VISOKA ŠKOLA STRUKOVNIH STUDIJA ZA INFORMACIONE TEHNOLOGIJE

Projekat

Tehnike i strategije unapređenja softverskog koda

Nastavnik:
Goran Radić

Student:
Cvetić Branko 65/06

Datum predaje:
22.08.2011.

SADRŽAJ

REZIME

Performanse su samo jedan aspekt ukupnog kvaliteta softvera i obično nisu ni najvažniji. Dalje, dobro optimizovani kod je samo jedan aspekt ukupnih performansi i skoro po pravilu nije i najznačajniji. Programska arhitektura, detaljni dizajn, upotrebljene strukture podataka i izbor algoritama u najvećem broju slučajeva imaju veći uticaj na performanse programa (vreme izvršavanja i zauzeće memorije) nego što to ima efikasnost njegovog programskog koda.

Kvantitativna merenja su ključni deo maksimizirananja performansi. Ona su neophodna da bi se pronašli delovi programa čija bi optimizacija performansi zaista dala rezultate. Takođe, merenja su ponovo potrebna kada treba verifikovati da su urađene optimizacije poboljšale (i to u dovoljnoj značajnoj meri) umesto da degradiraju performanse softvera. Većina aplikacija troši veliki deo vremena izvršavanja (*execution time*) u malim delovima koda. Ne možete znati koji su to delovi dok ne izvršite merenja. Kada pronađete te delove, tzv. uska grla (*hotspots*), obično je potrebno više iteracija u optimizaciji kako bi se postigla željena poboljšanja u performansama. Najbolji način da se pripremite za rad na performansama putem optimizacije koda (*code tuning*) tokom inicijalnog programiranja jeste da pišete "čist", standardni kod koji je lak za razumevanje i modifikaciju.

UVOD

Mnogi softverski inženjeri preporučuju ono što neki nazivaju "pristup odugovlačenja" optimizaciji softverskog koda. Taj pristup kaže da optimizaciju treba odugovlačiti što je moguće više, a ako je ikako moguće da optimizaciju u potpunosti zaobiđete – onda to i učinite. Prerana ili prečesta optimizacija prosto nije dobar pristup inženjerstvu. Svakako je bolje da imate program koji radi, nego da imate brz program koji pada ili daje netačne informacije. Sa druge strane, teško da ćete uspeti da napišete uspešnu aplikaciju u ovom trenutku bez bavljenja optimizacijom na nekom nivou u procesu stvaranja aplikacije. Kvalitetan kompajler može da vam pomogne, ali samo donekle. Istina je ipak, da vi kao programer daleko bolje razumete svoju aplikaciju (njenu primenu, ulazne vrednosti, uslove u kojima će raditi...) nego kompajler koji jednostavno nema takve informacije. Kao što kaže *Michael Abrash*, najbolji kompajler je onaj "između tvojih ušiju".

Postoji mnogo nivoa na kojima se može vršiti optimizacija performansi vaših aplikacija. Neke od tehnika koje ćemo obraditi bolje rade na kompajliranim jezicima, neke bolje na interpretiranim. Suština jeste u tome da kakve god optimizacije da primenite, važno je da izmerite efekat predviđenih optimizacija i da to činite redovno kako bi bili sigurni koja je optimizacija i kakve dala efekte.

STRATEGIJE ZA UNAPREĐENJE SOFTVERSKOG KODA

Kada govorimo o strategijama za unapređenje softverskog koda, moramo se prvo pozabaviti jednim, istorijski kontraverznim, pitanjem – optimizacijom tj. tjuningom performansi. Računarski resursi u pogledu raspoložive memorije i brzine procesora su bili ozbiljno ograničeni tokom 1960-tih godina i efikasnost je bila glavna briga. Kako su računarski sistemi postali moćniji tokom 1970-tih, programeri su počeli da shvataju kako je njihov fokus na performansama štetio čitljivosti i lakoći održavanja programerskog koda te je optimizacija koda (*code tuning*) tih godina zavređivalo sve manje pažnje. Povratak ograničenjima performansi sa revolucijom mikroprocesora 1980-tih je ponovo doveo efikasnost u prvi plan, koja je ponovo isčezla tokom 1990-tih. Tokom 2000-tih, ograničenja po pitanju memorije u ugrađenom (*embedded*) softveru za uređaje kao što su mobilni telefoni i PDA uređaji, kao i vreme izvršavanja interpretiranog koda su još jednom doveli do toga da efikasnost postane ključna tema.

Kada govorimo o performansama, stvari možemo gledati sa dva nivoa: strateškog i taktičkog nivoa. U ovom delu ćemo razmotriti performanse sa strateške tačke gledišta: šta su to performanse, koliko su važne i generalne principe za postizanje performansi.

PERFORMANSE

Optimizacija koda (*code tuning*) je samo jedna od metoda kojima se može postići poboljšanje performansi. Veoma često možete naći druge načine da poboljšate performanse više (i za manje vremena), a pri tome čineći manju štetu kodu, nego što biste to učinili optimizacijom koda. Ovde ćemo razmotriti neke od opcija.

KVALITATIVNE KARAKTERISTIKE I PERFORMANSE

Većina ljudi kao što smo mi (programeri) pretpostavlja da što bolje napravi programski kod, da će više naših klijenata i kupaca voleti naš softver. W.A. Wulf je rekao: „Više računarskih grehova je počinjeno u ime efikasnosti (bez da je obavezno i postignuta) nego iz bilo kog drugog razloga, uključujući i slepu glupost.“ Generalno, programeri imaju dosta iskrivljenu sliku šta je to što je najvažnije korisnicima softvera. Korinici su daleko više zainteresovani za „opipljive“ karakteristike koda od recimo kvaliteta samog koda. Korisnici su ponekad zainteresovani za sirove performanse softvera, ali samo kada to utiče na njihov rad u značajnijom meri. Oni su više zainteresovani za protokom programa (*program throughput*) nego za sirove performanse. Isporuka softvera na vreme, lak i pregledan korisnički interfejs, kao i zanemarljivo vreme prekida odnosno nedostupnosti u radu, često su daleko važniji.

Performanse su samo vezane za brzinu koda. I to do te mere da kada radite na performansama koda, zapravo ne radite na drugim kvalitativnom svojstvima softvera. Ako je neophodno da žrtvujete druge kvalitativne karakteristike da bi ste postigli bolje performanse, trebalo bi da se zabrinete. Vaš rad na brzini može da naškodi ukupnim performansama softvera, pre nego da im pomogne.

PERFORMANSE I OPTIMIZACIJA KODA

Jednom kada te izabrali efikasnost kao prioritet, bez obzira da li se ona odnosi na brzinu ili veličinu (memorijsku), trebalo bi da uzmete u obzir nekoliko opcija pre nego što se

odlučite da poboljšate brzinu ili veličinu na nivou programskog koda. Razmislite o efikasnosti iz svake od sledećih tački gledišta:

- Zahtevi softvera
- Dizajn softvera
- Dizajn klasa i rutina (funkcija)
- Interakcije sa operativnim sistemom
- Kompilacija koda
- Hardver
- Optimizacija koda (*code tuning*)

ZAHTEVI SOFTVERA

Performanse se navode kao korisnički zahtev daleko češće nego što to zaista jesu. Barry Boehm prepričava priču iz TRW inc. o sistemu koji je inicijalno imao zahtev za reakcijom ispod jedne sekunde. Ovakav zahtev je mogao biti rešen veoma kompleksnim dizajnom čija bi realizacija koštala dodatnih 100 miliona dolara. Dalje analize su utvrdile da bi korisnici bili zadovoljni i reakcijom sistema u prve 4 sekunde 90 posto vremena. Promena zahteva za reakcijom sistema smanjila je ukupne troškove za oko 70 miliona dolara. Poruka glasi: Pre nego što investirate vreme u rešavanje problema sa performansama, budite sigurni da rešavate problem koji treba da bude rešen.

DIZAJN SOFTVERA

Dizajn softvera uključuje glavne pravce dizajna za pojedinačni program, a to je uglavnom način na koji je program podeljen u manje celine, odnosno klase. Određeni dizajni softvera otežavaju pisanje koda za sisteme visokih performansi. Drugi, čine da pisanje koda za visoko-performansne sisteme bude lako i prirodno.

Pogledajmo primer iz realnog radnog okruženja programa koji se bavi sakupljanjem podataka za koji je protok merenja podataka identifikovan kao ključna osobina. Svako merenje uključuje vreme potrebno za elektronsko merenje, kalibrisanje vrednosti, skaliranje vrednosti i konverziju sa senzorskih jedinica podataka (kao što su milivolti) u inženjerske jedinice (kao što su recimo stepeni Celzijusovi).

U ovom slučaju, bez uzimanja u obzir rizika koji sa sobom nosi dizajn visokog nivoa, programeri bi se našli u situaciji da pokušavaju da optimizuju jednačine za rešavanje polinoma 13-og reda u softveru, odnosno polinoma sa 14 promenljivih uključujući i promenljive 13-og stepena. Umesto toga, oni su rešili problem sa drugačijim hardverom i dizajnom visokog nivoa koji koristi desetine polinoma 3. reda. Ovakva promena nije mogla da bude sprovedena putem prilagođavanja koda, a takođe postoji verovatnoća da bilo koja količina prilagođavanja koda ne bi mogla da reši problem. Ovo je samo jedan od primera problema koji je morao biti rešen na nivou dizajna softvera.

Ako znate da su veličina i brzina programa važni, dizajnirajte arhitekturu programa na takav način da uz razumne programerske napore možete da postignete ciljeve u pogledu veličine i brzine. Dizajnirajte arhitekturu koja je orjentisana na performance i onda postavite ciljeve u pogledu resursa za svaki pojedinačni podsistem, osobinu ili klasu. Ovo će vam pomoći na više načina:

- Postavljanje pojedinačnih ciljeva u pogleda zahtevanih resursa čini da konačne performance sistema odnosno programa budu predvidive. Ako svaka celina ispunjava

zahteve u pogledu resursa, onda će i čitav system ispuniti zahtev u pogledu resursa. Na ovaj način, možete da identifikujete podsisteme koji imaju problema sa ispunjavanjem sopstvenih ciljeva prilično rano i da ih ciljate u smislu potencijalnog redizajna ili prilagođavanja koda.
- Sama činjenica da ste postavili eksplicitne ciljeve povećava šansu da će oni biti postignuti. Programeri rade ka ispunjenju ciljeva kada znaju šta su ciljevi; što su eksplicitnije i preciznije objašnjeni, programerima je lakše da rade.

Možete da postavite i ciljeve koji ne postižu efikasnost direktno, ali promovišu efikasnost na duge staze. Efikasnost se često najbolje postiže u kontekstu drugih pitanja. Na primer, postizanje visokog nivoa mogućnosti modifikacije koda može da posluži kao odlična osnova za postizanje ciljeva u pogledu efikasnosti, nego li samo postavljanje efikasnosti za cilj. Sa visoko modularnim, promenljivim dizajnom, možete veoma lako da zamenite komponente manje efikasnosti onima koje su visoko efikasne.

DIZAJN KLASA I RUTINA

Dizajn unutrašnjosti klasa i rutina predstavlja još jednu od prilika da kreirate dizajn koji je optimizovan za performanse. Jedna od ključnih stavki koja se može iskoristiti na ovom nivou jeste izbor tipova podataka i algoritama, koji utiču kako na zauzeće memorije, tako i na samu brzinu izvršavanja programa. Ovo je nivo gde recimo možete recimo da izabrete *quicksort* umesto *bubblesort* algoritma za sortiranje ili binarnu pretragu umesto linarne.

INTERAKCIJE SA OPERATIVNIM SISTEMOM

Ako vaši programi rade sa externim fajlovima, dinamičkom memorijom ili izlaznim uređajima, velika je šansa da vrše i određene interakcije sa operativnim sistemom. Ako performanse takve aplikacije nisu zadovoljavajuće postoji velika verovatnoća da su uzrok tome rutine operativnog sistema koje su ili spore ili preglomazne. U nekim situacijama nećete biti svesni da program interreaguje sa operativnim sistemom: ponekad kompajler generiše sistemske pozive ili vaše biblioteke pozivaju sistemske rutine na takav način da to ne biste mogli ni da pretpostavite.

KOMPILACIJA KODA

Dobri kompajleri pretvaraju programski kod jezika visokog nivoa u dobro optimizovani mašinski kod. Ako izaberete odgovarajući kompajler, nećete morati da brinete o optimizacijama na niovu koda, čak šta više moguće je da će sitno doterivanje koda sa ciljem da se poboljšaju perfomanse samo usporiti program jer ćete na taj način samo zbuniti kompajler.

HARDVER

Ponekad je najjeftiniji, najčistiji i najbolji način da poboljšate performanse programa kupovina novog hardvera. Ako recimo distribuirate aplikaciju za široku upotrebu stotina ili hiljada korisnika kupovina novog hardvera nije realistična opcija. Ali, ako razvijate softver "po narudžbini" za nekoliko korisnika u jednoj kompaniji, nadogradnja hardvera je možda najjeftinija opcija. Nadogradnja hardvera štedi početno ulaganje u merenje performansi, štedi

buduće trokove održavanja, a takođe i poboljšava performance svih aplikacija koje rade na tom hardveru.

OPTIMIZACIJA KODA

Optimizacija koda je praksa odnosno posao izmene ispravnog koda na takav način da se on izvršava efikasnije. "Optimizacija" se odnosi na izmene manjeg obima koje utiču na pojedinačnu klasu, rutinu ili najčešće nekoliko linija programskog koda. Optimizacija se ne odnosi na krupnije promene u dizajnu koda ili druge vidove poboljšanja performansi visokog nivoa.

Vi možete da napravite dramatična poboljšanja na svakom od nivoa počevši od dizajna softvera pa do prilagođavanja koda. *Jon Bentey* navodi argument da je u nekim slučajevima poboljšanja na svakom od nivoa moguće pomnožiti. Kako je moguće da ostvarite poboljšanje od 10 puta na svakom od nivoa, to implicira da se potencijalno može napraviti poboljšanje od čak million puta po pitanju efikasnosti. Iako takvo umnožavanje potenijala za poboljšanje performansi zahteva aplikaciju u kojoj su dobici na performansama nezavisni na svakom nivou, što je veoma retko, takav potencijal ipak pedstavlja veliki motiv za dalji rad.

UVOD U OPTIMIZACIJU KODA (CODE TUNING)

Šta je to što je privlačno kod optimizacije koda? Optimizacija koda svakako nije najefikasniji način da se poboljšaju performanse – arhitektura programa, dizajn klasa i pravilan odabir algoritama veoma često dovode do dramatičnijih poboljšanja. Takođe, to nije ni najlakši način da se poboljšaju performanse – kupovina novog hardvera ili komplajlera sa boljom optimizacijom je daleko lakši način. A nije ni najjeftiniji način da se poboljšaju performanse – potrebno je znatno više vremena da se programski kod ručno podesi inicijalno, s tim da je takav kod kasnije i daleko teže održavati.

Međutim, podešavanje koda je privlačno iz nekoliko razloga. Jedna od stvari koje podešavanje koda čine privlačnim jeste činjenica da ono „kao da se protivi zakonima prirode". Veliko je zadovoljstvo uzeti recimo jednu rutinu koja se izvršava za 20 milisekundi, promeniti par linija koda i smanjiti vreme izvršavanja rutine na 2 milisekunde.

Ono je takođe privlačno zbog toga što ovladavanje veštinom pisanja efikasnog koda jeste siguran put ka tome da postanete ozbiljan programer. U tenisu recimo, ne dobijate nikakve poene zbog načina na koji podižete lopticu sa terena, ali ipak morate da naučite „pravilan" način da to uradite. Ne možete samo da se sagnete i dohvatite je rukom. Ako ste dobri, udarićete lopticu glavom reketa sve dok se ona ne odbije do visine kukova i tu ćete je uhvatiti, bez naginjanja. Iako se čini da ovakve stvari nisu od velikog značaja, način na koji skupljate lopticu daje određeni pečat u okviru teniskog miljea. Slično, nikoga sem vas i drugih programera neće zanimati koliko je vaš kod uredan i efikasan. Ipak, u okviru programerskog miljea, pisanje mikroefikasnog koda „dokazuje" da ste posebni. Međutim, postoji određeni problem – efikasan kod nije nužno i „bolji" kod.

PARETOV PRINCIP

Paretov princip, takođe poznat i kao pravilo 80/20, tvrdi da možete da postignete 80 procenata rezultata sa 20 procenata napora. Ovaj princip se može primeniti u mnogim oblastima sem programiranja, ali definitivno važi kada govorimo o optimizaciji programa.

Barry Boehm navodi da 20 procenata rutina u programu troši 80 procenata vremena izvršavanja (*execution time*) istog programa (1987b). U svom klasičnom radu "Empirijska studija Fortran programa", *Donald Knuth* je utvrdio da je manje od 4 procenta koda u programu odgovorno za 50 procenata vremena njegovog izvršavanja (1971).

