Trịnh Thành Trung (ThS)
trungtt@soict.hust.edu.vn

Bài 7
TĂNG HIỆU NĂNG CHƯƠNG TRÌNH

Nội dung

1. Tổng quan
2. Các nguyên lý cơ bản
3. Một số kỹ thuật tăng hiệu năng chương trình
4. Tinh chỉnh mã nguồn
5. Một số kỹ thuật tinh chỉnh mã nguồn

1.
Tổng quan

Tăng hiệu năng chương trình và tinh chỉnh mã nguồn

Làm thế nào để
tăng hiệu năng chương trình

- **Giải thuật**
  - *Dùng giải thuật tốt nhất có thể*
  - *Sau đó hãy nghĩ tới việc tăng tính hiệu quả của code*
  - *Ví dụ: Tính tổng của n số tự nhiên kể từ m*

```
void main() {
  long n,m,i, sum ;
  cout << ‘ vào n ‘ ; cin << n;
  cout << ‘ vào m ‘ ; cin << m;
  sum =0;
  for(i = m ; i < m+n; i++) sum += i;
  cout << ‘ Tổng = ‘ <<sum;
}
```

```
void main()
{
  long n,m ;
  cout << ‘ vào n ‘ ; cin << n;
  cout << ‘ vào m ‘ ; cin << m;
  cout << ‘ Tổng = ‘
       << (m + m+ n) * n / 2.0;
}
```

Dùng chỉ thị chương trình dịch

- Một số compilers có khả năng tối ưu chương trình
  - *Phân tích sâu mã nguồn và tự động tối ưu hóa*
  - *Ví dụ GNU g++ compiler trên Linux/Cygwin cho chương trình viết bằng C*

```
g++ –O5 –o myprog myprog.c
```

  - *Có thể cải thiện hiệu năng từ 10% đến 300%*

Tự
tối ưu hóa

- Tự thực hiện những cải tiến mà trình dịch không thể
- Loại bỏ tất cả những chỗ bất hợp lý trong code
  - *Làm cho chương trình hiệu quả nhất có thể*
- Có thể phải xem lại khi thấy chương trình chạy chậm
  - *Cần tập trung vào đâu để cải tiến nhanh nhất, tốt nhất?*
- Xác định nguồn gây kém hiệu quả
  - *Dư thừa tính toán - redundant computation*
  - *Chủ yếu*
    - Trong các hàm
    - Trong các vòng lặp

2.
Các nguyên tắc cơ bản

tăng hiệu năng chương trình

Quy tắc
cơ bản

- **Đơn giản hóa Code - Code Simplification**
- **Đơn giản hóa vấn đề - Problem Simplification**
- **Không ngừng nghi ngờ - Relentless Suspicion**
- **Liên kết sớm - Early Binding**

Quy tắc
tăng tốc độ

Caching

- Dữ liệu thường dùng cần phải dễ tiếp cận nhất, luôn hiện hữu.

Lazy Evaluation

- Không tính 1 phần tử cho đến khi cần để tránh những sự tính toán không cần thiết.

Quy tắc
tăng tốc độ

- Có thể tăng tốc độ bằng cách sử dụng thêm bộ nhớ (mảng)
- Dùng thêm các dữ liệu có cấu trúc:
  - *Thời gian cho các phép toán thông dụng có thể giảm bằng cách sử dụng thêm các cấu trúc dữ liệu với các dữ liệu bổ xung hoặc bằng cách thay đổi các dữ liệu trong cấu trúc sao cho dễ tiếp cận hơn.*
- Lưu các kết quả được tính trước:
  - *Thời gian tính toán lại các hàm có thể giảm bớt bằng cách tính toán hàm chỉ 1 lần và lưu kết quả, những yêu cầu sau này sẽ được xử lý bằng cách tìm kiếm từ mảng hay danh sách kết quả thay vì tính lại hàm.*

Quy tắc lặp
Loop rules

- Những điểm nóng - Hot spots trong phần lớn các chương trình đến từ các vòng lặp:
- Đưa Code ra khỏi các vòng lặp:
  - *Thay vì thực hiện việc tính toán trong mỗi lần lặp, tốt nhất thực hiện nó chỉ một lần bên ngoài vòng lặp (nếu được)*
- Kết hợp các vòng lặp – loop fusion:
  - *Nếu 2 vòng lặp gần nhau cùng thao tác trên cùng 1 tập hợp các phần tử thì cần kết hợp chung vào 1 vòng lặp.*

Quy tắc lặp
Loop rules

- Kết hợp các phép thử - Combining Tests:
  - *Trong vòng lặp càng ít kiểm tra càng tốt và tốt nhất chỉ một phép thử. LTV có thể phải thay đổi điều kiện kết thúc vòng lặp. "Lính gác" hay "Vệ sĩ" là một ví dụ cho quy tắc này.*
- Loại bỏ Loop :
  - *Với những vòng lặp ngắn thì cần loại bỏ vòng lặp, tránh phải thay đổi và kiểm tra điều kiện lặp*

Quy tắc hàm
Procedure Rules

- Khai báo những hàm ngắn và đơn giản (thường chỉ 1 dòng) là inline
  - *Tránh phải thực hiện 4 bước khi hàm được gọi,*
  - *Tránh dùng bộ nhớ stack*

