće implementacije. Često moramo da ponovo implementiramo komponente ili algoritme koji smo već jednom implementirali.

Optimizacije visokog nivoa su skupe čak i kada se rade unapred, pre nego što smo bilo šta implementirali. Problem je u tome što se njima obično podiže nivo apstrakcije posmatranog dela softvera, uvode se složeniji odnosi među komponentama ili složeniji algoritmi, pa se potencijalno otežava implementacija. A pri tome je sasvim moguće da to sve nije ni potrebno, zato što bi možda i jednostavnije rešenje radilo sasvim dobro. Zato se pri razmatranju optimizacije visokog nivoa, koja se preduzima unapred, obično teži da se najpre pažljivo sagledaju svi rizici i njihova cena, pa da se na osnovu toga izabere rešenje koje nam je povoljnije.

12.3.2 Optimizacija niskog nivoa

Optimizacije niskog nivoa se odnose na već postojeći programski kod i tehnike izgradnje programa. Pojedinačne tehnike optimizovanja već postojećeg programskog koda se najčešće odvijaju na niskom nivou. Zbog toga se pod užim značenjem pojma optimizacije softvera uobičajeno podrazumevanju samo optimizacije niskog nivoa ili čak samo pojedinačne tehnike optimizovanja.

Optimizacije niskog nivoa mogu da se preduzimaju na nivou:

- izvornog koda;
- prevođenja;
- izgradnje koda;
- mašinskog koda ili
- izvršavanja

Optimizacija na nivou izvornog koda predstavlja menjanje implementiranog programskog koda radi ostvarivanja veće efikasnosti. Može da počiva na promenama strukture koda, takvim da se dobija manje opšte ili slabije strukturirano rešenje, ali da pri tome rezultat bude efikasniji. Drugi vid optimizacija na nivou izvornog koda je prilagođavanje implementacije algortima specifičnostima konkretnog programskog jezika. Nekada ima prostora da se pređe sa upotrebe nekih

opštih programskih koncepata na upotrebu nekih koncepata specifičnih za konkretan jezik i da se tako dobije na performansama.

Optimizacija na nivou prevođenja se odnosi na upravljanje postupkom prevođenja, tako da se neke specifičnosti prevodilaca iskoriste za dobijanje efikasnijeg izvršnog koda. Obuhvata izbor verzije prevodioca i podešavanje opcija prevođenja. Praktično svi prevodioci imaju neke opcije kojima se uključuju različiti oblici internog optimizovanja prevedenog programa, ali neke od tih opcija ne donose uvek pozitivne efekte. Zato je ponekad potrebno da isprobamo različite kombinacije finih podešavanja prevodilaca da bismo dodatno podigli performanse. Nezgodna strana ovog nivoa implementacije je što mora posebno da se radi za različite platforme i prevodioce.

Optimizacija na nivou izgradnje programa se ostvaruje izborom odgovarajućih varijanti biblioteka i načina povezivanja. Na primer, neki delovi programa mogu biti toliko osetljivi u odnosu na performanse, da izbor statičkog ili dinamičkog povezivanja[66] može da proizvede značajne razlike. Ponekad izbor verzije neke biblioteke može da ima značajne posledice po efikasnost. Nije retkost da neka jednostavnija biblioteka pruža veće performanse, ali po cenu manjih mogućnosti. Ako su nam te umanjene mogućnosti dovoljne, onda takva biblioteka može da bude bolji izbor.

Optimizacija na nivou mašinskog koda se izvodi pisanjem delova programa na mašinskom jeziku, tj. na asembleru. To je jedini nivo pisanja programa na kome do kraja mogu da se iskoriste sve mogućnosti računarskog sistema. Osnovni problem je u tome što se pisanjem mašinskog koda naša implementacija striktno vezuje za jednu klasu procesora, a često čak i za konkretan operativni sistem. To nas primorava da pišemo i održavamo više verzija optimizovanog koda za različite procesore i operativne sisteme.

Prostor za primenu ovog nivoa optimizacije se sve više sužava, kako zbog problema koje on donosi, tako i zbog činjenice da savremeni prevodioci imaju ugrađene vrlo napredne algoritme za optimizaciju mašinskog koda. Današnji prevodioci često mogu da naš program prevedu na mašinski jezik na praktično idealan način, pa čak i da promene redosled mašinskih naredbi tako da one rade efikasnije a sa istim rezultatom. Sve je teže manuelno napisati mašinski kod koji je efikasniji od onog koji bi napravio prevodilac. Zato se danas na mašinskom jeziku

[66] Da ne bude nesporazuma, ovde je reč o povezivanju prevedenih modula, a ne o vezivanju metoda. Statičko povezivanje je povezivanje objektnog koda sa statičkim bibliotekama u fazi povezivanja programa, a dinamičko povezivanje je povezivanje izvršnog programa sa dinamičkim bibliotekama u fazi učitavanja programa u radnu memoriju, neposredno pre izvršavanja.

sasvim retko pišu celi programi pa i funkcije ili metodi, već je uobičajeno da se na taj način manuelno optimizuju samo pojedini osetljivi delovi koda, u slučajevima kada želimo da iskoristimo neku specifičnost konkretnog procesora ili operativnog sistema.

Optimizacija na nivou izvršavanja i arhitekture obuvata prilagođavanje programa specifičnostima arhitekture procesora i računara i načinu izvršavanja programa ili određenih operacija. Uobičajeno je da se na ovom nivou optimizacije vrši preraspoređivanje mašinskih instrukcija radi njihovog izvršavanja preko reda, upotreba operacija odloženog grananja ili paralelnog izvršavanja nekih instrukcija i slično. Takve optimizacije su na nižem nivou nego optimizacije na nivou mašinskog koda i u velikoj meri se preklapaju sa njima.

U optimizacije na nivou izvršavanja se ubrajaju i tehnike poput automatskog prevođenja ili optimizovanja neposredno pred izvršavanje, kao što je na primer prevođenje *na vreme* (ili *u pravo vreme*, engl. *JIT – just in time*) u slučaju JVM ili CLR.

12.4 Strategije

Najvažnija pitanja u vezi sa optimizacijom su *kada*, *šta* i *koliko* je potrebno da optimizujemo. Odgovori na ova pitanja određuju, redom, trenutak, predmet i dubinu optimizacije.

Odgovor na pitanje *kada* i odabir tačnog trenutka izvođenja optimizacije nazivaju se i *vremenskom lokalizacijom* optimizacije. Prema tome kada se određena optimizacija preduzima razlikujemo *optimizaciju unapred* i *optimizaciju unazad*.

12.4.1 Optimizacija unapred

Optimizacija unapred podrazumeva da se staranje o performansama odvija sve vreme tokom razvoja – od samog početka planiranja, kroz projektovanje arhitekture, oblikovanje ili odabir algoritma i kroz implementaciju i testiranje softvera koji se razvija. Cilj je da svi učesnici u razvoju tokom svih razvojnih aktivnosti rade na tome da ponude što efikasnija rešenja za probleme koje rešavaju. Optimizacija unapred se naziva i *kontiunalnim staranjem o performansama*.

Optimizacija unapred omogućava da se dođe do veoma kvalitetnih rešenja, u pogledu performansi. Posvećenost razvojnog tima omogućava da se blagovremeno uoče i primene efikasna rešenja, što omogućava da u svakom trenutku razvoja može da se sagleda da li su performanse u skladu sa planovima.

Međutim, optimizacija unapred nosi i nekoliko veoma važnih negativnih posledica. Najpre, zahteva mnogo više vremena tokom razvoja, što podiže inicijalnu cenu razvoja. Naravno, bar deo te cene će se isplatiti zato što neće biti potrebno da se preduzima naknadna optimizacija, ali ne možemo da se ne zapitamo da li je ta cena opravdana – pokazuje se da je u realnom razvoju veoma lako utrošiti na optimizaciju

mnogo više vremena nego što je potrebno, a radi ostvarivanja zanemarljivo većeg ili nepotrebno visokog nivoa performansi.

Drugi problem je što je optimizovan softver najčešće značajno teži za održavanje. Ili su uvedene složenije apstrakcije na nivou arhitekture, ili se koriste složeniji algoritmi, ili je programski kod pisan sa akcentom na efikasnost, pa zato nije dovoljno lep i teži je za razumevanje. Razvijaoci će trošiti više vremena na razumevanje, menjanje i testiranje takvog programskog koda, pa se dodatno podiže cena razvoja.

Pri optimizovanju unapred se oslanjamo na procene o tome koji deo softvera je potrebno da se optimizuje i do kog nivoa efikasnosti. Pravljenje takvih procena obično nije nimalo jednostavno, a eventualno pogrešna procena može da ima za rezultat nepotrebno visoku cenu razvoja.

Ako se optimizuje šire i dalje nego što je potrebno, onda može da se proizvede softver koji je efikasniji nego što je potrebno, što samo po sebi nije loše, ali se zbog toga u optimizaciju ulaže mogo više rada nego što je neophodno, a verovatno je umesto toga taj rad mogao da se uloži u razvoj neke dodatne funkcionalnosti, koja bi za korisnike imala veću vrednost nego visoka efikasnost. Dodatni problem je što nepotrebne optimizacije prouzrokuju i očigledno nepotrebne dodatne troškove pri kasnijem menjanju i održavanju softvera.

Sa druge strane, ako se optimizuje uže ili nedovoljno daleko, onda se za rezultat često dobija nedovoljno efikasan softver, pa je na kraju neophodno da se primeni i optimizacija unazad. U takvim slučajevima se sa pravom postavlja pitanje da li je od čitave optimizacije unapred uopšte bilo nekakve koristi ili taj uloženi rad predstavlja beskorisno utrošen resurs?

Optimizacija unapred može da ima smisla kada se razvijaju kritični delovi programa, čije performanse presudno utiču na njegovu upotrebljivost. Može da se primenjuje i na određene elemente optimizacije visokog nivoa. Ona nije uobičajena za optimizovanje celih složenih softverskih projekata.

12.4.2 Optimizacija unazad

Strategija *optimizacija unazad* odlaže najveći deo staranja o performansama za sam kraj razvoja. Umesto da se sve vreme tokom razvoja mnogo vremena i energije posvećuje iznalaženju što boljih rešenja, umesto toga se tokom akcenat stavlja na dobro strukturiranje i ispravnost programa. Tek na kraju, kada su razvijene sve funkcije i kada je provereno da sve radi ispravno, onda se proverava da li softver radi dovoljno efikasno ili je možda potrebno da se dodatno podigne efikasnost nekog njegovog dela. Ovakav pristup razvoju se ponekad naziva i metodom *funkcionalnost pre performansi*.

Ova strategija počiva na dve osnovne pretpostavke:

- programski kod se tokom razvoja relativno često menja i
- najveći deo performansi zavisi od vrlo ograničenog dela softvera.

Ako znamo da se kod često menja i da je menjanje optimizovanog koda značajno otežano, onda ćemo da težimo da optimizujemo što je moguće manje delove programa i da to radimo što je moguće kasnije. Ako pri tome znamo da je za dobre performanse najčešće dovoljno da se optimizuje relativno mali deo programa, pa to još povežemo sa prvom pretpostavkom, onda ćemo tim pre biti motivisani da primenjujemo ovu strategiju.

Jedan od glavnih kvaliteta ovakvog pristupa je izbegavanje suvišnog, nepotrebnog ili preuranjenog optimizovanja. Ostvaruju se velike uštede tokom razvoja, kako zbog toga što se ne posvećuje vreme optimizaciji, tako i zbog toga što razvijen kod ima bolje strukturiran i često jednostavniji dizajn, pa se lakše i piše i debaguje i kasnije proširuje ili modifikuje.

Tek kada je razvoj softvera pri kraju, kada su prisutne sve funkcionalnosti i kada je pri kraju testiranje ispravnosti programa, onda mogu da se objektivno izmere performanse. Sve do tada se radi o procenjivanju, a tek tada je u pitanju egzaktno merenje performansi. Samim tim, tek tada možemo da donesemo ispravan sud o tome koji je deo softvera neophodno da se optimizuje i do koje granice. Pre objektivnog merenja i donošenja odgovarajućeg egzaktnog suda, postoji značajan rizik da se optimizacija preduzme na pogrešnom delu koda ili da ima neprimerenu širinu ili dubinu. Zato je optimizacija unazad najčešće poželjnija od optimizacije unapred.

Najveći problem sa ovakvim pristupom je što je neke stvari veoma teško popraviti na samom kraju, kada je već napisan najveći deo programa. Ako se ispostavi da je potrebno da se zameni neki algoritam ili čak arhitektura softvera, onda to može da zahteva implementiranje veoma obimnih izmena, pa čak i ponovno implementiranje nekih delova softvera. Cena takvih zahvata može da bude veoma visoka.

12.4.3 Odmerena optimizacija

Odmerenost (ili lokalizovanost) optimizacije predstavlja zadržavanje procesa optimizacije softvera u okvirima koji su određeni odgovorima na pitanja *šta* i *koliko* optimizujemo. Strategija odmerene optimizacije nalaže precizno određivanje predmeta i dubine optimizacije pre njenog preduzimanja.

Određivanje predmeta optimizacije

Kao što smo već istakli u prehodnim odeljcima, najveći deo performansi softvera zavisi od njegovog relativno malog dela. Od presudnog značaja za uspeh optimizovanja softvera je da se vrlo precizno odredi koji deo softvera je potrebno da se optimizuje.

Programeri troše enormne količine vremena na razmišljanje i brigu
o brzini nekritičnih delova programa,
a takvi pokušaji postizanja efikasnosti zapravo imaju negativan uticaj
kada se uzmu u obzir debagovanje i održavanje.

Potrebno je da zanemarimo sitne dobitke u efikasnosti
u, recimo, 97% slučajeva:
preuranjena optimizacija je osnov svakog zla.

Sa druge strane, ne smemo da propustimo priliku
za optimizaciju preostalih kritičnih 3%.

Donald Knut

Nije nam posebno važno koliko je precizna procena koju je izneo Knut – u nekim slučajevima će se optimizovati i više od 10% programskog koda, ali u nekim drugim i manje od 0,1%. Važno je da se razume da najveći deo programskog koda ne utiče kritično na performanse. Možda neki deo programa može da se napiše i tako da radi 10 puta brže, ali ako to znači 1ms umesto 10ms, a pri tome nema značajan uticaj na ukupnu efikasnost softvera, onda eventualna optimizacija tog dela koda ne donosi praktično nikakvu korist.

Određivanje predmeta optimizacije počinje na vrlo grubom nivou, od uočavanja poslova koji nisu dovoljno efikasno implementirani. Preciznije lokalizovanje ima sličnosti sa postupkom lokalizovanja bagova – pažljivim posmatranjem i analiziranjem programa se trudimo da što preciznije odredimo deo programskog koda koji je potrebno da se optimizuje. To je nekada relativno lako, ali često ne može da se uradi dovoljno dobro bez primene odgovarajućih alata. O tome će biti više reči u odeljku ***Error! Reference source not found.Error! Reference source not found.***.