Knuth je koristio profajler za brojanje linija koda (*line-count profiler*) da bi otkrio ovaj iznenađujući odnos, a implikacije na optimizaciju su jasne. Prilikom optimizacije trebalo bi da „merite" kod kako bi našli najkorišćenije delove (*hot spots*) i onda da svoje resurse uložite u optimizaciju tih nekoliko procenata koda koji se najviše koriste. *Knuth* je takođe profilisao i sopstveni program za brojanje linija i pronašao da on troši polovinu vremena izvršavanja u samo 2 petlje. Nakon toga je promenio samo par linija koda u okviru pomenutih petlji i duplirao brzinu sopstvene aplikacije za nepunih sat vremena.

Jon Bentley opisuje slučaj u kojem je program od 1000 linija koda trošio 80 procenata vremena izvršavanja u rutini od samo 5 linija za računanje kvadratnog korena. Prepravkom te rutine ubrzao je samu rutinu trostruko, a kompletnu aplikaciju 2 puta (1988). Paretov princip je poslužio i kao izvor za generalni savet da pišete najveći deo koda u interpretiranom jeziku kao što je Python i da nakon toga prepišete najkorišćenije delove (*hot spots*) u nekom brzom kompajliranom jeziku kao što je C. *Bentley* takođe navodi slučaj u kojem je tim programera pronašao da operativni system polovinu svog izvršnog vremena provodi u maloj "neoptimizovanoj" petlji. Ponovo su napisali pomenutu petlju i učinili je 10 puta bržom. Međutim, nije došlo do bilo kakve promene u brzini izvrašavanja aplikacija. Kada su malo bolje pogledali shvatili su da su optimizovali sistemsku *idle* petlju (*idle loop*).

Tim koji je kreirao programski jezik ALGOL - deku većine modernih jezika i jedan od najuticajnijih jezika ikada dao je savet programerima: "Najbolje je neprijatelj dobrog. Rad ka savršenstvu može da ili veoma često sprečava da se zadatak uopšte i završi. Prvo završi, pa onda usavršavaj. Deo koji treba da bude savršen obično je veoma mali.", njihova je poruka.

BAPSKE PRIČE

Veliki deo onoga što ste čuli o optimizaciji koda je netačan, uključujući i česte zablude:

Smanjanje broja linija koda u jezicima visokog nivoa poboljšava brzinu ili smanjuje veličinu rezultujućeg mašinskog koda – neistina. Mnogi programeri se uporno drže verovanja da ako napišu kod koji ima samo jednu ili par linija, da će takav kod biti najefikasniji mogući. Pogledajmo sledeći kod koji inicijalizuje niz od 10 elemenata:

```
for i = 1 to 10
   a[ i ] = i
end for
```

Da li bi ste rekli da je ovakav kod brži ili sporiji od naredne verzije koda koja radi isti posao?

```
a[ 1 ] = 1
a[ 2 ] = 2
a[ 3 ] = 3
a[ 4 ] = 4
a[ 5 ] = 5
a[ 6 ] = 6
a[ 7 ] = 7
a[ 8 ] = 8
a[ 9 ] = 9
a[ 10 ] = 10
```

Ako biste primenili gorenavedenu dogmu da je kraći kod i brži, pomislili bi ste da je gornji kod brži. Međutim, testovi u Microsoft Visual Basic-u i Java-i pokazuju da je drugi deo koda brži za barem 60 procenata od prvog. U narednoj tabeli možete pogledati brojke:

Jezik	For petlja	Ručna inicijalizacija	Ušteda	Odnos performansi
Visual Basic	8.47	3.16	63%	2.5:1
Java	12.6	3.23	74%	4:1

Tabela 1. Inicijalizacija vrednosti niza od 10 elemenata

Ovo svakako ne znači da povećanje broja linija koda jezika visokog nivoa uvek poboljšava brzinu ili smanjuje veličinu mašinskog koda. Ali, ovo govori da bez obzira na "estetsku lepotu" pisanja nečega što ima manje linija, ne postoji nikakav prdvidiv odnos između broja linija koda u jezicima visokog nivoa i konačne veličine i brzine ukupnog programa.

Određene operacije su verovatno brže ili manje od drugih – neistina. Ne postoji nimalo prostora za odrednice kao što je "verovatno" kada pričamo o performansama. Morate uvek da merite performance da bi ste znali da li je načinjena promena dala rezultate ili nije. Pravila igre se menjaju svaki put kada promenite programski jezik, kompajler, verziju kompajlera, biblioteke ili njihove verzije, procesore, količinu raspoložive radne memorije itd…

Ova pojava ukazuje na to da postoji nekoliko razloga da se performance ne poboljšavaju putem prilagođavanja koda. Ako želite da program bude portabilan, morate imati na umu da tehnike koje poboljšavaju performance u jednom okruženju mogu da ih degradiraju u drugom. Ako recimo promenite kompajler ili ga nadogradite na noviju verziju, novi kompajler će možda automatski optimizovati kod na isti način na koji bi ste vi ručno optimizovali kod i vaš rad bi bio uzaludan. Ili još gore, "ručna" optimizacija koda može da "pobedi" optimizacije kompajlera koje su dizajnirane da rade sa standardnim kodom i da uspori vašu aplikaciju.

Kada optimizujete kod, vi praktično implicitno ponovno profilišete svaku optimizaciju svaki put kada promenite proizvođača kompajlera, njegovu verziju, verziju biblioteke itd... Ako ne uradite ponovno profilisanje, optimizacija koja poboljšava performanse u jednoj verziji kompajlera mogla bi da degradira performanse kada god promenite okruženje za kreiranje mašinskog koda. „Ne treba se baviti malim efikasnostima: prerana optimizacija je koren sveg zla" – Donald Knuth.

Kod treba optimizovati tokom rada – neistina. Jedna teorija kaže da ako težite pisanju najbržeg i najmanjeg mogućeg koda, trebalo bi da optimizujete kod tokom pisanja svake rutine, a vaš program će biti brz i mali. Ovaj pristup kreira situaciju da programer „ne vidi šumu od drveća" u kojoj se ignorišu značajne globalne optimizacije jer je programer prezauzet mikro-optimizacijama. Slede glavni problemi sa optimizovanjem koda tokom rada:

- Skoro da je nemoguće identifikovati uska grla pre nego što konačan program proradi. Programeri su veoma loši u proceni koja 4 procenta koda su važna za 50 procenata vremena izvršavanja pošto jednostavno nemaju uvid u buduće navike korisnika. Programeri koji optimizuju tokom normalnog programiranja, u proseku, provedu 96 procenata vremena na optimizaciju koda koji uopšte i nije potrebno optimizovati. To ostavlja jako malo vremena za optimizaciju onih najvažnijih 4 procenta koja se „stvarno" računaju.
- U onim retkim situacijama kada programeri identifikuju uska grla korektno, oni se fokusiraju na ta uska grla pružajući im preveliku pažnju. Na taj način dozvole drugim uskim grlima da postanu kritična. Ponovo, konačni efekat je pogoršanje performansi. Optimizacije koje se rade nakon što je program kompletiran mogu da identifikuju oblast u kojem se problem javlja i njegovu relativnu važnost tako da je moguće efikasno rasporediti vreme koje je namenjeno za optimizaciju koda.
- Fokusiranje na optimizaciju tokom inicijalnog razvoja programa odvlači pažnju sa postizanja drugih programskih ciljeva. Programeri „zarone" u analizu algoritama i volšebne debate koje na kraju ne donese vrednost za korisnike programa. Drugi ciljevi kao što su ispravnost koda, sakrivanje informacija (učaurivanje) i čitljivost koda postanu sekundarni ciljevi, iako je performanse lakše kasnije poboljšati nego druge navedene ciljeve. Naknadni rad na poboljšanju performansi obično utiče na manje od 5 procenata programskog koda. Da li biste se radije vratili i radili na optimizaciji performansi 5 procenata koda ili na čitljivosti na 100 procenata koda?

Ukratko, glavni nedostatak prerane optimizacije je njen nedostatak perspektive. Žrtve prerane optimizacije su konačna brzina koda, atributi performansi koji su važniji od brzine koda, kvalitet programa i konačno korisnici programa. „Ako bi se vreme za razvoj programa koje bi se uštedelo na implementaciji najjednostavnijeg programa koji će raditi, posvetilo optimizaciji programa koji je već funkcionalan, umesto da se uloži u optimizaciju koda od samog početka razvoja - rezultat bi uvek bio program koji je brži od ovog drugog (optimizovanog od početka razvoja)" – Stevens (1981).

Povremeno, optimizacije koda nakon završetka programa neće biti dovoljne kako bi se ispunili ciljevi po pitanju perfomansi i moraćete da načinite značajne promene na kompletiranom kodu. U tim situacijama, male, lokalne optimizacije ionako ne bi donele potrebne dobitke. Problem u tim situacijama nije neadekvatan kvalitet koda – problem je neadekvatna softverska arhitektura.

Ako je ipak potrebno da optimizujete program pre nego što je završen, minimizirajte rizik tako što ćete uključiti perspektivu u ovaj process. Jedan od načina da to uradite jeste da specificirate ciljeve u pogledu veličine i brzine za pojedinačne delove programa i da ih onda optimizujete tako da ispune zadate ciljeve dok radite na ukupnom programu. Postavljanje takvih ciljeva jeste način da jedno oko držite na šumi dok drugim pokušavate da sagledate koliko je visoko drvo kojim se trenutno bavite.

Brzina programa je podjednako važna kao i tačnost istog – neistina. Vrlo, vrlo retko će se desiti da je potrebno da program bude mali ili brz pre nego korektan, odnosno ispravan. Gerald Weinberg prenosi priču o programeru koji je pozvan u Detroit da bi pomogao u debagovanju problematičnog programa. Taj programer je radio sa timom programera koji su razvili kompletan program i nakon par dana je zaključio da je situacija beznadežna.

Na putu kući, on je intenzivno razmišljao o situaciji i shvatio je šta je problem. Do kraja leta kući, on je već imao okvir za nov kod koji bi rešio problem. Napisao je i testirao kod nekoliko dana i taman kada je krenuo nazad za Detroit, dobio je telegram u kome mu je saopšteno da je projekat otkazan zbog toga što je bilo nemoguće da se napiše program koji bi

ispunio zahteve korisnika. Međutim, on je ipak odleteo ponovo za Detroit i ubedio rukovodioce da je moguće završiti projekat.

Nakon toga je trebalo "samo" da ubedi prvobitne programere koji su radili na projektu. Oni su odslušali pažljivo njegovu prezentaciju i kada je završio, jedan od kreatora starog sistema je upitao: "A koliko je potrebno da se vaš program izvrši?" "To varira, ali oko deset sekundi po unosu", rekao je novi programer. "Aha, ali našem programu je potrebna samo jedna sekunda po unosu", reče veteran lagano se zavalivši unazad koji je već bio zadovoljan što je poentirao na samom startu razgovora, dok su se ostali programeri složili. Međutim, novi programer nije bio zastrašen njihovom konstatacijom: "Da, ali vaš program ***ne radi***! Ako i moj program ne mora da radi, ja mogu da ga napravim da se izvršava skoro trenutno."

Za određenu klasu projekata, brzina ili veličina predstavljaju osnovne ciljeve. Ovakvi projekti su prilično retki i dosta su ređi nego što većina ljudi misli. Za takve projekte, rizici koji se tiču performansi se moraju ukalkulisati u osnovni dizajn. Za ostale projekte, rana optimizacija predstavlja značajnu pretnju za ukupan kvalitet softvera, uključujući i performanse.

KADA OPTIMIZOVATI

Koristite dizajn visokog kvaliteta. Napravite da program radi tačno. Napravite da bude modularan i lako izmenljiv tako da možete da radite lako sa njim kasnije. Kada je program završen i tačan, proverite performanse. Ako primetite da je program spor ili glomazan, učinite ga efikasnim. Ali, nemojte da optimizujete stvari sve dok niste sigurni da to i treba da uradite. Jackson-ovo pravilo optimizacije:

- Pravilo broj 1. - Nemojte da radite (optimizaciju).
- Pravilo broj 2. (samo za eksperte) – Nemojte da je radite još uvek, tj. sve dok nemate savršeno jasno i neoptimizovano rešenje (aplikaciju).

Pre nekoliko godina autor knjige je radio na C++ projektu koji je kreirao grafički izlaz za analizu investicionih podataka. Nakon što je njegov tim završio prvi graf, urađena su testiranja koja su pokazala da je potrebno oko 45 minuta za iscrtavanje pojedinačnog grafa. To očigledno nije bilo prihvatljivo. Tim je održao sastanak da bi se videlo šta dalje raditi. Jedan od programera je, iritiran situacijom, uzviknuo: "Ako želimo da imamo bilo kakvu šansu da pustimo prihvatljiv proizvod, moramo da krenemo sa ponovnim pisanjem celog baznog koda u asembleru odmah!". Autor knjige je odgovorio da ne misli tako – da je vrlo verovatno da je 4 procenta koda odgovorno za 50 ili više procenata uskih grla koje utiču na performanse. „Bilo bi dobro da se potraže ta 4 procenta koda kako se projekat bliži kraju", nastavio je. Posle još malo vikanja, menadžer projekta je dodelio autoru knjige da uradi inicijalno testiranje performansi.

Kao što je to često slučaj, jedan dan testiranja je identifikovao nekoliko šljaštećih uskih grla u kodu. Samo par optimizacija koda je smanjilo vreme iscrtavanja grafova sa 45 minuta na manje od 30 sekundi. Daleko manje od jednog procenta koda je bilo odgovorno za 90 procenata vremena izvršavanja. Do trenutka kada je proizvod pušten na tržište nekoliko meseci kasnije, par dodatnih optimizacija koda je skratilo vreme iscrtavanja na nešto više od jedne sekunde.

OPTIMIZACIJE KOMPLAJLERA

Optimizacije modernih kompajlera su možda moćnije nego što biste očekivali. U situaciji sa grafovima koja je opisana, kompajler je odradio bolji posao sa ugnježdenim petljama, nego što je autor bio u stanju da napiše na navodno „efikasniji način". Kada kupujete kompajler, uporedite performanse svakog kompajlera na vašoj aplikaciji. Svaki kompajler ima različite prednosti i slabosti i neke od njih će više odgovarati vašoj aplikaciji ili tipu aplikacija koje izrađujete.

Optimizacije kompajlera su efikasnije kada se radi sa standardnijim kodom nego kada kompajler radi sa „optimizovanim", „lukavim" kodom. Ako ste u kodu radili sa „pametnim" stvarima kao što je igranje sa indeksima petlji, vašem kompajleru će biti teže da uradi svoj posao, a vaš program će zbog toga trpeti.

Sa dobrim optimizacijama kompajlera, vaš kod može imati ubrzanje od čitavih 40 procenata ili više. Mnoge od tehnika za optimizaciju koda, koje ćemo pogledati kasnije, mogu da proizvedu maksimalan dobitak od 15 do 30 procenata. Zašto onda ne pisati čist, standardan kod i pustiti kompajler da obavi svoj deo posla? U sledećoj tabeli se može videti rezultat nekoliko testova iz kojih se može zaključiti koliko je kompajler ubrzao rutinu za *insertion-sort*: jedina razlika između verzija rutina jeste da su one kompajlirane sa isključenim optimizacijama za prvo kompajliranje i uključenim za drugo kompajliranje. Očigledno, neki kompajleri bolje optimizuju nego drugi, dok neki bolje rade bez optimizacije. Neke Javine virtualne mašine (JVM) su očigledno bolje nego druge. Naravno, da bi dobili najbolje rezutate za vaše aplikacije, morate da proverite kompajler, JVM ili oba da bi izmerili efekte.

Jezik	Vreme bez optimizacija kompajlera	Vreme sa optimizacijama kompajlera	Ušteda u vremenu	Odnos performansi
C++ kompajler 1	2.21	1.05	52%	2:1
C++ kompajler 2	2.78	1.15	59%	2.5:1
C++ kompajler 3	2.43	1.25	49%	2:1
C# kompajler	1.55	1.55	0%	1:1
Visual Basic	1.78	1.78	0%	1:1
Java VM 1	2.77	2.77	0%	1:1
Java VM 2	1.39	1.38	<1%	1:1
Java VM 3	2.63	2.63	0%	1:1

Tabela 2. Brzina izvršavanja koda kompajliranog sa optimizacijama i bez optimizacija kompajlera

VRSTE ZAGUŠENJA

Kada radite optimizaciju koda nalazićete na delove programa koji su spori kao puž ili veliki kao Godzila i menjati ih tako da budu brzi kao munje i toliko mali (tanki) tako da se mogu sakriti u pukotine između bajtva u RAM-u. Uvek morate da profilišete program da biste znali, sa iole sigurnosti, koji delovi su spori i glomazni, ali i pored toga postoje određene operacije i strukture koje imaju dugu istoriju lenjosti i gojaznosti. Za početak, nije loša ideja da vaša analiza startuje proučavanjem baš ovih operacija.