Tối ưu mã
C/C++

- Đặt kích thước mảng = 2^n
  - *Với mảng, khi tạo chỉ số, trình dịch thực hiện các phép nhân, vì vậy, hãy đặt kích thước mảng bằng 2^n để phép nhân có thể được chuyển thành phép toán dịch chuyển nhanh chóng*
- Đặt các case trong phạm vi hẹp
  - *Nếu số case trong câu lệnh switch nằm trong phạm vi hẹp, trình dịch sẽ biến đổi thành if – else if lồng nhau, và tạo thành 1 bảng các chỉ số, như vậy thao tác sẽ nhanh hơn*

Tối ưu mã
C/C++

- Đặt các trường hợp thường gặp trong lệnh switch lên đầu
  - *Khi số các trường hợp tuyển chọn là nhiều và tách biệt, trình dịch sẽ biến lệnh switch thành các nhóm if – else if lồng nhau. Nếu bố trí các case thường gặp lên trên, việc thực hiện sẽ nhanh hơn*
- Tái tạo các switch lớn thành các switches lồng nhau
  - *Khi số cases nhiều, hãy chủ động chia chúng thành các switch lồng nhau, nhóm 1 gồm những case thường gặp, và nhóm 2 gồm những case ít gặp=> kết quả là các phép thử sẽ ít hơn, tốc độ nhanh hơn*

Ví dụ

```
switch (queue) {
    case 0: letter = ‘D'; break;
    case 1: letter = ‘H'; break;
    case 2: letter = ‘B'; break;
    case 3: letter = ‘K'; break;
}
// Hoặc có thể là:
if (queue == 0)
letter = ‘D';
else if (queue == 1)
letter = ‘H';
else if (queue == 2)
letter = ‘B';
else letter = ‘K';
```

```
static char *classes="DHBK";
letter = classes[queue];
```

Tối ưu mã
C/C++

- Minimize local variables
  - *Các biến cục bộ được cấp phát và khởi tạo khi hàm được gọi, và giải phóng khi hàm kết thúc, vì vậy mất thời gian*
- Khai báo các biến cục bộ trong phạm vi nhỏ nhất
- Hạn chế số tham số của hàm
- Với các tham số và giá trị trả về > 4 bytes, hãy dùng tham chiếu

Tối ưu mã
C/C++

- Đừng định nghĩa giá trị trả về, nếu không sử dụng ***void***
- Lưu ý vị trí của tham chiếu tới code và data
  - *Các dữ liệu hoặc code được lưu trong bộ nhớ cache để tham khảo về sau (nếu được). Việc tham khảo từ bộ nhớ cache sẽ nhanh hơn. Vì vậy mã và dữ liệu được sử dụng cùng nhau thì nên được đặt với nhau. Điều này với object trong C++ là đương nhiên. Với C: Không dùng biến tổng thể, dùng biến cục bộ…*

Tối ưu mã
C/C++

- Nên dùng int thay vì char hay short (mất thời gian convert), nếu biết int không âm, hãy dùng unsigned int
- Hãy định nghĩa các hàm khởi tạo đơn giản
- Thay vì gán, hãy khởi tạo giá trị cho biến
- Hãy dùng danh sách khởi tạo trong hàm khởi tạo

```
Employee::Employee(String name, String designation) {
    m_name = name;
    m_designation = designation;
}
/* === Optimized Version === */
Employee::Employee(String name, String designation):
m_name(name), m_destignation (designation) { }
```

- Đừng định nghĩa các hàm ảo tùy hứng: "just in case" virtual functions
- Các hàm gồm 1 đến 3 dòng lệnh nên định nghĩa inline

Sử dụng bộ nhớ và con trỏ

- Con trỏ (pointer) có thể được gọi là một trong những "niềm tự hào" của C/C++, tuy nhiên thực tế nó cũng là nguyên nhân làm đau đầu cho các LTV, vì hầu hết các trường hợp sụp đổ hệ thống, hết bộ nhớ, vi phạm vùng nhớ… đều xuất phát từ việc sử dụng con trỏ không hợp lý.
- Hạn chế pointer dereference: pointer dereference là thao tác gán địa chỉ vùng nhớ dữ liệu cho một con trỏ. Các thao tác dereference tốn nhiều thời gian và có thể gây hậu quả nghiêm trọng nếu vùng nhớ đích chưa được cấp phát.

Ví dụ

```
for (int i = 0; i < max_number; i++)
{
    SchoolData->ClassData->StudentData->Array[i] = my_value;
}
```

Di chuyển pointer dereference ra ngoài vòng lặp

```
unsigned long *Tmp = SchoolData->ClassData->StudentData->Array;
for (int i = 0; i < max_number; i++)
{
    Tmp[i] = my_value;
}
```

Tận dụng đặc tính xử lý của CPU

- Để đảm bảo tốc độ truy xuất tối ưu, các bộ vi xử lý (CPU) 32-bit hiện nay yêu cầu dữ liệu sắp xếp và tính toán trên bộ nhớ theo từng offset 4-byte. Yêu cầu này gọi là memory alignment.
- Khi biên dịch một đối tượng dữ liệu có kích thước dưới 4-byte, các trình biên dịch sẽ bổ sung thêm các byte trống để đảm bảo các dữ liệu được sắp xếp theo đúng quy luật. Việc bổ sung này có thể làm tăng đáng kể kích thước dữ liệu, đặc biệt đối với các cấu trúc dữ liệu như structure, class…

