Лекція 12. Стратегії та методи оптимізації програмного коду

Оптимізація - модифікація системи для поліпшення її ефективності. Система може бути одиночній комп'ютерною програмою, наборомкомп'ютерів або навіть цілою мережею, такий як Інтернет.

Хоча метою оптимізації є отримання оптимальної системи, істинно оптимальна система в процесі оптимізації досягається далеко не завжди. Оптимізована система зазвичай є оптимальною тільки для однієї задачі або групи користувачів: десь може бути важливіше зменшення часу, необхідного програмі для виконання роботи, навіть ціною споживання більшого обсягу пам'яті; в додатках, де важливіше пам'ять, можуть вибиратися більш повільні алгоритми з меншими запитами до пам'яті.

Більш того, часто не існує універсального рішення, яке працює добре у всіх випадках, тому інженери використовують компромісні (англ. *tradeoff*) Рішення для оптимізації тільки ключових параметрів. До того ж, зусилля, необхідні для досягнення повністю оптимальної програми, яку неможливо далі поліпшити, практично завжди перевищують вигоду, яка може бути від цього отримана, тому, як правило, процес оптимізації завершується до того, як досягається повна оптимальність. На щастя, в більшості випадків навіть при цьому досягаються помітні поліпшення.

Оптимізація повинна проводитися з обережністю. Тоні Хоар вперше вимовив, а Дональд Кнут згодом часто повторював відомий вислів: "Передчасна оптимізація - це корінь всіх бід". Дуже важливо мати для початку озвучений алгоритм і працюючий прототип.

Основи оптимізації коду

Деякі завдання часто можуть бути виконані більш ефективно. Наприклад, програма мовою Сі, яка підсумовує всі цілі числа від 1 до N:

```
int  i  ,  sum  =  0  ;
for  (  i  =  1  ;  i  <=  N  ;  i  + +  )
sum  + =  i  ;
```

Маючи на увазі, що тут немає переповнювання, цей код може бути переписаний у наступному вигляді за допомогою відповідної математичної формули:

```
 int  sum  =  (  N  *  (  N  +  1  )  )  /  2  ;
```

Поняття "оптимізація" зазвичай має на увазі, що система зберігає ту ж саму функціональність. Однак, значне поліпшення продуктивності часто може бути досягнуто і за допомогою видалення надлишкової функціональності. Наприклад, якщо допустити, що програмі не потрібно підтримувати більш, ніж 100 елементів при введенні, то можливо використовувати статичну виділення пам'яті замість більш повільного динамічного.

Компроміси (tradeoff)

Оптимізація в основному фокусується на одиночному або повторному часу виконання, використання пам'яті, дискового простору, пропускної здатності або деякому іншому ресурсі. Це звичайно вимагає компромісів - один параметр оптимізується за рахунок інших. Наприклад, збільшення розміру програмного кеша чогось покращує продуктивність часу виконання, але також збільшує споживання пам'яті. Інші поширені компроміси включають прозорість коду і його виразність, майже завжди ціною деоптімізації. Складні спеціалізовані алгоритми вимагають більше зусиль поналагодженні і збільшують ймовірність помилок.

Різні області

В дослідженні операцій, оптимізація - це проблема визначення вхідних значень функції, при яких вона має максимальне або мінімальне значення. Іноді на ці значення накладаються обмеження, таке завдання відома як *обмежена оптимізація.*

В програмуванні, оптимізація зазвичай позначає модифікацію коду і його установок компіляції для даної архітектури для виробництва більш ефективного ПЗ.

Типові проблеми мають настільки велику кількість можливостей, що програмісти зазвичай можуть дозволити використовувати тільки "досить добре" рішення.

Вузькі місця

Для оптимізації потрібно знайти вузьке місце (англ. *hotspot*), Іноді зване пляшковим горлечком (англ. *bottleneck*): Критичну частину коду, яка є основним споживачем необхідного ресурсу. Поліпшення приблизно 20% коду іноді тягне за собою зміну 80% результатів). Для пошуку вузьких місць використовуються спеціальні програми - профайлер. Витік ресурсів (пам'яті, дескрипторів і т.д.) також може привести до падіння швидкості виконання програми, для їх знаходження також застосовуються спеціальні програми (наприклад, BoundsChecker).

Архітектурний дизайн системи особливо сильно впливає на її продуктивність. Вибір алгоритму впливає на ефективність більше, ніж будь-який інший елемент дизайну. Більш складні алгоритми і структури дані можуть добре оперувати з великою кількістю елементів, в той час як прості алгоритми підходять для невеликих обсягів даних - накладні витрати на ініціалізацію складнішого алгоритму можуть переважити вигоду від його використання.