ČESTI IZVORI NEEFIKASNOSTI

Ulazno/Izlane (I/O) operacije: Jedan od najznačajnijih i najčešćih izvora neefikasnosti su nepotrebne I/O operacije. Ako imate mogućnost izbora da radite sa fajlovima u memoriji nasuprot fajlovima na disku, u bazi podataka ili preko mreže, uvek koristite strukturu podataka u radnoj memoriji, osim u situaciji kada je prostor kritičan.

Evo poređenja performansi između koda koji pristupa nasumičnim elementima u nizu od 100 elemenata koji se nalazi u radnoj memoriji i koda koji pristupa nasumičnim elementima u fajlu sa 100 elemenata koji se nalazi na hard disku:

Jezik	Vreme za fajl	Vreme za RAM	Ušteda u vremenu	Odnos performansi
C++	6.04	0.000	~100%	N/A
C#	12.8	0.010	~100%	1000:1

Tabela 3. Vremena pristupa nasumičnim elementima, niz od 100 elemenata

Prema izmerenim podacima, pristup elementima niza u memoriji je reda veličina 1000 puta brži od pristupa podacima u eksternom fajlu. Čak šta više, sa C++ kompajlerom koji je korišćen na testu, neophodno vreme koje je bilo potrebno za pristup podacima u memoriji je tako kratko da ga nije bilo moguće izmeriti. Poređenje performansi na sličnom testu, ali sa sekvencijalnom pristupom elementima niza izleda ovako:

Jezik	Vreme za fajl	Vreme za RAM	Ušteda u vremenu	Odnos performansi
C++	3.29	0.021	~99%	150:1
C#	2.60	0.030	~99%	85:1

Tabela 4. Vremena sekvencijalnog pristupa elementima niza, testovi izvršeni nad 13 puta većim podacima

Da je test koristio sporiji medijum za eksterni pristup, na primer tvrdi disk preko mrežne konekcije – razlika bi bila još značajnija. Test performansi u situaciji kada koristite sličan (slučajni) pristup elementima na lokaciji na mreži umesto na lokalnoj mašini izleda ovako:

Jezik	Vreme za fajl	Vreme za RAM	Ušteda u vremenu
C++	6.04	6.64	-10%
C#	12.8	14.1	-10%

Tabela 5. Vremena sekvencijalnog pristupa elementima niza na mrežnoj lokaciji

Naravno, ovi rezultati mogu da variraju drastično u zavisnosti od brzine vaše mreže, opterećenja, udaljenosti lokalne mašine od tvrdog diska kome se pristupa, brzine mrežnog diska u odnosu na lokalni disk, trenutne faze mesečevih mena i drugih faktora… Sveukupno gledano, efekat pristupa podacima u radnoj ili još bolje keš (*cache*) memoriji je dovoljno snažan da u svakoj situaciji razmislite o I/O operacijama u kritičnim delovima programa kod kojih se zahteva brzina, odnosno efikasnost.

Straničenje (*paging*): Operacija koja zahteva od operativnog sistema da menja stranice radne memorije je daleko sporija od operacije koja radi samo sa jednom stranicom memorije. Ponekad sitna promena ima za posledicu velike razlike. U sledećem primeru, programer je napisao petlju za inicijalizaciju koja je proizvela veliki broj pogrešnih memorijskih stranica (*page faults*), na sistemu koji koristi stranice veličine 4KB. Sledi primer urađen u Java programskom jeziku koji proizvodi veliki broj loših stranica:

```
for ( column = 0; column < MAX_COLUMNS; column++ ) {
  for ( row = 0; row < MAX_ROWS; row++ ) {
      table[ row ][ column ] = BlankTableElement();
   }
}
```

Kada se pogleda na prvu ovo izgleda kao sasvim lepo formatirana petlja sa dobrim imenima promenljivih. Pa šta je onda problem? Problem je u tome što je svaki element tabele "dugačak" 4000 bajtova. Ako tabela imam veliki broj redova, svaki put kada program pristupa drugom redu, operativni sistem će morati da promeni memorijsku stranicu. Ako pogledamo kako je strukturirana petlja u navedenom primeru videćemo da svaki pristup ovom dvodimenzionalnom nizu u osnovi menja redove, što praktično znači da svaki pristup tom nizu povlači sa sobom i promenu stranice, tj. prebacivanje na tvrdi disk. Programer je uvidevši ovo, restrukturirao patlju na sledeći način:

```
for ( row = 0; row < MAX_ROWS; row++ ) {
   for ( column = 0; column < MAX_COLUMNS; column++ ) {
      table[ row ][ column ] = BlankTableElement();
   }
}
```

Ovaj kod, kao i prethodni, takođe proizvodi pozive ka različitim memorijskim stranicama, ali on menja redove "samo" MAX_ROWS puta, umesto MAX_ROWS * MAX_COLUMNS puta. Kazna u performansama varira značajno od slučaja do slučaja. Na mašinama sa malo memorije, na kodu čije je vreme izvršavanja oko sekunde, može se izmeriti i 1000 puta brže izvršavanje sa drugom u odnosu na prvu varijantu koda. Na mašinama sa dosta radne memorije, izmerena je razlika od samo 2 puta i manje, a razlika se može uočiti isključivo kada se radi o veoma velikim vrednostima MAX_ROWS i MAX_COLUMNS.

Sistemski pozivi (*System calls*): Pozivi sistemskih rutina su čestio veoma skupi. Oni često uključuju promenu konteksta – snimanje stanja programa, uzimanje stanje kernela i obratno. Sistemske rutine uključuju I/O operacije sa diskovima, tastaturom, ekranom, štampačem ili drugim uređajima, zatim rutine koje upravljaju memorijom, kao i određene pomoćne rutine. Ako su performance bitne, treba pogledati koliko su skupi sistemski pozivi koje koristite. Ako shvatite da oni troše značajne resurse možete uzeti u obzir sledeće opcije:

- ***Napišite sopstvene servise***. Ponekad vam je potreban samo mali deo funkcionalnosti koje vam daje sistmska rutina: Vi možete da kreirate sopstveni servis koji bi zadovoljio isključivo vaše potrebe, bez obimnijeg korišćenja resursa. Pisanjem sopstvenog servisa dobijate nešto što je brže, manje i bolje prilagođeno vašim potrebama.
- ***Izbegavajte odlazak u sistem***.
- ***Sarađujte sa proizvođačem operativnog sistema kako bi on napravio poziv koji je efikasniji***. Većina proizvođača operativnih sistema želi da unapredi sopstvene proizvode i drago im je da čuju za delove sistema koji imaju slabe performance. Može da se desi da u početku budu malo džangrizavi, ali u suštini oni su veoma zainteresovani za takve stvari.

U pokušaju optimizacije koda koji je opisan u delu ***Kada optimizovati***, program je koristio klasu *AppTime* koja je izvedena iz komercijalno dostupne klase *BaseTime*. *AppTime* objekat je bio najčešće korišćen objekat u celom programu i instancirano je više desetina hiljada *AppTime* objekata. Nakon nekoliko meseci, otkiveno je da je klasa BaseTime prilikom inicijalizacije pozivala sistemsko vreme u konstruktoru. Za potrebe programa, sistemsko vreme je bilo irelevantno, što znači da je pravljeno desetine hiljada nepotrebnih sistemskih poziva. Prosto over-rajdovanje konstruktora klase BaseTime i inicijalizacija atributa *time* na 0, umesto

sistemskog poziva, doprinelo je više na poboljšanju performansi od svih ostalih promena i optimizacija zajedno, koje su sprovedene.

Interpretirani jezici: Interpretirani jezici proizvode značajne gubitke u performansama zbog toga što moraju da procesiraju svaku programsku instrukciju pre nego što naprave i izvrše mašinski kod. U sledećoj tabeli se može pogledati relativan odnos u brzini izvšavanja nekoliko standardnih i par interpretiranih jezika:

Programski jezik	Tip programskog jezika	Vreme izvršavanja u odnosu na C++
C++	Kompajlirani	1:1
Visual Basic	Kompajlirani	1:1
C#	Kompajlirani	1:1
Java	Bajt kod	1,5:1
PHP	Interpretirani	>100:1
Python	Interpretirani	>100:1

Tabela 6. Relativan odnos u brzini različitih tipova jezika (kompajlirani/interpretirani/…)

Kao što se može videti, C++, Visual Basic i C# imaju slične performanse. Java je tu negde blizu, ali sporija nego pomenuti kompajlirani jezici. PHP i Python su interpretirani jezici i kao takvi, njihov kod (u zavisnosti od polja primene) se izvršava oko 100 puta sporije. Brojke, tj. odnosi u ovoj tabeli se moraju uzeti sa rezervom pošto su oni drugačiji ako pogledamo različita polja primene: kompajlirani jezici mogu biti i 2 puta brži od ostalih ili 2 puta sporiji. Kasnije ćemo pogledati neke primere.

Greške: Poslednji standarni izvor problema sa performansama su greške u kodu. Greške uključuju zaboravljanje uključenog moda za proveru i pronalaženje grešaka (*debugging*) i posledične radnje koje troše resurse (kao što su zapisivanje informacija o logovima u fajlu), zaboravljanje da se dealocira, odnosno oslobodi radna memorija, loš dizajn tabela baze podataka, "povlačenje" sa nepostojećih uređaja i slične stvari.

Autor knjige je radio na verziiji 1.0 aplikacije u kojoj se nalazila operacija koja je bila neobično sporija od drugih sličnih operacija. Tokom rada na aplikaciji, zaživela je čak i svojevrsna mitologija koja se trebalo da objasni sporost ove operacije. Autori su pustili aplikaciju na tršište u verziji 1.0 bez da su ikada stvarno razumeli zašto je navedena operacija tako spora. Dok je radio na verziji 1.1, autor aplikacije je uvideo da tabela baze podataka koju koristi dotična operacija nije uopšte indeksirana! Prosto indeksiranje tabele je poboljšalo performanse za oko 30 puta za određene operacije. Definisanje indeksa na često korišćenim tabelama nije optimizacija, to je prosto standardna dobra programerska praksa.

RELATIVNA CENA U PERFORMANSAMA ČESTO KORIŠĆENIH OPERACIJA

Iako ne možete unapred da računate da su određene operacije zahtevnije od drugih, ipak postoje određene operacije koje imaju tendenciju da budu „skuplje" u odnosu na druge. Kada tražite mesta u vašem programu koja su troma, možete se poslužiti sledećom tabelom da bi napravili neka inicijalna nagađanja o tome gde se nalaze „lepljivi" delovi. Merenja u ovoj tabeli su veoma osetljiva na okruženje lokalne mašine, optimizacije kompajlera i kod koji generišu određeni kompajleri. Merenja između operacija koje se vrše u programskim jezicima C++ i Java nisu direktno uporediva.

Operacija	Primer	C++	Java
Osnovno (celobrojno dodeljivanje)	i = j	1	1
Pozivi rutina			
Poziv rutine koja nema parametre	foo()	1	n/a
Poziv privatne rutine koja nema parametre	this.foo()	1	0.5
Poziv privatne rutine koja ima 1 parametar	this.foo(i)	1.5	0.5
Poziv privatne rutine koja ima 2 parametra	this.foo(i, j)	2	0.5
Poziv rutine objekta	bar.foo()	2	1
Poziv izvedene rutine	derivedBar.foo()	2	1
Poziv polimorfne rutine	abstractBar.foo()	2.5	2
Referenciranje objekata			
Dereferenciranje objekata 1. nivoa	i = obj.num	1	1
Dereferenciranje objekata 2. nivoa	i = obj1.obj2. num	1	1
Svako dodatno dereferenciranje	i = obj1.obj2.obj3…	Nije merljivo	Nije merljivo
Operacije sa celim brojevima			
Dodeljivanje celobrojne vrednosti (lokalno)	i = j	1	1
Dodeljivanje celobrojne vrednosti (nasleđeno)	i = j	1	1
Dodavanje celih brojeva	i = j + k	1	1
Oduzimanje celih brojeva	i = j - k	1	1
Množenje celih brojeva	i = j * k	1	1
Celobrojno deljenje	i = j \ k	5	1.5
Operacije brojevima sa pokretnim zarezom			
Dodeljivanje brojeva sa pokretnim zarezom	x = y	1	1
Sabiranje brojeva sa pokretnim zarezom	x = y + z	1	1
Oduzimanje brojeva sa pokretnim zarezom	x = y - z	1	1
Množenje brojeva sa pokretnim zarezom	x = y * z	1	1
Deljenje brojeva sa pokretnim zarezom	x = y / z	4	1
Transcedentne funkcije			
Kvadratni koren brojeva sa pokretnim zarezom	x = sqrt(y)	15	4
Sinus broja sa pokretnim zarezom	x = sin(y)	25	20
Logaritam broja sa pokretnim zarezom	x = log(y)	25	20
Eksponencijalna funkcija broja sa pokretnim zarezom e^y	x = exp(y)	50	20
Nizovi			
Pristup nizu celobrojnih vrednosti sa konstantnim indeksom	i = a[5]	1	1
Pristup nizu celobrojnih vrednosti sa promenljivim indeksom	i = a[j]	1	1
Pristup dvo-dimenzionalnom nizu celobrojnih vrednosti sa konstantnim indeksima	x = z[3, 5]	1	1
Pristup dvo-dimenzionalnom nizu celobrojnih vrednosti sa konstantnim indeksima	i = a[j, k]	1	1
Pristup nizu brojeva sa pokretnim	x = z[5]	1	1

zarezom sa konstantnim indeksom			
Pristup nizu brojeva sa pokretnim zarezom sa promenljivim indeksom	x = z[j]	1	1
Pristup dvo-dimenzionalnom nizu brojeva sa pokretnim zarezom sa konstantnim indeksom	x = z[3, 5]	1	1
Pristup dvo-dimenzionalnom nizu brojeva sa pokretnim zarezom sa promenljivim indeksom	x = z[j, k]	1	1

Tabela 7. Relativna „cena“ najčešćih operacija (rezultati nisu direktno uporedivi za C++ i Java)

Većina čestih operacija ima sličnu "cenu" – pozivi rutina, dodeljivanja, celobrojna aritmetika, kao i aritmetika brojeva sa pokretnim zarezom i one su ugrubo podjednako zahtevne. Transcedentne matematičke funkcije su, sa druge strane, veoma zahtevne, dok su pozivi polimorfnih rutina malo zahtevniji u odnosu na druge vrste poziva rutina. Gorenavedena tabela 7 ili bilo koja druga slična koju bi ste napravili, je ključ koji otključava potencijalna poboljšanja koja ćemo opisati kasnije. U svakom slučaju, suština poboljšanja efikasnosti i performansi jeste u zameni skupih operacija onima koje to nisu (gde je to moguće).

MERENJA

Usled činjenice da mali delovi programa zauzimaju disproporcionlano veliki deo vremena za izvršavanje, uvek treba "meriti" kod kako bi se pronašla uska grla. Jednom, kada ste pronašli uska grla i optimizovali ih, testirajte kod ponovo da bi ste pravilno procenili koliko ste ga unapredili. Mnogi aspekti performansi su suprotni intuitivnom mišljenju. Primer koji smo nešto ranije pokazali, gde je 10 linija koda značajno brže od jedne linije koda istog semantičkog značenja, je samo jedan primer kako vas kod može iznenaditi. Takođe, ni ranije iskustvo u optimizacijama nije garant da ćete uspeti da optimizujete pravilno i dovoljno kod u ovom trenutku. Iskustvo može doći iz rada sa starijim mašinama, softverom, programskim jezikom ili kompajlerom. Kada se bilo koja od ovih stvari promeni, velika je verovatnoća da vam iskustvo neće pomoći. Nikada ne možete biti sigurni u efekte optimizacije, dok ne testirate "optimizovani" kod i izmerite konkretne efekte vaših optimizacija.

Autor knjige se pre nekoliko godina bavio programom koji je trebalo da sumira elemente dvodimenzionalnog niza, odnosno matrice. Originalni kod (C++) je izgledao ovako i predstavlja primer klasičnog (standardnog) koda za sumiranje elemenata matrice:

```
sum = 0;
for ( row = 0; row < rowCount; row++ ) {
   for ( column = 0; column < columnCount; column++ ) {
      sum = sum + matrix[ row ][ column ];
   }
}
```

Ovaj kod je sasvim standardan, a kako su performanse sumiranja matrice bile kritične za celu aplikaciju, autor knjige je znao da su pristupanje članovima niza i testiranja u petlji morala nositi sa sobom određene penale po performanse. Znao je da svaki put kada kod pristupa dvo-dimenzionalnom nizu, da se izvode "skupa" množenja i sabiranja. Za matrice reda 100, odnosno 100 X 100, to je bilo 10,000 množenja i sabiranja, plus *overhead* koji pravi petlja. Autor je razmiljšao, da bi konvertovanjem u pointersku notaciju mogao da inkrementira pointere i da praktično zameni 10,000 skupih množenja relativno jeftinim operacijama inkrementiranja pointera. Zatim je pažljivo konvertovao gornji kod u pointersku notaciju, u pokušaju da optimizuje sumiranje elemenata u matrici:

```
sum = 0;
elementPointer = matrix;
lastElementPointer = matrix[ rowCount - 1 ][ columnCount - 1 ] + 1;

while ( elementPointer < lastElementPointer ) {
   sum = sum + *elementPointer++;
}
```

Iako novi kod nije čitljiv kao prva verzija, posebno programerima koji nisu vični u C++ programskom jeziku, autor je bio veoma zadovoljan sobom. Za matricu 100 X 100, izračunao je da je uštedeo 10,000 množenja i *overhead* petlje. Bio je toliko zadovoljan sobom da je rešio da izmeri ubrzanje (što je nešto što nije radio tako često tada) da bi još jednom mogao da oda priznanje samom sebi.