Ví dụ

Theo nguyên tắc alignment 4-byte (hai biến "c" và "d" có kích thước 4 byte), các biến "a" và "b" chỉ chiếm 1 byte và sau các biến này là biến int chiếm 4 byte, do đó trình biên dịch sẽ bổ sung 3 byte cho mỗi biến này. Kết quả tính kích thước của lớp Test sẽ là 16 byte.

```
class Test
{
    bool a;
    int c;
    int d;
    bool b;
};
```

Ví dụ

Ta có thể sắp xếp lại các biến thành viên của lớp Test như sau theo chiều giảm dần kích thước

Khi đó, hai biến "a" và "b" chiếm 2 byte, trình biên dịch chỉ cần bổ sung thêm 2 byte sau biến "b" để đảm bảo tính sắp xếp 4-byte. Kết quả tính kích thước sau khi sắp xếp lại class Test sẽ là 12 byte.

```
class Test
{
    int c;
    int d;
    bool a;
    bool b;
};
```

Số phẩy động
Floating point

So sánh

```
x = x / 3.0;
```

và

```
x = x * (1.0/3.0);
```

- dùng float thay vì double
- Tránh dùng sin, exp và log (chậm gấp 10 lần *)
- Lưu ý : nếu x là float hay double thì : 3 * (x / 3) <> x
- Thứ tự tính toán: (a + b) + c <> a + (b + c).

Các quy tắc khác

- Tránh dùng ++, -- trong biểu thức lặp
  - *VD: while (n--) {...}*
- Dùng x * 0.5 thay vì x / 2.0.
- x+x+x thay vì x*3
- Mảng 1 chiều nhanh hơn mảng nhiều chiều
- Tránh dùng đệ quy

3.
Một số kỹ thuật

tăng hiệu năng chương trình

Phạm vi hàm & biến

Stack, Heap

- Khi thực hiện , vùng dữ liệu data segment của 1 chương trình được chia làm 3 phần
  - *static, stack, và heap data.*
- Static: global hay static variables
- Stack data:
  - *các biến cục bộ của chương trình con*
    - ví dụ: double_array trong ví dụ trên.
- Heap data:
  - *Dữ liệu được cấp phát động (vd, pchar trong ví dụ trên).*
  - *Dữ liệu này sẽ còn cho đến khi ta giải phóng hoặc khi kết thúc chương trình.*

Phạm vi biến

- Before

```
float f()
{ double value = sin(0.25);
//
...
}
```

- After

```
double defaultValue = sin(0.25);

float f()
{ double value = defaultValue;
//
...
}
```

Biến tĩnh
static

- Kiểu dữ liệu Static tham chiếu tới global hay static variables, chúng được cấp phát bộ nhớ lần đầu khi hàm được gọi và tồn tại cho đến lúc kết thúc chương trình.

```
int int_array[100];
int main() {
    static float float_array[100];
    double double_array[100];
    char *pchar;
    pchar = new char [100];
    /* .... */
    return (0);
}
```

Biến tĩnh
static

- Các biến khai báo trong chương trình con được cấp phát bộ nhớ khi chương trình con được gọi và chỉ bị loại bỏ khi kết thúc chương trình con.
- Khi bạn gọi lại chương trình con, các biến cục bộ lại được cấp phát và khởi tạo lại.
- Nếu bạn muốn 1 giá trị vẫn được lưu lại cho đến khi kết thúc toàn chương trình, bạn cần khai báo biến cục bộ của chương trình con đó là static và khởi tạo cho nó 1 giá trị.
  - *Việc khởi tạo sẽ chỉ thực hiện lần đầu tiên chương trình được gọi và giá trị sau khi biến đổi sẽ được lưu cho các lần gọi sau.*
  - *Bằng cách này 1 chương trình con có thể "nhớ" một vài mẩu tin sau mỗi lần được gọi.*

Hàm
inline

- Nếu 1 hàm trong C++ chỉ gồm những lệnh đơn giản, không có **for**, **while**... thì có thể khai báo inline.
  - *Inline code sẽ được chèn vào bất cứ chỗ nào hàm được gọi.*
  - *Chương trình lớn hơn*
  - *Nhanh hơn (không dùng stack– 4 bước khi 1 hàm được gọi)*

Ví dụ
Hàm inline

Ví dụ

```
#include <iostream.h>
#include <math.h>

inline double delta (double a,
double b)
{
   return sqrt (a*a + b*b);
}

void main () {
  double k = 6, m = 9;
  cout << delta (k, m) <<'\n';
  // tương đương với
  cout << sqrt (k*k + m*m) <<'\n';
}
```

Macro

```
#define max(a,b) (a>b?a:b)
```

- Các hàm Inline cũng giống như macros vì cả 2 được khai triển khi dịch - compile time
  - *macros được khai triển bởi preprocessor, còn inline functions được truyền bởi compiler.*
- Điểm khác biệt:
  - *Inline functions tuân thủ các thủ tục như 1 hàm bình thường.*
  - *Inline functions có cùng syntax như các hàm khác, chỉ có điều là có thêm từ khóa inline khi khai báo hàm.*
  - *Các biểu thức truyền như là đối số cho inline functions được tính 1 lần. Trong 1 số trường hợp, biểu thức truyền như tham số cho macros có thể được tính lại nhiều hơn 1 lần.*
  - *Bạn không thể gỡ rối cho macros, nhưng với inline functions thì có thể.*