Kada tačno prepoznamo deo softvera koji je potrebno da optimizujemo, onda u sledećem koraku moramo da odredimo tačnu meru potrebne optimizacije. Stepen optimizacije nekog izabranog dela softvera nazivamo i *dubinom optimizacije*. Kažemo da je optimizacija dublja ili dalja, ako se njom više približavamo pretpostavljenom idealnom rešenju.

Određivanje dubine optimizacije

Potrebna dubina optimizacije se određuje kroz definisanje kriterijuma performansi. Kriterijumi performansi mogu da budu izraženi na različite načine, a

obično imaju oblik *minimalnih* performansi i *planiranih* ili *ciljanih* performansi. Minimalne performanse predstavljaju zahtev koji mora da se ispuni. Ako se ne ispuni, onda se smatra da softver nije dovršen. Minimalne performanse moraju da se dostignu, čak i po cenu povećavanja trajanja i troškova razvoja. Za razliku od minimalnih, planirane performanse predstavljaju cilj kome se teži, ali koji uglavnom ne zaslužuje produžavanje rokova ili produžavanje troškova razvoja. Pri samom kraju razvoja se često vrši naknadna revizija ovih kriterijuma i njihovo usklađivanje sa rokovima i troškovima.

Proces određivanja kriterijuma performansi se razlikuje u zavisnosti od vrste softvera koji se razvija i planiranih načina njegove upotrebe. U tom smislu se najveća razlika pravi između načina ocenjivanja i određivanja kriterijuma performansi interaktivnog i neinteraktivnog softvera.

Kriterijumi performansi interaktivnog softvera

Kada se radi o softveru kojim korisnik interaktivno rukuje tokom obavljanja nekog posla, tada se pod dovoljnom efikasnošću podrazumeva da je softver dovoljno efikasan da bude upotrebljiv, tj. da ne izaziva neprijatnosti ili stres kod korisnika. Upotrebljivost korisničkog interfejsa zavisi od načina upotrebe softvera, odnosno od načina komunikacije korisnika i softvera. U kontekstu razmatranja efikasnosti softvera upotrebljivost se obično ocenjuje na osnovu tri kriterijuma [Card1991][67]:

- perceptivna obrada, do 0,1s;
- neposredan odgovor, do 1s;
- jednostavan zadatak, do 10s.

Ova tri slučaja mogu da se povežu sa interaktivnom komunikacijom među ljudima i sa normativima na koje smo naviknuti u toj komunikaciji.

Perceptivna obrada podrazumeva slučajeve kada je uobičajeno da se na akciju odgovara bez čekanja na razmišljanje, tj. instiktivno ili na osnovu osećaja. To bismo mogli da poredimo sa saradnjom sportista u nekom brzom timskom sportu. U kontekstu softvera, to su elementarne aktivnosti kao pritisak tastera na tastaturi ili mišu, pomeranje olovke po tabli ili izvođenje pokreta na ekranu osetljivom na dodir.

[67] Ove koncepte je prethodno mnogo detaljnije razradio Robert Miler [Miller1968]. On je opisao 19 različitih vrsta aktivnosti i diskutovao dopustive opsege čekanja na reakciju računara. Od tada su se značajno promenili i načini interakcije korisnika i računara, ali te procene su ipak uglavnom relevantne i danas. Upotrebljivosti softvera je temeljnije razmatrana u [Nielsen1993], gde su kao najvažniji kriterijumi upotrebljivosti interaktivnog softvera istaknuti isti ovi granični kriterijumi.

U takvim slučajevima, koji obično predstavljaju samo delove neke složenije interakcije, očekujemo da nam računar odgovara dovoljno brzo da nam čekanje na odgovor ne remeti tok aktivnosti. Obično se smatra da je gornja granica dopuštenog čekanja u takvim slučajevima 0,1s, ali ćemo često očekivati da reakcija bude i malo brža. Na primer, da bi putanja olovke ili miša bila verno opisana, često je potrebno da se detektuje više nego 10 promena u jednoj sekundi. Takođe, kod akcionih video igara vreme reakcije softvera mora da bude nešto kraće, zato što vrhunski igrači mogu da izdaju i do 600 komandi za minut.

Neposredan odgovor podrazumeva odgovaranje sagovornika na jednostavno pitanje. Na primer, ako pitamo sagovornika „Da li je veći mali slon ili veliki miš?“, on mora da razume pitanje, sasvim kratko razmisli o njemu i odgovori. U slučaju interaktivnog softvera, pod neposrednim odgovorom podrazumevamo brzo reagovanje na zahtev korisnika, koji očekuje odgovor da bi mogao da nastavi neku složeniju aktivnost. U kontekstu interaktivnog rada, smatra se da se na zahtev korisnika odgovara neposredno ako se u periodu od postavljanja pitanja do isporučivanja odgovora korisniku ne prikazuje nikakva dodatna povratna informacija (engl. *feedback*). Kao gornja granica dopuštenog čekanja na nesporedan odgovor obično se smatra 1s.

Treća vrsta interaktivnih poslova su jednostavni zadaci. Jednostavnim zadacima se obično smatraju svi oni zadaci kod kojih se na rezultat rada čeka duže nego na neposredne odgovore ali dovoljno kratko da komunikacija i dalje može da se nazove interaktivnom. I dalje pretpostavljamo da će korisnik na osnovu dobijenog odgovora želeti da postavi neki drugi zahtev (to je pretpostavka interaktivnosti), ali i da je on svestan složenosti zadatka i da može malo da sačeka. Ako je čekanje na odgovor duže od 1s (tj. iznad praga neposrednog odgovora), onda je neophodno da se korisniku pruži neka informacija o tome da je obrada u toku i približna procena koliko će da traje. U suprotnom, korisnik može da doživi neprijatan osećaj neizvesnosti – da li će odgovor uopšte da stigne, ili je nešto krenulo naopako?

Sve poslove na koje se čeka duže od 10s trebalo bi raditi u neinteraktivnom režimu. Na primer, njihovo izračunavanje može da se nastavi u pozadini, a da se korisniku omogući da zadaje nove zadatke.

Imajući u vidu navedena vremena reakcije koja određuju upotrebljivost interaktivnog softvera, možemo da okvirno prepoznamo i ciljeve optimizovanja pojedinih delova softvera. Za svaki interaktivan posao prvo moramo da prepoznamo da li mora da pruži utisak perceptivne obrade, ili je dovoljno da radi poput neposrednog odgovora ili može da spada u jednostavne zadatke. Zatim na osnovu prepoznatog nivoa interaktivnosti ustanovimo ciljno vreme odziva. Na taj način smo definisali i koliko taj posao moramo da optimizujemo.

Kriterijumi performansi neinteraktivnog softvera

U slučaju softvera koji ne radi interaktivno, cilj optimizacije se ne ustanovljava na osnovu nekih globalnih objektivnih merila, već na osnovu nefunkcionalnih zahteva ili nekih sistemskih ograničenja.

Na primer, u nekom sistemu prodavnica mogao bi da se postavi poslovni zahtev da rukovodstvo svakog dana dobija izveštaj o dnevnom prometu i to u roku od 15 minuta posle zatvaranja svih prodavnica. U takvom slučaju cilj optimizacije je sasvim precizno određen konkretnim potrebama korisnika i softver mora da se napravi tako da se te potrebe zadovolje.

Veoma često cilj optimizacije zavisi od nekih složenijih faktora. U slučaju složenih sistema, opterećenost neke od komponenti, koja radi veliki broj poslova ili opslužuje veliki broj korisnika, može u nekim periodima radnog vremena ili radne nedelje da predstavlja usko grlo čitavog sistema. Tada je neophodno da se radi na tome da se ta komponenta rastereti ili da se njeni poslovi optimizuju do mere koja garantuje ispravno funkcionisanje sistema. Na primer, neka je u onlajn prodavnici oglašen popust na neke proizvode, koji prestaje da važi u nekom trenutku. Može da se očekuje da će pred istek važenja popusta doći do povećanog opterećenja sistema za naručivanje i plaćanje. Da bismo mogli da rešimo takav problem potrebno je da napravimo procenu opterećenja i da probamo da softver optimizujemo tako da može da ga podnese.

Postoje i slučajevi kada nema nikakvih spolja nametnutih ciljeva. Na primer, ako pravimo softver koji mora da izvede neku složenu obradu podataka radi nekog naučnog istraživanja, onda nam limiti često nisu strogo definisani. Nekada ćemo biti u prilici da limite odredimo na osnovu toga da li je očekivano vreme prihvatljivo ili ne (na primer, ako procenimo da će obrada da traje više od 20 dana, to nekada može da bude prihvatljivo, ali nekada nije), ali nekada će trajanje biti prihvatljivo, a da ipak imamo utisak da bi bilo dobro da bude kraće (na primer, procenjeno trajanje obrade je 20 dana i to nam je u načelu prihvatljivo, ali bi bilo dobro da ga skratimo na 10 dana, da bismo mogli da stignemo da obradimo još jedan skup podataka). Tada je potrebno da sami odlučimo da li nam je optimizacija potrebna, pa ako jeste onda i da odredimo kriterijume i ciljeve.

12.4.4 Savremena praksa optimizovanja softvera

Savremenu praksu optimizovanja softvera možemo da predstavimo pomoću nekoliko osnovnih principa, koji se međusobno nadopunjuju:

- Procenjivanje performansi;
- Lenja optimizacija;

- Lokalizovanje optimizacije i
- Eksperimentisanje.

Ovde izloženi principi su donekle zasnovani na radu [Auer1996], u kome su Ken Aur i Kent Bek opisali agilni pristup optimizaciji softvera kroz predstavljanje skupa od 16 *obrazaca*. Njihovi obrasci optimizovanja su izloženi na primeru programskog jezika *Smalltalk*, ali uglavnom mogu da se odnose i na druge programske jezike. Neki od tih obrazaca predstavljaju specifične tehnike optimizacije, ali neki drugi, koji su nama interesantniji, su vrlo uopšteni i ostvarili su veliki uticaj na oblikovanje agilne strategije optimizovanja softvera.

Procenjivanje performansi

Procenjivanje performansi dela softvera se preduzima pre početka njegovog projektovanja i implementiranja. Osnovna namena procenjivanja performansi je da se blagovremeno uspostave merila, koja bi trebalo da nam pomognu da se odaberu ili oblikuju dovoljno dobra arhitektura i adekvatni algoritmi, koji omogućavaju dostizanje uspostavljenih kriterijuma performansi. Dobijene procene ne bi trebalo da se koriste kao osnov za forsiranje kontinualne optimizacije.

Procenjivanje performansi smanjuje rizik nastajanja ozbiljnijih problema pri optimizaciji unazad. Ima ulogu prevencije tzv. nedostižnih performansi, tj. slučajeva kada pri optimizaciji unazad nisu dovoljne samo optimizacije niskog nivoa, već je neophodno da se preduzimaju i skupe naknadne optimizacije visokog nivoa.

Uobičajeno je da rezultat procene ima oblik izveštaja o sagledanim minimalnim i planiranim performansama i uočenim potencijalnim problemima u realizaciji. U okviru pregleda potencijalnih problema bi trebalo da budu prepoznata i opisana moguća uska grla ili kritični resursi, i to kako u delu softvera koji će se razvijati, tako i u njegovom okruženju.

Procenjivanje performansi bi trebalo da uzme relativno malo vremena, ali u praksi potrebno vreme i obim izveštaja zavise od veličine i složenosti dela softvera čijem se razvoju pristupa, kao i od zahteva koji su postavljeni u odnosu na performanse. Ako se radi na nekom relativno jednostavnom slučaju upotrebe, koji opisuje, na primer, neku izdvojenu transakciju u informacionom sistemu, onda su obim i složenost problema relativno niski, a i sama priroda analize potencijalinh problema verovatno veoma liči na neke druge već razvijene transakcije. Zato procenjivanje performansi često zateva sasvim kratko vreme.

Međutim, u mnogim slučajevima stvari stoje sasvim drugačije. Ako nam je zadatak da postavimo osnovu za buduću arhitekturu dela sistema, ili da izaberemo algoritam koji je najprimereniji problemu, u smislu da je rešenje dovoljno efikasno i istovremeno što jednostavnije za implementaciju, onda ni analiza ni donošenje odluke nisu nimalo jednostavni. U takvim slučajevima moramo pažljivo da

izvagamo moguća rešenja i da donesemo odluke, koje mogu da imaju dalekosežne posledice.

Procenjivanje performansi se izvodi na početku razvoja, unapred, pa zbog toga obično nije na raspolaganju dovoljno informacija da bi mogla da se ostvari visoka preciznost procene. Radi dobijanja dodatnih informacija često se sprovode različiti eksperimenti (princip *Eksperimentisanje*). Problem ograničene preciznosti može donekle da se prevaziđe naknadnim revizijama. Revizije procene efikasnosti se preduzimaju u slučajevima kada dostignut stadijum razvoja ili neke izmenjene okolnosti pružaju osnov za tačnije procenjivanje performansi ili njihovo realnije sagledavanje. Na primer, trebalo bi da se napravi revizija procene performansi ako se bližimo prekoračenju vremenskih i finansijskih okvira. Ona bi mogla da prilagodi kriterijume minimalnih ili planiranih performansi na osnovu realno dostižnih okvira razvoja.

Lenja optimizacija

Videli smo da i optimizacija unapred i optimizacija unazad imaju svoje prednosti i slabosti. Zbog toga savremena praksa obično počiva na kombinovanju ovih strategija, uz uzimanje u obzir specifičnosti savremenog razvoja softvera i posebno agilnih metodologija. Pri tome izbor načina ili trenutka započinjanja procesa optimizacije ima sličnosti sa odnosom projektovanja arhitekture i dizajna softvera. Kao što skupe odluke o arhitekturi ima smisla doneti što ranije (i naravno što opreznije), tako i neke elemente optimizacije visokog nivoa ima smisla preduzimati unapred. Sa druge strane, optimizacija nižeg nivoa je najbolje da se ostavi za završni period razvoja softvera.

Lenja optimizacija podrazumeva da se svi vidovi optimizacija niskog nivoa preduzimaju tek pri kraju razvoja softvera, nakon dostizanja njegove planirane funkcionalnosti. Nasuprot tome, samo neki elementi optimizacije visokog nivoa će se preduzimati unapred, ali samo u minimalnom obimu, koji pomaže da se arhitektura i algoritmi oblikuju tako da potonjim razvojem i optimizacijom unazad mogu da se dostignu kriterijumi koji su ustanovljeni procenjivanjem performansi.