Чим більше пам'яті використовує програма, тим швидше вона зазвичай виконується. Наприклад, програма-фільтр зазвичай читає кожен рядок, фільтрує і виводить цей рядок безпосередньо. Тому вона використовує пам'ять тільки для зберігання одного рядка, але її продуктивність зазвичай дуже погана. Продуктивність може бути значно поліпшена читанням цілого файлу і записом потім відфільтрованого результату, однак цей метод використовує більше пам'яті. Кешування результату також ефективно, проте вимагає більшої кількості пам'яті для використання.

Найпростіші прийоми оптимізації програм за витратами процесорного часу

Оптимізація за витратами процесорного часу особливо важлива для розрахункових програм, в яких велику питому вагу мають математичні обчислення. Тут наведені деякі прийоми оптимізації, які може використовувати програміст під час написання початкового тексту програми.

Ініціалізація об'єктів даних

У багатьох програмах якусь частину об'єктів даних необхідно *ініціалізувати,* тобто присвоїти їм початкові значення. Таке присвоювання виконується або на самому початку програми, або, наприклад, в кінці циклу. Правильна ініціалізація об'єктів дозволяє заощадити дорогоцінний процесорний час. Так, наприклад, якщо мова йде про ініціалізації масивів, використання циклу, швидше за все, буде менш ефективним, ніж оголошення цього масиву прямим присвоєнням.

Програмування арифметичних операцій

У тому випадку, коли значна частина часу роботи програми відводиться арифметичним обчислень, чималі резерви підвищення швидкості роботи програми таяться в правильному програмуванні арифметичних (і логічних) виразів. Важливо, що різні арифметичні операції значно розрізняються по швидкодії. У більшості архітектур, найшвидшими є операції додавання і віднімання. Повільнішим є множення, потім йде поділ. Наприклад, обчислення значення виразу $\frac{x}{a}$, Де *a* - Константа, для аргументів з плаваючою точкою виробляється швидше у вигляді $x \cdot b$, Де $b = \frac{1}{a}$ - Константа, яка обчислюється на етапі компіляції програми (фактично повільна операція ділення замінюється швидкою операцією множення). Для цілочисельного аргументу *x* обчислення виразу 2 *x* швидше провести у вигляді *x* + *x* (Операція множення замінюється операцією додавання) або з використанням операції зсуву вліво (що забезпечує виграш не на всіх процесорах). Подібні оптимізації називаються зниженням сили операцій. Множення цілочисельних аргументів на константу на процесорах сімейства x86 може бути ефективно виконана з використанням асемблернихкоманд `LEA`, `SHL` і `ADD` замість використання команд `MUL/IMUL` :

```
    ; Вихідний операнд у регістрі EAX ADD EAX , EAX
; Множення на 2 LEA EAX , [ EAX + 2 * EAX ]
; Множення на 3 SHL EAX , 2 ; Множення на 4 LEA
EAX , [ 4 * EAX ] ; Інший варіант реалізації
множення на 4 LEA EAX , [ EAX + 4 * EAX ] ;
Множення на 5 LEA EAX , [ EAX + 2 * EAX ] ;
Множення на 6 ADD EAX , EAX , І т.д.
```

Подібні оптимізації є мікроархітектурнимі і зазвичай виробляються оптимизирующим компілятором прозоро для програміста.

Відносно багато часу витрачається на звернення до підпрограм (передача параметрів через стек, збереження регістрів і адреси повернення, виклик конструкторів копіювання). Якщо підпрограма містить малу кількість дій, вона може бути реалізована **Підставляємо** (англ. *inline*) - Всі її оператори копіюються в кожне нове місце виклику (існує ряд обмежень на inline-підстановки: наприклад, підпрограма не повинна бути рекурсивної). Це ліквідує накладні витрати на звернення до підпрограми, проте веде до збільшення розміру виконуваного файлу. Само по собі збільшення розміру виконуваного файлу не є суттєвим, проте в деяких випадках виконуваний код може вийти за межі кеша команд, що спричинить значне падіння швидкості виконання програми. Тому сучасні оптимізують компілятори зазвичай мають налаштування оптимізації за розміром коду і за швидкістю виконання.

Швидкодія також залежить і від типу операндів. Наприклад, у мові Turbo Pascal, через особливості реалізації цілочисельний арифметики, операція складання виявляється найбільш повільною для операндів типу `Byte` і `ShortInt` : незважаючи на те, що змінні займають один байт, арифметичні операції для них двобайтові і при виконанні операцій над цими типами виробляється обнуління старшого байта регістрів і операнд копіюється з пам'яті в молодший байт регістра. Це і призводить до додаткових витрат часу.

