Mục lục

[Kỹ thuật hiệu chỉnh mã nguồn 2](#_Toc4479184)

[1) Hiệu chỉnh biểu thức logic 2](#_Toc4479185)

[2) Hiệu chỉnh vòng lặp 7](#_Toc4479186)

[3) Chuyển đổi dữ liệu 15](#_Toc4479187)

[4) Hiệu chỉnh biểu thức 21](#_Toc4479188)

[5) Hiệu chỉnhchương trình con 30](#_Toc4479189)

[6) Chuyển sang hợp ngữ 31](#_Toc4479190)

**Kỹ thuật hiệu chỉnh mã nguồn**

Hiệu chỉnh mã nguồn là một chủ đề phổ biến trong hầu hết lịch sử của lập trình máy tính. Do đó, một khi bạn đã quyết định rằng bạn cần cải thiện hiệu suất và bạn muốn làm điều đó ở cấp độ mã, bạn có nhiều kỹ thuật để thực hiện điều đó.

Chương này tập trung vào việc cải thiện tốc độ và bao gồm một vài lời khuyên để làm mã ngắn hơn. Hiệu suất thường đề cập đến cả tốc độ và kích thước, nhưng kích thước giảm có xu hướng đến nhiều hơn từ việc thiết kế lại các lớp học và dữ liệu hơn từ hiệu chỉnh. Hiệu hiệu chỉnh mã đề cập đến những thay đổi quy mô nhỏ hơn là những thay đổi trong thiết kế quy mô lớn hơn.

Một số sách hiện kỹ thuật đang chỉnh là "quy tắc của ngón tay cái" hay trích dẫn các nghiên cứu đó cho thấy rằng một điều chỉnh cụ thể sẽ tạo ra hiệu ứng mong muốn. Như bạn muốn sớm nhìn thấy, khái niệm về "quy tắc của ngón tay cái" áp dụng kém để hiệu hiệu chỉnh mã. Duy nhất quy tắc đáng tin cậy của ngón tay cái là để đo lường hiệu quả của mỗi điều chỉnh trong bạn môi trường. Do đó chương này trình bày một danh mục các "điều để thử" - nhiều thứ trong số đó sẽ không hoạt động trong môi trường của bạn, nhưng một số sẽ hoạt động rất tốt.

1. **Hiệu chỉnh biểu thức logic**

Phần lớn các chương trình bao gồm các thao tác logic. Phần này mô tả cách thao tác với biểu thức logic để tạo ra lợi thế.

**Ngừng kiểm tra khi biết chắc câu trả lời**

Giả sử có một đoạn mã như :

if (5 < x) and (x < 10) then …

Khi đã xác định x nhỏ hơn 5, không cần kiểm tra điều kiện còn lại.

Một số ngôn ngữ cung cấp một hình thức đánh giá biểu hiện được gọi là "short-circuit evaluation", có nghĩa là trình biên dịch tạo mã tự động dừng lại kiểm tra ngay sau khi nó biết câu trả lời. short-circuit evaluation là một phần của các nhà khai thác tiêu chuẩn của C ++ và các toán tử điều kiện của Java.

Nếu ngôn ngữ của bạn không hỗ trợ short-circuit evaluation, bạn cần tránh sử dụng *and* hay *or*, thay bằng biểu thức logic. Với short-circuit evaluation, đoạn code trên trở thành:

if (5 < x) then

if (x < 10) then …

Các nguyên tắc không kiểm tra sau khi biết câu trả lời là tốt cho nhiều loại khác của các trường hợp tốt. Một vòng lặp tìm kiếm là trường hợp phổ biến. Nếu bạn đang quét một mảng các số đầu vào cho một giá trị âm và bạn chỉ đơn giản là cần phải biết liệu một giá trị âm là có hay không, một cách tiếp cận là để kiểm tra mỗi giá trị, thiết lập một biến *negativeFound* khi bạn tìm thấy một giá trị. Đây là cách các vòng lặp tìm kiếm sẽ xem xét:

|  |
| --- |
| negativeInputFound = false;  for ( i = 0; i < count; i++ ) {  if ( input[ i ] < 0 ) {  negativeInputFound = true;  }  } |

Một cách tiếp cận tốt hơn là nên dừng quét ngay sau khi bạn tìm thấy một giá trị âm.Những phương pháp này sẽ giải quyết các vấn đề:

* Thêm một câu lệnh break sau dòng *negativeInputFound = true*.
* Nếu ngôn ngữ của bạn không có *break*, sử dụng*goto* mà đi đến câu lệnh đầu tiên sau vòng lặp.
* Thay đổi vòng lặp *for* thành vòng lặp *while*, và kiểm tra *negativeInputFound* cũng như cách tăng số vòng lặp truy cập qua.
* Thay đổi vòng lặp *for* thành vòng lặp *while*, đặt một giá trị trọng điểm (sentinel value) trong các phần tử mảng đầu tiên sau khi nhập giá trị cuối cùng, và chỉ đơn giản là kiểm tra một giá trị âm trong các điều kiện *while*. Sau khi chấm dứt vòng lặp, xem vị trí của các giá trị tìm thấy đầu tiên là trong mảng hay không.

Đây là kết quả của sử dụng từ khoá *break* trong C++ và Java:

|  |  |  |  |
| --- | --- | --- | --- |
| Ngôn ngữ | Thời gian | Thời gian sau hiệu chỉnh | Thời gian tiết kiệm |
| C++ | 4.27 | 3.68 | 14% |
| Java | 4.85 | 3.46 | 29% |

Các tác động của sự thay đổi này thay đổi rất nhiều tùy thuộc vào có bao nhiêu giá trị mà bạn có và bao lâu bạn mong đợi để tìm một giá trị âm. Thử nghiệm này giả định một mức trung bình của 100 giá trị và giả định rằng một giá trị âm sẽ được tìm thấy 50 phần trăm thời gian.

