Chương 9

Gỡ lỗi, kiểm thử và tinh chỉnh mã nguồn

Tổng quan

Mục đích:

Giới thiệu về các cách tiếp cận khác nhau của việc gỡ lỗi, kiểm thử và tinh chỉnh mã nguồn

Mục tiêu:

Nắm vững các kĩ thuật của việc gỡ lỗi, các phương pháp kiểm thử, độ bao phủ của kiểm thử và tinh chỉnh mã nguồn

Nội dung

1. Gỡ lỗi

1.1. Khái niệm

- Gỡ lỗi là quá trình định vị và gỡ bỏ các lỗi của chương trình
- Ước tính có tới 85% thời gian của việc gỡ lỗi là định vị ra nơi xảy ra lỗi và 15% là sửa chữa các lỗi này
- Nếu chương trình được thiết kế với cấu trúc tốt, được viết bằng phong cách lập trình tốt và áp dụng các kỹ thuật viết chương trình hiệu quả, bẫy lỗi thì chi phí cho việc gỡ rối sẽ được giảm thiểu.

1.2. Phân loại lỗi

- Có thể phân loại thành lỗi syntax, lỗi run-time và lỗi logic.
- Lỗi syntax:
  - Các lỗi khiến chương trình bị sai cú pháp và không thể biên dịch được.
  - Ngoài ra cũng có thể bao gồm các cảnh báo của trình biên dịch như: biến cục bộ không được dùng hoặc gọi đến hàm chưa được khai báo (vẫn chạy được nếu dùng dynamic linking)
- Lỗi run-time: các lỗi chỉ phát hiện ra khi tiến hành chạy chương trình
- Lỗi logic: chương trình vẫn chạy đúng nhưng do tư duy sai, thuật toán sai dẫn đến sai kết quả, là loại lỗi khó phát hiện nhất

1.2. Phân loại lỗi

- Các lỗi run-time thường gặp:
  - Truy cập phần tử ngoài mảng
  - Lỗi gán không hợp lệ: biến toàn cục vô ý bị chỉnh sửa, thao tác gán thay vì so sánh (`a = b` thay vì `a == b`)
  - Sử dụng các biến chưa được khởi tạo ban đầu
  - Các case thực hiện xuyên suốt cho nhau do thiếu break
  - Không giải phóng bộ nhớ
  - Viết nhầm điều kiện (x < 5 thay vì viết x > 5)
  - Xử lý file (quên không đóng file hoặc mở/đóng file liên tục)

1.3. Quy trình gỡ lỗi

Quy trình sẽ trải qua các bước sau:

1) Xác định/mô tả về lỗi
2) Thu thập dữ liệu về trường hợp xảy ra lỗi
3) Dự đoán hoặc định vị về nơi/thời điểm xảy ra lỗi
4) Chạy lại hoặc dùng gdb để kiểm tra dự đoán/định vị của mình
5) Nếu bước (4) phát hiện ra dự đoán/định vị của mình sai, quay lại bước (2)
6) Nếu bước (4) khẳng định dự đoán/định vị, tiến hành sửa lỗi

1.3. Quy trình gỡ lỗi (2)

Để có thể đưa ra các dự đoán/định vị, hãy cố gắng trả lời các câu hỏi sau:

1. Liệu có chú thích nào bị đóng không đúng cách (khiến một số câu lệnh cũng bị biến thành chú thích)
2. Liệu tất cả các biến đều được khởi tạo?
3. Liệu tất cả các vòng lặp đều kết thúc?
4. Liệu tất cả các tham số truyền vào cho các hàm đều hợp lệ?
5. Liệu các khối { } đều đặt đúng chỗ?