Програмуючи арифметичні вираження, слід вибирати таку форму їх записи, щоб кількість "повільних" операцій було зведено до мінімуму. Розглянемо такий приклад. Нехай необхідно обчислити многочлен 4-го ступеня:

$$a\,x^4 + b\,x^3 + c\,x^2 + d\,x + e$$

За умови, що обчислення ступеня проводиться перемножуванням підстави певну кількість разів, неважко знайти, що в цьому виразі міститься 10 множень ("повільних" операцій) і 4 складання ("швидких" операцій). Це ж саме вираз можна записати у вигляді:

$$(((A\,x + b)\,x + c)\,x + d)\,x + e$$

Така форма запису називається схемою Горнера. У цьому виразі 4 множення і 4 складання. Загальна кількість операцій скоротилася майже в два рази, відповідно зменшиться і час обчислення многочлена. Подібні оптимізації є алгоритмічними і зазвичай не виконується компілятором автоматично.

Цикли

Різниться і час виконання циклів різного типу. Час виконання циклу з лічильником і циклу з постусловіем при всіх інших рівних умовах збігається, цикл з передумовою виконується трохи довше (приблизно на 20-30%).

При використанні вкладених циклів слід мати на увазі, що витрати процесорного часу на обробку такої конструкції можуть залежати від порядку проходження вкладених циклів. Наприклад, вкладений цикл з лічильником мовою Turbo Pascal :

```
 for  j  : =  1  to  100000  do
for  k  : =  1  to  1000  do
a  : =  1  ;
```

```
 for  j  : =  1  to  1000  do
for  k  : =  1  to  100000  do
a  : =  1  ;
```

Цикл в лівій колонці виконується приблизно на 10% довше, ніж у правій.

На перший погляд, і в першому, і в другому випадку 10 000 000 разів виконується оператор присвоювання, і витрати часу на це повинні бути однакові в обох випадках. Але це не так. Пояснюється наше спостереження тим, що ініціалізації циклу, тобто обробка процесором його заголовка з метою визначення початкового і кінцевого значень лічильника, а також кроку збільшення лічильника вимагає часу. У першому випадку 1 раз ініціалізується зовнішній цикл і 100 000 разів - внутрішній, тобто всього виконується 100001 ініціалізація. У другому випадку, як неважко підрахувати, таких ініціалізацій виявляється всього лише 1001.

Аналогічно поводяться вкладені цикли з передумовою та з постусловіем. Можна зробити висновок, що при програмуванні вкладених циклів по можливості слід робити цикл з найбільшим числом повторень самим внутрішнім, а цикл з найменшим числом повторень - самим зовнішнім.

Але, кращих результатів можна добитися, об'єднавши кілька циклів в один, коли таке допустимо. Наприклад:

```
 for  (  int  y  =  0  ;
y  <  height  ;  y  + +  )
for  (  int  x  =  0  ;
x  <  width  ;  x  + +  )
put_puxel  (  x  ,  y  ,  0x00FF00 );
```

```
 for  (  int  i  =  0  ,  n  =  width
*  height  ;  i  <  n  ;  i  + +  )
VRAMptr_dd  [  i  ]  =  0x00FF00  ;
```

Поліпшення за рахунок зниження кількості циклових команд, і чим вже width тим ефективніше.

Якщо в циклах містяться звернення до пам'яті (зазвичай при обробці масивів), порядок обходу адрес пам'яті повинен бути по можливості послідовним, що дозволяє максимально ефективно використовувати кеш і механізм апаратної предвибірки даних з пам'яті (англ. *Hardware Prefetch*). Класичним прикладом подібної оптимізації є зміна порядку проходження вкладених циклів при виконанні множення матриць.

При обчисленні сум часто використовуються цикли, що містять однакові операції, пов'язані з кожним доданку. Це може бути, наприклад, загальний множник (мова Turbo Pascal):

```
     sum  : =  0  ;  for  i  : =  1
to  1000  do  sum  : =  sum  +  a
*  x  [  i  ]  ;
```

```
     sum  : =  0  ;  for  i  : =  1  to
1000  do  sum  : =  sum  +  x  [  i  ]  ;
                  Sum  : =  a  *  sum;
```

Очевидно, що друга форма запису циклу виявляється більш економною.