Lenja optimizacija je u potpunosti u duhu principa agilnog razvoja softvera „neće biti potrebno“. Zaista, ona odlaže optimizaciju sve do trenutka kada je ona zaista i potrebna. Na taj način se štedi vreme, zato što se ne optimizuju delovi programa koje nije neophodno da optimizujemo, a uz to nam pomaže da se što duže očuva dobar dizajn softvera, te da se on „kvari“ optimizacijom tek na samom kraju razvoja.

Uobičajeno je da se na samom kraju razvojnog ciklusa predvidi jedan period za posvećivanje optimizaciji. Poslovi optimizacije se često odvijaju paralelno sa poslovima na doterivanju korisničkog interfejsa. U zavisnosti od vrste problema, korišćenih programskih jezika i drugih alata, stepena stručnosti tima i drugih

faktora, taj period može da obuhvata od 5% do 35% predviđenog vremenskog okvira.

Već smo istakli da pri optimizaciji unazad postoji opasnost da dođemo u situaciju da moramo naknadno da preduzimamo ozbiljne optimizacije visokog nivoa (na primer paralelizaciju ili distribuiranje izračunavanja), što je veoma skupo. Da bismo to izbegli, ili bar da bismo smanjili rizik a time i broj i cenu takvih slučajeva, oslanjamo se na princip *Procenjivanje performansi*.

Lokalizovanje optimizacije

Princip *Lokalizovanje optimizacije* se odnosi na primenu strategije odmerene optimizacije. U osnovi ovog principa je što preciznije odgovaranje na pitanja *šta* i *koliko* se optimizuje. Primenjuje se u skladu sa merilima utvrđenim pri procenjivanju performansi. Suština ovog principa može da se rezimira kroz tri jednostavne preporuke:

- Budi odmeren pri optimizovanju.
- Prvo postavi ciljeve, pa tek onda optimizuj.
- Ne optimizuj dalje od postavljenih ciljeva.

Ken Aur i Kent Bek su izdvojili četiri obrasca [Auer1996], koji odgovaraju ovom principu:

- Kriterijumi performansi – Pre početka razvoja je neophodno da se uz pomoć klijenta odrede kriterijumi performansi. Oni mogu da se kasnije revidiraju.
- Prag prihvatljivosti – Prag prihvatljivosti se određuje na osnovu kriterijumima performansi. Ne sme da se ide dalje od praga, čak ni kada izgleda da je to lako dostižno.
- Merenje performansi – Neophodno je da precizno merimo performanse delova programa za koje su definisani kriterijumi performansi i pragovi prihvatljijvosti.
- Ključna mesta – Pre preduzimanja optimizovanja je potrebno da se jasno utvrdi koja su to ključna mesta u softveru na čijim performansama moramo da radimo. To će biti mesta koja imaju veze sa kriterijumima performansi i pragovima prihvatljivosti i na kojima merenje performansi ukazuje na određene probleme.

Ova četiri obrasca mogu da predstavljaju vid uputstva za sprovođenje lokalizovane optimizacije. Obrasci *Merenje performansi* i *Ključna mesta* se odnose

prvenstveno na prostorno lokalizovanje optimizacije, a obrasci *Kriterijumi performansi* i *Prag prihvatljivosti* nam pomažu da odredimo dubinu optimizacije.

Priprema za lokalizovanje

Lokalizovanje optimizacije počinje od pripreme. Dobra priprema je ključna za uspešno lokalizovanje optimizacije, pa i za njeno kasnije sprovođenje. Priprema se oslanja na prethodno ustanovljene kriterijume performansi i ciljevi optimizacije. Ona se sastoji od širokog i iscrpnog izučavanja problema sa kojim se suočavamo i okolnosti u kojima se optimizacija preduzima.

Pod izučavanjem problema obično podrazumevamo dobro poznavanje zadataka koje softver mora da reši, primenjenih algoritama i izvedene implementacije. Svaka optimizacija predstavlja neki oblik kompromisa, pa zato moramo dobro da znamo kakve kompromise eventualno smemo da pravimo i koliko daleko u tome možemo da idemo u prostoru konkretnih zadataka i algoritama.

Poznavanje implementacije nam omogućava da steknemo uvid u to šta eventualno možemo da menjamo u postojećoj implementaciji i koliko toga možemo da unapredimo bez menjanja ili zamenjivanja algoritma.

Poznavanje i razumevanje okolnosti se odnosi na alate koje koristimo, kontekst projekta i različite oblike prisutnih ograničenja. Alati obuhvataju sve ono što koristimo u razvoju i što je vezano za deo softvera koji pokušavamo da optimizujemo. Tu spadaju programski jezici, prevodioci, arhitektura procesora, mašinski jezik procesora, ali i različiti alati za merenje performansi. Optimizacija se često svodi na korišćenje nekih karakteristika alata, koje se relativno retko neposredno upotrebljavaju. Zbog toga nam iscrpno poznavanje alata otvara širi prostor za unapređenja i veći izbor potencijalno korisnih tehnika optimizacije.

Pri optimizovanju moramo da dobro poznajemo i poštujemo razna ograničenja sa kojima se suočavamo. Neka od ograničenja su implicirana poslovnim zahtevima (kako funkcionalnim tako i nefunkcionalnim), neka druga su posledica arhitekture procesora ili računara, a neka ograničenja nastaju usled prethodno načinjenih izbora tokom razvoja, kao što su izbor arhitekture softvera, izbor tehnologije za povezivanje komponenti ili izbor nekih infrastrukturnih elemenata (na primer, sistem za upravljanje bazom podataka ili sistem za deljenje fajlova i drugo).

Eksperimentisanje

Svaka pojedinačna optimizacija softvera je uvek samo *pokušaj* optimizovanja. Nakon što *probamo* neku optimizaciju, moramo uvek da proverimo da li ona zaista donosi očekivan pozitivan efekat. Ako eksperimentalno potvrdimo da optimizacija donosi povećanje performansi, onda moramo da procenimo i da li je taj dobitak vredan odgovarajućeg kvarenja programskog koda ili nije. Tek ako procenimo da jeste, onda optimizaciju odobravamo i usvajamo kao implementiranu. U svim ostalim slučajevima bi trebalo da je obrišemo iz programa, tj. da vratimo programski

kod u stanje koje je prethodilo optimizaciji, ali i da dokumentujemo da smo tu optimizaciju probali i da nam nije odgovarala.

Eksperimentisanje sa optimizacijama niskog nivoa je relativno jednostavno, zato što su pojedinačne optimizacije najčešće dobro lokalizovane. Sa druge strane, eksperimentisanje sa optimizacijama visokog nivoa je složeno, obimno i skupo, ali nam je svejedno neophodno i korisno, zato što je cena eksperimentisanja ipak daleko manja nego što bi bila cena razvijanja dela softvera koji počiva na arhitekturi ili algoritmima koji ne mogu da pruže neophodne performanse.

12.5 Tehnike optimizacije

Optimizacija softvera se u praksi svodi na analizu i merenje performansi i odabir i primenu određenih tehnika, kojima se teži da se rastereti uočeno usko grlo. Skup mogućih tehnika je u osnovi veoma veliki. Na njihov broj utiče činjenica da optimizacije mogu da se izvode na više nivoa, kao i da možemo da imamo za cilj rasterećenje različitih uskih grla.

Tehnike optimizacije mogu da se podele na opšte i specifične. Opšte tehnike su one čija primena nije čvrsto vezana za konkretan programski jezik i mogu da se primenjuju na svim ili bar na većem broju različitih programskih jezika. Nasuprot njima, specifične tehnike optimizacije su one koje mogu da se primenjuju na jednom konkretnom programskom jeziku ili na manjem broju srodnih jezika. Neke specifične tehnike se čak odnose na konkretne prevodioce.

Tehnike optimizacije se značajno razlikuju i prema nivou na koji se odnose. Na primer, optimizacije visokog nivoa najčešće nisu vezane za konkretne programske jezike, pa se uglavnom svrstavaju u opšte tehnike. U skladu sa tim, specifične tehnike optimizacije obično imaju relativno nizak nivo.

Najpre ćemo da razmotrimo tehnike optimizacije prema nivoima, a zatim ćemo da predstavimo neke opšte tehnike i neke tehnike specifične za C++. Pregled tehnika ćemo da završimo ukazivanjem na neke od najčešćih grešaka pri optimizaciji.

12.5.1 Tehnike optimizacije visokog nivoa

Kao što se oblikovanje arhitekture i algoritama odvija na apstraktnijem nivou od pisanja delova programa, tako se i optimizacija na nivou arhitekture ili algoritama izvodi na apstraktnijem nivou. Ali apstraktnost se odnosi samo na principe i ciljeve, a ne i na konkretne tehnike optimizacije.

Optimizacije višeg nivoa se zapravo svode na revidiranje projekta. U tom smislu teško možemo da govorimo o velikom broju konkretnih tehnika, već je pre reč o širokom prostoru za menjanje postojećih ili pravljenje novih elemenata projekta. Što je optimizacija višeg nivoa, to je ona istovremeno i manje univerzalna i više zavisi od konkretnog problema, arhitekture i algoritama.

Na primer, ako imamo algoritme za pretraživanje kolekcije tekstova i algoritme za analizu astronomskih fotografija, u oba slučaja ćemo raditi sa velikim količinama podataka, ali ne možemo da na iste načine smanjujemo opterećenje radne memorije. Možemo da imamo sličan cilj, ili da pri razmatranju optimizacije primenjujemo neke slične principe, ali sama konkretna tehnika promene algoritma je potpuno drugačija, zato što se i algoritmi značajno razlikuju.

Ako ustanovimo da arhitektura ili projekat ne mogu da pruže neophodne performanse, to znači da imamo neke nove informacije. Po potrebi možemo da izvedemo i dodatne eksperimente i merenja i obezbedimo još dodatnih informacija. Sve prikupljene dodatne informacije predstavljaju dopunu skupa informacija o domenu i problemu, koje su ranije već bile prikupljene prilikom istraživanja sistema. Prikupljanjem dodatnih informacija smo praktično napravili reviziju istraživanja sistema.

Na osnovu novih ili izmenjenih informacija, moramo da ponovimo neke elemente projektovanja onih delova softvera na koje te informacije imaju uticaj. Na osnovu rezultata revizije istraživanja sistema, najpre se pravi revizija analize problema, pa zatim i revizija svih aspekata modela domena i softvera.

Drugim rečima, optimizacija visokog nivoa predstavlja vid naknadne razrade projekta i samim tim je daleko sličnija projektovanju softvera nego njegovoj implementaciji. Posledica je i da se tehnike optimizacije visokog nivoa praktično svode na tehnike projektovanja softvera. Zbog toga ćemo pri razmatranju konkretnih tehnika obraditi svega nekoliko tehnika koje spadaju u optimizacije visokog nivoa, a videćemo da će i one biti obrađene uglavnom na relativno apstraktnom nivou.

Ako se optimizacije visokog nivoa primenjuju nakon implementacije softvera, onda one zahtevaju mnogo promena, pa i ponavljanje pisanja programskog koda, što ima za rezultat njihovu visoku cenu. Zato se teži da se optimizacije visokog nivoa primenjuju unapred, pre implementiranja softvera.

12.5.2 Tehnike optimizacije niskog nivoa

Što je optimizacija nižeg nivoa, to se ona odnosi na manji deo konkretnog programskog koda. Optimizacije niskog nivoa su relativno bliske hardveru računara i najviše utiču na rad procesora i njegovih komponenti. Zato se optimizacije niskog nivoa odnose pre svega na rasterećenje procesora ili delova procesora, kao što su procesorska jezgra, keš memorija, procesorska (memorijska) magistrala i slično. One za osnovni cilj imaju povećavanje efikasnosti rada procesora, u smislu povećavanja brzine rada programa.

Nasuprot tome, optimizacijama niskog nivoa najčešće može da se ostvari samo ograničeno rasterećenje ostalih resursa, kao što su radna memorija, mreža, diskovi i drugo. Što je neki resurs fizički udaljeniji od procesora, to je za njegovo rasterećivanje

neophodno da se optimizacija preduzima na višem nivou: opterećenje radne memorije zavisi najviše od odabranog algoritma; opterećenje diskova zavisi i od algoritma i od arhitekture; opterećenje mreže zavisi najviše od arhitekture i slično.

Uglavnom ćemo se fokusirati na optimizacije niskog nivoa, koje se tiču rasterećenja procesora. Mnoge od njih počivaju na prilagođavanju programskog koda specifičnostima računarskog sistema, arhitekture procesora, magistrale podataka i drugo. Među njima su izdvajaju one koje ističu upotrebu keš memorije i one kojima je cilj smanjivanje broja skokova i grananja.

Optimizacije koje ističu upotrebu keš memorije obično se odnose na menjanje redosleda upotrebe podataka tako da se poveća verovatnoća da je naredni upotrebljen podatak već u keš memoriji. Povećanje efikasnosti usled intenziviranja upotrebe keš memorije može da bude zaista dramatično, čak i više od 100 puta. Dobitak je prvenstveno u tome što je rad sa keširanim podacima višestruko brži, ali i u tome što se izbegava ponovljeno punjenje keš memorije istim podacima.

Skokovi i grananja imaju veoma visoku cenu pri izvršavanju programa. Savremeni procesori dele obradu instrukcija u korake i istovremeno obrađuju, u različitim fazama, po nekoliko uzastopnih instrukcija. Skokovi remete sekvencijalno izračunavanje i poništavaju značaj (ali ne i cenu) već urađenog posla na izvršavanju nekoliko narednih instrukcija. U slučaju petlji i uslovnih naredbi, procesori pokušavaju da predvide tok izvršavanja, ali to često nije uspešno. Zbog toga smanjivanjem broja skokova i grananja možemo da doprinesemo boljoj iskorišćenosti procesorskog vremena. Dodatno, poželjno je da se izbegavaju skokovi na udaljene delove koda, zato što je manja verovatnoća da takvi delovi koda budu keširani.

12.5.3 Opšte tehnike optimizacije

Odbacivanje nepotrebne preciznosti

Što su podaci sa kojima radimo precizniji, to je njihov zapis veći i izračunavanje operacija je složenije. Znači, mogli bismo da očekujemo da smanjivanjem preciznosti možemo da dobijemo na efikasnosti, zato što se tako pojednostavljuju zadaci koje postavljamo pred procesor. Odbacivaje nepotrebne preciznosti može da se primenjuje kao smanjivanje preciznosti u radu sa realnim brojevima (tj. brojevima u zapisu sa pokretnom zapetom) i kao smanjivanje opsega celih brojeva. Oblik ove optimizacije je i zamenjivanje rada sa realnim brojevima radom sa celim brojevima.

Pri ovoj tehnici optimizacije potrebno je da imamo u vidu da se njena primena različito odražava na različite vrste operacija i da razmatramo efekte optimizacije u odnosu na svaku vrstu operacija posebno.