1.4. Lời khuyên khi gỡ lỗi

- Giữ lại một bản mô tả kĩ lưỡng về lỗi (người phát hiện lỗi, thời điểm, tình huống tái hiện lỗi, mức độ nguy hiểm, ai đã sửa, thời điểm sửa, phiên bản sửa)
- Giữ lại một bản sao chép mã nguồn trước khi cố gắng sửa lỗi. Như vậy có thể quay lại được phiên bản cũ nếu lỗi sửa mãi không được hoặc quá trình sửa lại gây thêm nhiều lỗi khác.
- Khi sửa xong phải kiểm thử tích hợp nhằm xác nhận phiên bản mới không ảnh hưởng đến mã nguồn cũ.

1.4. Lời khuyên khi gỡ lỗi (2)

- Cần ghi lại các chú thích để lý giải nguyên nhân sửa đổi của mình (Cả trong mã nguồn và trên github/gitlab/bất kỳ hệ quản lý mã nguồn nào)
- Hạn chế giả sử rằng lỗi là do máy tính bị lỗi phần cứng hoặc phần mềm hệ thống lỗi.
- Cố gắng tổng quát hóa quy luật/nguyên nhân xảy ra lỗi
- Có thể chèn thêm các câu lệnh printf để dò lỗi được nhanh hơn (thay vì debug từng dòng lệnh)

1.4. Lời khuyên khi gỡ lỗi (3)

- Có thể chèn thêm các câu lệnh `printf` để dò lỗi được nhanh hơn (thay vì debug từng dòng lệnh) (tiếp):
  - Chèn tại dòng đầu của hàm (để in ra tham số)
  - Chèn tại ngay trước lệnh `return` (để in ra kết quả)
  - Chèn tại phần đầu của vòng lặp

1.4. Lời khuyên khi gỡ lỗi (4)

- Thêm các điều khiển khi biên dịch
  - Chèn thêm các câu lệnh `printf` cũng có các phiền toái của nó
  - Giải pháp thay thế là thêm các điều khiển khi biên dịch
  - Điều khiển này sẽ khiến chỉ thực thi một số câu lệnh khi một (vài) biến môi trường đạt giá trị nào đó
  - Thường được kết hợp với các `MACRO`

1.4. Lời khuyên khi gỡ lỗi (5)

```
#include <stdio.h>
int foo(int);
int main( ){
   int n = 1;
   #ifdef TEST
      printf("Reached main( ) n = %d\n", n);
   #endif
}
```


1.4. Lời khuyên khi gỡ lỗi (6)

•Khi biên dịch, sẽ thêm tùy chọn tạo ra biến TEST trong khối lệnh gcc:

```
gcc -DTEST -o exam   examp.c
```

•Điều đó cũng tương tự: `#define TEST`

•Một cách khác để hỗ trợ gỡ lỗi là sử dụng hàm assert, đây là hàm có sẵn trong thư viện assert.h

•Nếu MACRO định nghĩa NDEBUG (dùng #define hoặc tùy chọn -D của trình biên dịch) thì các hàm assert bị bỏ qua không thực thi

•Chỉ dùng hàm assert này cho mục đích kiểm thử đơn vị hoặc gỡ lỗi

1.4. Lời khuyên khi gỡ lỗi (7)

- Hàm assert có thể truyền vào bất kỳ biểu thức C nào
- Nếu biểu thức có giá trị 0 khi thực thi, thì một thông báo lỗi sẽ hiện ra và việc thực thi của chương trình sẽ dừng lại.

```
#include <stdio.h>
#include <assert.h>
int main( ) {
   int x = 9, y = 9;
   assert(x = = (y + 1)) ;
   return 0 ;
}
```

1.5. gdb

- Công cụ nổi tiếng nhất để gỡ lỗi trong ngôn ngữ C/C++ là `gdb`
- Có thể tìm hiểu thêm thông tin về công cụ này: `man gdb` hoặc `gdb -help` hoặc `ginfo`
- Nhìn chung các công cụ gỡ lỗi đều có hỗ trợ tính năng:
  - Đặt điểm dừng (breakpoint) tại vị trí bất kỳ trong mã nguồn
  - Thực thi từng câu lệnh sau điểm dừng
  - Kiểm tra giá trị của các biến
  - Phân tích các lỗi liên quan đến bộ nhớ (core dump)