**Sắp xếp các điều kiện theo khả năng xuất hiện**

Sắp xếp các điều kiện để một trong đó là nhanh nhất và nhiều khả năng là sự thật được thực hiện đầu tiên. Nó sẽ được dễ dàng để thả qua các trường hợp bình thường, và nếu có sự thiếu hiệu quả, họ phải ở trong xử lý các trường hợp hiếm. Nguyên tắc này áp dụng đối với từ khoá case và chuỗi if-then-elses.Dưới đây là một *case* mà phản hồi đầu vào nhập từ bàn phím trong một trình xử lý (Code Visual Basic):

|  |
| --- |
| 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 |

Các trường hợp trong lệnh case này được sắp xếp trong một cái gì đó gần với thứ tự trong ASCII. Tuy nhiên, trong một câu lệnh *case*, tác dụng thường là giống như nếu bạn đã viết một bộ lớn của *if-then-elses*, vì vậy nếu bạn có "a" như một ký tự đầu vào, các điều kiện chương trình cho dù đó là một biểu tượng toán học, một dấu chấm câu đánh dấu, một chữ số, hoặc một dấu cách trước khi xác định rằng đó là một ký tự chữ cái. Nếu bạn biết tần số khả năng các kí tự đầu vào của bạn, bạn có thể đặt các trường hợp thông thường nhất. Đây là *case* sắp xếp lại:

|  |
| --- |
| 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 |

Bởi vì các trường hợp phổ biến nhất thường được tìm thấy sớm trong mã tối ưu, hiệu quả thuần sẽ được thực hiện các thử nghiệm ít hơn. Sau đây là kết quả của tối ưu hóa này với một kết hợp điển hình các kí tự:

| **Ngôn ngữ** | **Thời gian** | **Thời gian sau tối ưu** | **Thời gian tiết kiệm** |
| --- | --- | --- | --- |
| C# | 0.220 | 0.260 | -18% |
| Java | 2.56 | 2.56 | 0% |
| Visual Basic | 0.280 | 0.260 | 7% |
| Lưu ý: Benchmarked với một sự pha trộn đầu vào của 78 % các ký tự chữ, dấu cách 17 %, và 5 % biểu tượng dấu chấm câu. | | | |

Microsoft Visual Basic kết quả như mong đợi, nhưng Java và C # kết quả không được như mong đợi. Rõ ràng đó là vì *switch - case* được cấu trúc trong C ++ và Java vì mỗi giá trị phải được liệt kê riêng hơn là trong phạm vi, C ++ và Java không được hưởng lợi từ việc tối ưu hóa như mã Visual Basic thực hiện. Kết quả này nhấn mạnh tầm quan trọng của không tuân theo bất kỳ lời khuyên tối ưu hóa việc thực hiện biên dịch một cách mù quáng cụ thể sẽ ảnh hưởng đáng kể đến kết quả.

Bạn có thể giả định rằng các mã được tạo ra bởi trình biên dịch Visual Basic cho một bộ *if – then – else* mà thực hiện các thử nghiệm tương tự như *case*sẽ tương tự. Hãy nhìn vào những kết quả:

| **Ngôn ngữ** | **Thời gian** | **Thời gian sau tối ưu** | **Thời gian tiết kiệm** |
| --- | --- | --- | --- |
| C# | 0.630 | 0.330 | 48% |
| Java | 0.922 | 0.460 | 50% |
| Visual Basic | 1.36 | 1.00 | 26% |

Kết quả là khá khác nhau. Đối với cùng một số các điều kiện, trình biên dịch Visual Basic mất khoảng năm lần như lâu dài trong các trường hợp được tối ưu hóa, bốn lần trong trường hợp tối ưu. Điều này cho thấy rằng trình biên dịch được tạo ra các mã khác nhau cho phương pháp trường hợp hơn cho các phương pháp tiếp cận *if – then – else*.  
  
Những cải tiến với *if-then-else* là phù hợp hơn so với thời với lệnh *case*. Trong C # và Visual Basic, cả hai phiên bản của cách tiếp cận case là nhanh hơn so với cả hai phiên bản của cách tiếp cận *if – then – else*, trong khi ở cả hai phiên bản Java chậm hơn. sự thay đổi trong kết quả này cho thấy một tối ưu hóa khả năng thứ ba, được mô tả trong phần tiếp theo.

**So sánh hiệu suất của cấu trúc logic**

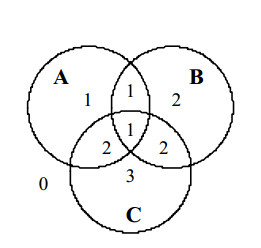
Các thử nghiệm được mô tả ở trên có thể được thực hiện bằng cách sử dụng một case hoặc if- then - else. Tùy thuộc vào môi trường, một trong hai phương pháp tiếp cận có thể làm việc tốt hơn.

| **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 |

Những kết quả này thách thức bất kỳ lời giải thích hợp lý. Trong một trong những ngôn ngữ, trường hợp là mạnh vượt trội hơn if - then - else, và trong một, if - then - else là mạnh vượt trội so với case. Trong ngôn ngữ thứ ba, sự khác biệt là khá nhỏ. Bạn có thể nghĩ rằng bởi vì C # và Java chia sẻ cú pháp tương tự như đối với lệnh báo cáo, kết quả của họ sẽ là tương tự, nhưng trên thực tế kết quả của chúng trái ngược nhau.

**Tra cứu bảng thay thế cho biểu thức phức tạp**

Trong một số trường hợp, một bảng tra cứu có thể nhanh hơn so với đi qua một chuỗi phức tạp của logic. Điểm của một chuỗi phức tạp thường là để phân loại một cái gì đó và sau đó để có một hành động dựa trên thể loại của nó. Như một ví dụ trừu tượng, giả sử bạn muốn chỉ định một số tiêu chuẩn để phân loại một cái gì đó trên cơ sở đó ba nhóm A, B, và C nó rơi vào:

****

Ví dụ sử dụng chuỗi logic phức tạp (Code C++):

|  |
| --- |
| 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;  } |

Ví dụ dùng bảng tra cứu để thay thế biểu thức logic phức tạp:

|  |
| --- |
| // 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 ]; |

Mặc dù định nghĩa của bảng là khó đọc, nếu nó cũng được mô tả sẽ không khó đọc hơn các mã cho các chuỗi logic phức tạp. Nếu định nghĩa thay đổi, bảng sẽ được dễ dàng hơn nhiều để duy trì so với chuỗi logic trước đó sẽ có được. Dưới đây là kết quả thực hiện:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** | **Performance Ratio** |
| --- | --- | --- | --- | --- |
| C++ | 5.04 | 3.39 | 33% | 1.5:1 |
| Visual Basic | 5.21 | 2.60 | 50% | 2:1 |

**Sử dụng “Lazy Evaluation”**

Giả sử, ví dụ, rằng chương trình của bạn có chứa một bảng 5000 giá trị, tạo ra toàn bộ bảng lúc khởi động, và sau đó sử dụng nó như là các chương trình thực thi. Nếu các chương trình sử dụng chỉ một phần nhỏ của các mục trong bảng, nó có thể làm cho ý nghĩa hơn để tính toán chúng khi chúng đang cần thiết chứ không phải là tất cả cùng một lúc. Một khi một mục được tính toán, nó vẫn có thể được lưu trữ để tham khảo trong tương lai (hay còn gọi là "cache").