Da li znate šta je izmerio? Nije bilo nikakvog ubrzanja. Ni sa matricom 100 X 100, niti sa matricom 10 X 10, niti sa matricom bilo koje veličine. Bio je toliko razočaran, da je odlučio da pogleda asemblerski kod koji je generisao kompajler da bi video zašto optimizacija nije dala nikakve rezultate. Na njegovo iznenađenje, ispostavilo se da on nije bio prvi programer kome je bilo potrebno da vrši iteriranje kroz elemente niza, odnosno matrice – optimizator kompajlera je već konvertovao pristupe nizu u pointere. Naučio je takođe, da je jedini rezultat optimizacije, u koji možete biti sigurni lošija čitljivost koda, a sa njom i naporniji i dugotrajniji rad. Ako ne uradite merenja da bi znali da li je novi kod efikasniji, onda nemojte da žrtvujete čitljivost koda za neizvesnost u performansama.

Jon Bentley je, u svom klasiku "Pisanje efikasnih C programa" iz 1991. godine, prijavio slično iskustvo prilikom konverzije u pointere, koje je pralktično usporilo aplikaciju za 10%. Ista konverzija je, u drugom okruženju, popravila performanse za čitavih 50%. Sve zavisi od okruženja.

„Nijedan programer nikada nije bio u mogućnosti da predvidi ili analizira uska grla bez podataka. Nije važno šta vi mislite u kom pravcu stvari idu, bićete iznenađeni kada otkrijete da idu negde drugde." - *Joseph M. Newcomer*.

MERENJA TREBA DA BUDU PRECIZNA

Merenja performansi treba da budu prezicna. Merenje brzine rada programa štopericom ili brojanjem "jedan slon, dva slona, tri slona…" nije prezicno. Za merenje performansi korisni su recimo programi za profilisanje ili možete koristiti sistemski sat i rutine koje beleže početna i krajnja vremena rada vašeg programa.

Bilo da koristite tuđe ili sopstvene alate za merenja, pobrinite se da merite isključivo delove koda, odnosno njihovo izvršavanje, koje pokušavate da optimizujete. Koristite broj taktova CPU-a koji su dodeljeni vašem programu, pre nego vreme dana, odnosno opšte vreme. U suprotnom (kada koristite opšte vreme), u situaciji kada sistem dodeli procesor nekom drugom programu, a ne vašem, neka od rutina vašeg programa koja je došla na red će nepravedno biti ocenjena kao sporija za vreme za koje je sistem dodelio CPU drugim programima. Takođe, pokušajte da faktorišete višak (*overhead*) koji nastaje samim procesom merenja, kao i višak koji nastaje prilikom startovanja programa, tako da ni originalni kod, a ni pokušaj optimizovanja ne budu nepravedno ocenjeni.

ITERACIJE

Jednom kada ste identifikovali usko grlo, bićete zapanjeni koliko je moguće poboljšati performanse putem optimizacije koda. Veoma retko ili skoro nikad ćete uspeti da postignete desetostruko ubrzanje korišćenjem jedne tehnike za optimizaciju koda, ali možete efektivno da kombinujete tehnike, tako da treba nastaviti sa potragom za tehnikama koje će na vašoj aplikaciji dati najbolja ubrzanja, čak i kada pronađete neku tehniku koja radi.

Autor knjige je jednom prilikom pisao softversku implementaciju standarda za enkripciju podataka (*Data Encryption Standard*) - DES. Enkripcija prema DES-u enkodira digitalne podatke tako da se oni ne mogu pročitati bez lozinke. Algoritam za enkripciju je tako kompleksan i uvijen da izgleda kao da je korišćen sam na sebi. Cilj u pogledu performansi za navedenu DES implementaciju je bio da se enkriptuje fajl veličine 18KB za 37 sekundi na originalnom IBM PC-u. Prva implementacija koju je autor uradio se izvršavala za 21 minut i 40 sekundi, tako da je sledio nešto duži put do cilja.

Iako je većina pojedinačnih optimizacija predviđenih za ovu konkretnu situaciju davala manje rezultate, kada se primene sve one zajedno, bile su dovoljno dobre da se završi zadatak. Kada se pogledaju procentualna poboljšanja, 3 ili čak 4 pojedinačne optimizacije koje su urađene ne bi ispunile cilj. Međutim, konačna kombinacija optimizacija je ispunila cilj. Poruka ove priče jeste da ako „kopate" dovoljno duboko, moćete postići fantastične dobitke.

Optimizacija koda koju je uradio autor implementacije je, prema njegovim rečima, najagresivnija optimizacija koda koju je ikada uradio. Istovremeno, finalni kod je bio najnečitljiviji i kod najteži za održavanje koji je ikada napisao. Sam inicijalni algoritam za enkripciju je dovoljno komplikovan. Kod koji je nastao optimizacijom koda jezika visokog nivoa je bio jedva čitljiv. Transformacija u asemblerski kod je recimo kreirala jednu rutinu od 500 linija koda koje bio trebalo da se uplaši svaki prosečan, odnosno „normalan" programer. Generalno, odnos koji smo pominjali između optimizacije i čitljivosti koda važi tj. obrnuto je proporcionalan. Pogledajmo tabelu primenjenih optimizacija za implementaciju DES algoritma:

Optimizacija	Vreme izvršavanja	Poboljšanje
Inicijalna implementacija	21:40	—
Konverzija iz bitova u nizove	7:30	65%
„Odmotavanje" najugnježdenije for petlje	6:00	20%
Uklanjanje poslednje permutacije	5:24	10%
Kombinovanje dve promenljive	5:06	5%
Upotreba logičkih identiteta da bi se kombinovala prva dva koraka u DES algoritmu	4:30	12%
Pravljenje da dve promenljive dele istu memoriju kako bi se smanjila razmena podataka u unutrašnjoj petlji	3:36	20%
Pravljenje da dve promenljive dele istu memoriju kako bi se smanjila razmena podataka u spoljašnjoj petlji	3:09	13%
„Odmotavanje" (*unfold*) svih petlji i potpisivanje literala nizovima (*array subscript*)	1:36	49%
Uklanjanje poziva rutina i ugrađivanje (*embeding*) pozivanog koda u osnovni kod (*inline*)	0:45	53%
Prepisivanje kompletne rutine u asembleru	0:22	51%
Na kraju (ukupno)	**0:22**	**98%**

Tabela 8. Primenjene optimizacije na kod za implementaciju DES algoritma

Stabilno napredovanje učinjenih optimizacija u gornjoj tabeli ne implicira da sve optimizacije koje probate daju pozitivan efekat. U tabeli nisu prikazane sve optimizacije koje je autor pokušao. Barem dve trećine optimizacija koje je probao da primeni nisu davale nikakav ili su stvarale negativan efekat u navedenoj situaciji.

PRISTUP OPTIMIZACIJI KODA

Prilikom razmatranja da li optimizacija koda može da učini efikasnijom vašu aplikaciju, trebalo bi preduzmete sledeće korake:

1. Razvijajte vaš softver korišćenjem dobro (široko) poznatih tehnika za pisanje softverskog koda koji je razumljiv i lak za modifikaciju.
2. Ako su performanse loše,
    a. Snimite radnu verziju koda tako da u svakom trenutku možete da se vratite na „poslednje dobro stanje".
3. Testirajte sistem kako bi pronašli uska grla.
    a. Utvrdite da li su slabe performanse posledica neadekvatnog dizajna (arhitekture), tipova podataka ili algoritama i da li bi optimizacija koda mogla da da željene rezultate. Ako optimizacija koda ne bi bila svrsishodna, vratite se na korak 1.
    b. Optimizujte uska grla identifikovana u koraku (c.).
    c. Merite svaku optimizaciju, jednu po jednu.
    d. Ako optimizacija ne poboljšava performanse, vratite se na kod koji ste sačuvali u koraku (a.). U većini slučajeva, više od polovine pokušanih optimizacija će proizvesti zanemarljiva poboljšanja ili čak degradirati performanse.
4. Ponovite proceduru od koraka (2).

TEHNIKE ZA OPTIMIZACIJU KODA

Optimizacija koda je oduvek bila popularna tema, praktično kroz čitavu istoriju komjuterskog programiranja. Stoga, ako ste odlučili da je potrebno da poboljšate performanse vašeg programa na nivou koda (putem optimizacije koda), a imajući na umu da ovo često nije ni najefikasniji niti najbrži način da se poboljšaju performanse, imaćete na raspolaganju bogat skup tehnika kojima je moguće postići značajna poboljšanja perfomansi.

Kada se govori o perfomansama, obično se misli na brzinu i veličinu, kako radne memorije, tako i na tvrdom disku. Treba imati u vidu da smanjenje veličine pre dolazi od redizajna klasa i rutina, kao i tipova podataka, nego od same optimizacije koda. Optimizacija koda se praktično odnosi na promene koda manjih razmera, a ne na redizajn aplikacije širokih razmera. Neke od tehnika koje ćemo razmatrati su toliko široko upotrebljive da je delove koda moguće iskopirati direktno u vaš program i pustiti u rad.

Neke od tehnika za optimizaciju koda se kozmetički mogu učiniti veoma sličnim određenim refaktorisanjima koda, ali treba napraviti jasnu razliku - refaktorisanje koda predstavlja promenu koda koja poboljšava internu strukturu programa, dok bi se optimizacija koda u cilju poboljašanja performansi mogla shvatiti kao anti-refaktorisanje. Promene koda koje nastaju optimizacijom degradiraju internu strukturu u zamenu za dobitke na polju performansi. Ovo je istina, po definiciji. Ako ovakve promene ne bi degradirale internu strukturu, ne bi smo ih smatrali optimizacijama, već bi ih podrazumevano koristili i smatrali standardnom programerskom praksom.

Neka izdanja, a i određeni autori, predstavljaju neke tehnike za optimizaciju koda kao „pozitivne principe" (*rules of thumb*) koji kažu da će specifične optimizacije dati željene efekte. Kao što ćemo videti, koncept „pozitivnih principa" se prilično teško može primeniti na optimizaciju koda. Jedini pouzdan princip jeste praktično merenje performansi programa, odnosno dela na koji se odnosi optimizacija, nakon svake optimizacije u vašem okruženju. To dalje znači, da tehnike koje ćemo obraditi u nastavku predstavljaju samo „stvari koje treba pokušati", od kojih mnoge neće ili će loše raditi u vašoj konkretnoj situaciji, ali i od kojih će neke sasvim lepo raditi i dati pozitivne rezultate po pitanju performansi.

LOGIKA

Veliki deo programiranja se sastoji od manipulisanja logikom. Pogledajmo nekoliko tehnika kojim se efikasnije upravlja logikom programa.

PREKINITE SA ISPITIVANJEM USLOVA KADA SAZNATE KONAČNU VREDNOST

Recimo da imate sledeću naredbu:

```
if ( 5 < x ) and ( x < 10 ) then ...
```

Jednom, kada ste utvrdili da je x veće od 5, nije potrebno da testirate drugi deo izraza – koja god da je vrednost ostatka logičkog izraza, ukupan izraz će imati vrednost *false*. Neki jezici imaju konstrukcije za utvrđivanje vrednosti celokupnog izraza poznato kao "kratko-spojeni operatori" (*short-circuit evaluation*), što znači da kompajleri generišu kod koji automatski prekida testiranja uslova čim sazna konačnu vrednost iskaza. Kratko-spojeni

operatori su deo standardnih operatora jezika C++ i javljaju se još recimo u jeziku Java kao uslovni operatori. Ako programski jezik u kojem radite standardno (*natively*) ne podržava kratko-spojene operatore, treba da izbegavate korišćenje logičkih strujtura I (*and*) i ILI (*or*), dodajući drugačiju logiku. Gornji primer bi mogao da izgelda ovako:

```
if ( 5 < x ) then
   if ( x < 10 ) then ...
```

Princip koji govori da prestanete sa testiranjima i proverama nakon što sigurno znate njihov konačni rezultat je dobar i za veliki broj drugih situacija u programiranju. Kao jedan takav primer može poslužiti i petlja za pretragu. Ako skenirate niz unetih brojeva i tražite negativnu vrednost, a jednostavno treba da znate da li u tom nizu postoji negativan broj, jedan od pristupa jeste da proverite sve unete vrednosti i da u neku pomoćnu promenljivu (recimo *negativeNumberFound*) unesete vrednost *true* kada pronađete negativnu vrednost. Ako takav niz ima neku (ili više negativnih vrednosti) na kon petlje imaćemo informaciju da li je pronađena bilo koja negativna vrednost. Evo kako bi takva petlja za pretragu izgledala u programskom jeziku C++:

```
negativeNumberFound = false;

for ( i = 0; i < count; i++ ) {
   if ( input[ i ] < 0 ) {
      negativeNumberFound = true;
   }
}
```

Bolji pristup bi bio da se prekine sa skeniranjem (petljom) čim se pronađe neka negativna vrednost. Bilo koja od sledećih konstrukcija bi rešila opisanu radnju:

- Dodajte *break* naredbu nakon što promenljiva *negativeNumberFound* dobije vrednost *true*.
- Ako jezik koji koristite nema *break* naredbu, emulirajte *break* naredbu sa nekom *goto* konstrukcijom koja seli tok izvršavanja programa na prvu liniju nakon petlje u trenutku kada se pronađe negativna vrednost.
- Promenite *for* petlju *while* petljom i proveravajte i vrednost promenljive *negativeNumberFound* dok proveravate i trenutno stanje brojača petlje.
- Promenite *for* petlju *while* petljom i postavite jedan marker (*sentinel* iliti *flag value*) – negativnu vrednost u prvi član niza koji se nalazi iza poslednjeg unetog broja i jednostavno proveravajte postojanje negativne vrednosti u proveri uslova *while* petlje. Nakon što se petlja završi, proverite da li je pozicija prve pronađene negativne vrednosti u nizu ili nakon kraja niza (na mestu markera).

U sledećoj tabeli vidimo rezultate postignute korišćenjem *break* naredbe u C++ i Java:

Jezik	Standardno vreme	Vreme sa optimizovanim kodom	Ušteda u vremenu
C++	4.27	3.68	14%
Java	4.85	3.46	29%

Tabela 9. Rezultati korišćenja *break* naredbe

- Vremena u ovoj i narednih nekoliko tabela u ovom delu su data u milisekundama i imaju smisla samo za poređenje u istim tabelama. Stvarna vremena će varirati u zavisnosti od kompajlera, korišćenih opcija kompajlera, kao i okruženja u kome se izvode testovi.

- Rezultati merenja su nastali kao plod izvršavanja nekoliko hiljada ili miliona prolaza kroz specifilne delove koda da bi se odbacile fluktuacije rezultata koje se javljaju od testa do testa.
- Precizne marke i verzije kompajlera nisu naznačene. Performanse značajno variraju od proizvođača do proizvođača i od različitih verzija kompajlera.
- Poređenja rezultata različitih programiskih jezika nemaju uvek smisla zbog toga što kompajleri za različite jezike ne nude uvek uporedive opcije za generisanje koda.
- Neke od ušteda u vremenu nisu mogle biti prikazane na pravilan način zbog zaokruživanja u samoj tabeli u kolonama za standardno i vreme sa optimizovanim kodom.

Uticaj urađenih promena (optimizacija) značajno varira u zavisnosti od toga koliko se brojnim vrednosti unosi i koliko često očekujete da ćete pronaći negativnu vredmnost. Ovaj test podrazumeva prosečnih 100 vrednosti koje se unose i da će negativna vrednost biti pronađena u 50% testiranih slučajeva.

