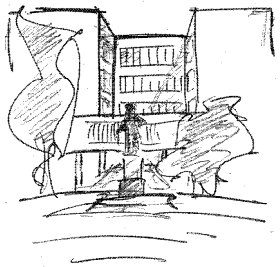
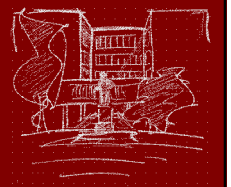
Развој софтвера 12



Саша Малков
Универзитет у Београду
Математички факултет
2023/2024

[P290]
Развој софтвера
Саша Малков



Тема 17

Оптимизација софтвера

Оптимизација софтвера

- ***Оптимизација софтвера*** је процес мењања структуре и имплементације програма у циљу постизања мањег заузећа једне или више врста ресурса:
  - процесорског времена
  - радне меморије
  - простора у трајном складишту
  - и друго

Оптимизација софтвера (2)

- Уштеда на једном ресурсу најчешће повећава оптерећење другог
- Зато је често у употреби мало другачија дефиниција:
  - ***Оптимизација софтвера*** је распоређивање оптерећења по врстама рачунарских ресурса, у складу са потребама и могућностима

Плански приступ

- Оптимизација је веома осетљив посао
  - може да испуни циљ, тј. да смањи оптерећење неког ресурса
  - али може и да знатно повећа оптерећење неког другог ресурса
  - уз то обично производи програмски код који је тежи за одржавање
- Зато мора да се предузима плански и уз претходно изучавање проблема

Потребне информације

- **Разумевање проблема и решења**
  - задатак
  - алгоритми
  - имплементација
- **Разумевање ограничења**
  - пословни захтеви
  - архитектура рачунара
  - архитектура процесора
- **Познавање алата**
  - програмски језик
  - преводилац
  - асемблер и машински језик
  - алати за мерење перформанси

Када оптимизовати

- Оптимизација унапред
  - Континуално старање о перформансама
- Оптимизација уназад
  - Након завршеног развоја

Оптимизација унапред

- Континуално старање о перформансама, током развоја
- Има смисла само ако у току развоја већ знамо које је тачно делове потребно да оптимизујемо
- Неопходно је да се пажљиво структурира код, са јасним дефинисањем циљних перформанси сваке од компоненти
- Потенцијални проблеми:
  - Да ли смо сигурни да знамо тачно који су то делови?
  - Да ли смо сигурни које су критичне границе перформанси?
  - Да ли смо сигурни да тај део кода неће бити мењан или избачен?

Оптимизација уназад

- Предузима се тек након завршеног развоја
- Мере се перформансе, сагледавају се проблеми и планирају оптимизације
- Потенцијални проблеми:
  - Да ли употребљен алгоритам уопште може да се оптимизује?
  - Да ли имплементирана архитектура софтвера може да се оптимизује?

Шта је боље?

- Најчешће је много боље да се оптимизује уназад
- Оптимизација унапред значајно отежава одржавање софтвера и има тенденцију да *квари дизајн*
- Добар дизајн софтвера омогућава релативно лако одржавање, па чак и замењивање алгоритама или неких делова архитектуре
- Често је корисно задржавање (чување) обе верзије кода (оптимизоване и неоптимизоване)
  - лакше је локализовати багове у неоптимизованој верзији

Превремена оптимизација

- ***Превремена оптимизација*** је оптимизација предузета пре него што знамо ***шта*** је и ***колико*** потребно да се оптимизује
- Сукобљава се са принципом агилног развоја "*неће бити потребно*"
- "***Оптимизуј касније***" је примена агилног принципа "*неће бити потребно*" на проблем оптимизације:
  - Имплементирати (оптимизовати) тек онда када је потребно, а не када се предвиђа или процењује да би могло бити потребно!