- Efekat ove optimizacije na operacije prepisivanja može da bude minimalan, pa čak i negativan, ako je ciljna veličina zapisa različita od veličine reči procesora.
- Cena konverzija nije lako predvidiva i efekti moraju da se izmere.
- Efikasnost izračunavanja često zavisi od procesora.

Danas se ova optimizacija relativno retko primenjuje na realne brojeve, zato što na većini savremenih procesora, koji imaju ugrađene jedinice za rad sa realnim brojevima, razlika u radu sa tipovima `double` i `long double` praktično ne postoji, a rad sa tipom `float` može čak da bude i sporiji nego rad sa većim tipovima.

Upotreba osnovnog celobrojnog tipa

Procesori obično najbrže rade sa celobrojim podacima čija veličina odgovara veličini procesorske reči. Operacije na većim podacima se uglavnom implementiraju pomoću više operacija na manjim podacima, pa su zato manje efikasne. Operacije na manjim celobrojnim podacima u većini slučajevima su jednako efikasne, ali u nekim slučajevima mogu da zahtevaju implictne konverzije, koje mogu da dovedu do smanjivanja efikasnosti.

Upotreba umetnutih funkcija i metoda

Pozivanje posebno implementirane funkcije obuhvata prenos argumenata, pozivanje funkcije i povratak iz nje, a često i čuvanje i restauriranje vrednosti nekih registara. U slučaju relativno jednostavnih funkcija, cena samog mehanizma pozivanja može da bude čak i veća nego cena izvršavanja tela funkcije. U takvim slučajevima može da se ostvari značajno povećavanje efikasnosti tako što se pozivanje funkcije zameni ugradnjom njenog tela na mestu pozivanja. To se naziva *umetanje* funkcija i metoda.

Ova tehnika može da se primenjuje manuelno (eksplicitnim prepisivanjem tela funkcije), pomoću odgovarajuće tehnike programskog jezika ili u okviru optimizacije koju preduzima prevodilac. Na primer, osnovna tehnika za umetanje funkcija u programskom jeziku C je bila upotreba makroa, a u programskom jeziku C++ se koristi posebna deklaracija `inline` kao sugestija prevodiocu da funkciju prevodi kao umetnutu[68].

[68] Zbog karakteristike programskog jezika C++ da se svi metodi definisani u okviru definicije klase tretiraju kao da su označeni da treba da budu umetnuti, u praksi nastaje prava poplava umetnutih metoda. Zato savremeni prevodioci sve više ignorišu odgovarajuće sugestije i automatski procenjuju da li se neka funkcija ili metod prevode kao umetnute ili ne.

Na primer, funkcija `swap(a,b)`, koja razmenjuje vrednosti dva cela broja, radi 10 do 15 puta brže ako se prevede kao umetnuta.

Integracija petlji

Slično kao što pozivanje funkcije ima svoju cenu, tako svoju cenu ima i implementacija petlje. Ako je telo petlje relativno jednostavno, onda cena mehanizma ponavljanja može da bude značajna u odnosu na cenu posla koji se radi u petlji. Ako imamo više petlji, takvih da svaka radi neki jednostavan posao, onda njihovo spajanje u jednu petlju sa složenijim korakom može da nam donese značajne uštede procesorskog vremena.

Integracija petlji može da uštedi i do 50% vremena.

Ova optimizacija je suprotna pravilima pisanja čistog i razumljivog koda, pa čak imamo i refaktorisanja koja rade upravo suprotno. Pri tome je često i sasvim suvišna, zato što će neki prevodioci sami da je primene.

Izmeštanje invarijanti van petlje

Sve ono što se računa u petlji, a ima uvek isti rezultat, potrebno je da se izmesti iz petlje i izračuna pre nje. Na primer, ako imamo petlju poput:

```
for( int i=0; i<limit(a,b,c); i++ )...
```

gde se promenljive `a`, `b`, `c`, i vrednost funkcije `limit(a,b,c)` ne menjaju tokom izračunavanja, onda je bolje da to zamenimo sa:

```
auto lim = limit(a,b,c);
for( int i=0; i<lim; i++ )...
```

Ako je telo petlje relativno jednostavno, ili je izraz koji izračunava invarijantu relativno složen, ova optimizacija može višestruko da ubrza petlju.

Na primer, izmeštanje računanja veličine vektora van petlje može da ubrza petlju sa jednostavnim telom čak i više od 3 puta (tj. za više od 65%):

```
auto sz = v.size();
for( size_t i=0; i<sz; i++ )...
```

Zamenjivanje dinamičkog uslova statičkim

Ako opseg broja ponavljanja nije fiksan, ili se obrada ne vrši za sve elemente kojima pristupamo, onda nekada može biti efikasnije da se posao ipak uradi za sve podatke nego da se stalno proveravaju granice ili uslovi. Zamenjivanje dinamičkog uslova statičkim može da ima sličnosti sa izmeštanjem invarijanti, ali ima dodatni efekat – osim što se iz petlje izbacuje računanje uslova ili dela uslova, čime se

neposredno štedi procesorsko vreme, ovde postoji i dodatna ušteda zbog toga što se eliminišu neka grananja.

Na primer, ako imamo neki niz i želimo da anuliramo vrednosti elemenata koji nisu već jednaki nuli:

```
for( size_t i=0; i<sz; i++ )
    if( niz[i] ) niz[i] = 0;
```

onda može da bude mnogo efikasnije da postavimo sve elemente na 0 nego da proveravamo njihove vrednosti:

```
for( size_t i=0; i<sz; i++ )
    niz[i] = 0;
```

U predstavljenom primeru razlika u brzini može biti i 5 do 10 puta.

Razmotavanje petlji

Jedan način da se smanji broj grananja je da se smanji broj ponavljanja u petlji, a da se telo petlje eksplicitno navede više puta uzastopno. Na primer ako u petlji u N prolaza obrađujemo po jedan podatak, a znamo da obrađujemo $8N$ podataka, onda umesto toga možemo da napravimo petlju kroz koju prolazimo N puta ali tako da u svakom koraku sekvencijalno obradimo po 8 podataka.

Na ovaj način se neposredno narušava princip izbegavanja ponavljanja koda, ali može da se dobije i više od 50% efikasnije izračunavanje.

Ova optimizacija ne radi uvek. Ako je telo petlje već relativno složeno, onda njegovim multipliciranjem može da se dobije programski kod koji se slabije kešira, pa zbog toga rezultat može da bude i sporiji nego originalni kod. Takođe, prevodilac može da odluči da neke od operacija u petlji optimizauje na neki drugi način, pa da razmotavanje uspori rad programa, umesto da ga ubrza.

Tablice unapred izračunatih vrednosti

Ako se neke funkcije više puta izračunavaju u petlji, a za ograničen broj različitih argumenata, onda može da bude efikasnije da se umesto izračunavanja konsultuju tablice, koje sadrže već izračunate vrednosti. Prvi doprinos je što se tako izbegavaju složena računanja i grananja. Drugi doprinos je što su takve tablice obično relativno male, pa efikasno koriste keš memoriju. Ako je izračunavanje relativno složeno ili sadrži granjanja, onda ova optimizacija može da donese veliko unapređenje efikasnosti.

Na primer, neka niz sadrži cele brojeve u rasponu od 0 do 20 i u petlji želimo da izbrojimo koliko brojeva je deljivo sa 5. Uobičajen način je da računamo neposredno:

```
for...
    if( niz[i] % 5 != 0 )
        n++;
```

Ali ako napravimo tablicu, koja za brojeve deljive sa 5 sadrži 1 a za ostale 0, onda možemo da pomoću tablice ubrzamo petlju i do 10 puta:

```
for...
    n += tablica[ niz[i] ];
```

Kao i u slučaju razmotavanja petlji, može da se desi da prevodilac sam izvede neke optimizacije da da dobitak bude svega 30-50%, a ne očekivanih 90%.

Eliminacija grananja i petlji

Ovu tehniku možemo da posmatramo kao uopštenje ili kombinaciju nekih od prethodnih tehnika, koje ostvaruju doprinos performansama tako što smanjuju broj grananja i petlji.

Odličan primer je poznati slučaj brojanja bitova u 32-bitnom broju pomoću tablice izračunatih vrednosti i razmotavanja petlji:

```
const short tablica[] = {0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4};
short brojBitova(int x)
{
   return tablica[(x) & 0xF] +
          tablica[(x >> 4) & 0xF] +
          tablica[(x >> 8) & 0xF] +
          tablica[(x >> 12) & 0xF] +
          tablica[(x >> 16) & 0xF] +
          tablica[(x >> 20) & 0xF] +
          tablica[(x >> 24) & 0xF] +
          tablica[(x >> 28)];
}
```

Dobijena implementacija nije sasvim laka za razumevanje i održavanje, ali zato i do 10 puta skraćuje vreme izvršavanja u odnosu na pojedinačno brojanje.

Smanjivanje broja argumenata funkcije

Prenošenje argumenata i rezultata potprograma se uobičajeno odvija pomoću registara i računskog steka. Upotreba registara procesora je daleko efikasnija, ali broj registara koji mogu da se koriste za te namene je ograničen. Znači, što je više argumenata, ili što su argumenti veći, to je manje verovatno da njihovo prenošenje može da se izvede pomoću registara i dolazi do intenzivnije upotrebe steka. Cilj smanjivanja broja argumenata funkcije je da se smanji upotreba steka i da se tako podigne efikasnost mehanizma za prenošenje argumenata, a sa tim i čitavog pozivanja funkcije.

Doprinos ove optimizacije zavisi od mnogo faktora – broja argumenata, broja registara procesora, veličine keša, načina i učestalosti pozivanja funkcije, načina upotrebe argumenata u funkciji i drugo, pa da ga je zato relativno teško predvideti. Zbog toga moramo da merimo i poredimo performanse pre i posle optimizacije, da bismo znali da li je optimizacija isplativa.

Problem je u tome što smanjivanje broja i veličine argumenata često mora da se nadoknadi na drugi način. Na primer, umesto da se prenosi pet argumenata, oni mogu da se zapišu u pomoćnoj strukturi, pa da se onda prenese samo adresa te strukture. Međutim, tako samo menjamo način prenošenja – umesto da prepisujemo argumente u registre ili na stek, mi ćemo ih prepisati u pomoćnu strukturu, koja će verovatno opet biti na steku (ako je lokalna).

Na značaj ove optimizacije utiču i savremene arhitekture procesora. One obezbeđuju veći broj registara, koji mogu da se koriste za prenošenje argumenata. Pored toga, moramo da imamo u vidu da je rad sa računskim stekom relativno brz, zato što se stek odlično kešira.

Izbegavanje globalnih promenljivih

Upotreba lokalnih promenljivih je često efikasnija od upotrebe globalnih promenljivih. Za to ima nekoliko razloga:

- lokalne promenljive se zapisuju na računskom steku, koji se dobro i efikasno kešira, pa zato predstavlja verovatno najefikasniji deo radne memorije;
- ako program radi u više niti, onda globalne promenljive mogu da se dele između niti i moramo da se staramo o njihovom deljenju, što komplikuje implementaciju i usporava rad;
- ako se lokalna promenljiva često koristi u nekoj funkciji, onda prevodilac može da optimizuje prevod na mašinski jezik tako da ta promenljiva bude smeštena u registru a ne u memoriji[69]; odgovarajuća optimizacija za globalne promenljive je dovoljno složena da se retko preduzima
- i drugo

Postoje i slučajevi kada je upotreba globalne promenljive bolji izbor. Na primer, ako se neki podatak koristi na veoma velikom broju mesta, onda lokalizacija njegove

[69] U nekim programskim jezicima postoje deklaracije kojima se sugeriše prevodiocima da neku promenljivu čuvaju u registru. Na primer, u C/C++-u za to služi ključna reč `register`. Međutim, savremeni prevodioci uglavnom ignorišu takve sugestije i sami pokušavaju da procene da li je isplativije da se koristi registar ili memorija.

upotrebe podrazumeva da se on prenosi kao argument velikom broju funkcija, pa se stalno plaća cena njegovog prepisivanja (ili bar prepisivanja njegove adrese). U takvim slučajevima je obično potrebno da se eksperimentisanjem i merenjem proveri da li je efikasnije koristiti lokalne ili globalne promenljive.

Lokalnost upotrebe podataka

Savremeni procesori raspolažu višeslojnom i veoma efikasnom keš-memorijom. Međutim, da bismo iskoristili njene mogućnosti moramo da pazimo šta radimo. Efikasnost keša počiva na prostornoj i vremenskoj lokalnosti upotrebe podataka i programskog koda. Vremenska lokalnost podrazumeva ponovljenu upotrebu istih podataka u kratkim vremenskim intervalima, a prostorna lokalnost podrazumeva uzastopnu upotrebu podataka koji su zapisani jedni blizu drugih u memoriji.

Na primer, uobičajeno je da se dobro keširaju računski stek, lokalne promenljive (koje se uobičajeno nalaze na računskom steku) ili nizovi koji se sekvencijalno obrađuju. Dinamičke strukture podataka, kao što su povezane liste i stabla, se daleko teže keširaju, što utiče na njihovu efikasnost. Takođe, dobro se kešira programski kod koji se često ponavlja (telo petlje), a lošije kod koji divergira (velike funkcije sa mnogo grananja, kod koji nema duže sekvencijalne delove i slično).

Razmotrimo, na primer, naredni primer programskog koda, koji računa sumu elemenata matrice:

```
const unsigned len = 10000;
int niz[len][len];
...
int sum = 0;
for( unsigned i=0; i<len; i++ )
    for( unsigned j=0; j<len; j++ )
        sum += niz[j][i];
```

Ovaj programski kod je ispravan, ali ne koristi lokalnost podataka pri obradi elemenata niza. Nakon što se sumi doda element `niz[j][i]`, naredni element koji će se dodati je (najčešće) `niz[j+1][i]`. Ova dva elementa se nalaze u memoriji 40000 bajtova jedan od drugog i malo je verovatno da će oba da budu istovremeno u keš memoriji, a čak i da jesu, sledeći element će biti još 40000 bajtova dalje, pa sledeći još 40000 itd.

Očigledno je da ova implementacija ne poštuje lokalnost upotrebe podataka. Ako bismo je popravili tako da se posle elementa `niz[j][i]` obrađuje `niz[j][i+1]`, a ne `niz[j+1][i]`:

```
for( unsigned i=0; i<len; i++ )
    for( unsigned j=0; j<len; j++ )
        sum += niz[i][j];
```

onda bi uzastopno obrađivani podaci bili susedni u memoriji i bio bi poštovan princip lokalnosti upotrebe podataka. Mogli bismo sa pravom da očekujemo da će izmenjen programski kod biti efikasniji.

Zaista, ova mala izmena može da napravi veliku razliku u performansama. U zavisnosti od arhitekture procesora i veličine keša, ali i od veličine matrice koja se obrađuje, povećanje efikasnosti može da bude čak i do 100 puta.