1. **Hiệu chỉnh vòng lặp**

Bởi vì vòng lặp được thực hiện nhiều lần, các hot spot trong một chương trình thường bên trong vòng lặp. Các kỹ thuật trong phần này làm cho vòng lặp nhanh hơn.

**Unswitching**

Switchinglà ra một quyết định bên trong một vòng lặp mỗi khi nó thực hiện. Nếu quyết định không thay đổi trong khi vòng lặp được thực hiện, bạn có thể unswitch vòng lặp bằng cách làm cho các quyết định bên ngoài vòng lặp. Thông thường điều này đòi hỏi xoay vòng trong ra ngoài, đặt vòng bên trong có điều kiện hơn là đặt các điều kiện bên trong vòng lặp. Dưới đây là một ví dụ về một vòng trước khi unswitching (Code C++):

|  |
| --- |
| for ( i = 0; i < count; i++ ) {  if ( sumType == SUMTYPE\_NET ) {  netSum = netSum + amount[ i ];  }  else {  grossSum = grossSum + amount[ i ];  }  } |

Trong đoạn mã trên, điều kiện if (sumtype == SUMTYPE\_NET) được lặp đi lặp lại qua mỗi lần lặp, mặc dùmỗi lần qua vòng lặp giống nhau. Bạn có thể viết lại mã cho một đạt được tốc độ theo cách này:

|  |
| --- |
| 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 ];  }  } |

Điều này tiết kiệm được khoảng 20% thời gian:

| **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% |

Một mối nguy hiểm khác nhau cho trường hợp này là hai vòng phải được duy trì song song. Nếu biến đếm thay đổi *clientCount*, bạn phải nhớ để thay đổi nó trong cả hai nơi, đó là một ít phiền toái cho bạn và đau đầu bảo trì cho bất cứ ai khác đã làm việc với các mã.

Ví dụ này cũng cho thấy một thách thức quan trọng trong hiệu hiệu chỉnh mã: ảnh hưởng của bất kỳ hiệu chỉnh mã cụ thể là không thể đoán trước được. Mã điều chỉnh sản xuất cải tiến đáng kể trong ba trong bốn ngôn ngữ nhưng không phải trong Visual Basic. Để thực hiện tối ưu hóa đặc biệt này trong phiên bản đặc biệt này của Visual Basic sẽ sản xuất mã duy trì ít hơn mà không cần bất kỳ được bù đắp trong hoạt động. Bài học chung là bạn phải đo lường hiệu quả của mỗi tối ưu hóa cụ thể để chắc chắn về hiệu quả-không có trường hợp ngoại lệ của nó.

**Ghép các vòng lặp (Jamming)**

Jamming, hoặc "fusion" là kết quả của việc kết hợp hai vòng lặp hoạt động trên cùng một tập hợp của các yếu tố. Việc đạt được nằm trong việc cắt giảm chi phí vòng từ hai vòng một.

Ví dụ (Visual Basic):

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

Khi bạn kết hợp vòng lặp, bạn tìm thấy mã trong hai vòng mà bạn có thể kết hợp thành một. Thông thường, có nghĩa là các biến đếm của 2 vòng lặp giống nhau.

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

Kết quả, tiết kiệm thời gian:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** |
| --- | --- | --- | --- |
| C++ | 3.68 | 2.65 | 28% |
| PHP | 3.97 | 2.42 | 32% |
| Visual Basic | 3.75 | 3.56 | 4% |

Như trước đây, các kết quả khác nhau đáng kể giữa các ngôn ngữ. Việc kết hợp các vòng lặp có hai mối nguy hiểm chính. Đầu tiên, các chỉ số cho hai bộ phận đã kết hợp có thể thay đổi do đó chúng không còn tương thích. Thứ hai, bạn có thể không có khả năng kết hợp các vòng một cách dễ dàng. Trước khi bạn kết hợp các vòng, chắc chắn chúng vẫn hoạt động theo thứ tự đúng đối với phần còn lại của code.

**Giảm kích thước vòng lặp (Unrolling)**

Mục tiêu của unrolling là để giảm số lượng housekeeping. Trong chương 25, một vòng lặp là hoàn toàn trải ra và 10 dòng mã được hiển thị để được nhanh hơn 3. Trong trường hợp đó, các vòng lặp đó đi 3-10 đường đã được trải ra để tất cả 10 truy cập mảng được thực hiện riêng rẽ.

Dưới đây là một ví dụ về một vòng lặp chung (Code Java):

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

Để giảm kích thước vòng lặp, cần xử lý hai hay nhiều trường hợp trong mỗi lần lặp.

|  |
| --- |
| i = 0;  while ( i < count - 2 ) {  a[ i ] = i;  a[ i + 1 ] = i+1;  a[ i + 2 ] = i+2;  i = i + 3;  }  if ( i <= count - 1 ) {  a[ count - 1 ] = count - 1;  }  if ( i == count - 2 ) {  a[ count -2 ] = count - 2;  } |

Đây là kết quả với biến*count = 100*:

| **Language** | **Straight Time** | **Double Unrolled Time** | **Time Savings** |
| --- | --- | --- | --- |
| C++ | 1.75 | 1.01 | 42% |
| Java | 1.01 | 0.581 | 43% |
| PHP | 5.33 | 3.70 | 31% |
| Python | 2.51 | 2.79 | -12% |

**Giảm thiểu phép toán bên trong vòng lặp**

Chìa khoá để viết vòng lặp hiệu quả là giảm thiểu phép tính bên trong vòng lặp. Bạn có thể tính toán biểu thức hoặc một phần biểu thức bên ngoài vòng lặp và kết quả được sử dụngở bên trong vòng lặp. Trong một vài trường hợp, điều này còn giúp code dễđọc hơn.

Giả sử có một biểu thức chứa con trỏ phức tạp (Code C++):

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

Trong trường hợp này, việc lưu kết quả của biểu thức chứa con trỏ và một biến giúp làm code dễđọc và tăng hiệu năng chương trình.