Цитати

- *"Програмери троше енормне количине времена на размишљање и бригу о брзини некритичних делова програма, а такви покушаји постизања ефикасности заправо имају негативан утицај када се узму у обзир дебаговање и одржавање. Потребно је да занемаримо ситне добитке у ефикасности у, рецимо, 97% случајева: превремена оптимизација је извор свих зала. Са друге стране, не смемо пропустити прилику за оптимизацију преосталих критичних 3%."*

  Доналд Кнут

Цитати

1. *Направи да ради.*
2. *Направи исправно.*
3. *Направи брзо.*
4. *Направи јевтино.*

наводно Алан Кеј

Цитати

- У пракси може да испадне и овако:
  1. *Направи да ради*
     … и имаћеш багове
  2. *Направи исправно*
     … или бар довољно добро
  3. *Направи брзо*
     … и зато мењаш код, додајеш багове … GO TO 1
  4. *Направи јевтино*
     … што је врло тешко након свих претходних корака

"Правила" оптимизовања

- Неформална правила оптимизовања кода
  1. *Немој да оптимизујеш.*
  2. *Немој да оптимизујеш, још увек (само за експерте).*

  Мајкл Џексон

- Уобичајена верзија:
  - Немој да оптимизујеш.
  - Немој да оптимизујеш, још увек.
  - Прво пажљиво измери перформансе.

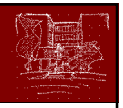
Принципи оптимизовања кода

1. Не оптимизовати.
2. Не оптимизовати без претходног мерења.
3. Ако перформансе нису ограничене програмским кодом, него спољашњим факторима, онда је оптимизација завршена.
4. Оптимизовати само код који је потпуно покривен тестовима јединица кода.
5. Оптимизовати једну по једну ствар.
6. Не оптимизовати уз нерешене багове или тесне рокове.
7. Тестирање мора да траје онолико колико је потребно.

Лења оптимизација

- “Лења оптимизација”
  - Чланак о оптимизацији у агилном развоју
  - Аутори: Кен Ауер и Кент Бек
    - Ken Auer, Kent Beck: **Lazy Optimization: Patterns for Efficient Smalltalk Programming**, in *Pattern Languages of Program Design 2*, eds. John Vlissides, James Coplien, Norman Kerth
    - http://web.archive.org/web/20060703205625/http://www.rolemodelsoftware.com/moreAboutUs/publications/articles/lazyopt.php

Ипак планирање, пре свега...



- Неки аспекти дизајна морају да се благовремено испланирају, зато што није добро да се одлажу за фазу оптимизације
  - т.ј. ако се одложе, онда оптимизација може да имплицира пратично понављање имплементирања
- То су, пре свега:
  - конкурентност
  - дистрибуираност

Нивои оптимизације

- Оптимизације високог нивоа
  - ниво пројекта
  - ниво алгоритма
- Оптимизације ниског нивоа
  - ниво изворног кода
  - ниво превођења
  - ниво изградње кода
  - ниво машинског кода
  - ниво извршавања

Оптимизације високог нивоа

- Ниво пројекта
  - архитектуром се прилагођавамо проблему и ресурсима
- Ниво алгоритма
  - алгоритам се бира и прилагођава према расположивим ресурсима

Оптимизације ниског нивоа

- Ниво изворног кода
  - имплементација алгоритма се прилагођава програмском језику
- Ниво превођења
  - избор опција преводилаца
- Ниво изградње кода
  - избор библиотека и начина повезивања
- Ниво машинског кода
  - ако не може никако другачије, пишемо део кода на машинском језику
- Ниво извршавања и архитектуре
  - при писању кода узимамо у обзир специфичности процесора
    - извршавање преко реда
    - одложено гранање
    - паралелно извршавање инструкција
    - ...

Оптимизације ниског нивоа (2)

- Најзаступљеније су оптимизације на нивоу изворног кода
  - њима ћемо да посветимо највише пажње
- Али и остале могу да имају велики значај

Оптимизације ниског нивоа (3)

- Пример – сумирање матрице

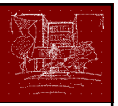
```
int sum = 0;
for( int i=0; i<len; i++ )
    for( int j=0; j<len; j++ )
          sum += niz[j][i];
```

- Шта може да се оптимизује?
- Колики допринос можемо да очекујемо?