Lokalnost upotrebe programskog koda

Kao što mogu da se keširaju upotrebljavani podaci, tako se kešira i programski kod. Zbog toga je važno da se program piše tako da se njegovi delovi bolje keširaju. Ova tehnika optimizacije počiva na izbegavanju nekoliko osnovnih oblika slabog keširanja programskog koda:

- ako imamo relativno veliku funkciju sa mnogo grananja, onda je povećana verovatnoća da će neki skokovi unutar te funkcije biti skokovi na delove koda koji nisu keširani;
- ako koristimo dinamičko vezivanje funkcija, onda konkretne pozivane funkcije često neće biti keširane;
- ako iz jedne funkcije često pozivamo brojne druge funkcije, onda ni one verovatno neće biti dobro keširane
- i drugo.

Neki od vidova unapređivanja lokalnosti koda su:

- skraćivanje funkcija;
- umetanje funkcija koje se najčešće pozivaju;
- eliminacija grananja;
- pažljivo određivanje redosleda proveravanja uslova;
- upotreba statičkog umesto dinamičkog vezivanja funkcija
- i drugo.

Pažljivo određivanje redosleda proveravanja uslova

Cilj ove optimizacije je da se smanje broj poređenja i broj skokova. Osnovna ideja je da se u slučajevima gde se proverava više uslova i bira jedna od više grana, najpre proveravaju oni uslovi za koje se očekuje da će češće biti ispunjeni, da bi se tako smanjio i broj proveravanih uslova i broj skokova.

Na primer, ako imamo izraze `uslov1`, `uslov2` i `uslov3`, takve da je najčešće ispunjen `uslov1`, pa onda `uslov2`, pa `uslov3`, a najređe nije ispunjen nijedan od njih, onda bi njihovo proveravanje trebalo da ide tim redom:

```
if( uslov1 ){...}
else if( uslov2 ){...}
else if( uslov3 ){...}
else {...}
```

Jedna od varijanti ove optimizacije je i hijerarhijska organizacija granjanja, tako da se umesto sekvencijalnog proveravanja velikog broja uslova oni podele u hijerarhiju približno ravnomerne dubine:

```
if( A ){
    if( uslov1 ) {...}
    else if( uslov2 ) {...}
    else {...}
}
else {
    if( uslov3 ) {...}
    else if( uslov4 ) {...}
    else {...}
}
```

Ova optimizacija može da se primeni i na višestruka grananja, tj. na naredbu `switch`.

Varijanta ove optimizacije je da se prevodiocu sugeriše koja uslovna grana će češće da se bira, da bi mogao da optimizuje mašinski kod tako da se linearno izvršava ona grana koja je verovatnija i da se manje vremena gubi na uslovnim skokovima. U programskom jeziku C++, od standarda C++20, postoje atributi `likely` i `unlikely`. Oni služe da se označe grane za koje očekujemo da predstavljaju verovatniji odnosno manje verovatan smer uslovnog grananja. Na primer, ako znamo da funkcija `skoroUvekTacna` najčešće vraća `true`, a sasvim retko `false`, onda možemo da napišemo:

```
if( skoroUvekTacna(...)) [[likely]] {...}
else {...}
```

ili:

```
if( skoroUvekTacna(...)) {...}
else [[unlikely]] {...}
```

Snižavanje složenosti operacije

Neke operacije mogu da se zamene efikasnijim operacijama. Na primer, umesto `a*8` možemo da napišemo `a<<3`. Takve optimizacije su nekada bile veoma cenjene, ali danas obično očekujemo da ih prevodioci sami izvode, čak i u složenijim slučajevima, i to bolje nego što bi prosečni programeri mogli da urade. Zato se takve optimizacije danas relativno retko primenjuju.

Snižavanje složenosti algoritma

Jedna od osnovnih opštih tehnika optimizacije je zamenjivanje neefikasnog algoritma nekim drugim algoritmom koji ima nižu složenost izračunavanja. Ova optimizacija ima nivo algoritma i zato je njena primena nešto složenija. Zato što spada u optimizacije visokog nivoa, uglavnom je poželjno da se primenjuje unapred.

Osnovna ideja je da se algoritam zameni alternativnim algoritmom, koji ima nižu složenost. Neposredna posledica je da će efikasnost novog algoritma sporije da se smanjuje sa porastom obima posla, u odnosu na prethodni algoritam.

Iako je konceptualno sasvim jednostavna, ova tehnika je među najsloženijim tehnikama optimizacije. Ne postoji opšte pravilo kako napraviti algoritam sa nižom složenošću, pa čak najčešće ne možemo da znamo unapred ni da li takav algoritam postoji.

Čak i kada znamo da postoji algoritam koji ima nižu složenost, ipak moramo da budemo oprezni pri njegovoj primeni. Problem je u tome što algoritmi sa nižom složenošću često imaju skuplje korake. Zbog toga može da se desi da za neke manje skupove podataka ovakvi algoritmi budu manje efikasni. Isplativost algoritama mora da se procenjuje na osnovu njihove složenosti, složenosti jednog koraka i karakteristika problema.

Izbor rešenja prema najčešćem slučaju

Neki algoritmi su za neke slučajeve efikasniji a za neke druge manje efikasni. Zbog toga je potrebno da se izbor algoritma pravi u skladu sa očekivanim slučajevima upotrebe.

Na primer, za uređivanje kratkih nizova primitivan algoritam sortiranja *bubble-sort* može da bude efikasniji od naprednijeg algoritma *quick-sort*.

Može se reći da optimizacija „Pažljivo određivanje redoseda proveravanja uslova“ predstavlja specijalan slučaj ove optimizacije, ali i da ova optimizacija predstavlja specijalan slučaj optimizacije „Snižavanje složenosti algoritma“.

Pisanje zatvorenih funkcija

Funkcija (ili drugi potprogram) je *zatvorena* ako ne poziva nijednu drugu funkciju. Nešto šira definicija dopušta da zatvorena funkcija poziva zatvorene umetnute funkcije. Značajna karakteristika zatvorenih funkcija je da prevodiocu za njihovo

prevođenje i optimizovanje nisu potrebne nikakve druge informacije o programu osim tela konkretne funkcije i informacija o podacima kojima ona rukuje.

Takve funkcije se bolje optimizuju od strane prevodilaca, pa se zato podstiče njihovo pisanje. Prevodioci često mogu da naprave efikasniji izvršni kod ako se koristi jedna složenija zatvorena funkcija nego ako je odgovarajuće ponašanje podeljeno na nekoliko manjih funkcija.

I ova optimizacija je suprotstavljena pravilima dobrog dizajna i refaktorisanja. Osim što proizvodi veće funkcije sa potencijalno grupisanim odgovornostima, ova tehnika posredno dovodi do ponavljanja delova koda, pa se njenom primenom otežava održavanje programa.

Uvođenje konkurentnog ili distribuiranog izračunavanja

Uvođenje konkurentnog ili distribuiranog izračunavanja se uobičajeno naziva *paralelizacijom* softvera.

Savremeni procesori imaju više izvršnih jezgara, što omogućava istovremeno izvršavanje više niti. To je posebno važno kada se radi o programima koji se pišu za servere i radne stanice, zato što takvi računari mogu da imaju po više procesora sa po više od 100 jezgara. Naredni korak je uvođenje distribuiranog izračunavanja, tako da se izvršavanje programa podeli ne samo na više niti ili procesa već i da se ti procesi pokreću na više čvorova (tj. odvojenih računara). Uvođenje konkurentnog i distribuiranog izračunavanja može da donese veoma značajno ubrzanje, koje može da pravi razliku između upotrebljivosti i neupotrebljivosti softvera.

Paralelizovanje izračunavanja je često veoma složeno, a može da pruži najviše linearan dobitak u performansama u odnosu na broj jezgara procesora. Na primer, ako program pokrenemo na 10 računara sa po 20 jezgara, onda će on raditi do 200 puta brže. U praksi je ubrzanje najčešće značajno niže, zato što se gubi vreme na razmenjivanju podataka i sinhronizaciji.

Imajući to u vidu, uvek je važnije da prvo odaberemo algoritam što niže složenosti, pa tek onda da se bavimo njegovom paralelizacijom. U suprotnom možemo da dobijemo softver koji radi veoma brzo za male količine podataka ali nije upotrebljiv za velike.

Veoma je teško naknadno paralelizovati algoritme koji su po svojoj prirodi sekvencijalni. Na primer, ako algoritam u nekom koraku zahteva da su svi prethodni koraci dovršeni i koristi njihove rezultate, onda se takav problem teško paralelizuje. U takvim slučajevima je potrebno mnogo sinhronizacije, što otežava programiranje i usporava izvršavanje.

Implementacija paralelizacije može da se ostavi za kasnije faze razvoja, ali razmišljanje o paralelizaciji i njeno planiranje je poželjno da se urade već u ranijim fazama razvoja softvera, tj. pri određivanju arhitekture softvera i odabiru

odgovarajućih algoritama. Zato se paralelizacija obično svrstava u optimizacije visokog nivoa.

12.5.4 Primeri tehnika specifičnih za C++

Svaki programski jezik ima neke specifične karakteristike koje otvaraju prostor za specifične tehnike optimizacije. Ovde ćemo predstaviti neke tehnike optimizacije koje su specifične za programski jezik C++.

Neke od tih tehnika nisu izvorno zamišljene kao tehnike optimizacije koda, ali mogu da doprinesu performansama ili da olakšaju optimizaciju i održavanje programskog koda. Zato se podjednako često razmatraju i kao tehnike pisanja efikasnog koda, onda kada se on po prvi put piše, i kao tehnike za naknadnu optimizaciju.

Koristiti standardnu biblioteku

Standardna biblioteka programskog jezika C++ je u svim svojim aspektima projektovana i implementirana tako da obezbedi visoke performanse. Ona u velikom broju slučajeva pruža visok nivo apstrakcije, koja se prevashodno zasniva na upotrebi parametarskog polimorfizma. Za razliku od hijerarhijskog polimorfizma, koji podrazumeva upotrebu dinamičkog vezivanja metoda i nešto nižu efikasnost, parametarski polimorfizam omogućava da se pri prevođenju programa u potpunosti iskoriste mogućnosti strogog proveravanja tipova, statičkog vezivanja, pa i optimizacije prema konkretnim slučajevima upotrebe.

Veoma je teško napisati programski kod koji radi posao efikasnije nego odgovarajući elementi standardne biblioteke, koji su godinama razvijani, debagovani i optimizovani. Čak i kada je to moguće, razumno je postavi pitanje isplativosti zato što je dobitak u performansama obično relativno mali, a cena koju plaćamo je dobijanje nefleksibilnijeg koda.

Zbog toga se korišćenje standardne biblioteke svrstava u osnovna pravila pisanja efikasnog koda na programskom jeziku C++. Naravno, to je pre svega jedna od osnovnih tehnika programiranja na C++-u, a ne tehnika optimizacije. Međutim, standardna biblioteka se sa svakom novom verzijom standarda i sa svakom novom verzijom implementacije prevodilaca i biblioteke povećava i unapređuje, što nam pruža priliku da upotrebu standardne biblioteke iskoristimo i kao vid optimizacije.

Relativno često se dešava da su neki delovi programa implementirani eksplicitno, bez upotrebe standardne biblioteke, zato što u trenutku njihovog pisanja odgovarajuća funkcionalnost nije bila obuhvaćena standardnom bibliotekom. Nakon što je odgovarajuća funkcionalnost uvrštena u standardnu biblioteku, ima smisla da se razmotri zamenjivanje postojeće implementacije upotrebom tih novih elemenata biblioteke. Može da se očekuje i dobitak u performansama i dobitak u preglednosti programskog koda.

Upotrebljavati reference za prenošenje argumenata

Prenošenje objekata po vrednosti može da bude relativno skupo, kako zbog veličine tako i zbog eventualne složenosti unutrašnje strukture objekata. Svako prepisivanje objekata podrazumeva i primenu konstruktora kopije ili operatora dodeljivanja, koji sprečavaju da novi i stari primerak objekta nastave da nekontrolisano dele neke elemente unutrašnje strukture.

Umesto prenošenja objekata po vrednosti mnogo je bolje upotrebiti prenošenje po referenci na konstantan objekat. U tom slučaju se objekat prenosi po imenu (adresi), ali tako da ne može da se menja. To je semantički skoro isto kao prenošenje po vrednosti, ali je mnogo efikasnije. Prenošenje putem referenci je veoma slično prenošenju putem pokazivača, ali je sintaksno čistije i onemogućavaju se neke greške poput promene vrednosti pokazivača, praznih pokazivača i drugo.

Reference moraju oprezno da se koriste za vraćanje rezultata. Vraćena referenca mora da se odnosi na objekat koji će preživeti povratak iz funkcije, tj. ne sme da se odnosi na lokalnu promenljivu ili drugi objekat koji će biti obrisan pri povratku iz funkcije.

Ova tehnika se koristi prevashodno kao tehnika pisanja programa, ali može da se upotrebi i kao tehnika optimizacije.

Uvoditi lokalne promenljive što bliže mestu upotrebe

U programskom jeziku C++ pravljenje objekta pretpostavlja i njegovu inicijalizaciju, a brisanje objekta njegovu deinicijalizaciju. To praktično znači da svako pravljenje nekog privremenog objekta u okviru funkcije ima svoju cenu, koja može da bude relativno značajna, posebno ako se često poziva. Zbog toga je poželjno da se lokalne promenljive ne uvode na početku tela funkcije, već neposredno pre nego što budu potrebne. Ako je to u nekom ugneždenom bloku, na primer u jednoj od grana uslovne naredbe, onda će taj lokalni objekat biti napravljen (a kasnije i obrisan) samo ako se ta grana izvrši. Na taj način se dobija na performansama, a pored toga i programski kod postaje razumljiviji.

Ova optimizacija nije specifična samo za C++, ali u njemu ima značajnije posledice nego u većini drugih jezika. Na primer, u jezicima kao što su Java ili C#, objekti se koriste putem referenci, pa ni deklarisanje ni oslobađanje promenljive ne podrazumevaju značajan dodatni posao, a pravljenje i brisanje mogu da se rade dublje u telu. U programskom jeziku C ne postoje automatska konstrukcija i destrukcija, pa se uvođenje lokalne promenljive na početku funkcije reflektuje samo na njenu alokaciju, a to se i inače radi odjedanput na početku funkcije za sve lokalne promenljive definisane u funkciji. Tako ispada da u mnogim drugim jezicima ovo nije optimizacija, već samo pravilo za preglednije pisanje programskog koda.

Odložena inicijalizacija objekata

Odložena inicijalizacija objekata je u suprotnosti sa principom *RAII* (pravljenje resursa je njegova inicijalizacija, engl. *resource acquisition is initialization*), koji se svrstava u najvažnije principe programiranja na programskom jeziku C++. Osnovna ideja principa je da je svaki napravljen objekat u potpunosti inicijalizovan i spreman za upotrebu.