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

Lưu kết quả của biểu thức chứa con trỏ phức tạp vào biến ngoài vòng lặp giúp tiết kiệm 3 lần truy cậpđịa chỉ thông qua con trỏ trong mỗi lần lặp. Kết quả được thể hiện như bảng dưới (*countRate = 100):*

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** |
| --- | --- | --- | --- |
| C++ | 3.69 | 2.97 | 19% |
| C# | 2.27 | 1.97 | 13% |
| Java | 4.13 | 2.35 | 43% |

**Sử dụng lính canh (Sentinel Values)**

Khi bạn có một vòng lặp kiểm tra, bạn có thể tiết kiệm thời gian bằng cách đơn giản hoá việc kiểm tra. Nếu vòng lặp dùng để tìm kiếm, một cách đơn giản là dùng lính canh, đó là giá trị đặtở cuối phạm vi tìm kiếm vàđảm bảo việc tìm kiếm kết thúc.

Ví dụ điển hình của một vòng lặp có thể được cải thiện bằng cách sử dụng lính canh là các vòng lặp tìm kiếm để kiểm tra cho dù đã tìm thấy giá trị nó tìm kiếm hay chưa. Dưới đây là các code (C#):

|  |
| --- |
| found = FALSE;  i = 0;  while ( ( !found ) && ( i < count ) ) {  if ( item[ i ] == testValue ) {  found = TRUE;  }  else {  i++;  }  }  if ( found ) {  ... |

Trong code trên, mỗi lần lặp đều kiểm tra 2 điều kiện*!found* và*i < count.* Mụcđích của*!found*  là xác định xem phần tử mong muốn được tìm thấy chưa, mụcđích của*I < count* là tránh vượt quá phạm vi của mảng. Trong vòng lặp kiểm tra giá trị của*item[]* và như vậy mỗi lần lặp có 3 biểu thức kiểm tra.

Với loại vòng lặp tìm kiếm, bạn có thể kết hợp 3 lần kiểm tra bằng một lần trong mỗi lần lặp bằng cách thêm “lính canh” vào cuối phạm vi tìm kiếm. Trong trường hợp này, có thể gán giá trị cần tìm vào cuối phạm vi tìm kiếm. Sau đó kiểm tra từng phần tử, nếu không tìm thấy cho đến khi kiểm tra phần tử cuối, bạn sẽ biết được không tông tại giá trị cần tìm trong mảng. Đây là code:

|  |
| --- |
| // set sentinel value, preserving the original value  initialValue = item[ count ];  item[ count ] = testValue;  i = 0;  while ( item[ i ] != testValue ) {  i++;  }  // check if value was found  if ( i < count ) {  ... |

Kết quả trong trường hợp các phần tử của mảng là số nguyên (mảng kích thước 100 phần tử):

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** | **Performance Ratio** |
| --- | --- | --- | --- | --- |
| C# | 0.771 | 0.590 | 23% | 1.3:1 |
| Java | 1.63 | 0.912 | 44% | 2:1 |
| Visual Basic | 1.34 | 0.470 | 65% | 3:1 |

Kết quả với mảng gồm 100 phần tử số thực 4 byte:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** |
| --- | --- | --- | --- |
| C# | 1.351 | 1.021 | 24% |
| Java | 1.923 | 1.282 | 33% |
| Visual Basic | 1.752 | 1.011 | 42% |

Kĩ thuật lính canh có thể áp dụng trong hầu hết các trường hợp tìm kiếm tuyến tính (đối với danh sách liên kết cũng tương tự mảng). Điều cần phải cẩn thận là lựa chọn giá trị của lính canh và đặt lính canh vào trong cấu trúc dữ liệu.

**Đặt vòng lặp thực hiện nhiều phép tính nhất vào bên trong**

Khi có các vòng lặp lồng nhau, điều cần nghĩ đến là vòng lặp nào đặt trong, vòng lặp nào đặt ngoài. Ví dụ đoạn code sau (Java):

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

Chìa khóa để cải thiện vòng lặp là các vòng ngoài thực hiện nhiều thường xuyên hơn so với các vòng trong. Mỗi lần vòng lặp thực hiện, nó có thể khởi tạo các chỉ số vòng lặp, tăng nó sau mỗi lần thực hiện vòng lặp, và kiểm tra xem nó vượt quá giới hạn. Tổng số lần thực hiện lặp là 100 cho các vòng ngoài và 100 \* 5 = 500 cho các vòng trong, với tổng số 600 lần lặp lại. Bằng cách chỉ đơn thuần là chuyển đổi các vòng bên trong và bên ngoài, bạn có thể thay đổi tổng số lần lặp lại đến 5 cho các vòng ngoài và 5 \* 100 = 500 cho các vòng trong, với tổng số 505 lần lặp lại. Theo những phân tích, bạn mong muốn tiết kiệm khoảng (600 - 505) / 600 = 16 phần trăm bằng cách chuyển các vòng lặp. Đây là sự khác biệt trong kết quả thực tế:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** |
| --- | --- | --- | --- |
| C++ | 4.75 | 3.19 | 33% |
| Java | 5.39 | 3.56 | 34% |
| PHP | 4.16 | 3.65 | 12% |
| Python | 3.48 | 3.33 | 4% |

Các kết quả khác nhau đáng kể, trong đó cho thấy một lần nữa rằng bạn phải đo lường hiệu quả trong môi trường cụ thể trước khi bạn có thể chắc chắn sự tố ưu đem lại hiệu quả.

**Giảm bớt phép toán phức tạp (Strength Reduction)**

Strength reduction có nghĩa là thay thế một phép toán“tốn kém” như phép nhân với một phép toánít “tốn kém” hơn như phép cộng. Đôi khi bạn sẽ có một biểu thức bên trong một vòng lặp mà phụ thuộc vào nhân chỉ số vòng lặp với một số. Phép cộng thường nhanh hơn so với phép nhân, và nếu bạn có thể tính toán cùng một số bằng cách cộng thêm sau mỗi lần lặp của vòng lặp chứ không phải bằng cách nhân, code thường sẽ chạy nhanh hơn. Dưới đây là một ví dụ về code Visual Basic sử dụng phép nhân:

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

Code sau khi thay thế phép nhân băng phép cộng:

|  |
| --- |
| incrementalCommission = revenue \* baseCommission \* discount  cumulativeCommission = incrementalCommission  For i = 0 to saleCount - 1  commission( i ) = cumulativeCommission  cumulativeCommission = cumulativeCommission + incrementalCommission  Next |

Điều quan trọng là các phép nhân ban đầu có phụ thuộc vào chỉ số vòng lặp. Trong trường hợp này, chỉ số vòng lặp là phần duy nhất của biểu thức khác nhau, vì vậy các biểu thứccó thể được viết lại nhằm tiết kiệm hơn. Dưới đây là kết quả thực tế (*saleCount = 20, các biến được tính toán kiểu số thực dấu phẩy động)*:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** |
| --- | --- | --- | --- |
| C++ | 4.33 | 3.80 | 12% |
| Visual Basic | 3.54 | 1.80 | 49% |

1. **Chuyển đổi dữ liệu**

Những thay đổi trong các kiểu dữ liệu có thể là một trợ giúp mạnh mẽ trong việc giảm kích thước chương trình và cải thiện tốc độ thực thi. thiết kế cấu trúc dữ liệu nằm ngoài phạm vi của cuốn sách này, nhưng thay đổi khiêm tốn trong việc thực hiện một kiểu dữ liệu cụ thể cũng có thể cải thiện hiệu suất. Dưới đây là một vài cách để điều chỉnh các kiểu dữ liệu của bạn.