Tính toán trước giá trị

Tính toán trước giá trị

- Nếu bạn phải tính đi tính lại 1 biểu thức, thì nên tính trước 1 lần và lưu lại giá trị, rồi dùng giá trị ấy sau này

```
int f(int i) {
     if (i < 10 && i >= 0) {
         return i * i - i;
      }
     return 0;
}
```

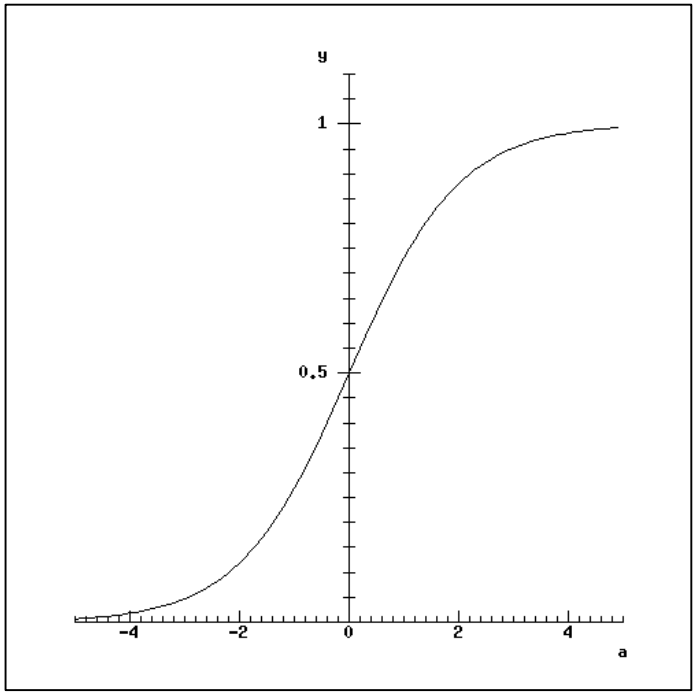
```
static int values[] =
     {0, 0, 2,3*3-3, ..., 9*9-9};

int f(int i) {
     if (i < 10 && i >= 0) {
         return values[i];
     }
     return 0;
}
```

Ví dụ
Hàm sigmoid

- Trong mô phỏng Neural Network người ta thường dùng hàm có tên sigmoid
- Với X dương lớn ta có sigmoid(x) ≅ 1
- Với x âm "lớn"

$$sigmoid\,(x) \cong 0$$

$$sigmoid\,(x) = \frac{1}{1+e^{-kx}}$$



Tính Sigmoid

$$sigmoid\,(x) = \frac{1}{1+e^{-kx}}$$

ex = 1+x/1!+ x2/2! + … + xn/n!

```
float sigmoid (float x )
{
  return 1.0 / (1.0 + exp(-x))
};
```

Ví dụ
Hàm sigmoid

- Hàm exp(-x) mất rất nhiều thời gian để tính!
  - *Những hàm kiểu này người ta phải dùng khai triển chuỗi*
    - Chuỗi Taylor /Maclaurin
    - Tính tổng các số hạng dạng $((-x)^n / n!)$
    - Mỗi số hạng lại dùng các phép toán với số chấm động
- Các mô phỏng neural network gọi hàm này trăm triệu lần trong mỗi lần thực hiện.
- Chính vì vậy , sigmoid(x) chiếm phần lớn thời gian (khoảng 70-80%)

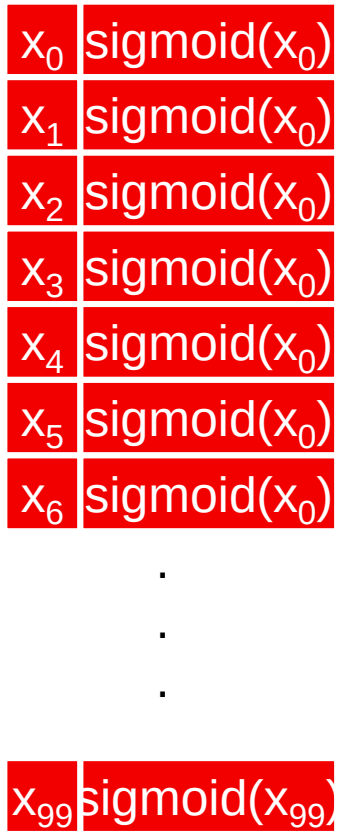
Ví dụ
Hàm sigmoid

- Thay vì tính hàm mọi lúc
  - *Tính hàm tại N điểm và xây dựng 1 mảng.*
  - *Trong mỗi lần gọi sigmoid*
    - Tìm giá trị gần nhất của x và kết quả ứng với giá trị ấy
    - Thực hiện nội suy tuyến tính - linear interpolation

x_0	sigmoid(x_0)
x_1	sigmoid(x_0)
x_2	sigmoid(x_0)
x_3	sigmoid(x_0)
x_4	sigmoid(x_0)
x_5	sigmoid(x_0)
x_6	sigmoid(x_0)
.	
.	
.	
x_{99}	sigmoid(x_{99})

Ví dụ
Tính sigmoid

if (x <x0) return (0.0);



if (x > x99) return (1.0);

Ví dụ
Hàm sigmoid

- Chọn số các điểm (N = 1000, 10000, v.v.) tùy theo độ chính xác mà bạn muốn
  - *Tốn kếm thêm không gian bọ nhớ cho mỗi điểm là 2 float hay double tức là 8 – 16 bytes/ điểm*
- Khởi tạo giá trị cho mảng khi bắt đầu thực hiện