Nasuprot tome, ova tehnika optimizacije počiva na pretpostavci da potpuna inicijalizacija objekata može da bude relativno složena i skupa, a da nama često nisu neophodni svi aspekti te složene inicijalizacije. U takvim slučajevima, kada su neki aspekti inicijalizacije potrebni samo povremeno i relativno retko, možda ima smisla da pri pravljenju objekata izvedemo samo najosnovniji deo inicijalizacije, a da ostatak posla radimo tek ako nam bude potreban.

Na taj način mogu da se ubrzaju neke jednostavnije operacije sa objektima, ali može i da se uspori rad u nekim kompleksnijim slučajevima, zato što ćemo morati prvo da proveravamo da li je odgovarajući aspekt inicijalizacije obavljen, pa da ga obavimo ako nije, pa tek onda da nastavimo posao.

Ova optimizacija mora da se izvodi vrlo oprezni i da se dobro dokumentuje. Potencijalni problemi nastaju ako se pri nekom narednom menjanju programa pretpostavi da je neki objekat potpuno inicijalizovan pri konstrukciji, a da to nije slučaj. Takve greške mogu da budu prikrivene i da se relativno teško pronalaze i ispravljaju.

Koristiti liste inicijalizacija članova i baznih objekata

Pri konstrukciji objekata u programskom jeziku C++, inicijalizacija članova objekta i nasleđenih delova objekta može da se obavlja na dva načina – u telu konstruktora i u okviru liste inicijalizacija. Značajno je jednostavnije i efikasnije upotrebljavati listu inicijalizacija.

Jedna od osnovnih pretpostavki pri pisanju konstruktora je da u trenutku kada započinje izvršavanje tela konstruktora već imamo dovršenu konstrukciju svega što čini objekat koji pravimo. Znači, svi članovi objekta, kao i svi nasleđeni delovi su već konstruisani. U telu konstruktora možemo da ih menjamo i da usklađujemo njihov sadržaj prema potrebnim ciljevima, ali sve te promene se obavljaju nad objektima i podacima koji su već inicijalizovani. To znači da se postavljanjem vrednosti u telu konstruktora u izvesnoj meri poništava ono što je već urađeno pri konstrukciji odgovarajućih delove, tj. da se njihov sadržaj dva puta postavlja – najpre pri konstrukciji, pa onda pri menjanju.

Namena liste inicijalizacija je da se pomoću nje opiše kako će koji član i koji nasleđeni deo klase da se inicijalizuje, tj. koji konstruktori i sa kojim parametrima će biti za to upotrebljeni. Ako podatak ili nasleđeni deo nisu navedeni u listi inicijalizacija, onda će se za njihovu inicijalizaciju upotrebiti podrazumevani

konstruktor. Zbog toga je mnogo efikasnije da navedemo kako će već pri konstrukciji ti elementi dobiti odgovarajući sadržaj, umesto da ih prvo pravimo i inicijalizujemo pomoću podrazumevanih konstruktora a zatim da tako određeni sadržaj menjamo.

Koristiti konstrukciju objekata a ne dodeljivanje

Motivacija za ovu tehnika je slična kao i za upotrebu liste inicijalizacija. Umesto da objekat prvo napravimo, pa mu onda dodelimo vrednost, bolje je da ga odmah napravimo na odgovarajući način.

Znači, umesto da pišemo nešto poput:

```
A a;
a = 5;
```

ili nešto kao:

```
A a;
a = (A)5;   // Isto kao: a = A(5);
```

bolje je da pišemo nešto poput:

```
A a = 5;   // Ovo je isto kao: A a { 5 };
```

U prvom slučaju, prvo pravimo objekat `a`, pomoću konstruktora bez argumenata, pa mu onda dodeljujemo vrednost 5.

U drugom slučaju prvo pravimo objekat `a`, pomoću konstruktora bez argumenata, pa mu onda dodeljujemo objekat konstruisan konstruktorom koji je dobio parametar 5. Ako ne postoji eksplicitno napisana operacija dodeljivanja celog broja objektu klase `A`, onda se prvi slučaj prevodi potpuno isto kao i drugi.

U trećem slučaju pravimo novi objekat `a` pomoću konstruktora koji ima celobrojni parametar 5. Samo u ovom slučaju imamo samo jednu operaciju, dok u prethodna dva slučaja prvo pravimo objekat pa ga onda menjamo.

Iz delova koda koji se optimizuju izbaciti rukovanje izuzecima

Ova optimizacija se relativno često predlaže, ali je ustvari diskutabilna. Njena motivacija je relativno jasna – ako u delu programskog koda nema mogućnosti da se pojavi izuzetak, zato što se u njemu koriste samo operacije i funkcije koje ne mogu da proizvedu izuzetak, onda nema ni razloga da se u njemu staramo o izuzecima. Ako znamo da staranje o izuzecima ima cenu, onda možemo da probamo da uštedimo tako što ćemo da izbegnemo taj deo posla.

Problemi sa ovako optimizovanim kodom nastaju pri njegovom menjanju. Ako označimo da funkcija ne proizvodi izuzetke, a pri njenom menjanju iskoristimo neku

operaciju ili funkciju koja može da proizvede izuzetak, onda ćemo imati posledice i po pouzdanost i po efikasnost programa.

Ako nastupi izuzetak, a nismo u stanju da ga obradimo, biće prekinut rad programa. Ipak, ako smo sigurni da upotrebljene operacije, koje načelno mogu da izazovu izuzetke, u konkretnom slučaju koristimo tako da ih one sigurno neće izazvati, onda nas prethodni problem ne brine mnogo. Bar ne u trenutku pisanja, ali mogu da nastupe problemu kada dođe do menjanja programa.

Problem je u tome što prevodioci često ugrađuju poseban deo koda za obezbeđivanje prelaza iz potprograma koji rade sa izuzecima u one koji ne rade sa njima, tako da u nekim slučajevima rezultat može da bude *pad performansi*.

Ovu „optimizaciju" je važnije razumeti kao uputstvo da se izuzeci ne koriste za stvari za koje nisu namenjeni, na primer za upravljenje kontrolom toka programa, tj. kao „napredni" oblik naredbe *GO-TO*. Obrada izuzetaka je složena i relativno spora, što nam nije posebno važno u slučejevima za koje su namenjeni (prepoznavanje i obrađivanje neočekivanih specijalnih slučajeva, problema i grešaka), ali je veoma skupo za druge vidove upotrebe.

Upotreba constexpr izraza i uslova

Od verzije C++11 dodata je ključna reč `constexpr`, koja služi da se deklarišu promenljive, izrazi i uslovi koji mogu da se jednokratno izračunavaju u fazi prevođenja programa. Ova tehnika je preporučljivija nego upotreba makroa, zato što je čistija i lakša za debagovanje.

Naravno postoje određena ograničenja, prvenstveno u odnosu na tip vrednosti izraza ili funkcije. Ukratko (ali ne i sasvim precizno), `constexpr` može da se koristi za sve tipove koji imaju trivijalnu inicijalizaciju kopije i deinicijalizaciju.

Upotreba meta-programiranja

Primenom šablona i parametarskog polimorfizma smo se bavili u poglavlju *10 - x...Polimorfizam.* Parametarski polimorfizam je vid statičkog polimorfizma, koji se u potpunosti razrešava u fazi prevođenja, pa zato primena šablona funkcija i klasa može da se veoma dobro optimizuje.

Poseban aspekt šablona je *meta-programiranje*, odnosno pisanje programskog koda koji se izračunava u fazi prevođenja programa. To je značajno uopštenje u odnosu na `constexpr` izraze. Na taj način neki aspekti izračunavanja mogu da se ubrzaju zato što se neće ponavljati svaki put pri izvršavanju programa već samo jedanput pri njegovom prevođenju. Meta-programiranje može da se upotrebi za pravljenje različitih tablica, uslova i slično. Meta-programiranje pomoću šablona je Tjuring-kompletno, tj. sve što može da se izračuna u C++-u, načelno može da se izračuna i primenom meta-programiranja u fazi prevođenja. U praksi nije sve baš tako lepo,

zato što prevodioci rade relativno neefikasno sa meta-programima, a i uvode različita ograničenja.

Radi se o relativno složenoj tehnici za čije opisivanje nemamo dovoljno prostora. Zainteresovanima preporučujemo da pročitaju neku od knjiga [Alex2001, Abra2004].

Zamenjivanje dinamičkog vezivanja statičkim

Upotreba hijerarhijskog polimorfizma je neposredno vezana za dinamičko vezivanje metoda. Odlučivanje o verziji metoda koji će se u nekom trenutku pozvati zavisi od klase konkretnog objekta na kome je metod pozvan, pa zato mora da se odvija u fazi izvršavanja programa. Programski jezik C++ je jedan od retkih OOPJ koji podržava i statičko i dinamičko vezivanje metoda. Štaviše, podrazumevano vezivanje je statičko, a metode koji želimo da se vezuju dinamički moramo eksplicitno da označavamo pomoću ključne reči `virtual`.

Dinamičko vezivanje metoda je sporije. Za složenije metode ta razlika u brzini obično nije značajna, ali za jednostavnije metode može da bude veoma važna. Na efikasnost pozivanja statički vezanih metoda najviše utiču mogućnost bolje predikcije skokova i činjenica da jednostavni statički metodi mogu da budu umetnuti na mestu pozivanja (engl. *inline*). Dinamički metodi ne smeju da se umeću, zato što u fazi prevođenja još uvek ne znamo koja verzija metoda će biti potrebna u kom slučaju.

U slučaju kada je neki metod deklarisan kao dinamički (tj. kao virtualan), a zapravo ne postoji više verzija tog metoda (tj. nije izmenjeno njegovo ponašanje u izvedenim klasama), onda ima smisla da se vezivanje tog metoda promeni tako da ne bude dinamičko nego statičko. Ovakva optimizacija posebno ima smisla ako se radi o jednostavnom metodu koji bi mogao da bude umetnut.

Ako se pri nekom menjanju programa naknadno pojavi izvedena klasa koja bi trebalo da ima izmenjeno ponašanje statički vezanog metoda, obično je relativno jednostavno da se metod (ponovo) proglasi za virtualan i da se tako obezbedi njegovo dinamičko vezivanje.

Implementirati operatore alokacije i dealokacije

Kada se dinamički alociraju i dealociraju objekti u programskom jeziku C++, upotrebljavaju se operatori `new` i `delete`. Svaka klasa može da ima svoje verzije ovih operatora. Ako oni nisu eksplicitno napisani, onda se koriste globalne univerzalne varijante. Globalne univerzalne operacije se koriste i pri alokaciji i dealokaciji nizova, a možemo i sami eksplicitno da ih upotrebimo, ako ih referišemo sa `::new` i `::delete`.

Ako imamo neku klasu čiji se objekti stalno iznova dinamički prave i brišu, onda može da bude korisno da implementiramo odgovarajuće verzije metoda `new` i `delete` u toj klasi. To je posebno korisno ako se radi o malim objektima, zato što

globalni ugrađeni operatori rade za sve različite veličine objekata, pa zbog svoje univerzalnosti nisu posebno optimizovani za neke specifične namene. Optimizovanjem ovih metoda za našu klasu, možemo da značajno podignemo performanse pravljenja i brisanja objekata. Dodatna korist može da bude eventualno ostvarivanje lokalnosti takvih objekata u prostoru.

12.5.5 Optimizacije u hodu

Programski jezik C++ i njegova standardna biblioteka su posebno posvećeni pisanju efikasnih programa. I jezik i biblioteka su definisani i i dalje se unapređuju imajući u vidu efikasnost kao jedan od najvažnijih ciljeva. Posledica takvog pristupa je da i proces programiranja na C++-u počiva na značajnom broju tehnika koje se odnose na staranje o performansama. Pridržavanje takvih tehnika u toku pisanja programa, kao vid *optimizacije unapred*, je uobičajeno među programerima koji pišu na C++-u.

Pri predstavljaju specifičnih tehnika optimizacije za C++ smo nagovestili, a ponegde i eksplicitno naglasili da se neke tehnike koriste ne samo kao tehnike optimizacije već i kao uobičajene tehnike pri pisanju programa. I neke opšte tehnike optimizacije se upotrebljavaju na sličan način.

Takve tehnike se ponekad nazivaju *optimizacijama u hodu*. U prethodnim odeljcima smo već istakli neke od njih pa im se nećemo ponovo posvećivati:

- upotreba umetnutih funkcija i metoda;
- izbegavanje globalnih promenljivih;
- lokalnost upotrebe podataka;
- lokalnost upotrebe programskog koda;
- upotreba standardne biblioteke;
- prenošenje objekata po referenci;
- definisanje privremenih promenljivih što dublje u kodu;
- upotreba liste inicijalizacija;
- upotreba konstrukcije objekata a ne dodeljivanja;
- upotreba constexp izraza i uslova i
- upotreba meta-programiranja.

12.5.6 Prepuštanje optimizacije prevodiocu

Savremeni prevodioci mogu da se pohvale automatizacijom velikog broja različitih optimizacija niskog nivoa. Konkretne raspoložive tehnike zavise od verzije

prevodica. Prevodioci obično nude i preporučene pakete optimizacija, koji obuhvataju one optimizacije za koje se procenjuje da ostvaruju najveći doprinos.

Pri primeni optimizacija na nivou prevodilaca moramo da imamo u vidu da se tehnike optimizacije stalno unapređuju i inoviraju, pa se zbog toga u njihovoj implementaciji relativno često pronalaze bagovi. Preporučeni paketi opcija su uglavnom bezbedni za upotrebu, zato što se sastoje od optimizacije koje su relativno dobro testirane, pa je verovatnoća da sadrže bagove uglavnom relativno niska. Sa druge strane, uključivanje dodatnih pojedinačnih optimizacija zahteva posebnu pažnju.

Pregled konkretnih opcija prevodilaca izlazi van okvira ove knjige. Dokumentacija prevodilaca sadrži detaljan opis svih opcija i podržanih optimizacija.

12.5.7 Česte greške

Pregled tehnika optimizacije ćemo da dovršimo podsećanjem na neke od grešaka koje se često prave pri optimizovanju softvera. Većina ovih grešaka je u nekom obliku već pomenuta u ovom poglavlju, ali zbog njihovog velikog značaja ne smemo da propustimo priliku da im se vratimo još jedanput.

Pogrešne pretpostavke

Nikada ne smemo da optimizujemo na osnovu pretpostavke da je „A efikasnije nego B". Da budemo do kraja precizni, možemo da pokušamo i trebalo bi da pokušamo, ali svakako uvek moramo da proverimo da li je promena donela povećanje performansi ili nije. Slično kao i u slučaju debagovanja, ako promena nije donela ono što smo od nje očekivali, onda bi trebalo da se poništi. Ne smemo da kvarimo dizajn programa zato što nešto *možda* radi efikasnije. Ako nekada odlučimo da pravimo kompromis sa dobrim dizajnom, onda bar moramo da budemo sigurni da od toga imamo i neku korist, a ne samo štetu.