**Sử dụng số nguyên thay cho số thực dấu phẩy động**

Các phép toán thực hiện trên số nguyên nhanh hơn số thực dấu phẩy động. Chuyển chỉ số vòng lặp từ số thực sang số nguyên giúp tiết kiệm thời gian. Ví dụ (Code Visual Basic):

|  |
| --- |
| Dim x As Single  For x = 0 to 99  a( x ) = 0  Next |

Ngược lại điều này với một vòng lặp Visual Basic tương tự một cách rõ ràng sử dụng kiểu số nguyên:

|  |
| --- |
| Dim i As Integer  For i = 0 to 99  a( i ) = 0  Next |

Dưới đây là kết quả cho mã Visual Basic này và cho mã tương tự như trong C ++ và PHP:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** | **Performance Ratio** |
| --- | --- | --- | --- | --- |
| C++ | 2.80 | 0.801 | 71% | 3.5:1 |
| PHP | 5.01 | 4.65 | 7% | 1:1 |
| Visual Basic | 6.84 | 0.280 | 96% | 25:1 |

**Sử dụng mảng với số chiều ít nhất có thể**

Theo lẽ thường, cho rằng nhiều kích thước trên mảng là đắt. Nếu bạn có thể cấu trúc dữ liệu của bạn để nó trong một mảng một chiều chứ không phải là một hai chiều hoặc mảng ba chiều, bạn có thể có thể tiết kiệm thời gian. Giả sử bạn có mã khởi tạo như thế này:

Ví dụ: Mảng 2 chiều trong Java:

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

Khi mã này được chạy với 50 hàng và 20 cột, phải mất gấp đôi thời gian với trình biên dịch Java hiện tại của tôi như khi mảng được cơ cấu lại để nó là một chiều. Dưới đây là cách mã sửa đổi sẽ xem xét:

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

Và đây là một bản tóm tắt các kết quả, với việc bổ sung các kết quả tương đương trong một số ngôn ngữ khác:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** | **Performance Ratio** |
| --- | --- | --- | --- | --- |
| C++ | 8.75 | 7.82 | 11% | 1:1 |
| C# | 3.28 | 2.99 | 9% | 1:1 |
| Java | 7.78 | 4.14 | 47% | 2:1 |
| PHP | 6.24 | 4.10 | 34% | 1.5:1 |
| Python | 3.31 | 2.23 | 32% | 1.5:1 |
| Visual Basic | 9.43 | 3.22 | 66% | 3:1 |

Các kết quả của tối ưu hóa này là tuyệt vời trong Visual Basic và Java, tốt trong PHP và Python, nhưng tầm thường trong C ++ và C #. Tất nhiên, thời gian được tối ưu hóa các biên dịch C # là cách dễ dàng nhất của nhóm, vì vậy bạn không thể quá cứng nhắc.  
Đây loạt các kết quả một lần nữa cho thấy sự nguy hiểm của bất kỳ lời khuyên sau đây hiệu chỉnh mã một cách mù quáng. Bạn không bao giờ có thể chắc chắn cho đến khi bạn thử những lời khuyên trong hoàn cảnh cụ thể của bạn.

**Giảm thiểu việc truy cập mảng**

Ngoài giảm thiểu các truy cập đến gấp đôi hoặc gấp basố chiều của mảng, nó thường có lợi thế để giảm thiểu mảng truy cập, thời gian. Một vòng lặp mà nhiều lần sử dụng một phần tử của mảng là một ứng cử viên tốt cho các ứng dụng của kỹ thuật này. Dưới đây là một ví dụ (C++) về một mảng truy cập không cần thiết:

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

Các truy cập đến*discount[discountType]* không thay đổi khi *discountLevel* thay đổi trong các vòng lặp trong. Do đó, bạn có thể di chuyển nó ra khỏi vòng lặp bên trong để bạn sẽ chỉ có một mảng truy cập mỗi thực hiện các vòng ngoài chứ không phải là một cho mỗi lần thực hiện của các vòng trong. Ví dụ tiếp theo cho thấy mã sửa đổi:

|  |
| --- |
| for ( discountType = 0; discountType < typeCount; discountType++ ) {  thisDiscount = discount[ discountType ];  for ( discountLevel = 0; discountLevel <levelCount; discountLevel++ ) {  rate[ discountLevel ] = rate[ discountLevel ] \* thisDiscount;  }  } |

Đây là kết quả:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** |
| --- | --- | --- | --- |
| C++ | 32.1 | 34.5 | -7% |
| C# | 18.3 | 17.0 | 7% |
| Visual Basic | 23.2 | 18.4 | 20% |

L*ưu ý: Benchmark được tính toán cho các trờng hợp trong đó typeCountbằng 10 và levelCountbằng 100.*

**Sử dụng chỉ số bổ sung**

Sử dụng một chỉ số bổ sung có nghĩa là thêm dữ liệu liên quan mà làm cho truy cập vào một kiểu dữ liệu hiệu quả hơn. Bạn có thể thêm các dữ liệu liên quan đến các kiểu dữ liệu chính, hoặc bạn có thể lưu trữ nó trong một cấu trúc song song.

Chỉ số độ dài xâu:

Một ví dụ của việc sử dụng một chỉ số bổ sung có thể được tìm thấy trong các chiến lược chuỗi-lưu trữ khác nhau. Trong C, chuỗi được kết thúc bằng một byte đó là thiết lập là 0. Trong định dạng chuỗi Visual Basic, một byte chiều dài giấu ở đầu mỗi chuỗi cho biết bao lâu chuỗi là. Để xác định độ dài của một chuỗi trong C, một chương trình đã bắt đầu vào lúc bắt đầu của chuỗi và đếm từng byte cho đến khi nó tìm thấy các byte đó là thiết lập là 0. Để xác định độ dài của một chuỗi Visual Basic, chương trình chỉ nhìn độ dài byte. Visual Basic byte chiều dài là một ví dụ về việc làm tăng một loại dữ liệu với một chỉ số để làm cho một số hoạt động giống như máy tính độ dài của một chuỗi nhanh hơn.