POREĐAJTE USLOVE PO UČESTALOSTI ISPUNJAVANJA

Poređajte uslove tako da se onaj koji je najbrži i koji će najverovatnije biti istinit prvi izvršava. U suštini,ređajte uslove tako da najčešće ispunjeni uslovi budu podrazumevani slučajevi, dok bi one koji su najzahtevniji i manje korišćeni trebalo procesirati u izuzetnim slučajevima. Ovaj princip se odnosi na *case* naredbu i ulančane *if-then-else* konstrukcije. Pogledajmo primer *select-case* naredbe koji odgovara na unos sa tastature u procesor teksta u programskom jeziku *Visual Basic* kod kojeg je loše izveden redosled logičkih testova:

```
Select inputCharacter

   Case "+", "="
      ProcessMathSymbol( inputCharacter )

   Case "0" To "9"
      ProcessDigit( inputCharacter )

   Case ",", ".", ":", ";", "!", "?"
      ProcessPunctuation( inputCharacter )

   Case " "
      ProcessSpace( inputCharacter )

   Case "A" To "Z", "a" To "z"
      ProcessAlpha( inputCharacter )

   Case Else
      ProcessError( inputCharacter )

End Select
```

Slučajevi u navedenoj *case* naredbi su poređani veoma slično originalnom ASCII rasporedu. Kod *case* naredbi, efekat je često veoma sličan većem setu *if-then-else* naredbi, tako da ako sa unosa dobijete simbol „a", program prvo testira da li je uneti znak matematički simbol, pa znak interpunkcije, cifra, blanko (*space*) pre nego što utvrdi da se u stvari radi o alfabetskom karakteru. Ako imate predstavu o tome koji će karakteri najčešće biti unošeni, možete da reorganizujete *case* naredbu tako da najčeće slučajeve stavite bliže vrhu i na taj način izbegnete veći broj nepotrebnih provera. Evo preuređene *case* naredbe:

```
Select inputCharacter

   Case "A" To "Z", "a" To "z"
      ProcessAlpha( inputCharacter )

   Case " "
      ProcessSpace( inputCharacter )

   Case ",", ".", ":", ";", "!", "?"
      ProcessPunctuation( inputCharacter )

   Case "0" To "9"
      ProcessDigit( inputCharacter )

   Case "+", "="
      ProcessMathSymbol( inputCharacter )

   Case Else
      ProcessError( inputCharacter )

End Select
```

Zbog toga što se najčešći slučajevi pronalaze ranije u optimizovanom kodu, ukupan efekat će biti poboljšane performanse usled manjeg broja testiranja. U sledećoj tabeli možemo pogledati rezultate optimizacije sa tipičnim miksom ulaznih karaktera:

Jezik	Vreme sa prvim kodom	Vreme sa optimizovanim kodom	Uštede u vremenu
C#	0.220	0.260	-18%
Java	2.56	2.56	0%
Visual Basic	0.280	0.260	7%

Tabela 10. Merenja su izvršena sa ulaznim skupom: 78% slova, 17% blanko i 5% interpunkcijskih karaktera

Rezultati merenja su očekivani za Visual Basic, ali za jezike Java i C# nisu. Očigledno, ovakvi rezultati su posledica načina na koji je strukturirana *case* naredba u jezicima C# i Java – zbog toga što se svaka vrednost mora enumerisati pojedinačno, a nije ih moguće enumerisati u intervalima (kao recimo u *Visual Basic*-u), pa zbog toga kod C# i Java koda nema poboljšanja usled optimizacije. Opet, ovakav rezultat naglašava važnost tvrdnji da ne treba slepo pratiti bilo koji vid predložene optimizacije i da će konkretne implementacije kompajlera značajno uticati na rezultate. Gledajući ovo, mogli bi da pretpostavite će kod koji generiše kompajler *Visual Basic*-a za skup *if-then-else* naredbi koje vrše isti test biti sličan, ako ne i isti sa prethodnim. Pogledajmo sada nove rezultate za *if-then-else* konstrukciju:

Jezik	Vreme sa prvim kodom	Vreme sa optimizovanim kodom	Uštede u vremenu
C#	0.630	0.330	48%
Java	0.922	0.460	50%
Visual Basic	1.36	1.00	26%

Tabela 11. Vremena dobijena korišćenjem *if-then-else* umesto *case* konstrukcije na istom setu ulaznih karaktera

Odmah se vidi da su rezultati prilično različiti. Za isti broj testova, kompajleru *Visual Basic*-a je potrebno oko 5 puta više vremena u neoptimizovanom slučaju, a oko 4 puta više u optimizovanom slučaju. Ovo govori da kompajer generiše potpuno drugačiji asemblerski kod za slučaj sa *case* naredbom od slučaja za *if-then-else* naredbu.

Poboljšanje sa *if-then-else* konstrukcijom je konzistentnije nego sa *case* naredbom. Kod *C#* i *Visual Basic*-a, obe verzije *case* naredbe su brže nego obe verzije *if-then-else* naredbi, dok su u Javi obe verzije sporije. Ova varijacija u rezultatima otvara vrata trećoj mogućoj optimizaciji.

POREDITE PERFORMANSE SLIČNIH LOGIČKIH STRUKTURA

Test opisan gore se može izvesti bilo korišćenjem *case*, bilo *if-then-else* naredbe. U zavisnosti od okruženja, bilo koji od pristupa bi mogao da radi efikasnije. U tabeli koja sledi vidimo podatke iz prethodne dve tabele formatirane tako da predstavljaju vremena za „optimizovani" kod kada se porede performanse *if-then-else* i *case* konstrukcija.

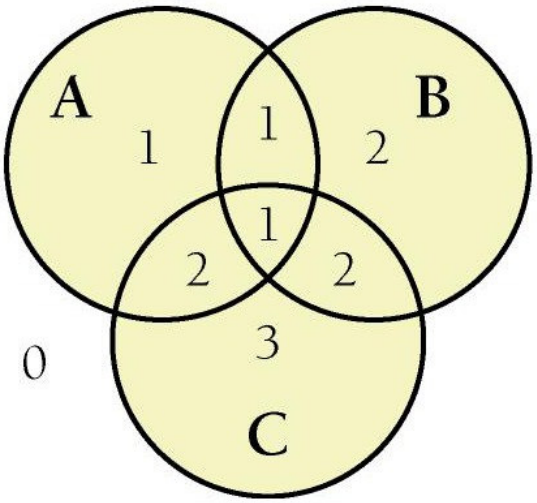
Jezik	case	if-then-else	**Uštede u vremenu**	**Odnos performansi**
C#	0.260	0.330	-27%	1:1
Java	2.56	0.460	82%	6:1
Visual Basic	0.260	1.00	258%	1:4

Tabela 12. Poređenje *if-then-else* i *case* konstrukcija sa „optimizovanim" kodom

Ovi rezultati se kose sa bilo kakvim logičkim objašnjenjem. U jednom od jezika, *case* naredba je superiorna u odnosu na *if-then-else*; u drugom jeziku, *if-then-else* je dramatično superiorna u odnosu na *case*. U trećem jeziku, ta razlika je relativno mala. Možete razmišljati da će zbog toga što C# i Java dele sličnu sintaksu za *case* naredbu, njihovi rezultati biti slični, ali oni su praktično suprotni jedan drugom. Ovaj primer jasno ilustruje težinu prihvatanja bilo kakvog „pozitivnog principa", odnosno „logike" za optimizaciju koda – prosto ne postoji pouzdana zamena za merenje rezultata.

ZAMENITE KOMPLIKOVANE IZRAZE TABELAMA (*TABLE LOOKUPS*)

U pojedinim situacijama, pronalaženje iz tabele može da bude brže nego prolaz kroz komplikovani lanac logike. Poenta komplikovane logike jeste da se kategoriše nešto i da se onda preduzmu određene akcije na osnovu pripradnosti kategoriji. Kao apstraktan primer, recimo da nečemu želite da dodelite broj kategorije na osnovu tri grupe: A, B i C i to spada u:



Slika 1. "Komplikovani" lanac logike

Evo primera u C++ jeziku koji dodeljuje brojeve kategorije na osnovu "komplikovanog" lanca logike, kao sa gornjeg grafa:

```
if ( ( a && !c ) || ( a && b && c ) ) {
   category = 1;
}

else if ( ( b && !a ) || ( a && c && !b ) ) {
   category = 2;
}

else if ( c && !a && !b ) {
   category = 3;
}

else {
   category = 0;
}
```

Logika iz ovog primera se može zameniti sa daleko izmenljivijom tabelom za pretraživanje koja pritom ima i daleko bolje performanse. Sledi primer u jeziku C++:

```
// define categoryTable

static int categoryTable[ 2 ][ 2 ][ 2 ] = {       <-- 1
   // !b!c !bc b!c bc
        0, 3, 2, 2, // !a
        1, 2, 1, 1 // a
};

...

category = categoryTable[ a ][ b ][ c ];
```

Iako je tabela nešto teža za razumevanje, ona je dobro dokumentovana i neće biti ništa teža za čitanje od komlikovanog lanca logike. Ako se definicija promeni, tabela će biti daleko lakša za održavanje od komplikovane logike. Evo rezultata performansi:

Jezik	Osnovno vreme	Vreme sa optimizovanim kodom	Ušteda	Odnos performansi
C++	5.04	3.39	33%	1.5:1
Visual Basic	5.21	2.60	50%	2:1

Tabela 13. Tabela umesto komplikovanog lanca logike

KORISTITE "LENJU" PROCENU (*LAZY EVALUATION*)

Jedan od bivših cimera autora je bio čovek koji često okleva da nešto uradi. Opravdavao je svoju lenjost rečima da mnoge stvari koje se u prvi trenutak čine važnima i bivaju proglašene da su hitne naprosto ne treba da se urade. Tvrdio je, da ako čeka dovoljno dugo, da bi se stvari koje nisu tako važne prosto otegle do večnosti i da on ne treba, a i neće da gubi vreme na njih.

Lenja procena se zasniva na principu koji je koristio cimer. Ako program koristi lenju procenu, on izbegava da vrši operacije, odnosno da vrši neki posao, sve do onog trenutka kada je stvarno neophodno da se posao uradi. Lenja procena je slična *just-in-time* strategijama koje obavljaju posao samo trenutak pre nego što je njihov rezultat neophodan.

Pretpostavimo, na primer, da vaš program sadrži tabelu sa 5000 vrednosti i da generiše kompletnu tabelu prilikom startovanja. Pretpostavimo i da koristi vrednosti iz te tabele tokom svog izvršavanja. Ako program koristi samo mali broj od ovih 5000 unosa iz tabele, možda bi imalo više smisla da ih računate tek u onom trenutku kada je potrebno da se iskoriste, pre nego da ih inicijalizujete sve odjednom. Jednom, kada se unos izračuna, on se može sačuvati u posebnom delu namenjenom za korišćene vrednosti za buduće potrebe (inače poznato kao "keširanje"). Na ovaj način se ne vrše računanja onih vrednosti koja neće biti korišćena u programu, čime se skraćuje ukupno vreme izvršavanja aplikacije, a i bolje distribuira opterećenje resursa.

PETLJE

Usled činjenice da se petlje izvršavaju veliki broj puta, uska grla u programima se često nalaze baš u petljama. Pogledaćemo nekoliko tehnika koje petlje čine bržima.

RAZBIJANJE CIKLUSA (*UNSWITCHING*)

Prebacivanje (*switching*) se odnosi na donošenje odluka unutar petlji svaki put kada se petlja izvrši. Ako se odluka ne menja tokom izvršavanja petlje, možete da „razbijete" petlju (*unswitch*) donošenjem odluke izvan petlje. Obično, ovo zahteva okretanje petlje iznutra ka spolja, odnosno stavljenje petlje unutar uslova umesto stavljenje uslova unutar petlje. Evo jednog primera petlje pre „razbijana" u jeziku C++:

```
for ( i = 0; i < count; i++ ) {
   if ( sumType == SUMTYPE_NET ) {
      netSum = netSum + amount[ i ];
   }
   else {
      grossSum = grossSum + amount[ i ];
   }
}
```

U gornjem kodu, upit *if (sumType == SUMTYPE_NET)* se ponavlja u svakoj iteraciji petlje, iako će rezultat biti isti svaki put kada se prođe kroz petlju. Ako imamo takvu situaciju, možemo da preradimo gornju petlju na sledeći način, mada moramo imati na umu da ovakva konstrukcija narušava nekoliko pravila „dobrog" programiranja, a to su recimo čitljivost koda i lakoća održavanja:

```
if ( sumType == SUMTYPE_NET ) {
   for ( i = 0; i < count; i++ ) {
      netSum = netSum + amount[ i ];
   }
}
else {
   for ( i = 0; i < count; i++ ) {
      grossSum = grossSum + amount[ i ];
   }
}
```

Loša strana ovakvog rešenja leži u činjenici da se dve petlje moraju održavati paralelno. Ako se recimo promenljiva *count* promeni u promenljivu *clientCount*, morate je promeniti na dva mesta. Ovo morate držati na umu što zna da bude umarajuće, a i povećava šanse za greškama, kao i to da komplikuje održavanje. Ovaj primer takođe ilustruje ključni izazov u optimizaciji koda, a to je da **efekat bilo koje konkretne optimizacije nije predvidiv**. Kao što možemo videti u tabeli 14, ovakva optimizacija daje značajno ubrzanje od 20% u 3 programska jezika, ali ne i u Visual Basic-u. Ovakva optimizacija, iako se može očekivati da uvek daje konkretna ubrzanja, na ovom primeru u Visual Basic-u ne bi donela ubrzanja, ali bi čitljivost sigurno bila lošija. **Uvek moramo meriti rezultate optimizacija!**

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu
C++	2.81	2.27	19%
Java	3.97	3.12	21%
Visual Basic	2.78	2.77	<1%
Python	8.14	5.87	28%

Tabela 14. Razbijanje petlje (*Unswitching*)

SPAJANJE PETLJI (*LOOP JAMMING* ILITI *LOOP FUSION*)

Spajanje petlji (*jamming*) je tehnika u kojoj se spajaju tj. kombinuju dve petlje koje operišu nad istim skupom podataka, odnosno elemenata. Ovde dobitak leži u odbacivanju nepotrebnih iteracija obe petlje i završavanju posla jednim prolazom kroz skup podataka. Pogledaćemo primer pogodnog kandidata za spajanje petlji u Visual Basic-u:

```
For i = 0 to employeeCount - 1
   employeeName( i ) = ""
Next

...

For i = 0 to employeeCount - 1
   employeeEarnings( i ) = 0
Next
```

Kada spajate petlje, vi tražite kod u dve petlje koji ste u stanju da iskombinujete u jednu petlju. Obično, to znači da brojači petlji moraju da budu isti. U ovom primeru, obe petlje kreću od 0 i „vrte" se do *employeeCount – 1*, tako da možemo da ih spojimo u jednu petlju:

```
For i = 0 to employeeCount - 1
   employeeName( i ) = ""
   employeeEarnings( i ) = 0
Next
```

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu
C++	3.68	2.65	28%
PHP	3.97	2.42	32%
Visual Basic	3.75	3.56	4%

Tabela 15. Uštede nastale spajanjem petlji (*employeeCount* je u testiranom slučaju jednak 100)

Kao i ranije, rezultati značajno variraju u zavisnosti od jezika. Spajanje petlji nosi sa sobom najmanje 2 značajne opasnosti. Prvo, indeksi za spojene petlje bi tokom vremena mogli da se promene, tako da bi te osnovne petlje postale nekompatibilne. Drugo, možda nećete moći tako lako da spojite petlje koje na prvi pogled deluju kao odlični kandidati za spajanje. Pre nego što spojite petlje, budite sigurni da će delovi koda koji se spajaju biti u istom rasporedu (sa ispunjenim istim međuzavisnostima kao i početne petlje) u odnosu na ostali kod.

RAZVIJANJE PETLJE (*UNROLLING*)

Cilj "razvijanja" petlje jeste da se smanji količina održavanja petlje. U jednom od prethodnih primera, petlja je bila u potpunosti razvijena da bi se pokazalo da je 10 linija koda razvijene petlje brže od 3 linije koda „normalne petlje". U tom slučaju, petlja je „porasla" sa 3 na 10 linija koda da bi se svih 10 pristupa elementima niza uradilo pojedinačno.

Iako je potpuno razvijanje petlje brzo rešenje koje pri tome i radi dobro kada baratate malim brojem elemenata, ono nije praktično kada se bavite velikim brojem elemenata koje treba obraditi ili kada unapred ne znate koliko ćete elemenata (iteracija) imati. Pogledajmo generalni primer petlje napisane u jeziku Java koji je moguće razviti (*unroll*):

```
i = 0;        <-- 1

while ( i < count ) {
   a[ i ] = i;
   i = i + 1;
}
```

Ovakvu petlju bi ste verovatno najčešće realizovali korišćenjem for naredbe, ali da bi je optimizovali, potrebno je da je konvertujete u *while* petlju (kao ovde). Ako želite da razvijete petlju delimično, treba da obradite dve ili više situacija u svakoj iteraciji petlje umesto jedne situacije. Ovakvo razvijanje petlje šteti čitljivosti koda, ali generalno ne stvara druge probleme samoj petlji. Sledi primer petlje koja je razvijena jednom na osnovu gornje:

```
i = 0;
while ( i < count - 1 ) {
   a[ i ] = i;
   a[ i + 1 ] = i + 1;
   i = i + 2;
}

if ( i == count ) {        <-- 1
   a[ count - 1 ] = count - 1;|
}        <-- 1
```

Ovom tehnikom smo zamenili originalnu liniju *a[i] = i* sa dve linije, a promenljiva *i* se inkrementira za 2 umesto za 1. Na taj način smo dobili petlju koja ima 2 puta manje iteracija, a dodatni kod nakon *while* petlje je neophodan kada je brojač neparan broj, a petlji je ostalo da odradi još jednu iteraciju nakon što se okonča.