Ví dụ
Hàm sigmoid

- Bạn đã biết X0
  - *Tính Delta = X1-X0*
  - *Tính Xmax = X0 + N * Delta;*
- Với X đã cho
  - *Tính i = (X – X0)/Delta;*
    - 1 phép trừ số thực và 1 phép chia số thực
  - *Tính sigmoid(x)*
    - 1 phép nhân float và 1 phép cộng float

Ví dụ
Hàm sigmoid

- Nếu dùng exp(x) :
  - *Mỗi lần gọi mất khoảng 300 nanoseconds với 1 máy Pentium 4 tốc độ 2 Ghz.*
- Dùng tìm kiếm trên mảng và nội suy tuyến tính :
  - *Mỗi lần gọi mất khoảng 30 nanoseconds*
- Tốc độ tăng gấp 10 lần
  - *Đổi lại phải tốn kếm thêm từ 64K to 640 K bộ nhớ.*

Tối ưu hóa biểu thức

Loại bỏ
biểu thức thông thường

- KHÔNG tính cùng một biểu thức nhiều lần!
- Một số compilers có thể nhận biết và xử lý.

```
for (i = 1; i<=10; i++) x += strlen(str);
Y = 15 + strlen(str);
```

```
len = strlen(str);
for (i = 1; i<=10; i++) x += len;
Y = 15 + len;
```

Dịch chuyển biểu thức bất biến ra khỏi vòng lặp

- Đừng lặp các biểu thức tính toán không cần thiết
- Một số Compilers có thể tự xử lý

```
for (i =0; i<100;i++)
      plot(i, i*sin(d));
```

```
M = sin(d);
for (i =0; i<100;i++)
      plot(i, i*M);
```

Sử dụng các biến đổi số học

- Trình dịch không thể tự động xử lý

```
if (a > sqrt(b))
   x = a*a + 3*a + 2
```

```
if (a * a > b)
    x = (a+1)*(a+2);
```

Không dùng các vòng lặp ngắn

```
for (i =j; i< j+3;i++)
    sum += q*i -i*7;
```

```
i = j;
sum += q*i -i*7;
i ++;
sum += q*i -i*7;
i ++;
sum += q*i-i*7;
```

Biểu thức điều kiện

Tối ưu các
biểu thức điều kiện

- Đưa các điều kiện có xác suất xảy ra cao nhất, tính toán nhanh nhất lên đầu biểu thức.
- Đối với các biểu thức luận lý, ta có thể linh động chuyển các biểu thức điều kiện đơn giản và xác suất xảy ra cao hơn lên trước, các điều kiện kiểm tra phức tạp ra sau.
- Ví dụ: ((A || B) && C) => (C && (A || B)) Vì điều kiện C chỉ cần một phép kiểm tra TRUE, trong khi điều kiện (A || B) cần đến 2 phép kiểm tra TRUE và một phép OR (||). Như vậy trong trường hợp C có giá trị FALSE, biểu thức logic này sẽ có kết quả FALSE và không cần kiểm tra thêm giá trị (A || B).

Tối ưu các
biểu thức điều kiện

- Đối với các biểu thức kiểm tra điều kiện phức tạp, ta có thể viết đảo ngược bằng cách kiểm tra các giá trị cho kết quả không thoả trước, giúp tăng tốc độ kiểm tra.
- Ví dụ: Kiểm tra một giá trị thuộc một miền giá trị cho trước.

```
    if (p <= max && p >= min && q <= max && q >= min)
    {       ...     }
    else
    {       .....   }
=>
    if (p > max || p < min || q > max || q < min)
    {   ...     }
    else
    {   ...     }
```

Tối ưu các
biểu thức điều kiện

- `(x >= min && x <= max)` có thể chuyển thành
  `(unsigned)(x - min) < (max - min)`

- `fact2_func` nhanh hơn, vi phép thử `!=` đơn giản hơn `<=`

```
int fact1_func(int n) {
    int i, fact = 1;
    for (i = 1; i <= n; i++) fact *= i;
    return (fact);
}
int fact2_func(int n) {
    int i, fact = 1;
    for (i = n; i != 0; i--) fact *= i;
    return (fact);
}
```

Lính canh Sentinels

Lính canh
Sentinels

- Tránh những kiểm tra không cần thiết

Ví dụ

```
char s[100], searchValue;
int pos, tim, size;
//Gán giá trị cho s, searchValue
…
size = strlen(s);
pos = 0;
while (pos < size) && (s[pos] != searchValue) {
    pos++;
}
if  (pos >= size) tim = 0;
else tim = 1;
```

Lính canh
Sentinels

- Ý tưởng chung
  - *Đặt giá trị cần tìm vào cuối xâu*
  - *Luôn tìm thấy!*
  - *Nếu vị trí > size => không tìm thấy !*

```
size = strlen(s);
strcat(s, searchValue);
pos = 0;
while (s[pos] != searchValue) {
    pos++;
}
if  (pos >= size) tim = 0
else tim = 1;
```