Технике оптимизације

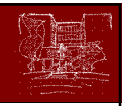
- Опште технике
  - технике које могу да се примене на све или бар на већи број различитих програмских језика или проблема
- Специфичне технике
  - технике које могу да се примене на мањи број програмских језика
  - често се односе на конкретан програмски језик, па чак и на конкретан преводилац

Опште технике оптимизације



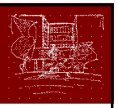
- Навешћемо само неке од важнијих техника оптимизације:
  - Одбацивање непотребне прецизности
  - Употреба уметнутих функција и метода
  - Интеграција петљи
  - Измештање инваријанте изван петље
  - Размотавање петљи
  - Таблице унапред израчунатих вредности
  - Елиминација гранања и петљи
  - Замењивање динамичког услова статичким
  - Снижавање сложености операције
  - Снижавање сложености алгоритма
  - Писање затворених функција
  - Смањивање броја аргумената функције
  - Избегавање глобалне променљиве
  - Пажљиво одређивање редоследа проверавања услова
  - Избор решења према најчешћем случају
  - Увођење конкурентности и дистрибуираног израчунавања

Одбацивање непотребне прецизности



- Смањивање прецизности у покретном зарезу
- Смањивање опсега целих бројева
- Може да смањи време извршавања и више од 50%
  - посебно ако је дужина података иницијално већа од дужине процесорске речи или ширине магистрале података
- Код савремених процесора често не утиче значајно
  - али може да утиче код векторских операције
  - такође, конверзије могу да утичу на перформансе

Употреба уметнутих функција и метода



- У случају једноставних функција (нпр. *swap*) може да се смањи време извршавања и до 50%
- Претерана употребе може да обесмисли концепт
  - на пример (C++), ако се целе класе дефинишу у заглављу, онда су сви методи уметнути
  - зато преводици у последње време игноришу сугестије и сами процењују шта ће како да преведу

Интеграција петљи

- Смањивање броја петљи, уместо две узастопне петље прављење једне са сложенијим кораком
- У случају једноставних корака може да допринесе и до 35%
- Приметимо да је то супротно правилу писања чистог кода (и рефакторисања) да би свака петља требало да ради тачно један посао

Измештање инваријанти ван петље

- Све оно што може, требало би да се израчунава ван петље
  - `for( int i=0; i<limit(a,b,c); i++ )...`
- Користити помоћне променљиве
- Променити смер итерације

Размотавање петљи

- Смањивање броја понављања уз повећавање сложености корака петље
- Може да допринесе и до 50%
- Може и да успори извршавање у неким случајевима
  - због отежавања кеширања кода

Таблице унапред израчунатих вредности

- Ако се неке функције израчунавају за ограничен број различитих аргумената, онда може да буде ефикасније да се уместо израчунавања консултују таблице
- У случају сложенијих израчунавања може доћи и до драматичног убрзавања, по неколико пута

Елиминација гранања и петљи

- У неким случајевима гранања или петље могу да се замене сложенијим израчунавањем без гранања и петљи
- Може да се добије и по неколико пута ефикаснији код
- Пример, бројање битова у 32-битном броју:

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

  - Скраћивање времена извршавања је око 90% у односу на појединачно бројање

Замењивање динамичког услова статичким

- Ако опсег броја понављања није фиксан, некада може бити ефикасније урадити посао за пун обим него проверавати границе
- Као наредни корак може да се примени и елиминација петље

Снижавање сложености операције

- Неке операције могу да се замене ефикаснијим операцијама
- На пример, уместо `a*8` можемо да напишемо `a<<3`
- Такве оптимизације данас обично изводе преводиоци, чак и боље него што би то програмери могли у разумном времену

Снижавање сложености алгоритма

- Уобичајено је да алгоритми са нижом сложеношћу израчунавања раде ефикасније
- Међутим, морамо да будемо опрезни, зато што алгоритми са нижом сложеношћу често имају скупље кораке
  - За неке скупове података могу да буду мање ефикасни
- Исплативост процењујемо на основу сложености и величине проблема