Kada se 5 linija standardnog koda razvije u 9 linija "optimizovanog", on postaje teži za razumevanje i održavanje. Osim dobitka u brzini, kvalitet koda je loš. Međutim, deo svake discipline koja se bavi dizajnom jeste povremeno pravljenje neophodnih kompromisa. Tako da, iako određena tehnika predstavlja deo loše programerske prakse, specifična situacija može od vas zahtevati da je upotrebite radi postizanja određenih ciljeva, najčešće brzine ili veličine. Pogledajmo rezultate razvijanja petlje (promenljiva *count* iznosi 100):

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu
C++	1.75	1.15	34%
Java	1.01	0.581	43%
PHP	5.33	4.49	16%
Python	2.51	3.21	-27%

Tabela 16. Uštede nastale razvijanjem petlje jedanput

Dobitak u rasponu od 16 do 43 procenta je značajan, mada morate paziti i na potencijalnu redukciju performansi, kao u slučaju jezika Python. Glavni nedostatak razvijanja petlji jeste obuhvatanje slučaja nakon završetka petlje u situaciji gde je brojač takav. Prirodno se postavlja pitanje, šta ako odemo dalje, sa dva ili više razvoja? Pogledajmo prethodni primer, ali razvijen dva puta:

```
i = 0;

while ( i < count - 2 ) {
   a[ i ] = i;
   a[ i + 1 ] = i+1;
   a[ i + 2 ] = i+2;
   i = i + 3;
}

if ( i <= count - 1 ) {
   a[ count - 1 ] = count - 1;
}

if ( i == count - 2 ) {
   a[ count -2 ] = count - 2;
}
```

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu
C++	1.75	1.01	42%
Java	1.01	0.581	43%
PHP	5.33	3.70	31%
Python	2.51	2.79	-12%

Tabela 17. Petlja razvijena dva puta, broj iteracija je 100

Gornji rezultati govore da dalje razvijanje petlje može, ali i ne mora doneti uštede u vremenu. Ako krenete u pravcu daljeg razvoja, glavna briga postaje izgled vašeg koda. Iako ovaj kod ne izgleda preterano komplikovano, podsetimo se da je petlja startovala sa 5 linija koda. U svakom slučaju, odaberite pravi odnos (za vaš slučaj) između brzine i čiljivosti koda.

MINIMIZIRAJTE POSAO KOJI SE OBAVLJA UNUTAR PETLJI

Jedan od ključeva za pisanje efikasnih petlji jeste minimiziranje posla koji se obavlja unutar same petlje. Ako imate mogućnost da ocenite iskaz ili deo iskaza izvan petlje na način da se samo rezultat koristi u petlji, tako i uradite. To je dobra programerska praksa, a u nekim situacijama čak i poboljšava čitljivost koda. Recimo da imate komplikovani izraz koji koristi pokazivače (pointere) unutar često korišćene petlje (u jeziku C++):

```
for ( I = 0; I < rateCount; i++ ) {
   netRate[ I ] = baseRate[ I ] * rates->discounts->factors->net;
}
```

U ovom slučaju, dodeljivanje komplikovanog pokazivačkog izraza pravilno imenovanoj promenljivoj poboljšava čitljivost koda, a u nekim situacijama i brzinu. Pogledajmo izmenu:

```
quantityDiscount = rates->discounts->factors->net;

for ( i = 0; i < rateCount; i++ ) {
   netRate[ i ] = baseRate[ i ] * quantityDiscount;
}
```

Dodatna promenljiva, *quantityDiscount*, sada jasno govori da se članovi niza *baseRate* množe sa faktorom popusta na količinu da bi se izračunala neto stopa. To nije bilo sasvim očigledno iz originalnog izraza koji se nalazio u petlji. Dodeljivanje vrednosti komplikovanog pokazivačkog izraza promenljivoj izvan petlje "štedi" i vreme koje je potrebno da se pokazivač dereferencira tri puta za svaki prolaz kroz petlju, rezultujući sledećim uštedama:

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu
C++	3.69	2.97	19%
C#	2.27	1.97	13%
Java	4.13	2.35	43%

Tabela 18. Minimiziranje koda unutar petlje, rateCount je 100

Osim za Javu (i njen kompajler), uštede koje se postižu nisu prevelike, tako da tokom inicijalnog kodiranja možete da koristite bilo koju tehniku koja je čitljivija, bez brige o brzini koda. Naravno, sve do trenutka kada brzina postane kritičan faktor.

MARKERSKE VREDNOSTI (*SENTINEL VALUES*)

Kada imate petlju sa složenim uslovom, vrlo često ćete biti u situaciji da uštedite vreme tako što ćete pojednostaviti uslov. Ako se radi o petlji za pretragu (*search loop*), jedan od načina da pojednostavite uslov jeste da iskoristite signalnu (markersku) vrednost, vrednost koju postavite odmah iza kraja opsega za pretragu i koja garantovano prekida pretragu. Klasičan primer složenog uslova koji je moguće poboljšati upotrebom signalne vrednosti je petlja za pretragu koja proverava da li je pronađena tražena vrednost i da li je petlja iscrpela sve slučajeve koje treba obraditi. Evo primera u jeziku C#:

```
found = FALSE;
i = 0;

while ( ( !found ) && ( i < count ) ) {        <-- složeni uslov
   if ( item[ i ] == testValue ) {
      found = TRUE;
   }
   else {
      i++;
   }
}

if ( found ) {
   ...
```

U gornjem kodu, svaka iteracija petlje testira delove (*!found*) i (*i* < *count*). Svrha testa (*!found*) jeste da odredi da li je traženi element pronađen. Svrha testa (*i* < *count*) jeste da spreči prelazak kraja niza koji se pretražuje. Unutar petlje, svaka vrednost niza *item* se testira pojedinačno, tako da petlja realno vrši 3 testa u svakoj iteraciji.

U ovakvoj petlji za pretraživanje, možemo da kombinujemo ova tri testa na način da se izvrši samo jedno testiranje po iteraciji stavljenjem markerske vrednosti na kraj opsega za pretragu da bi se petlja zaustavila. U ovom slučaju, možemo jednostavno da dodelimo vrednost koju tražimo kao dodatni element odmah posle kraja opsega za pretragu (upamtimo da ostavimo još jedno mesto za taj element kada deklarišemo sam niz). Sada, proveravamo svaki element, od početka do kraja niza, i ako ne nađemo takav element sve do onog kojeg

smo postavili na kraj niza, znamo da vrednost koju tražimo ne postoji u originalnom nizu. Pogledajmo prerađeni deo gornjeg koda sa dodatom markerskom vrednošću:

```
// postavljamo markersku (traženu) vrednost, čuvamo originalnu vrednost
initialValue = item[ count ];
item[ count ] = testValue;       <-- 1
i = 0;

while ( item[ i ] != testValue ) {
   i++;
}

// proveravamo da li je pronađena vrednost (ako je i < count onda je nađena)
if ( i < count ) {
   ...
```

U zavisnosti od tipa podataka koji se pretražuju, uštede variraju, ali ako se radi o recimo celim brojevima razlike mogu biti dramatične. Ova tehnika se može upotrebiti u bilo kojoj situaciji u kojoj koristite linearnu pretragu (bilo da su to nizovi ili povezane liste). Jedina stvar na koju morate obratiti pažnju jeste izbor same markerske vrednosti, kao i to da morate biti oprezni kako postavljate markersku vrednost u strukturu podataka.

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu	Odnos performansi
C#	0.771	0.590	23%	1.3:1
Java	1.63	0.912	44%	2:1
Visual Basic	1.34	0.470	65%	3:1

Tabela 19. Pretraga niza celih brojeva (100 elemenata) upotrebom markerske vrednosti

POSTAVLJANJE NAJČEŠĆE KORIŠĆENE PETLJE UNUTRA

Kada radite sa ugnježdenim petljama, morate razmisliti koje petlje želite sa spoljne strane, a koje sa unutrašnje. Sledi primer ugnježdenih petlji koje je moguće poboljšati u Javi:

```
for ( column = 0; column < 100; column++ ) {
   for ( row = 0; row < 5; row++ ) {
      sum = sum + table[ row ][ column ];
   }
}
```

Ključna stvar kod ovakvih petlji jeste da se spoljna petlja izvršava mnogo češće nego unutrašnja. Svaki put kad se petlja izvrši, ona mora da inicijalizuje brojač petlje, da ga inkrementira sa svakom iteracijom i da proveri brojač nakon svakog prolaza. Ukupan broj izvršavanja petlji je 100 za spoljnu i 100 x 5 = 500 za unutrašnju, što daje ukupno 600 iteracija. Samom zamenom unutrašnje i spoljne petlje, možemo da promenimo broj iteracija za spoljnu petlju na 5 i 5 x 100 = 500 za unutrašnju, što daje ukupno 505 iteracija. Analitički gledano, moće se očekivati oko (600 – 505) / 600 = 16% manje iteracija samom zamenom petlji. Pogledajmo i izmerene performanse:

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu
C++	4.75	3.19	33%
Java	5.39	3.56	34%
PHP	4.16	3.65	12%
Python	3.48	3.33	4%

Tabela 20. Poboljšanje performansi nastalo zamenom pozicija ugnježdenih petlji

SMANJENJE SNAGE (*STRENGTH REDUCTION*)

Smanjenje snage zapravo podrazumeva zamenu „skupih" operacija kao što je množenje „jeftinijim" operacijama kao što je sabiranje. Ponekad imamo situaciju da se unutar petlje nalazi izraz koji zavisi od množenja brojača petlje određenim faktorom. Sabiranje je skoro uvek brže od množenja, pa ako možete da izračunate isti broj dodavanjem prilikom svake iteracije petlje, a ne množenjem, kod će se brže izvršavati. Pogledajmo primer koda koji koristi množenje u jeziku Visual Basic:

```
For i = 0 to saleCount - 1
   commission( i ) = (i + 1) * revenue * baseCommission * discount
Next
```

Ovakav kod je intuitivan, ali i skup. Možemo ponovo da napišemo petlju tako da akumulira množioce umesto da ih računa svaki put. Ovo smanjuje „snagu" operacije od množenja ka sabiranju. Pogledajmo prerađenu petlju:

```
incrementalCommission = revenue * baseCommission * discount
cumulativeCommission = incrementalCommission

For i = 0 to saleCount - 1
   commission( i ) = cumulativeCommission
   cumulativeCommission = cumulativeCommission + incrementalCommission
Next
```

Originalni kod je inkrementirao *i* u svakoj iteraciji i množio sa (*revenue* * *baseCommission* * *discount*) , prvo sa 1, pa sa 2, pa 3 itd. Optimizovani kod definiše promenljivu *incrementalCommission* i inicijalizuje je (*revenue* * *baseCommission* * *discount*). Onda dodeljuje vrednost *incrementalCommission* promenljivoj *cumulativeCommission* svakim prolazom kroz petlju. Kod prvog prolaza, dodeljuje je jedan put, kod drugog - dodato je dva puta, kod trećeg - tri puta itd. Efekat je potpuno isti kao i kada bi se *incrementalCommission* množilo sa 1, 2, 3…, ali je u smislu resursa jeftinije. Ključ je u tome da originalno množenje mora da zavisi od brojača petlje. U ovom slučaju, brojač petlje je samo deo izraza koji se menja, tako da je sam izraz moguće napisati ponovo na opisani način, ali ekonomičnije.

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu
C++	4.33	3.80	12%
Visual Basic	3.54	1.80	49%

Tabela 21. Ušteda postignute smanjenjem snage operacije, 20 iteracija petlje

TRANSFORMACIJE PODATAKA

Promena tipa podataka može da bude velika pomoć u smanjenju veličine i poboljšanju brzine izvršavanja programa. Dizajn struktura podataka sam za sebe je krupna i značajna oblast i izlazi iz domena ovoga rada, ali skromne promene u implementaciji tipova podataka mogu značajno uticati na performanse. Pogledaćemo neke od tehnika koje mogu da pomognu u optimizaciji aplikacija.

KORISTITE CELE BROJEVE UMESTO BROJEVA SA POKRETNIM ZAREZOM

Sabiranje, kao i množenje, celih brojeva je daleko brže nego kada se radi sa brojevima sa pokretnim zarezom. Ako recimo imamo situaciju da je brojač petlje realan broj, njegova promena u celi broj (kada je to moguće), može značajno ubrzati petlju. Pogledajmo primer petlje u Visual Basic-u koja kao brojač koristi realan broj (*floating point*):

```
Dim x As Single

For x = 0 to 99
   a( x ) = 0
Next
```

Naravno, pošto je index petlje takav da ga je moguće zameniti celim brojem, probajmo petlju sa indexom koji je ceo broj (*integer*), dok se u donjoj tabeli mogu videti i razlike u brzini izvršavanja između ovih petlji:

```
Dim i As Integer

For i = 0 to 99
   a( i ) = 0
Next
```

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu	Odnos performansi
C++	2.80	0.801	71%	3.5:1
PHP	5.01	4.65	7%	1:1
Visual Basic	6.84	0.280	96%	25:1

Tabela 22. Razlika u brzini petlje sa brojačem koji je ralan ili ceo broj

KORISTITE NAJMANJI MOGUĆI BROJ DIMENZIJA NIZA

Konvencionalna mudrost kaže da je upotreba višestrukih dimenzija niza "skupa". Ako imate mogućnost da strukturirate vaše podatke tako da se nađu u jedno-dimenzionalnom nizu umesto dvo-dimenzionalnom ili tro-dimenzionalnom, možda ćete biti u mogućnoti da uštedite nešto operacija tj. vremena. Recimo da imamo kod koji inicijalizuje podatke na sledeći način (jezik Java, dvo-dimenzionalni niz):

```
for ( row = 0; row < numRows; row++ ) {
   for ( column = 0; column < numColumns; column++ ) {
      matrix[ row ][ column ] = 0;
   }
}
```

Kada se ova petlja pokrene sa 50 redova i 20 kolona, njoj je potrebno duplo više vremena nego kada je kod strukturiran tako da koristi jedno-dimenzionalnu petlju. Pogledajmo revidirani kod koji predstavlja jedno-dimenzionalnu reprezentaciju niza i rezultate merenja:

```
for ( entry = 0; entry < numRows * numColumns; entry++ ) {
   matrix[ entry ] = 0;
}
```

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu	Odnos performansi
C++	8.75	7.82	11%	1:1
C#	3.28	2.99	9%	1:1
Java	7.78	4.14	47%	2:1
PHP	6.24	4.10	34%	1.5:1
Python	3.31	2.23	32%	1.5:1
Visual Basic	9.43	3.22	66%	3:1

Tabela 23. Rezultat smanjenja broja dimenzija niza

Rezultat ovakve optimizacije je u Visual Basicu i Javi odličan, dobar u PHP i Python-u, ali osrednji u C++ i C#. Naravno, ne treba bit preoštar prema C# jer je on već imao najbolje vreme sa neoptimizovanim kodom, a i nakon optimizacije je u ostao u samom vrhu. Veoma veliki raspon rezultata i promena kod ove optimizacije još jednom govori da savete za optimizaciju ne treba slepo pratiti, već konstantno meriti rezultate.

MINIMIZIRAJTE REFERENCIRANJE NIZA

Pored preporučenog smanjenja pristupa dvo, tro i više-dimenzionalnim nizovima, često će biti povoljno po performanse da minimizirate pristup nizovima uopšte. Petlja koja u više navrata koristi jedan element niza je odlčan kandidat za primenu ove tehnike. Pogledajmo primer nepotrebnog referenciranja niza u jeziku C++:

```
for ( discountType = 0; discountType < typeCount; discountType++ ) {
   for ( discountLevel = 0; discountLevel < levelCount; discountLevel++ ) {
      rate[ discountLevel ] = rate[ discountLevel ] * discount[ discountType ];
   }
}
```

Referenca na *discount[discountType]* se ne menja prilikom promene *discountLevel* u unutrašnjoj petlji. Pošto je takva situacija, možemo da pomerimo *discount[discountType]* izvan unutrašnje petlje tako da imamo samo jedan pristup tom nizu prilikom izvršavanja spoljne petlje umesto da imamo po jedan pristup prilikom svakog izvršavanja unutrašnje petlje:

```
for ( discountType = 0; discountType < typeCount; discountType++ ) {
   thisDiscount = discount[ discountType ];

   for ( discountLevel = 0; discountLevel < levelCount; discountLevel++ ) {
      rate[ discountLevel ] = rate[ discountLevel ] * thisDiscount;
   }
}
```

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu
C++	32.1	34.5	-7%
C#	18.3	17.0	7%
Visual Basic	23.2	18.4	20%

Tabela 24. Rezultati pomeranja referenciranog elementa niza, *typeCount* je 10, *levelCount* je 100

KORISTITE DODATNE INDEKSE

Korišćenje dodatnih indeksa podrazumeva dodavanje povezanih podataka koji čine da pristupanje određenoj strukturi bude efikasnije. Dodavanje podataka (indeksa) može da se izvrši na glavnoj strukturi ili se može izvršiti na paralelnoj (za to optimizovanoj) strukturi:

- **Indeks za dužinu stringa (*string-length index*):** Jedan od primera korišćenja dodatnih indeksa se može naći kod različitih strategija za čuvanje stringova: U C-u, stringovi se završavaju bajtom koji je postavljen na vrednost 0. U formatu stringova kod Visual Basic-a, skriveni bajt se nalazi na početku stringa i on govori koliko je string dugačak. Kako bi utvrdili dužinu stringa u C-u, program mora da krene od početka stringa I da broji svaki bajt dok ne stigne do bajta koji je setovan na 0. Sa druge strane, kako bi utvrdio dužinu stringa u Visual Basic-u, program samo pogleda u informaciju o dužini stringa zapisanu u početnom bajtu. Bajt sa dužinom stringa u Visual Basic-u je primer uvećanja strukture podataka indeksom koji pomaže ubrzanju određenih operacija (kao što je recimo računanje dužine stringa). Ideju o ovakvim indeksiranjima možete da primenite na bilo kojem tipu podataka koji ima promenljivu dužinu. Često je efikasnije da informaciju o dužini strukture sačuvate u njoj samoj nego da je izračunavate svaki put kad vam je potrebna.
- **Nezavisna, paralelna struktura indeksa:** Ponekad je efikasnije da manipulišete indeksom nekih podataka nego samim podacima. Ako su elementi strukture podataka veliki ili zahtevni za premeštanje (recimo na tvrdom disku), sortiranje i pretraga referenci indeksa je daleko brža nego sortiranje i pretraga samih podataka. Ako su sami elementi (podaci) u strukturi glomazni, možete da kreirate dodatnu strukturu koja se sastoji od „lakih" podataka koja sadrži ključne vrednosti i pokazivače (pointere) na osnovnu strukturu za detaljne informacije. Ako je razlika u veličini osnovne strukture u kojoj se nalaze podaci i strukture sa indeksima dovoljno velika, ponekad možete ključne vrednosti da čuvate i u radnoj memoriji. Svako sortiranje i indeksiranje se tada izvodi u radnoj memoriji i izuzetno je brzo. Tek kada ste to uradili (saznali ste tačnu lokaciju neophodnog elementa), potrebno je pristupiti tvrdom disku za detaljne podatke.