4.
Tinh chỉnh mã nguồn

Code tuning

Tăng hiệu năng chương trình

- Sau khi áp dụng các kỹ thuật xây dựng chương trình phần mềm
- Chương trình đã có tốc độ đủ nhanh
  - *Không nhất thiết phải quan tâm đến viêc tối ưu hóa hiệu năng*
  - *Chỉ cần giữ cho chương trình đơn giản và dễ đọc*
- Hầu hết các thành phần của 1 chương trình có tốc độ đủ nhanh
  - *Thường chỉ một phần nhỏ làm cho chương trình chạy chậm*
  - *Tối ưu hóa riêng phần này nếu cần*

Tăng hiệu năng chương trình

- Các bước làm tăng hiệu năng thực hiện chương trình
  - *Tính toán thời gian thực hiện của các phần khác nhau trong chương trình*
  - *Xác định các "hot spots" – đoạn mã lệnh đòi hỏi nhiều thời gian thực hiện*
  - *Tối ưu hóa phần chương trình đòi hỏi nhiều thời gian thực hiện*
  - *Lặp lại các bước nếu cần*

Tăng hiệu năng chương trình

- Cấu trúc dữ liệu tốt hơn, giải thuật tốt hơn
  - *Cải thiện độ phức tạp tiệm cận (asymptotic complexity)*
    - Tìm cách khống chế tỉ lệ giữa số phép toán cần thực hiện và số lượng các tham số đầu vào
    - Ví dụ: thay giải thuật sắp xếp có độ phức tạp **O(n2)** bằng giải thuật có độ phức tạp **O(n log n)**
  - *Cực kỳ quan trọng khi lượng tham số đầu vào rất lớn*
  - *Đòi hỏi LTV phải nắm vững kiến thức về CTDL và giải thuật*

Tăng hiệu năng chương trình

- Mã nguồn tốt hơn: viết lại các đoạn lệnh sao cho chúng có thể được trình dịch tự động tối ưu hóa và tận dụng tài nguyên phần cứng
  - *Cải thiện các yếu tố không thể thay đổi*
    - Ví dụ: Tăng tốc độ tính toán bên trong các vòng lặp: từ 1000n thao tác tính toán bên trong vòng lặp xuống còn 10n thao tác tính toán
  - *Cực kỳ quan trọng khi 1 phần của chương trình chạy chậm*
  - *Đòi hỏi LTV nắm vững kiến thức về phần cứng, trình dịch và quy trình thực hiện chương trình*

Tinh chỉnh
mã nguồn

- Thay đổi mã nguồn đã chạy thông theo hướng hiệu quả hơn nữa
- Chỉ thay đổi ở phạm vi hẹp, ví dụ như chỉ liên quan đến 1 chương trình con, 1 tiến trình hay 1 đoạn mã nguồn
- Không liên quan đến việc thay đổi thiết kế ở phạm vi rộng, nhưng có thể góp phần cải thiện hiệu năng cho từng phần trong thiết kế tổng quát

Tinh chỉnh mã nguồn

- Có 3 cách tiếp cận để cải thiện hiệu năng thông qua cải thiện mã nguồn
  - *Lập hồ sơ mã nguồn (profiling): chỉ ra những đoạn lệnh tiêu tốn nhiều thời gian thực hiện*
  - *Tinh chỉnh mã nguồn (code tuning): tinh chỉnh các đoạn mã nguồn*
  - *Tinh chỉnh có chọn lựa (options tuning): tinh chỉnh thời gian thực hiện hoặc tài nguyên sử dụng để thực hiện chương trình*

Tinh chỉnh
mã nguồn

- Khi nào cần cải thiện hiệu năng theo các hướng này
- Sau khi đã kiểm tra và gỡ rối chương trình
  - *Không cần tinh chỉnh 1 chương trình chạy chưa đúng*
  - *Việc sửa lỗi có thể làm giảm hiệu năng chương trình*
  - *Việc tinh chỉnh thường làm cho việc kiểm thử và gỡ rối trở nên phức tạp*
- Sau khi đã bàn giao chương trình
  - *Duy trì và cải thiện hiệu năng*
  - *Theo dõi việc giảm hiệu năng của chương trình khi đưa vào sử dụng*

Tinh chỉnh mã nguồn và tăng hiệu năng chương trình

- Việc giảm thiểu số dòng lệnh viết bằng 1 NNLT bậc cao KHÔNG có nghĩa là
  - *Làm tăng tốc độ chạy chương trình*
  - *Làm giảm số lệnh viết bằng ngôn ngữ máy*

```
for (i = 1;i<11;i++)  a[i] = i;
```

```
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 ;
```

Language	*for*-Loop Time	Straight-Code Time	Time Savings	Performance Ratio
Visual Basic	8.47	3.16	63%	2.5:1
Java	12.6	3.23	74%	4:1

Tinh chỉnh mã nguồn và tăng hiệu năng chương trình

- Luôn định lượng được hiệu năng cho các phép toán
- Hiệu năng của các phép toán phụ thuộc vào:
  - *Ngôn ngữ lập trình*
  - *Trình dịch / phiên bản sử dụng*
  - *Thư viện / phiên bản sử dụng*
  - *CPU*
  - *Bộ nhớ máy tính*
- Hiệu năng của việc tinh chỉnh mã nguồn trên các máy khác nhau là khác nhau.