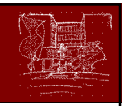
Писање затворених функција

- То су функције које не позивају друге функције
  - или евентуално позивају само инлајн функције
  - називају се и функције-листови
- Такве функције се боље оптимизују од стране преводилаца
- Често је ефикасније да се направи једна сложенија затворена функција него неколико мањих функција
- И ова оптимизација је супротстављена правилима доброг дизајна и рефакторисања

Смањивање броја аргумената функције

- Ако функција има мање аргумената, преводилац ће лакше моћи да употреби регистре за преношење аргумената (уместо уобичајене употребе стека)
- Посебно је корисно ако се функција често позива (на пример у петљи)
- Вид ове оптимизације је и преношење података у оквиру структуре која се преноси по адреси

Избегавање глобалне променљиве



- Локалне променљиве могу да се оптимизују записивањем у регистрима
- Није уобичајено да се глобалне променљиве замењују регистрима, зато што се не зна да ли те регистре користи још неко (потпрограм, библиотека, друге нити и сл.)
- У последње време преводиоци игноришу сугестије овог типа, зато што боље могу сами да процене за шта ће се користити регистри

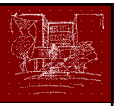
Пажљиво одређивање редоследа проверавања услова

- Ако се проверава већи број услова и затим поступа у складу са више варијанти, редослед проверавања може да утиче на ефикасност одлучивања
- Редослед проверавања услова би требало да се одреди тако да просечан број провера буде што мањи

Избор решења према најчешћем случају

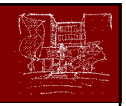
- Неки алгоритми су ефикаснији за неке случајеве а мање ефикасни за неке друге
- Избор алгоритма мора да се обавља у складу са очекиваним случајевима употребе
- Пример, за кратке низове примитиван алгоритам сортирања *bubble-sort* може да буде ефикаснији од алгоритма *quick-sort*

Увођење конкурентности и дистрибуирања



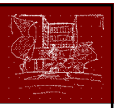
- Ако се конкурентност или дистрибуираност природно уклапају у алгоритам, онда је то добро решење
- Ако се проблем вештачки паралелизује, то често може да донесе више проблема него користи

Примери специфичних техника за C++ (1)



- Специфичне технике користе специфичности конкретног програмског језика и тешко могу да се примене у другим језицима
- Неке технике нису саме по себи замишљене као оптимизација, али често могу да допринесу перформансама или да олакшају оптимизацију и одржавање кода

Примери специфичних техника за C++ (2)



- Користити стандардну библиотеку
  - тешко је нешто урадити ефикасније него у стандардној библиотеци
  - чак и када у томе успемо, то ће вероватно већ следећом верзијом библиотеке бити превазиђено
- Употребљавати референце уместо показивача
  - једнако ефикасно а много чистије

Примери специфичних техника за C++ (3)

- Одложена иницијализација објеката
  - супротно од *RAII*
  - за неке операције нам можда нису неопходни комплетно иницијализовани објекти
  - уместо да се при конструкцији изведе пуна иницијализација, можда је довољно да се изведе само припрема за иницијализацију, а да се остало уради при првој употреби неког од сложенијих метода
- Из делова кода који се оптимизују избацити руковање изузецима
  - обрада изузетака је релативно неефикасна
  - никако не употребљавати изузетке као механизам за управљање током програма (“напредна верзија” *GOTO*)
  - ово је веома осетљива техника која може да донесе више штете него користи

Примери специфичних техника за C++ (4)

- Употреба мета-програмирања
  - применом шаблона многе ствари могу да се израчунају већ у фази превођења програма
- Употреба *constexpr* израза и услова
  - слично као и метапрограмирање, омогућава се одлучивање већ у фази превођења програма

Оптимизације у ходу

- Неке оптимизације могу да се праве у ходу
- Али веома опрезно...

Оптимизације у ходу

- Преношење објеката по референци
  - осим, евентуално, у случају сасвим малих објеката
- Декларисање привремених промењивих што дубље у коду
  - да се не праве ако није потребно
- Користити конструкцију објеката (иницијализацију) пре него додељивање

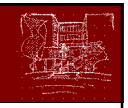
Оптимизације у ходу

- Користити листе иницијализација чланова и базних објеката
- Имплементирати сопствене операторе алокације и деалокације (*new* и *delete*)
- Употреба шаблона функција и метапрограмирања

Препуштање оптимизације преводиоцу

- Спада у специфичне технике оптимизације
- Преводиоци имају бројне опције за оптимизацију, како појединачне тако и збирне
- ОПРЕЗНО!
- Релативно често се проналазе багови преводилаца при примени неких техника оптимизације!