KORISTITE KEŠIRANJE (CACHING)

Keširanje predstavlja tehniku čuvanja određenog seta vrednosti na takav način da se zapisima koje se najčešće koriste lakše i brže pristupa nego manje korišćenim zapisima. Ako recimo program koristi nasumične podatke sa tvrdog diska, rutina bi mogla da kešira (sačuva) zapise koji se najčešće koriste. Kada rutina primi zahtev za određenim zapisom, ona proveri da li ima sačuvanu tu vrednost u kešu. Ako je tu, zapis se uzima direktno odatle, a to je obično iz radne memorije, a ne iz osnovne strukture gde se nalaze svi zapisi (obično na tvrdom disku).

Pored keširanja podataka na disku, ono (keširanje) se može primeniti i u mnogim drugim oblastima. Kod Microsoft Windows-a, program za proveru fontova je imao usko grlo kod dohvatanja širine svakog prikazanog karaktera. Keširanje širine najskorije korišćenog karaktera je bukvalno dupliralo brzinu priakaza.

Keširanje je još veoma pogodno za čuvanje rezultata vremenski zahtevnih izračunavanja, posebno ako su parametri za kalkulaciju jednostavni. Pretpostavimo, na primer, da vam je potrebno izračunavanje dužine hipotenuze pravogulog trougla, kada su vam poznate dužine kateta. Pogledajmo kako bi mogla da izgleda stnadardna implementacija ove rutine u jeiku Java:

```
double Hypotenuse(double sideA, double sideB) {
   return Math.sqrt( ( sideA * sideA ) + ( sideB * sideB ) );
}
```

Ako znate da se vrednosti kateta često ponavljaju, možete da keširate rezultat izračunavanja na sledeći način:

```
private double cachedHypotenuse = 0;
private double cachedSideA = 0;
```

```
private double cachedSideB = 0;

public double Hypotenuse(double sideA, double sideB) {

   // proverava da li se trougao vec nalazi u kesu
   if ( ( sideA == cachedSideA ) && ( sideB == cachedSideB ) ) {
      return cachedHypotenuse;
   }

   // izracunava novu hipotenuzu i kesira je
   cachedHypotenuse = Math.sqrt( ( sideA * sideA ) + ( sideB * sideB ) );
   cachedSideA = sideA;
   cachedSideB = sideB;

   return cachedHypotenuse;
}
```

Druga verzija rutine je komplikovanija nego prva i zauzima više prostora i memorije, tako da brzina treba da bude izrazit prioritet da bi se opravdala ovakva implementacija. Mnoge šeme za keširanje koriste više od jednog elementa, tako da one umeju da prave još veći *overhead*. Pogledajmo i razlike u brzini između ovih verzija rutine:

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu	Odnos performansi
C++	4.06	1.05	74%	4:1
Java	2.54	1.40	45%	2:1
Python	8.16	4.17	49%	2:1
Visual Basic	24.0	12.9	47%	2:1

Tabela 25. Brzina normalne i keširane rutine, pretpostavlja se da je keš pogođen 2 puta svaki put kada se setuje

Uspeh keširanja zavisi od relativne cene pristupanja keširanom elementu, kreiranja nekešerianog elementa i snimanja novog elementa u keš. Uspeh dalje značajno zavisi i od toga koliko često se keširana informacija potražuje. U nekim slučajevima, uspešnost može zavisiti i od keširanja koje se izvodi u hardveru. Generalno, što je „skuplje" generisanje novog elementa i što se više puta ista informacija potražuje, keš je vredniji tj. daje bolje rezultate. Dalje, što je lakši pristup elementima keša i snimanje elementa u keš, keš daje veću brzinu. Kao i sa drugim optimizacionim tehnikama, keširanje povećava kompleksnost koda, kao i mogućnost grešaka u kodu (koje se teže detektuju).

IZRAZI

Veliki deo programa se odvija unutar matematičkih ili logičkih izraza. Što su izrazi komplikovaniji, to više resursa i vremena zauzimaju, pa ćemo pogledati načine na koje možemo da ih učinimo „jeftinijim".

ISKORISTITE ALGEBARSKE IDENTITETE

Možete da iskoristite algebarske identitete kako bi zahtevnije operacije zamenili manje zahtevnim. Na primer, sledeći izrazi su ekvivalentni:

```
not a and not b
not (a or b)
```

Ako izaberete da iskoristite drugi izraz umesto prvog uštedećete jednu *not* operaciju. Iako je ušteda od jedne *not* operacije zanemarljiva, generalni princip je veoma moćan. *John Bentley* opisuje program koji testira da li je *sqrt(x) < sqrt(y)*. Kako je *sqrt(x)* manje od *sqrt(y)* isključivo kada je i *x* manje od *y*, test možete da zamenite sa *x < y*. Poznajući cenu rutine *sqrt()*, očekivano je da će uštede biti dramatične. One to i jesu:

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu	Odnos performansi
C++	7.43	0.010	99.9%	750:1
Visual Basic	4.59	0.220	95%	20:1
Python	4.21	0.401	90%	10:1

Tabela 26. Zamena uslova uz poznavanje algebarskog odnosa

KORISTITE REDUKCIJU SNAGE (STRENGTH REDUCTION)

Kao što smo pomenuli ranije, redukcija snage predstavlja zamenu skupe operacije jefitnijom. Evo nekih od mogućih zamena u tom smislu:

- Zamenite množenja sabiranjima;
- Zamenite stepenovanja množenjima;
- Zamenite trigonometrijske funkcije njihovim trigonometrijskim identitetima;
- Zamenite celobrojni tip *longlong* kraćim celobrojnim tipom (*long* ili *int*). Kako su celobrojni tipovi često osnovni (*native*) tipovi određenih jezika, treba pripaziti na performanse prilikom korišćenja *native* tipova naspram ne-*native* celobrojnih vrednosti;
- Zamenite brojeve u pokretnom zarezu brojevima fiksne širine ili celim brojevima;
- Zamenite brojeve u pokretnom zarezu duple preciznosti (*double precision*) brojevima u pokretnom zarezu jednostruke preciznosti (*single precision*);
- Zamenite množenja i deljenja celih brojeva sa 2 adekvatnim operacijama sa bitovima.

Recimo da treba da izračunate polinom. Opšti kod za računanje polinoma n-tog reda izgleda ovako (jezik *Visual Basic*):

```
value = coefficient( 0 )

For power = 1 To order
   value = value + coefficient( power ) * x^power
Next
```

Ako razmišljate o redukciji snage, u ovom kodu ćete gledati u deo koji izvršava stepenovanje. Jedno od rešenja bi mogla da bude zamena stepenovanja množenjima u svakom prolazu kroz petlju, što je praktično analaogno redukciji snage koju smo imali ranije – zamena množenja sabiranjem. Evo kao bi izgledalo računanje polinoma sa redukcijom snage:

```
value = coefficient( 0 )
powerOfX = x

For power = 1 to order
   value = value + coefficient( power ) * powerOfX
   powerOfX = powerOfX * x
Next
```

Ovakva optimizacija daje značajne rezultate ako radite sa polinomima drugog reda, odnosno polinomima čiji je najviši stepen kvadrat celog broja, ili sa polinomima viših redova:

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu	Odnos performansi
Python	3.24	2.60	20%	1:1
Visual Basic	6.26	0.160	97%	40:1

Tabela 27. Redukcija snage, promena operacije sa stepenovanja na množenje

Međutim, tu nije kraj. Ako dalje primenite princip redukcije snage u petlji, možete da sumirate stepene, a ne da ih množite svaki put, čime bi eliminisali dodatnu promenljivu *powerOfX* i zamenili je sa dva množenja u svakom prolazu kroz petlju umesto jednog:

```
value = 0

For power = order to 1 Step -1
   value = ( value + coefficient( power ) ) * x
Next

value = value + coefficient( 0 )
```

Jezik	Standardno vreme	Prva optimizacija	Druga optimizacija	Uštede u odnosu na prvu optimizaciju
Python	3.24	2.60	2.53	3%
Visual Basic	6.26	0.16	0.31	-94%

Tabela 28. Dalja redukcija snage, dvostruko množenje uz izbacivanje promenljive koja čuva stepen X

Rezultati u ovoj tabeli su pomalo neočekivani i dobar su primer kako se teorija ne realizuje u praksi. Drugi kod sa redukovanog snagom bi trebalo da je brži, ali nije.

INICIJALIZUJTE U VREME KOMPAJLIRANJA

Ako koristite imenovanu konstantu ili magični broj (*magic number*) u pozivu rutine i to je njen jedini argument, to je znak da možete da izračunate broj odnosno povratnu vrednost rutine, stavite je u konstantu i kompletno izbegne poziv rutine. Isti princip se odnosi na množenje, deljenje, sabiranje i druge operacije.

Pogledajmo primer računanja logaritma celog broja sa osnovom 2, koji je odsečen do najbližeg celog broja. Ako iskoristimo činjenicu da je *log(x)base = log(x) / log(base)*, možemo napisati rutinu u jeiku C++ na sledeći način:

```
unsigned int Log2( unsigned int x ) {
   return (unsigned int) ( log( x ) / log( 2 ) );
}
```

Ovako napisana rutina je zaista spora, a kako se vrednost *log(2)* praktično nikada ne menja, možemo je zameniti sa preračunatom vrednošću 0,69314718 na sledeći način:

```
const double LOG2 = 0.69314718;

unsigned int Log2( unsigned int x ) {
   return (unsigned int) ( log( x ) / LOG2 );
}
```

Kako je *log()* skupa rutina (daleko skuplja nego obična konverzija ili deljenje), mogli bi ste da očekujete da će prepolovljavanje broja poziva rutine *log()* smanjiti vreme izvršavanja kompletne rutine za skoro polovinu. Pogledajmo izmerene rezultate:

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu
C++	9.66	5.97	38%
Java	17.0	12.3	28%
PHP	2.45	1.50	39%

Tabela 29. Rezultati inicijalizacije korišćene vrednosti izvan izraza

U ovom slučaju, pretpostavke o relativnoj važnosti za brzinu deljenja i konverzije, kao i pretpostavka o potencijalnom ubrzanju usled smanjenja broja poziva rutine *log()* od 50 procenata tipova su bile prilično tačno. Međutim, ako se pogledaju i ostale pretpostavke o rezultatima optimizacija, možemo reći da smo dokazali jedino to da i „ćorava koka ponekad ubode zrno". Stalno merite rezultate optimizacija!

ČUVAJTE SE SISTEMSKIH POZIVA (RUTINA)

Sistemske rutine su skupe i obezbeđuju tačnost koja često nije neophodna. Tipične sistemske rutine koje se bave matematikom, su dizajnirane tako da mogu da smeste astronauta na Mesec ±60 cm od ciljane lokacije. Ako vam nije neophodan takav nivo preciznosti, nije potrebno da trošite vreme na njihov proračun.

U prethodnom primeru, *Log2()* rutina vraća celobrojnu vrednost, ali koristi rutinu *log()* koja barata brojevima u pokretnom zarezu da bi je izračunala. Takva rutina (*log()*) je prejaka za računanje celobrojnog rezultata. Pogledajmo sada kako se može preraditi Log2 rutina da radi sa celim brojevima u jeziku C++. Ova rutina koristi isključivo celobrojne operatore, nikada ne konvertuje u brojeve sa pokretnim zarezom i u smislu performansi daleko nadmašuje obe prethodne verzije koji koriste brojeve u pokretnom zarezu:

```
unsigned int Log2( unsigned int x ) {
   if ( x < 2 ) return 0;
   if ( x < 4 ) return 1;
   if ( x < 8 ) return 2;
   if ( x < 16 ) return 3;
   if ( x < 32 ) return 4;
   ...
   if ( x < 2147483648 ) return 30;
   return 31;
}
```

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu	Odnos performansi
C++	9.66	0.662	93%	15:1
Java	17.0	0.882	95%	20:1
PHP	2.45	3.45	-41%	2:3

Tabela 30. Funkcija za računanje celobrojnog logaritma osnove 2 koja radi isključivo sa celim brojevima

Većina takozvanih transcedentnih funkcija su dizajnirane tako da tačno rade i u najgorim situacijama, to jest one konvertuju ulazne argumente i rade sa brojevima u pokretnom zarezu duple preciznosti (*double-precision*) interno čak i ako im kao ulazne argumente date cele brojeve. Ako naiđete na neku od ovih (transcedentnih) funkcija u uskom grlu vaše aplikacije, a nije vam potreban taj nivo preciznosti, posvetite adekvatnu pažnju optimizaciji iste.

Druga opcija bi bila da se iskoristi činjenica da je pomeraj u desno (*bitwise right shift*) isto što i deljenje sa 2. Broj možete da podelite sa 2 i da vam ostane vrednost koja nije jednaka nuli isto onoliko puta koliko je i logaritam sa osnovom 2 tog broja. Na osnovu ovog zapažanja možemo da napišemo drugačiju varijantu prethodne rutine (jezik C++):

```
unsigned int Log2( unsigned int x ) {
   unsigned int i = 0;

   while ( ( x = ( x >> 1 ) ) != 0 ) {
      i++;
   }
   return i;
}
```

Programerima koji nisu vešti u programskom jeziku C++, ovaj kod je posebno težak za razumevanje. Komplikovani deo se nalazi u uslovu petlje *while* i predstavlja primer kodiranja koji bi trebalo izbegavati, osim ako nemate posebno dobar razlog da rutine realizujete na ovakav način. Ova rutina se izvršava sporije (i to za 350 procenata) od prethodne, ali ipak značajno brže od prve. Međutim, poslednja varijanta rutine se veoma lako prilagođava 32-bitnim, 64-bitnim i drugim okruženjima, što nije slučaj sa prethodnima.

KORISTITE PRAVILAN TIP KONSTANTI

Koristite imenovane konstante i literale koji su istog tipa kao i promenljive kojima su dodeljenje. Kada su konstanta i povezana promenljiva različitog tipa, kompajler mora da odradi konverziju tipova da bi mogao da izvrši dodelu vrednosti konstante promenljivoj. Kvalitetni kompajleri konverziju tipova završavaju u vreme kompajliranja tako da to ne utiče na performanse u toku rada programa (*run-time*). Manje kvalitetni kompajleri ili interpreteri generišu kod za konverziju u vreme izvršavanja programa, tako da može doći do zagušenja.

PRERAČUNAJTE (*PRECOMPUTE*) REZULTATE

Česta odluka na niskom nivou dizajna jeste izbor da li ćete rezultate odnosno međurezultate računati „u letu" ili ih izračunati, snimiti i uzeti kada i ako za to dođe vreme. Ako se takvi rezultati često koriste u aplikaciji, često je optimalnije (a i brže) izračunati ih jednom, snimiti i koristiti po potrebi.

Ovaj izbor se manifestuje na nekoliko načina. Na najednostavnijem nivou, možete da izračunate deo izraza izvan petlje, a ne u petlji (kao što smo pokazali ranije). Na nešto komplikovanijem nivou, možete da kreirate čitavu tabelu indeksa (ili *data* fajl) jednom kada program krene sa izvršavanjem i koje ćete intenzivno koristiti tokom rada programa.