Tinh chỉnh mã nguồn và tăng hiệu năng chương trình

- Một số kỹ thuật viết mã hiệu quả được áp dụng để tinh chỉnh mã nguồn
- Nhưng nhìn chung không nên vừa viết chương trình vừa tinh chỉnh mã nguồn
  - *Không thể xác định được những nút thắt trong chương trình trước khi chạy thử toàn bộ chương trình*
  - *Việc xác định quá sớm các nút thắt trong chương trình sẽ gây ra các nút thắt mới khi chạy thử toàn bộ chương trình*
  - *Nếu vừa viết chương trình vừa tìm cách tối ưu mã nguồn, có thể làm sai lệch mục tiêu của chương trình*

5.
Một số kỹ thuật

Tinh chỉnh mã nguồn

Một số kỹ thuật tinh chỉnh mã nguồn

- *Tinh chỉnh các biểu thức logic*
- *Tinh chỉnh các vòng lặp*
- *Tinh chỉnh việc biến đổi dữ liệu*
- *Tinh chỉnh các biểu thức*
- *Tinh chỉnh dãy lệnh*
- *Viết lại mã nguồn bằng ngôn ngữ Assembler*

Tinh chỉnh các biểu thức logic

- Không kiểm tra khi đã biết kết quả rồi
  - *Initial code*

```
if ( 5 < x ) && ( x < 10 ) ….
```

  - *Tuned code*

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

Tinh chỉnh các biểu thức logic

- Không kiểm tra khi đã biết kết quả rồi

Ví dụ

```
negativeInputFound = False;
for ( i = 0; i < iCount; i++ ) {
    if ( input[ i ] < 0 ) {
        negativeInputFound = True;
    }
}
```

Dùng break

Language	Straight Time	Code-Tuned Time	Time Savings
C++	**4.27**	**3.68**	**14%**
Java	4.85	3.46	29%

Tinh chỉnh các biểu thức logic

- Sắp xếp thứ tự các phép kiểm tra theo tần suất xảy ra kết quả đúng
  - *Initial code*

```
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
```

Tinh chỉnh các biểu thức logic

- Sắp xếp thứ tự các phép kiểm tra theo tần suất xảy ra kết quả đúng
  - *Tuned code*

```
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
```

Language	Straight Time	Code-Tuned Time	Time Savings
C#	0.220	0.260	-18%
Java	2.56	2.56	0%
Visual Basic	**0.280**	**0.260**	**7%**

Tinh chỉnh các biểu thức logic

- Sắp xếp thứ tự các phép kiểm tra theo tần suất xảy ra kết quả đúng
  - *Tuned code: chuyển lệnh switch thành các lệnh if - then - else*

Language	Straight Time	Code-Tuned Time	Time Savings
C#	0.630	0.330	48%
Java	0.922	0.460	50%
Visual Basic	**1.36**	**1.00**	**26%**

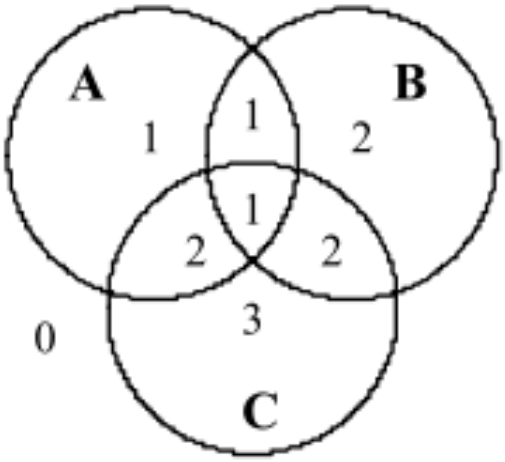
Tinh chỉnh các biểu thức logic

- So sánh hiệu năng của các lệnh có cấu trúc tương đương

Language	*case*	*if-then-else*	Time Savings	Performance Ratio
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

Tinh chỉnh các biểu thức logic

- Thay thế các biểu thức logic phức tạp bằng bảng tìm kiếm kết quả

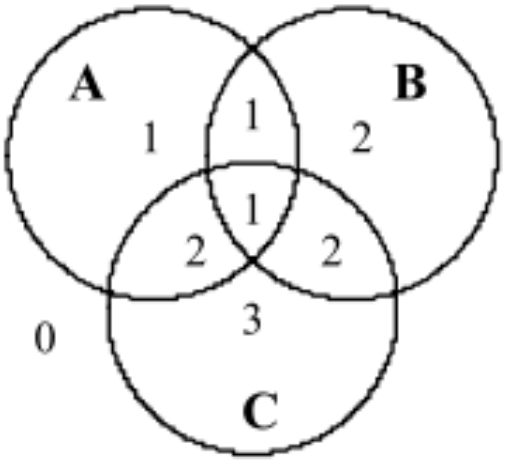


Initial code

```
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;
}
```

Tinh chỉnh các biểu thức logic

- Thay thế các biểu thức logic phức tạp bằng bảng tìm kiếm kết quả




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

Tuned code

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

Tinh chỉnh các vòng lặp

- Loại bỏ bớt việc kiểm tra điều kiện bên trong vòng lặp
  - *Initial code*

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

Tinh chỉnh các vòng lặp

- Loại bỏ bớt việc kiểm tra điều kiện bên trong vòng lặp
  - *Tuned code*

```
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 ];
    }
}
```

Language	Straight Time	Code-Tuned Time	Time Savings
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%

Tinh chỉnh các vòng lặp

- Nếu các vòng lặp lồng nhau, đặt vòng lặp xử lý nhiều công việc hơn bên trong
  - *Initial code*