Bạn có thể áp dụng ý tưởng của lập chỉ mục cho chiều dài cho bất kỳ loại dữ liệu có độ dài biến. Nó thường hiệu quả hơn để theo dõi độ dài của cấu trúc hơn là tính toán chiều dài mỗi khi bạn cần nó.

Cấu trúc chỉ số song song, độc lập:

Đôi khi nó hiệu quả hơn để thao tác một chỉ số để một kiểu dữ liệu hơn là để thao tác các kiểu dữ liệu riêng của mình. Nếu các mục trong các loại dữ liệu lớn hoặc khó di chuyển (chẳng hạn trên đĩa), phân loại và tìm kiếm chỉ số là nhanh hơn so với làm việc với các dữ liệu trực tiếp. Nếu mỗi mục dữ liệu quá lớn, bạn có thể tạo ra một cấu trúc phụ trợ bao gồm các giá trị quan trọng và con trỏ đến thông tin chi tiết. Nếu sự khác biệt về kích thước giữa các mục dữ liệu cấu trúc và mục phụ trợ cấu trúc là đủ lớn, đôi khi bạn có thể lưu trữ các mục quan trọng trong bộ nhớ ngay cả khi mục dữ liệu phải được lưu trữ bên ngoài. Tất cả các tìm kiếm và phân loại được thực hiện trong bộ nhớ, và bạn phải truy cập vào ổ đĩa chỉ có một lần, khi bạn biết chính xác vị trí của các mục mà bạn muốn.

Sử dụng caching:

Caching có nghĩa là tiết kiệm một vài giá trị trong một cách mà bạn có thể lấy các giá trị phổ biến nhất được sử dụng một cách dễ dàng hơn so với giá trị ít được sử dụng. Nếu một chương trình đọc ngẫu nhiên các bản ghi từ một đĩa, ví dụ, một thói quen có thể sử dụng một bộ nhớ cache để lưu hồ sơ đọc thường xuyên nhất. Khi thường xuyên nhận được yêu cầu cho một bản ghi, nó sẽ kiểm tra bộ nhớ cache để xem liệu nó có hồ sơ. Nếu có, hồ sơ được trả về trực tiếp từ bộ nhớ thay vì từ đĩa.

Ngoài hồ sơ bộ nhớ đệm trên đĩa, bạn có thể áp dụng bộ nhớ đệm trong các lĩnh vực khác. Trong một chương trình font-proofing trên Microsoft Windows, nút cổ chai hiệu năng nằm trong việc lấy độ rộng của mỗi kí tự được hiển thị. Caching kí tự sử dụng gần đây nhất tăng gần gấp đôi tốc độ hiển thị.

Bạn có thể cache các kết quả tính toán tốn thời gian quá, đặc biệt là nếu các thông số để tính toán rất đơn giản. Giả sử, ví dụ, rằng bạn cần phải tính toán chiều dài của cạnh huyền của một tam giác vuông, với độ dài của hai cạnh kia. Việc thực hiện đơn giản của các thói quen sẽ trông như thế này (Java Code):

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

Nếu bạn biết rằng các giá trị cùng một xu hướng được yêu cầu nhiều lần, bạn có thể cache các giá trị theo cách này:

|  |
| --- |
| private double cachedHypotenuse = 0;  private double cachedSideA = 0;  private double cachedSideB = 0;  public double Hypotenuse(  double sideA,  double sideB  ) {  // check to see if the triangle is already in the cache  if ( ( sideA == cachedSideA ) && ( sideB == cachedSideB ) ) {  return cachedHypotenuse;  }  // compute new hypotenuse and cache it  cachedHypotenuse = Math.sqrt( ( sideA \* sideA ) + ( sideB \* sideB ) );  cachedSideA = sideA;  cachedSideB = sideB;  return cachedHypotenuse;  } |

Phiên bản thứ hai phức tạp hơn so với trước và chiếm nhiều không gian hơn, bù lại tốc độ thực hiện lớn hơn. Nhiều chương trình bộ nhớ đệm bộ nhớ cache nhiều hơn một phần tử, do đó chi phí nhiều hơn. Dưới đây là các tốc độ khác biệt giữa hai phiên bản:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** | **Performance Ratio** |
| --- | --- | --- | --- | --- |
| C++ | 4.06 | 1.05 | 74% | 4:1 |
| Java | 2.54 | 1.40 | 45% | 2:1 |
| Python | 8.16 | 4.17 | 49% | 2:1 |
| Visual Basic | 24.0 | 12.9 | 47% | 2:1 |

Sự thành công của bộ nhớ cache phụ thuộc vào chi phí tương đối của các truy cập vào một yếu tố lưu trữ, tạo ra một yếu tố uncached, và tiết kiệm một yếu tố mới trong bộ nhớ cache. Thành công cũng phụ thuộc vào mức độ thường xuyên các thông tin lưu trữ được yêu cầu. Trong một số trường hợp, thành công cũng có thể phụ thuộc vào bộ nhớ cache được thực hiện bởi phần cứng. Nói chung, càng có nhiều chi phí để tạo ra một yếu tố mới và nhiều lần cùng một thông tin được yêu cầu, có giá trị hơn một bộ nhớ cache. Như với các kỹ thuật tối ưu hóa khác, bộ nhớ đệm thêm phức tạp và có xu hướng dễ bị lỗi.

1. **Hiệu chỉnh biểu thức**

Nhiều công việc trong một chương trình được hoàn thành bên trong các biểu thức toán học hoặc logic. Biểu thức phức tạp dẫn đến sự tốn kém, phần này trình bày cách để giảm bớt chúng.