Kod svemirskih simulacija (video igara), na primer, programeri su u početku računali koeficijent gravitacije za različite udaljenosti od Sunca u realnom vremenu. Računanje koeficijenta gravitacije je bila skupa operacija koja je značajno uticala na ukupne performanse igre. Dubljom analizom, utvrđeno je da se u toku igre distanca od Sunca menja mali broj puta, tako da je realno postajalo samo nekoliko različitih koeficijenata gravitacije. To je značilo da programeri mogu preračunatii koeficijent gravitacije i sačuvati ga u nizu od samo 10 elemenata. Kao što možete da pretpostavite, uzimanje vrednosti iz niza je bilo daleko brže nego stalno računanje koeficijenta gravitacije.

Recimo da imate rutinu koja računa iznose rata za otplatu zajma za automobil. U jeziku Java, programski kod bi mogao da izgleda ovako:

```
double ComputePayment(long loanAmount, int months, double interestRate) {
   return loanAmount /
      (
      ( 1.0 - Math.pow( ( 1.0 + ( interestRate / 12.0 ) ), -months ) ) /
      ( interestRate / 12.0 )
      );
}
```

Ovakva formula za računanje rate zajma je relativno komplikovana i zahtevna. Stavljanje podataka u tabelu umesto stalnog računanja bi verovatno bilo jeftinije za aplikaciju. Kolika bi trebalo da bude velika ta tabela u kojoj bi se čuvale informacije? Promenljiva *interestRate* može biti u opsegu od nekih 5 do 20 procenata, sa korakom od jedne četvrtine (i to je samo 61 različita stopa). Promenljiva *months* može ići od 12 do 72 i to je takođe potencijalnih ukupno 61 različit period. Promenljiva *loanAmount* može da varira ubedljivo najviše i to od recimo 100,000 do 10,000,000 dinara, što je veći broj unosa nego što bi verovatno trebalo čuvati u tabeli indeksa. Veći deo računice ne zavisi od promenljive *loanAmount*, tako da onaj „ružniji" deo kalkulacije (imenilac ukupnog izraza) možete da stavite u tabelu koja je indeksirana vrednostima *interestRate* i *months*:

```
double ComputePayment(long loanAmount, int months, double interestRate) {
   int interestIndex =
      Math.round( ( interestRate - LOWEST_RATE ) * GRANULARITY * 100.00 );

   return loanAmount / loanDivisor[ interestIndex ][ months ];
}
```

U ovom kodu, komplikovani deo kalkulacije je zamenjen sa računanjem indeksa niza i jednim pristupom nizu. Evo i rezultata optimizacije koda:

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu	Odnos performansi
Java	2.97	0.251	92%	10:1
Python	3.86	4.63	-20%	1:1

Tabela 31. Zamena zahtevnog dela preračunatom vrednošću iz niza

Optimizacija programa putem preračunavanja (vrednosti, izraza...) može da ima nekoliko oblika:

- Izračunavanje vrednosti pre nego što program počne da se izvršava i vezivanje vrednosti sa konstantom koja se dodeljuje za vreme kompajliranja;
- Izračunavanje rezultata pre nego što program počne da se izvršava i ručno dodeljivanje promenljivima za vreme izvršavanja programa;
- Izračunavanje rezultata pre nego što program počne da se izvršava i snimanje istih u fajl koji se učitava za vreme izvršavanja programa;
- Izračunavanje rezultata jedanput, na početu programa, i njihovo referenciranje svaki put kada su potrebni;
- Izračunavanje što je moguće više vrednosti pre početka petlje i minimiziranje posla koji se obavlja unutar same petlje;
- Izračunavanje rezultata prvi put kada su potrebni i njihovo čuvanje tako da se mogu pozvati kada budu ponovo neophodni.

ELIMINIŠITE ČESTE PODIZRAZE

Ako imate izraz koji se ponavlja nekoliko puta, dodelite ga promenljivoj i pozivajte tu promenljivu, a ne da računate vrednost izraza svaki put ili na nekoliko mesta. Gorepomenuta kalkulacija rate za zajam u sebi sadrži čest podizraz koji se može eliminisati. Podsetimo se dela originalnog koda:

```
payment = loanAmount / (
      ( 1.0 -- Math.pow( 1.0 + ( interestRate / 12.0 ), -months ) ) /
      ( interestRate / 12.0 )
   );
```

U ovom primeru, podizraz *interestRate*/12.0 možemo da dodelimo promenljivoj koja se poziva 2 puta, a ne da računamo vrednost podizraza 2 puta. Ako pravilno izaberete ime za promenljivu, ova optimizacija ne samo što može poboljšati performanse već i čitljivost koda. Pogledajmo optimizaciju:

```
monthlyInterest = interestRate / 12.0;

payment = loanAmount / (
      ( 1.0 -- Math.pow( 1.0 + monthlyInterest, -months ) ) /  monthlyInterest
   );
```

„Uštede" koje su postignute ovom optimizacijom ne izgledaju impresivno:

Jezik	Standardno vreme	Vreme optimizovanog koda	Ušteda u vremenu
Java	2.94	2.83	4%
Python	3.91	3.94	-1%

Tabela 32. Zamena čestog podizraza promenljivom

Izgleda da je rutina *Math.pow()* u ovom primeru toliko zahtevna da prosto zasenjuje uštede koje bi trebalo da nastanu eliminacijom podizraza. Ili možda sam kompajler eliminiše podizraz? Da je podizraz značajniji deo ukupnog izraza, onda bi ostatak izraza ili kompajler imali manjeg uticaja, pa bi ova optimizacija došla do izražaja. Ovako, možete videti da postoje situacije u kojima i optimizacije koje treba da daju rezutate, to ne čine, usled niza faktora koji utiču na krajnji rezultat. To nas opet podseća na najznačajnije pravilo optimizacije koda – uvek merite rezultate konkretne optimizacije!

RUTINE

Jedan od najmoćnijih alata u optimizaciji koda jeste dobra dekompozicija rutina. Male, dobro definisane rutine, štede prostor tako što posao obavljaju odvojeno, na različitim mestima. Takve rutine čine da je program lako optimizovati zbog toga što refaktorisanje koda u jednoj rutini donosi poboljšanje performansi na svim mestima gde se ta rutina poziva. Manje rutine su lakše za rekodiranje u jezicima nižeg nivoa. Dugačke, komplikovane rutine same po sebi je dovoljno teško razumeti, a rekodiranje u jeziku nižeg nivoa je prosto nemoguće.

PREPISIVANJE RUTINA (*INLINE*)

U ranim danima programiranja, određene mašine (hardver) su „skupo" kažnjavale performanse prilikom zvanja rutina. Poziv rutine je značio da je operativni sistem morao da isčita (*swap out*) program, da učita (*swap in*) direktorijum rutina, da učita određenu rutinu, da je izvrši, da isčita rutinu i da ponovo učita kontekst osnovnog programa. Sva ova učitavanja i isčitavanja su značajno trošila resurse i činile pograme sporima samo zbog činjenice da pozivaju rutine. Moderni računarski sistemi daleko manje „naplaćuju" pozive rutina. Pogledajmo rezultate stavljanja rutine (funkcije) za kopiranje stringova *inline*:

Jezik	Vreme normalne rutine	Vreme inline rutine	Ušteda u vremenu
C++	0.471	0.431	8%
Java	13.1	14.4	-10%

Tabela 33. Kopiranje funkcije direktno na mesto poziva, pomoću rezervisane reči *inline*

U nekim situacijama, na ovaj način (stavljanjem koda rutine direktno na mesto poziva iste) bi mogli da uštedite par nanosekundi. Naravno, ako koristite mogućnosti jezika kao što je C++ komanda *inline*. Sa druge strane, ako radite u jeziku koji ne podržava komandu *inline* direktno, ali ima makro predprocesor, možete da iskoristite makro da umetnete kod direktno. Međutim, moderni računari praktično ne kažnjavaju poziv rutina. Kao što se iz gornje tabele može i videti, umetanjem koda rutine pomoću komande *inline* ćete pre degradirati performanse nego ih optimizovati.

PONOVNO PISANJE (*RECODE*) U JEZIKU NISKOG NIVOA

Jedna od poznatih konvencionalnih mudrosti koju treba pomenuti jeste savet da kada imate uska grla po pitanju performansi, trebalo bi da ponovo napišete delove koda, koji su pogodni za to, u jeziku niskog, odnosno nižeg nivoa. Ako recimo kodirate u C++, onda bi jezik nižeg nivoa mogao da bude asembler, a ako kodirate u Python-u onda bi jezik niskog nivoa mogao da bude C. Ponovno kodiranje u jeziku nižeg nivoa obično poboljšava i brzinu i veličinu koda. Evo tipičnog pristupa koji se koristi prilikom optimizacije koda u jeziku nižeg nivoa:

- Napišite 100% aplikacije u jeziku visokog nivoa.
- Testirajte podrobno aplikaciju i verifikujte da je korektna.
- Ako su neophodna poboljšanja performansi posle prva dva koraka, profilišite aplikaciju da bi ste identifikovali uska grla. Pošto je obično oko 5% koda odgovorno za 50% vremena izvršavanja aplikacije, pronaći ćete male delove koda koje je potrebno optimizovati.
- Rekodirajte, odnosno napišite ponovo, te manje (identifikovane) delove koda u jeziku niskog nivoa da bi ste poboljšali ukupne performanse.

Da li ćete pratiti ovaj dobro poznati pristup zavisi najviše od toga koliko ste vični sa jezicima niskog nivoa, koliko je problem pogodan za rekodiranje u jeziku niskog nivoa, pa i od stepena očaja u koji ste zapali. Autor knjige se prvi put sreo sa ovom tehnikom za optimizaciju koda prilikom pisanja programa za enkripciju podataka, odnosno DES algoritma. Pre nego što se odlučio da proba i ovu tehniku, program je bio i dalje duplo sporiji od zacrtanog cilja po pitanju brzine iako je isprobao sve moguće tehnike za koje je čuo do tada. Rekodiranje dela aplikacije u asembleru je bila jedina preostala opcija. Kao potpuni početnik u svetu programiranja u asembleru, sve što je mogao da pokuša jeste čista translacija sa jezika visokog nivoa na asembler. Iako je uradio baš to (čistu translaciju koda), dobio je ubrzanje od čak 50% iako se radilo o rekodiranju na tako rudimentarnom nivou! Recimo da imate rutinu koja konvertuje binarne podatke u velike (*uppercase*) ASCII karaktere. Sledeći primer pokazuje kod za tu operaciju u jeziku Delphi:

```
procedure HexExpand(
   var source: ByteArray;
   var target: WordArray;
   byteCount: word
);

var
   index: integer;
   lowerByte: byte;
   upperByte: byte;
   targetIndex: integer;

begin
   targetIndex := 1;

   for index := 1 to byteCount do begin
      target[ targetIndex ] := ( (source[ index ] and $F0) shr 4 ) + $41;
      target[ targetIndex+1 ] := (source[ index ] and $0f) + $41;
      targetIndex := targetIndex + 2;
   end;
end;
```

Iako je iz ovog koda dosta teško uočljivo koji deo koda je trom (*fat*), može se uočiti da se dosta bavi manipulacijom bitovima, što baš i nije jača strana Delphi-ja. Međutim, manipulacija bitovima jeste jača strana asemblera, tako da ovaj deo koda predstavlja dobrog kandidata za rekodiranje u asembleru. Sledi translirani kod u asembleru:

```
procedure HexExpand(
   var source;
   var target;
   byteCount : Integer
);

   label

   EXPAND;

   asm
         MOV   ECX,byteCount      // load number of bytes to expand
         MOV   ESI,source         // source offset
         MOV   EDI,target         // target offset
         XOR   EAX,EAX            // zero out array offset

   EXPAND:
         MOV   EBX,EAX            // array offset
```

```
        MOV   DL,[ESI+EBX]       // get source byte
        MOV   DH,DL              // copy source byte

        AND   DH,$F              // get msbs
        ADD   DH,$41             // add 65 to make upper case

        SHR   DL,4               // move lsbs into position
        AND   DL,$F              // get lsbs
        ADD   DL,$41             // add 65 to make upper case
        SHL   BX,1               // double offset for target array offset
        MOV   [EDI+EBX],DX       // put target word

        INC   EAX                // increment array offset
        LOOP  EXPAND             // repeat until finished
end;
```

Rekodiranje u asembleru je u ovom slučaju bilo prilično profitabilno, dajući ubrzanje od čak 41%. Bilo bi logično da pretpostavimo da bi osnovni kod u jeziku viskog nivoa koji je pogodan za manipulaciju bitovima (C++ na primer), imao manje dobitke nego Delphi, prilikom translacije u asemblerski kod. Pogledajmo rezultate:

Jezik	Vreme u jeziku visokog nivoa	Vreme u asembleru	Ušteda
C++	4.25	3.02	29%
Delphi	5.18	3.04	41%

Tabela 34. Razlike u jezicima visokog nivoa i asembleru

Rutina napisana u asembleru pokazuje da rekodiranje u asembleru ne proizvodi nužno velik i "ružan" kod. Takve rutine su često vrlo jednostavne, kao ova gore. Ponekad je asemblerski kod skoro podjednako kompaktan kao i njegov ekvivalent u jeziku visokog nivoa.

Relativno prosta i efektivna strategija za rekodiranje u asembleru jeste da počnete od koda koji generiše kompajler kao nus proizvod kompajliranja. Izdvojte asemblerski kod za rutinu koju treba da optimizujete i sačuvajte je na nezavisnoj lokaciji, recimo novom fajlu. Koristeći kompajlerov asemblerski kod kao osnovu, optimizujte kod ručno, pri tome proveravajući tačnost, kao i performanse prilikom svake izmene koda. Neki kompajleri ostavljaju kod jezika visokog nivoa u kreiranom asemblerskom kodu, objašnjavajući kako je transliran kod visokog nivoa. Ako to radi i vaš kompajler, možete da sačuvate taj kod kao dokumentaciju koja naknadno može biti od velike koristi.

ZAKLJUČAK

Možemo očekivati da su se atributi performansi značajnije promenili u poslednjih 10-15 godina. Na neki način i jesu. Kompjuteri su dramatično brži nego tada, a i memorije ima u izobilju. U to doba, kada se radila optimizacija koda, bilo je potrebno da se izvrši od 10,000 do 50,000 testova na kodu da bi se dobili razumljivi, merljivi rezultati. Danas, potrebno je da se izvrši od milion do 100 miliona testova da bi se dobili isto tako merljivi rezultati. A kada je potrebno da izvršite recimo 100 miliona testova da bi se dobio merljiv rezultat, morate da se zapitate da li će iko ikada primetiti uticaj optimizacija u realnoj eksploataciji programa. Računari (hardver) su postali tako moćni da su brojne optimizacije performansi o kojima smo pričali, za mnoge široko korišćene primene, postale irelevantne.

Drugim rečima, problemi vezani za performanse su ostali prilično isti. Programerima koji pišu desktop aplikacije ove informacije možda više nisu neophodne, ali onima koji pišu za *embedded* sisteme, sisteme u realnom vremenu i druge sisteme sa striktnim zahtevima vezano za brzinu i memoriju mogu biti od koristi.

Potreba da se meri svaki pojedinačni pokušaj optimizacije koda ostala je konstanta još od objavljivanja studije o Fortran programima Donalda Knuth-a 1971. godine. Prema merenjima koje smo obradili u ovom radu, efekat bilo koje pojedinačne optimizacije je u stvari manje predvidiv nego što je to bio ranije, pre recimo 10-15 godina. Efekat svake optimizacije koda zavisi od programskog jezika, kompajlera, verzije kompajlera, biblioteke koda, kao i podešavanja kompajlera između ostalog.

Optimizacija koda uvek podrazumeva ustupke između kompleksnosti, čitljivosti, jednostavnosti i lakoće održavanja koda sa jedne strane i želje da se poboljšaju performanse na drugoj. Ona sa sobom nosi povećanje nivoa održavanja koda usled reprofilisanja koje je neophodno.

Može se reći da je insistiranje na merljivim poboljšanjima dobar put da se oduprete iskušenjima prerane optimizacije, kao i siguran put da se nametne pisanje „čistog", standardnog koda. Ako je optimizacija dovoljno važna da „prizove" *profajler* (softver za profilisanje softvera), onda je verovatno i dovoljno važna da bi se i dozvolila optimizacija, naravno sve dok kod radi ispravno. Međutim, ako optimizacija nije dovoljno važna da bi prizvala mašineriju za profilisanje, onda optimizacija svakako nije dovoljno bitna da bi se zbog nje degradirali čitljivost, lakoća održavanja i druge karakteristike koda. Uticaj neizmerenih optimizacija koda na performanse je u najbolju ruku špekulativan, dok je negativan uticaj na čitljivost koda siguran isto kao i štetni uticaji na druge kvalitativne karakteristike koda.

LITERATURA

1. Steve McConnell : Code complete, Second Edition, 2004.
2. http://www.tantalon.com/pete/cppopt/main.htm (22.08.2011.)