2. Kiểm thử

2.1. Khái niệm
2.2. Phương pháp kiểm thử
2.3. Độ bao phủ kiểm thử
2.4. Các phương pháp đo

2. Kiểm thử

- Khó có thể khẳng định 1 chương trình lớn có làm việc chuẩn hay không
- Khi xây dựng 1 chương trình lớn, 1 lập trình viên chuyên nghiệp sẽ dành thời gian cho việc viết test code không ít hơn thời gian dành cho viết bản thân chương trình
- Lập trình viên chuyên nghiệp là người có khả năng, kiến thức rộng về các kỹ thuật và chiến lược testing

2.1. Khái niệm

- Beizer: Việc thực hiện test là để **chứng minh tính đúng đắn giữa 1 phần tử và các đặc tả của nó.**
- Myers: Là quá trình thực hiện 1 chương trình **với mục đích tìm ra lỗi.**
- IEEE: Là quá **trình kiểm tra hay đánh giá 1 hệ thống hay 1 thành phần hệ thống** một cách thủ công hay tự động để kiểm chứng rằng nó **thỏa mãn những yêu cầu đặc thù** hoặc để **xác định sự khác biệt giữa kết quả mong đợi và kết quả thực tế**

2.2. Phương pháp kiểm thử

- **Black-Box**: Testing chỉ dựa trên việc phân tích các yêu cầu - requirements (unit/component specification, user documentation, v.v.). Còn được gọi là functional testing.
- **White-Box**: Testing dựa trên việc phân tích các logic bên trong - internal logic (design, code, v.v.). (Nhưng kết quả mong đợi vẫn đến từ requirements.) Còn được gọi là structural testing.

Kiểm thử hộp đen

- Black-box testing sử dụng mô tả bên ngoài của phần mềm để kiểm thử, bao gồm các đặc tả (specifications), yêu cầu (requirements) và thiết kế (design).
- Không có sự hiểu biết cấu trúc bên trong của phần mềm
- Các dạng đầu vào có dạng hàm hoặc không, hợp lệ và không không hợp lệ và biết trước đầu hợp lệ và không biết trước đầu ra
- Được sử dụng để kiểm thử phần mềm tại mức: mô đun, tích hợp, hàm, hệ thống và chấp nhận.

Kiểm thử hộp đen (2)

- Ưu điểm của kiểm thử hộp đen là khả năng đơn giản hoá kiểm thử tại các mức độ được đánh giá là khó kiểm thử
- Nhược điểm là khó đánh giá còn bộ giá trị nào chưa được kiểm thử hay không

Kiểm thử hộp trắng

- Còn được gọi là clear box testing, glass box testing, transparent box testing, hoặc structural testing, thường thiết kế các trường hợp kiểm thử dựa vào cấu trúc bên trong của phần mềm.
- WBT đòi hỏi kĩ thuật lập trình am hiểu cấu trúc bên trong của phần mềm (các đường, luồng dữ liệu, chức năng, kết quả)
- Phương thức: Chọn các đầu vào và xem các đầu ra

Kiểm thử hộp trắng (2)

- ***Đặc điểm***
  - Phụ thuộc vào các cài đặt hiện tại của hệ thống và của phần mềm, nếu có sự thay đổi thì các bài test cũng cần thay đổi theo.
  - Được ứng dụng trong các kiểm tra ở cấp độ mô đun (điển hình), tích hợp (có khả năng) và hệ thống của quá trình test phần mềm.