**Đồng nhất biểu thứcđại số**

Bạn có thể sử dụng đồng nhất biểu thứcđại số để giảm bớt toán tử. Ví dụ, 2 biểu thức dưới đây là tương đương:

|  |
| --- |
| not a and not b  not (a or b) |

Nếu sử dụng biểu thức thứ hai, bạn sẽ tiết kiệm một toán tử *not.*

Mặc dù việc tiết kiệm bằng cách tránh dùng một toán tử *not* gần như không hiệu quả, tuy nhiên nguyên tắc này lại rất mạnh mẽ. Jon Bentley mô tả một chương trình được kiểm tra điều kiện *sqrt(x) < sqrt(y)* (1982). Vì *sqrt(x) < sqrt(y)* khi *x < y* nên ta có thể thay điều kiện kiểm tra bằng *x < y*. Điều này tiết kiệm việc dử dụng hàm *sqrt()*. Đây là kết quả rất ấn tượng:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** | **Performance Ratio** |
| --- | --- | --- | --- | --- |
| C++ | 7.43 | 0.010 | 99.9% | 750:1 |
| Visual Basic | 4.59 | 0.220 | 95% | 20:1 |
| Python | 4.21 | 0.401 | 90% | 10:1 |

**Sử dụng giảm khối lượng**

Như đã nói, giảm khối lượng nghĩa là thay thế một hoạt động tốn kém bằng một hoạt động rẻ hơn. Dưới đây là một vài cách thay thế:

* Thay thế phép nhân bừng phép cộng
* Thay thế hàm mũ bằng phép nhân
* Thay thế tỉ số lượng giác bằng hàm lượng giác
* Thay thế số nguyên kiểu long long bằng long hoặc int
* Thay thế số kiểu dấu chấm đọng bằng dấu chấm tĩnh
* Thay thế số thực có độ chính xác kép bằng chính xác đơn
* Thay việc nhân hai hoặc chi hai một số bằng cách dịch

Giả sử bạn đánh giá một đa thức. Đa thức dạng Ãx2+ Bx +C, với A,B,C là hệ số, x là biến. mã chung cho đa thức bậc n như sau:

|  |
| --- |
| **Visual Basic Example of Evaluating a Polynomial**  value = coefficient( 0 )  For power = 1 To order  value = value + coefficient( power ) \* x^power  Next |

Nếu bạn nghĩ về việc giảm khối lương, bạn sẽ nhìn vào hàm lũy thừa với ánh mắt không tốt. Một giải pháp là thay thế hàm lũy thừa bằng phép nhân thông qua một vòng lặp, tương tự như cách giảm hối lượng ở chương trc, trong đó phép nhân thay bằng phép cộng :

|  |
| --- |
| Visual Basic Example of a Reduced-Strength Method of Evaluating a Polynomial value = coefficient( 0 )  powerOfX = x  For power = 1 to order  value = value + coefficient( power ) \* powerOfX  powerOfX = powerOfX \* x  Next |

Phương pháp này giúp loại bỏ biến PowerofX và thay thế hai phép nhân bằng một. Kế quả là:

| **Language** | **Straight Time** | **First Optimization** | **Second Optimization** | **Savings over First Optimization** |
| --- | --- | --- | --- | --- |
| Python | 3.24 | 2.60 | 2.53 | 3% |
| Visual Basic | 6.26 | 0.16 | 0.31 | -94% |

Đây là ví dụ tốt cho thấy lí thuyế không ứng với thực hành. Đoạn mã có giảm khối lượng trông như nhanh hơn, nhưng không phải. Một tình trạng có thể là giảm òng lặp đi 1, thay viif tăng vòng lặp lên 1 trong Visual Basic có thể ảnh hưởng xấu tới sự thực hiện, nhưng bạn phải đo lường giả thuyết để chắc chắn.

**Khởi chạy tại thời điểm biên dịch**

Nếu bạn đang sử dụng một hằng số đc đặt tên hoặc một số kìa ảo trong gọi chương trình, và nó chỉ là tham số, đó là đầu mối để bạn có thể tính toán trước số, đặt nó vào hằng số và tránh các lơi gọi . Các nguyên tắc tương tự áp dụng cho phép nhân, chia , công và các hoạt động khác

Có lần tôi cần tính log cơ số hai của một số , bỏ bớt phần nguyên gần nhất. Hệ thống không có hàm log cơ số 2, vì vậy tôi dã viết riêng:

log(x)base = log(x) / log(base)

Với cách này, tôi có thể viết chương trình như sau:

|  |
| --- |
| C++ Example of a Log-Base-Two Routine Based on System Routines unsigned int Log2( unsigned int x ) {  return (unsigned int) ( log( x ) / log( 2 ) );  } |

Chương trình chạy rất chậm, vì log2 không bao giơi thay đổi, , tôi thay thế log2 bằng giá trị 0.69314 , đoạn mã sẽ như thế này:

|  |
| --- |
| C++ Example of a Log-Base-Two Routine Based on a System Routine and a Constant const double LOG2 = 0.69314718;  ...  unsigned int Log2( unsigned int x ) {  return (unsigned int) ( log( x ) / LOG2 );  } |

Từ khi log() thành một chương trình tốn kém, tốn hơn nhều so với chuyển đổi hoặc phép chia, bạn hi vọng cắt giảm các lần gọi đến hàm log() đi một nửa sẽ giảm thời gian cần để chay chương trình đi một nửa, dưới đây là kết quả:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** |
| --- | --- | --- | --- |
| C++ | 9.66 | 5.97 | 38% |
| Java | 17.0 | 12.3 | 28% |
| PHP | 2.45 | 1.50 | 39% |

Trong trường hợp này, ta đoán về ý nghĩa liên qua giữa phép chia và chuyển đổi kiểu và sự ước lượng giảm 50% là rất gần. Xem xét khả năng dự đoán của các kết quả đươpcj mô tả trong chương này, độ chính xác của dự đoán của tôi trong trường hợp này chỉ chứng minh ngay cả một con sóc mù cũng có thể tìm thấy hạt dẻ

**Cảnh giác với các chương trình hệ thống.**

Chương trình hệ thống thương tốn kém và sự chính xác của nó là lãng phí .Tiêu biểu là chương trình toán hệ thống, ví dụ, được thiết kế để dưa phi hành gia lên mặt trăng trong vòng +- 2 feet của các mục tiêu. Nếu bạn không cần độ chính xác cao, bạn không cần dành thời gian để tính toán nó, hoặc