Честе грешке (1)

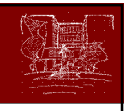


- Ништа не претпостављати
  - често се погрешно претпоставља да је “*А ефикасније него Б*”
  - свака претпоставка мора да се провери
  - пример бесмислене оптимизације:
    - Уместо да напишемо
      ```
      a = b * 40;
      ```
    - Можемо да напишемо оптимизовано
      ```
      // a = b * 40;
      a = (b << 5) + (b << 3);
      ```
    - Мотив: померање је брже него множење
    - Међутим, већина преводилаца то може да уради уместо нас
    - Штавише, то неће даураде ако је процесор такав да то није брже

Честе грешке (2)

- Смањивање кода није увек и његово оптимизовање
  - добар пример је управо елиминација петљи
- Оптимизовање током иницијалног кодирања
  - најчешћи проблем
  - прво написати исправан код (са тестовима, наравно)
  - па тек онда оптимизовати
- Посвећивање више пажње перформансама него коректности
  - некада перформансе јесу примаран циљ
  - али најчешће су коректност и прецизност много важнији

Честе грешке (3)



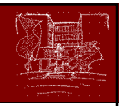
- Мерење перформанси је пресудно, али погрешно мерење може да направи озбиљне проблеме
- Мерење перформанси изолованог дела кода може да нам пружи веома искривљене информације
- Перформансе у изолованом окружењу могу да буду и много веће и много мање него у оквиру оптерећеног система

Цитати

- *Оптимизација увек упропасти ствари, зато што је свака оптимизација, шире посматрано, облик варања, а варалице увек на крају буду ухваћене*

  Лери Вол

Добре смернице



- Одложити оптимизовање кода за крај развојног циклуса
- Пажљиво поставити циљеве
- Не оптимизовати даље од постављених циљева
- Проверити да ли је архитектура одговарајућа
  - ако су циљне перформансе далеко од остварених, можда морају да се предузму драматичне измене - ако је тако, не губити време на појединачне мање оптимизације него одмах предузети измене на нивоу архитектуре
- Изабрати одговарајуће алгоритме
  - ако перформансе значајно опадају са порастом димензија проблема, онда је врло вероватно неопходно наћи ефикаснији алгоритам, иако можда његова имплементација може да буде много сложенија и његова примена на мање димензије проблема нешто мање ефикасна
- Пажљиво мерити остварене перформансе
- Не оптимизовати верзију за дебаговање
  - већ и само превођење верзије за публиковање ће значајно унапредити перформансе
  - верзија за дебаговање може да има другачија (и уз то ирелевантна) уска грла

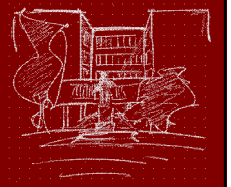
Алати за подршку оптимизовању

- Профајлер (*Profiler*)
  - За сваки потпрограм (или чак блок или наредбу програма) рачунају
    - број извршавања
    - укупно време извршавања
    - време проведено у позивима и време проведено у самом коду
    - ...
  - Могу да мере и
    - заузеће меморије
    - употребу кеш меморије
    - разне друге ствари
- *GNU gprof*
- *Valgrind*
- ...

Литература за тему

- *Ken Auer, Kent Beck:* **Lazy Optimization: Patterns for Efficient Smalltalk Programming**, in *Pattern Languages of Program Design 2*, eds. John Vlissides, James Coplien, Norman Kerth
- *Agner Fog*, **Optimizing software in C++ - An optimization guide for Windows, Linux and Mac platforms**, *Technical University of Denmark, 2014*.
  `http://www.agner.org/optimize/optimizing_cpp.pdf`

Хвала на пажњи!



МАТФ
Универзитет у Београду
Математички факултет