Kiểm thử hộp xám

- Là sự kết hợp của kiểm thử hộp đen và kiểm thử hộp trắng khi mà người kiểm thử biết được một phần cấu trúc bên trong của phần mềm
- Khác với kiểm thử hộp đen
  - Là dạng kiểm thử tốt và có sự kết hợp các kĩ thuật của cả kiểm thử hộp đen và hộp trắng

Những ai cần biết đến kiểm thử

- Programmers
  - *White-box testing*
    - *Ưu điểm: Người triển khai nắm rõ mọi luồng dữ liệu*
    - *Nhược: Bị ảnh hưởng bởi cách thức code được thiết kế/viết*
  - Quality Assurance (QA) engineers
    - *Black-box testing*
    - *Pro: Không có khái niệm về implementation*
    - *Con: Không muốn test mọi logical paths*

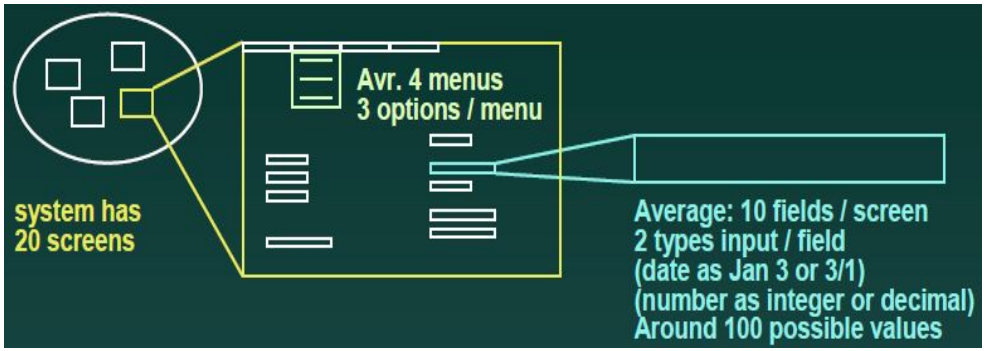
Các mức độ kiểm thử

- Unit: kiểm thử các công việc nhỏ nhất của lập trình viên để có thể lập kế hoạch và theo dõi hợp lý (vd : function, procedure, module, object class,…)
- Component: kiểm thử tập hợp các units tạo thành một thành phần (vd: program, package, task, interacting object classes,…)
- Product: kiểm thử các thành phần tạo thành một sản phẩm (subsystem, application,…)

Các mức độ kiểm thử (2)

- System: kiểm thử toàn bộ hệ thống
- Việc kiểm thử thường:
  - Bắt đầu = functional (black-box) tests,
  - Rồi thêm = structural (white-box) tests, và
  - Tiến hành từ unit level đến system level với một hoặc một vài bước tích hợp

Kiểm thử tất cả mọi thứ?



Chi phí cho 'exhaustive' testing:
20 x 4 x 3 x 10 x 2 x 100 = 480,000 tests
Nếu 1 giây cho 1 test, 8000 phút, 133 giờ, 17.7 ngày
(chưa kể nhầm lẫn hoặc test đi test lại)
nếu 10 secs = 34 wks, 1 min = 4 yrs, 10 min = 40 yrs

Bao nhiêu testing là đủ?

- Không bao giờ đủ!
- Khi bạn thực hiện những test mà bạn đã lên kế hoạch
- Khi khách hàng/người sử dụng thấy thỏa mãn
- Khi bạn đã chứng minh được/tin tưởng rằng hệ thống hoạt động đúng, chính xác
- Phụ thuộc vào risks for your system

Bao nhiêu testing là đủ? (2)

- Dùng RISK để xác định:
  - Cái gì phải test trước
  - Cái gì phải test nhiều
  - Mỗi phần tử cần test kỹ như thế nào? Tức là đâu là trọng tâm
  - Cái gì không cần test (tại thời điểm này…)

2.3. Độ bao phủ kiểm thử

- Độ bao phủ kiểm thử (test coverage) là một độ đo xác định xem các trường hợp kiểm thử có thực sự bao trùm mã ứng dụng hay không, nói cách khác có bao nhiêu phần trăm dòng mã được thực hiện khi chạy các trường hợp kiểm thử đó.
- Áp dụng cho kiểm thử hộp trắng.