Інваріантні фрагменти коду

Оптимізація інваріантних фрагментів коду тісно пов'язана з проблемою оптимального програмування циклів. Всередині циклу можуть зустрічатися вирази, фрагменти яких ніяк не залежать від керуючої змінної циклу. Їх називають *інваріантними фрагментами* коду. Сучасні компілятори часто визначають наявність таких фрагментів і виконують їх автоматичну оптимізацію. Таке можливо не завжди, і іноді продуктивність програми залежить цілком від того, як запрограмований цикл. Як приклад розглянемо наступний фрагмент програми (мова Turbo Pascal):

```
  for i := 1 to n do
begin
...
for k := 1 to p do
for m := 1 to q do
begin
a [ k , m ] := Sqrt ( x * k * m - i ) + Abs ( u * i - x * m + k ) ;
B [ k , m ] := Sin ( x * k * i ) + Abs ( u * i * m + k ) ;
end ;
 ...
am := 0 ;
Bm := 0 ;
  for k := 1 to p do
for m := 1 to q do
begin
am := am + a [ k , m ] / c [ k ] ;
Bm := bm + b [ k , m ] / c [ k ] ;
end ;
end ;
```

Тут інваріантними фрагментами коду є доданок `Sin(x * k * i)` в першому циклі по змінній `m` і операція ділення на елемент масиву `c[k]` у другому циклі по `m`. Значення синуса і елемента масиву не змінюються в циклі по змінній `m`, отже, в першому випадку можна обчислити значення синуса і привласнити його допоміжної змінної, яка буде використовуватися у виразі, що знаходиться всередині циклу. У другому випадку можна виконати розподіл після завершення циклу по `m`. Таким чином, можна істотно скоротити кількість трудомістких арифметичних операцій.

Пріоритети оптимізації

- інтерфейсний, т.е.желательно заздалегідь ВСЕ погодити з іншими учасниками проекту, включаючи кому скільки МАКСИМУМ% CPUuse на конкретному ПК.
- PS: ПК не повинен бути сучасним і багатопроцесорним ... Якщо такого немає - свідомо сильно обмежити сумарний CPU витрата, можна утилітами типу slowcpu; на MP доп.CPU відключаючи в BIOS / фіксуючи всі потоки на один CPU (але, це не еквівалент SP, т.к.ОС буде висіти на др.CPU , тільки для процесу програмування, не кінцевого тестування) - виходячи з того що реліз буде значно гальмівні очікуваного, і чим складніше тема, жорсткіше терміни, і менше досвіду в темі - тим сильніше ... До того ж запас ЗАВЖДИ потрібен ще й на скільки-то нормальну роботу в режимі DEBUG, TRACE

- PS: узгодити і обговорити - не поспішаючи, і кожну дрібницю аж до того де використовувати ООП, а де ні - для прискорення. PS: Підказка - STL і прч .- не використовувати НІДЕ ... C # - аналогічно, Скрипти - ТІЛЬКИ там де не можна без них обійтися ..., тим більше вони часто привносять багато проблем з багами ...
- PS: причому не забувши кожну ітерацію тесту - штучно скидати L12 кеші і BTB, ... Щоб з'єднавши всі модулі потім не дивуватися ...
- алгоритмічний, мінус - принцип ніж оптимізований - тим менш зрозуміло, тим більше багів, але це найефективніший спосіб, тим більше найбільш оптимізуються алгоритми вже давно оптимізовані і можна подивитися на їх реалізацію.

Цей пункт можна можна розділити на:

- математично-алгоритмічний
- логічно-алгоритмічний
- інше-функціонально-алгоритмічний, наприклад використання асинхронного виконання всього що можна

...

- кеш-оптимізації на рівні алгоритму, включаючи HW L1, L2, програмні-для даних, програмні для ресурсів
- використання спецкоманда, наприклад MMX, SSE, ... Правда сумарно мінусів більше ніж плюсів, так як це чисто піарівські технології, наприклад трудовитрати (тим більше на вивчення їх роботи - 1) на різних архітектурах 2) у різних виробників CPU 3) у різних їх моделях ...)

PS: але, повинен завжди бути варіант коду - без їх наявності, в т.ч.для возм.портірования в майбутньому [можливо іншими людьми] і просто для Safe версії исполнимого файлу, т.к.некоторие PC-сумісні ПК, тим більше лаптопи і прч. - Не зовсім сумісні і тим більше безглючною ...

- заміна коду на асемблерні, вельми спірна, т.к.не вважаючи непортірованності, і непоганого рівня оптимізації (деяких) сучасних компіляторів, щоб у разі великого обсягу коду або застосування нестандартних для програмування трюків призводить до дуже плачевних результату - помилок, джерело які найчастіше так і не виявляється ... Але оптимізація теж має право на життя, особливо в inline або невеликих ф-иях, типу rol і т.п. І особливо якщо програміст хоч скільки знайомий з асемблером та архітектурою ПК, так як асемблер дозволяє іноді прискорити (деякі) блоки в десятки і сотні разів за рахунок візуального уявлення про функціонування і як наслідок оптимізації так щоб ефективно використовувати L1, L2, промах яких на сучасних ПК обчислюється сотнями тактів.

Для кожної ітерації оптимізації в ідеалі повинен зберігатися неоптимізований приклад для розуміння алгоритму іншими людьми, або самому розробнику, для тестів на швидкість і головне еквівалентність роботи з і без оптимізації.