Kao što razvijaoci softvera teže da ubrzaju konkretan softver na kome rade, tako se projektanti procesora i razvijaoci prevodilaca trude da obezbede bolje tehnike implementacije, koje više doprinose performansama. Pri tome se i pri projektovanju procesora i pri razvoju prevodilaca, najviše pažnje posvećuje onim elementima za koje se pokazalo da su relativno neefikasni. Sa novim verzijama procesora nam dolaze i bolje performanse, a sa novim verzijama prevodilaca nam dolaze napredniji optimizatori.

Šta se dešava ako mi optimizujemo program, a u međuvremenu se procesori ili prevodioci unaprede tako da se podignu performanse neoptimizovanog programa? Pa, dešava se, i to ne sasvim retko, da neoptimizovani kod postane efikasniji od optimizovanog. Optimizacija softvera je stalna igra između lepo i dobro strukturiranog programa i različitih „trikova" koji kvare dizajn ali koriste

specifičnosti prevodilaca ili arhitekture računara da bi se neki posao uradio brže. Ali kada se te specifičnosti promene, onda se promene i posledice odgovarajućih trikova.

Pre nekoliko decenija se čak moglo okarakterisati kao neprofesionalno ponašanje ako bi neko u kodu napisao `a*8`, a ne `a<<3`, a danas o takvim stvarima praktično više nema razloga da razmišljamo. Moramo da budemo spremni da nešto što je juče bila optimizacija danas može da bude suvišno kvarenje koda, zato što više ne doprinosi performansama. Štaviše, u nekim slučajevima može da dođe i do pada efikasnosti, zato što dodatnim komplikovanjem otežavamo prevodiocu ili procesoru da iskoriste svoje tehnike optimizacije.

Možda bismo mogli da zamislimo neku idealnu budućnost u kojoj optimizovanje nije potrebno, već procesori i prevodioci sve što smo napisali sami modifikuju tako da i dalje radi tačno, ali najefikasnije što je moguće? Izvesno je da ćemo se u narednim godinama (ili pre decenijama) postepeno približavati takvom idealu, ali je jednako izvesno da će još dugo biti potrebe za dodatnim manuelnim optimizacijama.

Smanjivanje koda nije uvek i njegovo optimizovanje

Iako intuicija može da nas navede da pomislimo da je manji kod istovremeno i efikasniji, na primeru razmotavanja petlji smo videli da nije uvek tako. Umesto da mnogo puta prolazimo kroz petlju i ponavljamo proveravanje uslova i skakanje na početak, videli smo da je nekada efikasnije da se telo petlje eksplicitno napiše više puta i da se smanji broj ponavljanja.

Ali kako se to slaže sa keširanjem koda? Zar ne bi trebalo da dovede do usporenja? Ispada da treba imati dobru meru. Ako preteramo pri razmotavanju petlji, lako može da se dogodi da telo petlje preraste veličinu keš-memorije i da efikasnost počne da pada.

Kada kažemo da je cilj optimizacije rasterećenje procesora, to ne znači da je dovoljno da se smanji broj instrukcija koje procesor mora da izvrši. Procesor je složena kolekcija resursa i rasterećenje procesora često znači dobro upravljanje svim tim resursima, a ne puko smanjivanje broja instrukcija:

- da, bolje je da imamo manje instrukcija;
- ali neke instrukcije su *mnogo* skuplje od drugih;
- skokovi (granjanja, ponavljanja) su među najskupljim operacijama;
- čitanje instrukcije iz memorije je mnogo skuplje nego čitanje iz keša;
- izvršavanje uzastopnih instrukcija na međusobno nezavisnim podacima u nekim slučajevima može da se odvija istovremeno;

Vidimo da smo od vrlo jednostavnih pretpostavki, koje je lako primeniti na pisanje programa, došli do nekih čija primena nije tako jednostavna. Na osnovu toga

možemo da uopštimo zaključak ovog odeljka – pojednostavljivanje zaključivanja o optimizovanju često može da nas odvede na pogrešnu stranu. Optimizovanje softvera je složeno i ta složenost ne sme da se zanemaruje. Smanjivanje koda je samo jedan od primera.

Optimizovanje tokom inicijalnog kodiranja

Istakli smo da preuranjena optimizacija može da donese mnogo više problema nego koristi. A onda smo opisali tehnike optimizacije koje se u hodu preduzimaju pri programiranju u C++-u. Zar to nije protivrečno?

Optimizacija i pisanje dobro dizajniranog programskog koda su u nekim jezicima jasno i drastično razdvojeni, ali u nekim jezicima (pre svih C i C++) nisu, zato što je jedan od osnovnih principa programiranja u tim jezicima upravo pisanje efikasnih programa.

Ako pogledamo tehnike za koje smo istakli da mogu da se primenjuju već pri pisanju koda, možemo da primetimo da te tehnike uglavnom ne kvare dizajn. Naprotiv, u njima se koriste specifičnosti programskog jezika koje omogućavaju da se zadrži čist i jasan dizajn, a da pri tome kod bude efikasan (na primer, liste inicijalizacija, prenošenje podataka po referenci, umetnute funkcije, `constexpr`) ili tehnike koje čak doprinose preglednosti i razumljivosti programa (na primer, upotreba lokalnih promenljivih, definisanje privremenih promenljivih što bliže mestu upotrebe, upotreba standarne biblioteke, konstrukcija umesto dodeljivanja).

Ako nam je cilj da pišemo efikasne programe, onda ćemo programski kod *odmah* da pišemo tako da bude efikasan. Ali pri tome bi pricipi agilnog razvoja i težnja dobrom dizajnu ipak trebalo da nam budu u prvom planu. Tokom inicijalnog kodiranja je prihvatljivo, pa čak i poželjno da se koriste tehnike pisanja efikasnog koda, koje ne kvare dizajn.

Sa druge strane, upotreba tehnika koje kvare dizajn i otežavaju dalji razvoj i održavanje programa bi trebalo da se zaobilazi u širokom luku sve do dostizanja funkcionalnosti doftvera, zato što predstavlja aspekt preuranjenog optimizovanja.

Preuranjena ili preterana optimizacija

Preuranjena optimizacija je optimizacija preduzeta pre nego što je ustanovljeno šta je potrebno da se optimizuje.

Kao što je Knut napisao, „preuranjena optimizacija je osnov svakog zla“. Ona se sukobljava sa principom agilnog razvoja „neće biti potrebno“. Ako taj princip agilnog razvoja prevedemo na domen optimizacije, onda dobijamo preporuku „optimizuj kasnije“. Sve dok možemo da odlažemo optimizaciju, to znači da ona nije neophodna i da je dobro da je i dalje odlažemo.

Preterana optimizacija je optimizacija koja je obavljana dublje nego što je potrebno. Nastaje ako nismo postavili jasne ciljeve optimizacije i kriterijume performansi.

Optimizovanje programskog koda je vrlo izazovan posao. Kada se u njega jednom uđe, onda može vrlo lako da proguta naše resurse kao neka programerska crna rupa. Veoma često se dešava da se nepotrebno gubi vreme na optimizacije koje nisu neophodne, umesto da se to vreme posveti nekom drugom, korisnijem problemu.

Preuranjena i preterana optimizacija su veoma česte greške u razvoju softvera. Nastaju kao posledica nepridržavanja principa savremene prakse optimizovanja softvera. Predstavljaju neposrednu posledicu propuštanja da se pažljivo lokalizuje predmet optimizovanja i odmere neophodne performanse, a posredno mogu da budu i posledica lošeg procenjivanja potrebnih performansi.

Posvećivanje više pažnje performansama nego korektnosti

Nekada efikasnost softvera koji pišemo spada u osnovne zahteve koji su nam postavljeni. U takvim slučajevima je posebno velika opasnost od preuranjene optimizacije. Ako trpimo pritisak da se staramo o efikasnosti (bilo u obliku pozitivne motivacije da ispunimo zahteve ili stalnog insistiranja nekog od naručilaca ili rukovodilaca), onda možemo da dođemo u iskušenje da već pri planiranju i implementaciji softvera posvetimo mnogo više pažnje performansama nego što je poželjno. Posledica takvog rada može da bude stavljanje ispravnosti softvera u drugi plan, a zatim i vrlo verovatno preduzimanje preuranjene ili preterane optimizacije.

To je nedopustivo. Koliko god da su nam važne performanse, ispravnost mora uvek da nam bude važnija. Čak i kada je program neefikasan, on će možda biti upotrebljiv za neke manje skupove podataka, ali ako je neispravan, onda je svaka njegova upotreba rizična, zato što ne znamo da li je rezultat ispravan ili ne.

Neispravno merenje performansi

Ako se merenje performansi softvera sprovodi u uslovima koji nisu identični uslovima njegove produkcione upotrebe, onda postoji opasnost da se merenjem dobiju neispravni rezultati, na osnovu kojih kasnije mogu da se donesu neke pogrešne odluke.

Jedna od čestih grešaka pri merenju performansi je merenje u izolovanom okruženju. Ako posmatramo izolovan deo koda (na primer jednu funkciju) i ustanovimo da je jedna implementacija efikasnija nego druga, to ne mora da važi kada se taj deo koda ugradi u veću celinu.

Na primer, neka imamo dve funkcije `f1` i `f2` koje su funkcionalno ekvivalentne, ali su različito implementirane. Ako merenje efikasnosti pokaže da je `f1` efikasnija od `f2`, da li to znači da je svakako bolje da koristimo `f1` u našem softveru? Ne. To može da nam ukaže da je vrlo verovatno da je bolje da koristimo `f1`, ali ne i da je to sigurno. Postoje mnoge situacije u kojima može da se pokaže da je efikasnije koristiti `f2`, na primer:

- ako `f2` može da se umetne a `f1` ne može;
- ako se pored `f1`/`f2` poziva i neka funkcija `g` koja se poziva iz `f2` a ne iz `f1`, pa njeno keširanje doprinosi efikasnosti;
- ako se iz `f1`/`f2` pozivaju dinamički vezani metodi, onda konkretne instance klasa koje se koriste mogu da značajno utiču na merenje;
- i drugo.

Druga česta greška je da se efekti neke optimizacije mere u odnosu na neku operaciju koja nema istu složenost kao operacija na koju se ta optimizacija na kraju zaista primenjuje. Na primer, ako merimo efikasnost razmotavanja neke petlje, rezultati mogu drastično da se promene ako se neka od operacija u petlji zameni složenijom operacijom. Prva posledica je da efekat ubrzavanja može da se izgubi, zato što relativna cena ponavljanja (proveravanja uslova i grananja) postaje značajno manja nego što je cena tela petlje. Druga posledica je da čak može da dođe i do značajnog usporavanja, zato što je telo kratke petlje moglo da se dobro kešira, a nakon razmotavanja to više nije slučaj.

Merenje performansi je presudno za ocenu uspešnosti optimizacije. Već smo istakli da ne smemo da se oslanjamo na pretpostavke da je nešto efikasnije već da to mora da se proveri u praksi. Međutim, veoma je lako da se pogreši pri merenju performansi. Zato je neophodno da se merenje performansi preduzima u realnim uslovima, koji su praktično isti kao uslovi produkcione primene softvera. Svako drugačije merenje predstavlja samo procenu, a ne i pouzdanu meru efikasnosti.

Optimizovanje verzije za debagovanje

Optimizovanje verzije za debagovanje je specifičan slučaj neispravnog merenja performansi.

Kada se program prevede „za debagovanje“, onda prevodilac po pravilu ne primenjuje brojne interne optimizacije, već ostavlja prevod programa na mašinski kod u obliku koji je pregledniji i lakši za čitanje i analiziranje. Takav prevod po pravilu ima ugrađene i dodatne elemente koji olakšavaju debagovanje i praćenje toka izvršavanja programa ili menjanja njegovog stanja. Sa druge strane, kada se program prevede „za produkciju“, onda se iz prevoda sklanjaju svi viškovi i mašinski kod se dodatno optimizuje.

Zbog toga je uobičajeno da se program preveden (i izgrađen) za debagovanje izvršava daleko sporije nego program koji je preveden (i izgrađen) za produkciju. To nas dovodi u situaciju da pravimo različite greške pri optimizovanju, zbog toga što dobijamo pogrešne rezultate merenja performansi.

Zbog velike razlike u brzini (pa čak i u zauzeću radne memorije) različito prevedenih verzija programa, može da se desi da ocenimo da program ne radi

dovoljno efikasno, a da on zapravo radi odlično kada se prevede za produkciju. U takvim slučajevima se preduzima nepotrebno duboka optimizacija, zato što se insistira na nepotrebno visokim performansama verzije za debagovanje.

Drugi problem je što verzija za debagovanje i verzija za produkciju mogu da imaju potpuno različita uska grla. Ako procenjujemo usko grlo na osnovu verzije za debagovanje, onda možemo da odredimo pogrešan predmet optimizacije i da zatim nepotrebno gubimo vreme na optimizovanje pogrešno odabranog dela softvera.

12.6 Profajleri

Profajleri (engl. *profiler*) su programi koji prate i analiziraju upotrebu resursa tokom rada programa. Osnovna namena profajlera je da olakšaju uočavanje delova programa koji predstavljaju usko grlo u pogledu zauzeća nekog od resursa. Imaju nezamenljivu ulogu pri prepoznavanju predmeta optimizacije, pa predstavljaju jedan od najvažnijih alata koje koristimo pri optimizovanju softvera.

Prema vrsti resursa koji se prati, profajleri mogu da se podele na procesorske profajlere, memorijske profajlere i drugo. Pri optimizacijama niskog nivoa se najčešće koriste procesorski profajleri, pa se često pod nazivom profajler podrazumeva da se radi o procesorskom profajleru. I ovde ćemo se u daljem tekstu baviti prvenstveno procesorskim profajlerima. Pored njih se često koriste i memorisjki profajleri. Pri merenju performansi složenijih aplikacija koriste se profajleri koji mere zauzeće mreže, diskova i drugih resursa.

Procesorski profajleri prate zauzeće procesora i procesorskih resursa. Neki od osnovnih parametara koji se prate su brojanje koliko puta je izvršena pojedina funkcija ili metod, merenje vremena provedenog u funkciji ili metodu, merenje upotrebe keš memorije, brojanje promašaja stranice virtualne memorije i drugo. Prema tome da li mere vreme provedeno u funkcijama i metodima ili vreme provedeno u određenim linijama programskog koda, profajleri mogu da budu funkcijski ili linijski. Dobri savremeni profajleri obično omogućavaju da se prati i jedno i drugo.

Prema načinu na koji prikupljaju informacije o izvršavanju programa, procesorske profajlere delimo na (1) profajlere zasnovane na praćenju događaja, (2) profajlere sa ugradnjom brojača i (3) statističke profajlere.