2.3. Độ bao phủ kiểm thử (2)

- Khi ta đo đạc các giá trị độ bao phủ trong một tập các lần thực thi các trường hợp kiểm thử:
  - Nếu sớm đạt giá trị 100% thì nghĩa là thừa các trường hợp kiểm thử
  - Nếu toàn bộ các lần thực thi không đạt 100% nghĩa là cần bổ sung các trường hợp kiểm thử mới
  - Nếu bổ sung mà mãi không đạt được giá trị 100% nghĩa là mã nguồn có những nhánh không thể thực thi được.

2.4. Các phương pháp đo

a) Statement Coverage:

Statement Coverage đảm bảo rằng tất cả các dòng lệnh trong mã nguồn đã được kiểm tra ít nhất một lần.

```
if(A)
  F1();
F2();
```

Test Case: `A = 1`
Độ bao phủ Statement Coverage đạt 100%

```
int* ptr = NULL;
if (B)
   ptr = &variable;
*ptr = 10;
```

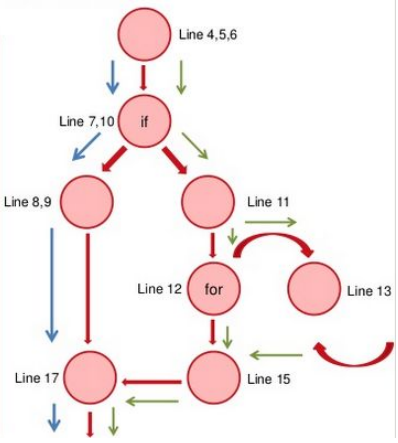
Test Case:
`B = 1`
Độ bao phủ Statement Coverage đạt 100%

2.4. Các phương pháp đo (2)

Ở ví dụ này, để Statement Coverage đạt 100%, chúng ta cần thực thi hai test case với n < 0 (màu xanh dương) và n > 0 (màu xanh lá)

Statement Coverage

```
1 #include <stdio.h>
2 main ()
3 {
4     int i, n, f;
5     printf ("n = ");
6     scanf ("%d", &n);
7     if (n < 0) {
8         printf ("Invalid: %d\n", n);
9         n= -1;
10    } else {
11        f = 1;
12        for (i = 1; i <= n; i++) {
13            f *= i;
14        }
15        printf("d! = %d\n", n, f);
16    }
17    return n;
18 }
```



2.4. Các phương pháp đo (3)

b) Branch Coverage:

Branch Coverage đảm bảo rằng tất cả các nhánh chương trình trong mã nguồn đã được kiểm tra ít nhất một lần.

```
if(A)  F1();
else   F2();
if(B)  F3()
else   F4();
```

Test Case: `A = B = 1`
Và test case: `A = B = 0`
Sẽ giúp độ bao phủ đạt 100%

```
if (A && (B || F1()))
  F2();
else
  F3();
```

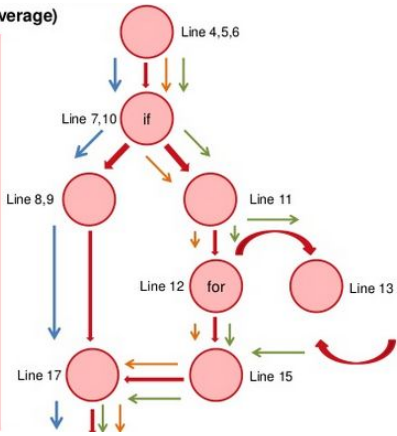
Test Case: `A = B = 1`
Và test case: `A = 0`
Sẽ giúp độ bao phủ đạt 100%
Nhưng độ bao phủ này không đảm bảo rằng hàm F1( ) sẽ được thực thi