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

  - *Tuned code*

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

Tinh chỉnh các vòng lặp

- Một số kỹ thuật viết các lệnh lặp hiệu quả đã học
  - *Ghép các vòng lặp với nhau*
  - *Giảm thiểu các phép tính toán bên trong vòng lặp nếu có thể*

```
for (i=0; i<n; i++) {
    balance[i] += purchase->allocator->indiv->borrower;
    amounttopay[i] = balance[i]*(prime+card)*pcentpay;
}
newamt = purchase->allocator->indiv->borrower;
payrate = (prime+card)*pcentpay;
for (i=0; i<n; i++) {
    balance[i] += newamt;
    amounttopay[i] = balance[i]*payrate;
}
```

Tinh chỉnh việc
biến đổi dữ liệu

- Một số kỹ thuật viết mã hiệu quả đã học:
  - *Sử dụng kiểu dữ liệu có kích thước nhỏ nếu có thể*
  - *Sử dụng mảng có số chiều nhỏ nhất có thể*
  - *Đem các phép toán trên mảng ra ngoài vòng lặp nếu có thể*
  - *Sử dụng các chỉ số phụ*
  - *Sử dụng biến trung gian*
  - *Khai báo kích thước mảng = 2^n*

Tinh chỉnh các biểu thức

- Thay thế phép nhân bằng phép cộng
- Thay thế phép lũy thừa bằng phép nhân
- Thay việc tính các hàm lượng giác bằng cách gọi các hàm lượng giác có sẵn
- Sử dụng kiểu dữ liệu có kích thước nhỏ nếu có thể
  - *long int* → *int*
  - *floating-point* → *fixed-point, int*
  - *double-precision* → *single-precision*
- Thay thế phép nhân đôi / chia đôi số nguyên bằng các toán tử bit: <<, >>
- Sử dụng hằng số hợp lý
- Tính trước kết quả
- Sử dụng biến trung gian
- Sử dụng các hàm inline

Viết lại mã nguồn bằng Assembler

- Viết chương trình hoàn chỉnh bằng 1 NNLT bậc cao
- Kiểm tra tính chính xác của toàn bộ chương trình
- Nếu cần cải thiện hiệu năng thì áp dụng kỹ thuật lập hồ sơ mã nguồn để tìm "hot spots" (chỉ khoảng 5 % chương trình thường chiếm 50% thời gian thực hiện, vì vậy ta có thể thường xác định đc 1 mẩu code như là hot spots)
- Viết lại những mẩu nhỏ các lệnh bằng assembler để tăng tốc độ thực hiện

Tận dụng khả năng của trình dịch

- Trình dịch có thể thực hiện 1 số thao tác tôi ưu hóa tự động
  - *Cấp phát thanh ghi*
  - *Lựa chọn lệnh để thực hiện và thứ tự thực hiện lệnh*
  - *Loại bỏ 1 số dòng lệnh kém hiệu quả*
- Nhưng trình dịch không thể tự xác định
  - *Các hiệu ứng phụ (side effect) của hàm hay biểu thức: ngoài việc trả ra kết quả, việc tính toán có làm thay đổi trạng thái hay có tương tác với các hàm/biểu thức khác hay không*
  - *Hiện tượng nhiều con trỏ trỏ đến cùng 1 vùng nhớ (memory aliasing)*
- Tinh chỉnh mã nguồn có thể giúp nâng cao hiệu năng
  - *Chạy thử từng đoạn chương trình để xác định "hot spots"*
  - *Đọc lại phần mã viết bằng assembly do trình dịch sản sinh ra*
  - *Xem lại mã nguồn để giúp trình dịch làm tốt công việc của nó*

Khai thác hiệu quả phần cứng

- Tốc độ của 1 tập lệnh thay đổi khi môi trường thực hiện thay đổi
- Dữ liệu trong thanh ghi và bộ nhớ đệm được truy xuất nhanh hơn dữ liệu trong bộ nhớ chính
  - *Số các thanh ghi và kích thước bộ nhớ đệm của các máy tính khác nhau*
  - *Cần khai thác hiệu quả bộ nhớ theo vị trí không gian và thời gian*
- Tận dụng các khả năng để song song hóa
  - *Pipelining: giải mã 1 lệnh trong khi thực hiện 1 lệnh khác*
    - Áp dụng cho các đoạn mã nguồn cần thực hiện tuần tự
  - *Superscalar: thực hiện nhiều thao tác trong cùng 1 chu kỳ đồng hồ (clock cycle)*
    - Áp dụng cho các lệnh có thể thực hiện độc lập
  - *Speculative execution: thực hiện lệnh trước khi biết có đủ điều kiện để thực hiện nó hay không*

Tổng kết

- Hãy lập trình một cách thông minh, đừng quá cứng nhắc
  - *Không cần tối ưu 1 chương trình đủ nhanh*
  - *Tối ưu hóa chương trình đúng lúc, đúng chỗ*
- Tăng tốc chương trình
  - *Cấu trúc dữ liệu tốt hơn, giải thuật tốt hơn: hành vi tốt hơn*
  - *Các đoạn mã tối ưu: chỉ thay đổi ít*
- Các kỹ thuật tăng tốc chương trình
  - *Tinh chỉnh mã nguồn theo hướng*
    - Giúp đỡ trình dịch
    - Khai thác khả năng phần cứng

Thanks!

Any questions?

Email me at trungtt@soict.hust.edu.vn