Profajleri zasnovani na praćenjenju događaja (engl. *event-based*) prate i evidentiraju različite događaje o kojima ih obaveštava izvršno okruženje ili operativni sistem tokom izvršavanja programa. Na ovaj način mogu da se prate različiti sistemski događaji ili resursi, kao što su operacije sa kešom ili virtualnom memorijom, različiti hardverski događaji koje proizvode procesori, kao i specifični događaji koje proizvode izvršna okruženja. Na primer, profajleri `perf` i *OProfile* za *Linux* su zasnovani na praćenju događaja o kojima ih izveštava jezgro operativnog

sistema [Melo2010]. Ovakav način rada je uobičajen i za profajlere za interpretirane programske jezike ili virtualne mašine, kao što su *JVM* ili *CLR*.

Drugi način prikupljanja informacija je ugradnja brojača u sam programski kod (engl. *instrumentation-based*) [Graham1982]. Na početku i na kraju svakog potprograma se ugrađuje programski kod koji obaveštava profajler o ulasku i izlasku iz potprograma. Ovakvi profajleri imaju visoku, čak apsolutnu tačnost brojanja događaja. Posledica je da se profajleru veoma često dostavlja veoma veliki broj i obim informacija, što zahteva dosta vremena, pa se pod ovakvim profajlerima programi najčešće izvršavaju značajno sporije. Samim tim, merenje vremena može da izgubi na tačnosti, pa čak može da opadne i tačnost merenja relativnog vremena provedenog u određenim delovima koda.

Savremeni procesorski profajleri za kompilirane jezike najčešće rade pomoću povremenog uzimanja uzoraka (engl. *sampling-based*) [Arnold2000]. Nazivaju se i statističkim profajlerima. Pomoću sistema prekida ili pomoću zamki procesora, rad procesora se prekida u otprilike jednakim vremenskim intervalima i profajler se obaveštava o tome koja se mašinska naredba trenutno izvršava i u kom kontekstu. Uobičajeno je da se program prekida 100-10.000 puta u sekundi, zavisno od mogućnosti procesora i profajlera. Svaki put kada se izvršavanje programa zaustavi, beleži se vrednost brojača instrukcija, ali i stanje steka, da bi moglo da se ustanovi kako se do tog mesta u programu došlo. Zbog beleženja stanja steka, pojedinačna obrada je složenija nego u slučaju ugradnje u kod, ali se evidentira daleko manji broj događaja, pa ova vrsta profajlera nešto manje usporava izvršavanje programa. Posledica ovakvog rada je da brojanje upotrebe potprograma nije sasvim tačno, ali se ispostavlja da je relativno merenje vremena čak i nešto tačnije nego u slučaju ugradnje brojača.

Za interpretirane programske jezike, profajleri se obično prave kao dodatni moduli, koji se pri pokretanju programa povezuju sa izvršnim okruženjem i prate zauzeće resursa. Na primer, za interpretator za Pajton postoji više profajlerskih modula, među kojima su `cProfile` (funkcijski procesorski profajler zasnovan na praćenju događaja), `line_profiler`, (linijski procesorski profajler zasnovan na praćenju događaja), `statprof` (funkcijski statističi procesorski profajler) i drugi.

U slučaju programskih jezika koji se prevode, profajleri se obično prave kao posebni programi. Oni pri izvršavanju nemaju neposrednog dodira sa prevodiocem, osim u slučaju kada se koriste brojači ugrađeni u programski kod. U tom slučaju programi moraju da budu prevedeni na odgovarajući način i profajler mora da bude kompatibilan sa konkretnom tehnikom ugradnje brojača koju koristi prevodilac. Na primer, *GNU* profajler `gprof` je procesorski profajler zasnovan na ugrađenim brojačima [Graham1982]), a *Microsoft* profajler integrisan u vizualno razvojno okruženje *Visual Studio* je procesorski statistički profajler. Slični profajleri postoje i za neke interpretirane jezike, na primer `py-spy` ili `vprof` za Pajton.

Za profajliranje može da se koristi i *Valgrind*. Iako je izvorno namenjen za druge stvari, njegov modul `cachegrind` je nadgrađen modulom `callgrind`, koji omogućava praćenje pozivanja funkcija i može da se upotrebi za profajliranje. Program `kcachegrind` je koristan vizualni alat za analizu izveštaja koje prave moduli `cachegrind` i `callgrind`.

Primer optimizacije pomoću profajlera gprof

Upotrebu profajlera ćemo ilustrovati na jednostavnom primeru programa `primer.cpp`, koji popunjava matricu jedinicama i zatim je sumira:

```
#include <iostream>
using namespace std;

int constexpr len = 10000;
int niz[len][len];

void init()
{
    for( int i=0; i<len; i++ )
        for( int j=0; j<len; j++ )
            if( niz[i][j] != 1 )
                niz[i][j] = 1;
}

int sumline( int n )
{
    int sum = 0;
    for( int j=0; j<len; j++ )
        sum += niz[j][n];
    return sum;
}

int sum()
{
    int sum = 0;
    for( int i=0; i<len; i++ )
        sum += sumline(i);
    return sum;
}

int main()
{
    init();
    cout << sum() << endl;
    return 0;
}
```

Ovaj primer je sasvim jednostavan i trebalo bi da je očigledno gde su problemi. Poslužiće nam da ilustrujemo upotrebu profajlera. Za merenje performansi ćemo da upotrebimo profajler `gprof`, koji počiva na ugradnji brojača u programski kod. Pri

prevođenju programa je potrebno je da se uključe odgovarajuće opcije, čime će se prevodiocu reći da u svaku funkciju ugradi odgovarajući kod za merenje broja prolazaka kroz funkciju i vremena provedenog u njoj. Postoji više opcija, ali osnovne su:

- `-pg`, u prevod se automatski ugrađuju brojači za profjaliranje pomoću profajlera `gprof`;
- `-g`, prevod uključuje informacije za debagovanje, što može da bude potrebno ako se performanse mere na nivou linija koda, a ne samo na nivou potprograma.

Ako se samo neki moduli prevode sa opcijama za profajliranje, evidentiranje informacija će i dalje raditi ali samo za odgovarajuće delove programa, tj. neće biti na raspolaganju informacije za ceo program. Iste opcije moraju da se koriste i pri povezivanju.

Naš primer ćemo da prevedemo sa opisanim opcijama za profajliranje i paketom opcija za optimizaciju `-O3`. Dodatnom opcijom `-fno-inline-small-functions` ćemo zahtevati da se ne ugrađuju jednostavne funkcije, zato što bi inače ceo naš program stao u jednu funkciju:

```
g++ -g -pg -O3 -fno-inline-small-functions primer.cpp -o primer
```

Sledeći korak je pokretanje programa. Program se pokreće i izvršava na sličan način kao da je preveden bez dodatnih opcija, ali će zbog uključenih opcija za profajliranje napraviti datoteku `gmon.out`, u kojoj će zapisati prikupljene podatke o toku izvršavanja programa:

```
$ ./primer
100000000
```

Sada nam preostaje da pokrenemo program `gprof`. Kao parametar se navodi program koji se profajlira, da bi iz njega mogli da se prikupe podaci o programskom kodu. Opciono se navodi i naziv datoteke sa podacima o izvršavanju, a ako se ne navede onda se podrazumeva da se koristi `gmon.out`. Navešćemo i dodatni parametar `-b`, da ne bi bila ispisivana legenda o podacima. Kao rezultat ćemo dobiti sledeći izveštaj:

```
$ gprof -b ./primer
Flat profile:

Each sample counts as 0.01 seconds.
```

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 87.07      1.28     1.28    10000     0.00     0.00  sumline(int)
 12.93      1.47     0.19        1     0.19     0.19  init()
  0.00      1.47     0.00        1     0.00     1.28  sum()

            Call graph

granularity: each sample hit covers 4 byte(s) for 0.68% of 1.47
seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    1.47                 main [1]
                0.00    1.28       1/1           sum() [3]
                0.19    0.00       1/1           init() [4]
-----------------------------------------------
                1.28    0.00   10000/10000       sum() [3]
[2]     87.1    1.28    0.00   10000         sumline(int) [2]
-----------------------------------------------
                0.00    1.28       1/1           main [1]
[3]     87.1    0.00    1.28       1         sum() [3]
                1.28    0.00   10000/10000       sumline(int) [2]
-----------------------------------------------
                0.19    0.00       1/1           main [1]
[4]     12.9    0.19    0.00       1         init() [4]
-----------------------------------------------

Index by function name

   [3] sum()                   [4] init()                  [2]
sumline(int)
```

Prvi deo izveštaja obuhvata podatke o funkcijama čije je izvršavanje zahtevalo najviše vremena. Vidimo da je ukupno trajanje izvršavanja programa bilo 1,47*s* i da je najviše vremena utrošeno u funkciji `sumline`, koja je pozvana 10.000 puta.

U drugom delu izveštaja se vidi koja je funkcija odakle pozivana i koliko je vremena utrošeno u kom njenom delu. Svaka sekcija opisuje po jednu funkciju:

- U prvoj sekciji se vidi da je funkcija `main` pozvana od strane sistema („*<spontaneous>*“) i da je ona utrošila 100% vremena, tj. 1,47*s*. Međutim, vidimo i da je od toga 1,28*s* utrošeno pri pozivanju funkcije `sum`, a još 0,19*s* pri pozivanju funkcije `init`;
- Funkcija `sumline` je pozvana ukupno 10.000 puta i to svaki put iz funkcije `sum`. Pri tom je utrošila 1,28*s*, što je 87% ukupnog vremena i sve je to utrošeno u njenom telu, tj. nije pozivala druge funkcije;
- Funkcija `sum` je pozvana jednaput iz funkcije `main` i potrošila je 1,28*s*, ali vidimo da je sve to utrošeno pri 10.000 pozivanja funkcije `sumline`;

- Funkcija `init` je pozvana jedanput iz funkcije `main` i potrošila je 0,19*s* ili 12,9% vremena.

Treći deo izveštaja služi kao indeks, za lakše snalaženje u drugom delu izveštaja. Veoma je koristan u slučaju većih programa sa velikim brojem funkcija, zato što tada drugi deo izveštaja može da bude prilično obiman.

Analizom dobijenih podataka vidimo da je najviše vremena uzelo sumiranje, što je i očekivano. Znači, predmet eventualne optimizacije bi trebalo da bude, pre svega, funkcija `sumline`. Lako se uočava da je uzrok neefikasnosti ove funkcije u tome što ne poštuje princip lokalnosti. Znači, možemo da probamo da optimizujemo program tako što bismo elemente sumirali red po red, a ne vrstu po vrstu:

```
int sumline( int n )
{
    int sum = 0;
    for( int j=0; j<len; j++ )
        sum += niz[n][j];
    return sum;
}
```

Kada ponovimo prevođenje, izvršavanje i pokretanje profajlera, dobićemo izmenjene podatke:

```
Flat profile:
...
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 86.96      0.20     0.20        1   200.00   200.00  init()
 13.04      0.23     0.03    10000     0.00     0.00  sumline(int)
  0.00      0.23     0.00        1     0.00    30.00  sum()

            Call graph
...
-----------------------------------------------
                0.20    0.00       1/1           main [1]
[2]     87.0    0.20    0.00       1          init() [2]
-----------------------------------------------
...
```

Vidimo da je sada funkcija `sumline` daleko efikasnija i da troši samo 0,03*s* umesto 1,28*s*, što je oko 42 puta manje vremena. Ukupno izvršavanje programa je sada svedeno na 0,23*s* umesto početnih 1,47*s*, pa je program ubrzan za oko 82%.

Sada je najneefikasnija funkcija `init`, na koju odlazi skoro 7 puta više vremena nego na sumiranje. Potencijalna optimizacija je da iz nje izbacimo proveru uslova, tako da se vrednosti matrice bezuslovno postavljaju na 1:

```
void init()
{
    for( int i=0; i<len; i++ )
        for( int j=0; j<len; j++ )
            niz[i][j] = 1;
}
```

Kada ponovimo prevođenje, izvršavanje i pokretanje profajlera, dobijamo sledeće podatke:

```
Flat profile:
...
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 75.00      0.09     0.09        1    90.00    90.00  init()
 25.00      0.12     0.03    10000     0.00     0.00  sumline(int)
  0.00      0.12     0.00        1     0.00    30.00  sum()
...
```

Sada je funkcija `init` potrošila svega 0,09*s*, što je 55% efikasnije nego ranije. Izvršavanje programa je trajalo ukupno 0,12*s*, što je 48% manje nego ranije, a oko 92% manje nego na početku.

12.7 Umesto zaključka

Jedan od osnovnih razloga što su računari postali sveprisutni je njihova visoka efikasnost. Mi koristimo računare i pišemo programe za njih baš zbog toga što su efikasni. To onda ima za posledicu da pri pisanju programa prirodno težimo da tu efikasnost ne dovedemo u pitanje svojom lošom implementacijom, već da nasuprot tome, pokušamo da je još više istaknemo i iskoristimo. Kada god primetimo da imamo problem sa performansama, pokušavaćemo da taj problem prevaziđemo, nekada ispravljanjem eventualno napravljenih grešaka, a nekada traženjem efikasnijih načina da se uradi neki posao. To je ono što nazivamo optimizacijom softvera.

Kada se početnici prvi put susretnu sa nekom optimizacijom, ona može da im izgleda kao magija – neko dođe i promeni nešto u kodu, čak na način koji izgleda besmisleno ili trivijalno, a program posle toga počne da radi mnogo brže!? Naravno, nema tu nikakve magije. U pitanju je vrlo egzaktna oblast, koja zahteva iscrpno poznavanje alata koje koje koristimo – od programskog jezika, preko prevodioca, pa do konkretnog računarskog sistema. Posledica širine i dubine znanja, koja su neophodna za uspešno optimizovanje, doprinosi tom „magijskom“ doživljaju.

Svima koje ova oblast zanima preporučujemo da ne idu preko reda, već da prvo nauče dobro i lepo dizajniranje softvera, zatim debagovanje, pa onda konkretne alate koje koriste i da se tek onda posvete izučavanju tehnika optimizacije.

Ima mnogo izvora iz kojih se mogu izučavati principi i tehnike optimizacije. Za upoznavanje sa opštim principima optimizovanja programa u kontekstu dobrog dizajniranja i agilnog razvoja preporučujemo članak Kena Aura i Kenta Beka [Auer1996]. Za temeljnije upoznavanje sa tehnikama optimizacije na programskom jeziku C++ predlažemo [Gunt2016] ili [Fog2022]. Knjiga [Gerber2006] se bavi karakteristikama arhitekture Intelove familije procesora IA-32 i specifičnim tehnikama optimizacije. Iako je u osnovi posvećena arhitekturi IA-32, veći deo sadržaja može da se primeni i na druge savremene procesore.