2.4. Các phương pháp đo (4)

Ở ví dụ này, để Branch Coverage đạt 100%, chúng ta cần thực thi ba test case với n < 0 (màu xanh dương), n > 0 (màu xanh lá) và n = 0 (màu vàng)

Decision Coverage (branch coverage)

```
1 #include <stdio.h>
2 main ()
3 {
4    int i, n, f;
5    printf ("n = ");
6    scanf ("%d", &n);
7    if (n < 0) {
8        printf ("Invalid: %d\n", n);
9        n= -1;
10   } else {
11       f = 1;
12       for (i = 1; i <= n; i++) {
13           f *= i;
14       }
15       printf("d! = %d\n", n, f);
16   }
17   return n;
18 }
```

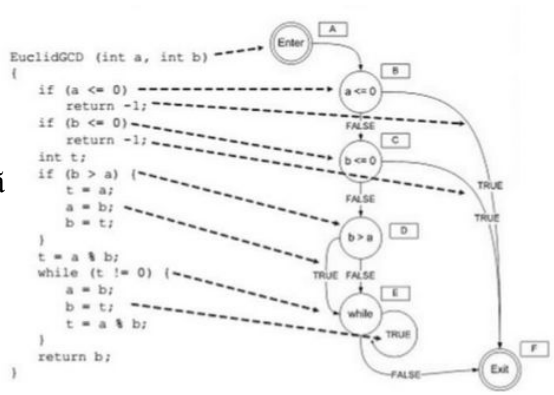


2.4. Các phương pháp đo (5)

c) Path Coverage:

Path Coverage đảm bảo rằng tất cả các đường chạy (là tổ hợp của các nhánh) chương trình trong mã nguồn đã được kiểm tra ít nhất một lần.

Với các bộ giá trị (a, b) nào sẽ khiến độ bao phủ đạt 100%???



3. Tinh chỉnh mã nguồn

3.1. Hiệu năng
3.2. Các phương pháp
3.3. Kết luận

3.1. Hiệu năng

Sau khi áp dụng các kỹ thuật xây dựng CT PM:

- CT đã có tốc độ đủ nhanh
  - Không nhất thiết phải quan tâm đến viêc tối ưu hóa hiệu năng
  - Chỉ cần giữ cho CT đơn giản và dễ đọc
- Hầu hết các thành phần của một CT có tốc độ đủ nhanh
  - Thường chỉ một phần nhỏ làm cho CT chạy chậm
  - Tối ưu hóa riêng phần này nếu cần
- Các bước làm tăng hiệu năng thực hiện CT
  - Tính toán thời gian thực hiện của các phần khác nhau trong CT
  - Xác định các "hot spots" – đoạn mã lệnh đòi hỏi nhiều thời gian thực hiện
  - **Tối ưu hóa phần CT đòi hỏi nhiều thời gian thực hiện**
  - Lặp lại các bước nếu cần

Tối ưu hóa hiệu năng của CT ?

- 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 $O(n^2)$ 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

Tối ưu hóa hiệu năng của CT ?

- 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 CT chạy chậm
  - Đòi hỏi LTV nắm vững kiến thức về phần cứng, trình dịch và quy trình thực hiện CT

 Tinh chỉnh mã nguồn

Tinh chỉnh mã nguồn là gì ?

- 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 một chương trình con, một tiến trình hay một 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

Cải thiện hiệu năng qua cải thiện mã nguồn

- 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 CT
- 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 CT chạy chưa đúng
    - Việc sửa lỗi có thể làm giảm hiệu năng CT
    - Việc tinh chỉnh thường làm cho việc kiểm thử và gỡ rối trở nên phức tạp
  - Sau khi đã bàn giao CT
    - Duy trì và cải thiện hiệu năng
    - Theo dõi việc giảm hiệu năng của CT khi đưa vào sử dụng

Tương quan hiệu năng và tinh chỉnh mã nguồn

- Việc giảm thiểu số dòng lệnh viết bằng 1 NNLT bậc cao KHÔNG có nghĩa là :
  - Làm tăng tốc độ chạy CT
  - 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

Tương quan hiệu năng và tinh chỉnh mã nguồn

- 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.

Tương quan hiệu năng và tinh chỉnh mã nguồn

- 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

Hiệu năng chương trình

- Băng thông thiết bị (Tốc độ tăng dần):user input device, tape drives, network, CDROM, hard drive, memory mapped local BUS device (graphics memory), uncached main memory, external cached main memory, local/CPU cached memory, local variables (registers.)
- Tốc độ thực hiện các phép toán : Lượng giác > Căn > % > / > * > - > + > << > >>
- Tốc độ thực hiện lệnh : indirect function calls, switch() statements, fixed function calls, if() statements, while() statements

3.2. Các phương pháp

- 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
- Lưu ý: Càng thay đổi nhiều thì càng không cải thiện được hiệu năng

3.2.1. 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 )
     ....
```


3.2.1. Tinh chỉnh các biểu thức logic

- Không kiểm tra khi đã biết kết quả rồi
- Ví dụ: tinh chỉnh như thế nào ???

```
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%

3.2.1. 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
```


3.2.1. Tinh chỉnh các biểu thức logic

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%**

```
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
```


3.2.1. 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%**

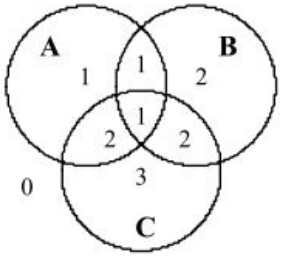
3.2.1. Tinh chỉnh các biểu thức logic

- 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

3.2.1. Tinh chỉnh các biểu thức logic

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

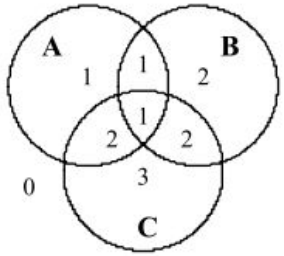


Initial code

```
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;
}
```


3.2.1. 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
};
...

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


3.2.2. 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 ];
}
}
```

```
if ( sumType == SUMTYPE_NET ) {
for ( i = 0; i < count; 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%

3.2.2. 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 ];
}
}
```

3.2.2. 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;
```


3.2.2. Tinh chỉnh các vòng lặp

- Thay thế phép nhân trong vòng lặp bằng phép cộng

```
for (i=0; i<n; i++)
    a[i] = i*conversion;
=>
sum = 0;
for (i=0; i<n; i++) {
    a[i] = sum;
    sum += conversion;
}
```

Tốt hơn:

```
a[0] = 0;
for (i=1; i<n; i++)
   a[i] = a[i-1]+conversion;
```


3.2.2. Tinh chỉnh các vòng lặp

```
for (i=0; i<r; i++)
    for (j=0; j<c; j++)
        A[j] = B[j] + C[i];

=>
for (i=0; i<r; i++) {
    temp = C[i];
    for (j=0; j<c; j++)
        A[j] = B[j] + temp;
}
```


3.2.3. 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

3.2.4. 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 phép bitwise : <<, >>
- Sử dụng hằng số hợp lý
- Tính trước kết quả
- Sử dụng biến trung gian

3.2.5. Tinh chỉnh dãy lệnh (đã học)

- Sử dụng các hàm inline

3.2.6. Viết lại bằng ngôn ngữ 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 % CT 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 = assembler để tăng tốc độ thực hiện

Giúp trình dịch làm tốt công việc của nó

- Trình dịch có thể thực hiện 1 số thao tác tối 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

Khai thác hiệu quả phần cứng (tiếp)

- 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

3.3. Kết luận

- Hãy lập trình một cách thông minh, đừng quá cứng nhắc
  - Không cần tối ưu một 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

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions!

soict.hust.edu.vn/ fb.com/groups/soict