Trong ví dụ trươc, hàm log() trả lại giá trị nguyên nhưng sử dụng dấu phẩy động hàm log() để tính toán nó. Đó là quá cần thiết cho kết quả là số nguyên. Vì vậy , sau nỗ lực ddaaauf tiên, tôi đã viết laotj các bài kiểm tra số nguyên hoàn toán chính xác cho tính toán số nguyên log cơ số 2

|  |  |
| --- | --- |
| |  | | --- | | C++ Example of a Log-Base-Two Routine Based on Integers unsigned int Log2( unsigned int x ) {  if ( x < 2 ) return 0 ;  if ( x < 4 ) return 1 ;  if ( x < 8 ) return 2 ;  if ( x < 16 ) return 3 ;  if ( x < 32 ) return 4 ;  if ( x < 64 ) return 5 ;  if ( x < 128 ) return 6 ;  if ( x < 256 ) return 7 ;  if ( x < 512 ) return 8 ;  if ( x < 1024 ) return 9 ;  ...  if ( x < 2147483648 ) return 30;  return 31 ;  } | |

Chương trình này sử dụng các hoạt động dạng số nguyên không bao giờ chuyển thành dạng dấu chấm động, và phủ định hai chương trình dạng thực. Dưới đây là kết quả:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** | **Performance Ratio** |
| --- | --- | --- | --- | --- |
| C++ | 9.66 | 0.662 | 93% | 15:1 |
| Java | 17.0 | 0.882 | 95% | 20:1 |
| PHP | 2.45 | 3.45 | -41% | 2:3 |

Hầu hết các hàm siêu việt đều tính tới trường hợp tồi nhất, chúng chuyển đổi thành só thực độ chính sác kép ngay lập tức, kể cả bạn đưa một số nguyên. Nếu bạn tìm thấy một phần chặt chẽ của mã nguồn, và bạn không cân fnhieeuf chính xác, đưa cho nó sự quan tâm trước mắt của bạn

Một cách khác để tận dụng lợi thế thực tế, là trượt phải giống như chia hai. Số lần bạn chia hai mà vẫn có giá trị khác không gióng như log cơ số 2 của số đó. Dưới đâ ylà đoạn mã dựa trên nhận xét đó:

|  |
| --- |
| unsigned int Log2( unsigned int x ) {  unsigned int i = 0;  while ( ( x = ( x >> 1 ) ) != 0 ) {  i++;  }  return i ;  } |

Với những người không lập trình C++, đoạn mã là khó đọc. Các biểu thức phức tạp trong diều kiện while là một trong những ví dụ bạn cần tránh trừ khi bạn có lí do tốt để dùng nó.

Chương trình này tốn nhiều thời gian hơn 350% so với chương trình trc, từ 0,66 lên 2.4. Nhưng nó nhanh hơn so với phương pháp đầu tiên, và dễ thích nghi với 32-bit, 64-bit và các môi trường khác.

Ví dụ này nhấn mạnh giá trị cảu việc không dừng lại sau khi tối ư hóa thành công. Cách tối ưu đầu tiên tiết kiệm đươc 30-40% nhưng không gần sự tác động của cách tối ưu thứ hai hoặc thứ ba.

**Sử dụng đúng loại hăng số**

Sử dụng hằng số đc đặt tên và chữ mà cung kiểu với biến mà chúng gán cho.Nếu hằng số và biến liên qua với nó có khác kiểu, trình dịch phải chuyển đổi kiểu để gán hằng số cho biến. một trình dịch tốt sẽ chuyển đổi kiểu ngay lúc dịch , nên sẽ không ảnh hươgr tới thời gian đang thi hành

Một chương trình dịch ít tiên tiến hoặc một thông dịch tạo ra code khi đang thi hành, có thể bị mắc kẹt. Dưới đây là một số điểm khác biệt trong thi hành giữa khởi tạo biến thực x và biến nguyên i trong 2 trường hợp:

Trường hợp 1:

X=5

i=3.14

và cần chuyển kiểu, giả sử x là dấu chấm động, I là kiểu nguyên.

Trương hợp 2:

X=3.14

i=5

và không yêu cầu chuyển kiểu, đây là kết quả, và sự khác nhau giữa trình dịch một lần nữa đáng lưu ý:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** | **Performance Ratio** |
| --- | --- | --- | --- | --- |
| C++ | 1.11 | 0.000 | 100% | not measurable |
| C# | 1.49 | 1.48 | <1% | 1:1 |
| Java | 1.66 | 1.11 | 33% | 1.5:1 |
| Visual Basic | 0.721 | 0.000 | 100% | not measurable |
| PHP | 0.872 | 0.847 | 3% | 1:1 |

**Kết quả tính toán trước**

Một quyết định thiết kế ở mức thấp phổ biến là lựa chon kết quả tính toán ở đâu đó hoặc tính chúng một lần, lưu chúng, và tìm chúng nếu cần. Nếu kết quả dùng nhiều lần, thường rẻ hơn tính chúng một lần và timf chúng cho phần còn lại

Lựa chọn này thể hiện bằng nhiều cách. ở mức đơn giản nhất, bạn có thể tính toán một biểu thức trong vòng lặp ở bên ngoài thay vì bên trong. Một ví dụ về điều này xuất hiện ở chương trước. Ở mức độn phức tạp hơn, bạn có thể tạo ra một bảng tra cứu một lần khi chương trình thực hiện bắt đầu, sử dụng chúng mọi lần về sau, hoặc lưu kiết quả trong file dữ liệu hoặc nhúng nó vào chương trình

Ví dụ trong trò chơi star wars, người lập trình tính các hệ số hấp dẫn cho các khoảng cách khác nhau từ mặt trời. Các tính toán cho các hệ số trọng lực là tốn kém và ảnh hưởng hiệu suất. Chương trình thừa nhậ có tương đối ít sai khác khoảng cách từ mặt trời, tuy nhiên, lập trình viên có thể tính toán trước hệ số trọng lượng và lưu trữ chúng trong một mảng 10 phần tử. Mảng này tìm kiếm nhanh hơn là các tính toán tốn kém

Giả sử bạn có chương trình tính toán số tiền vay cho ô tô, đoạn mã sẽ như sau:

|  |
| --- |
| **Java Example of a Complex Computation That Could Be Precomputed**  double ComputePayment(  long loanAmount,  int months,  double interestRate  ) {  return loanAmount /  (  ( 1.0 - Math.pow( ( 1.0 + ( interestRate / 12.0 ) ), -months ) ) / ( interestRate / 12.0 )  );  } |

Công thức tính toán tiền vay là phức tạp và tốn kém. Đặt thông tin vào bảng thay vì tính nó sẽ đơn giản hơn

Bảng có thể to như thê nào? Biến rộng nhất là loanAmount. Biến interestRate có thể trải từ 5% thông qua 20% bởi một phần tư, nhưng chỉ có 61 khác biệt tỉ lệ, month trải từ 12 tơi 72, nhưng chỉ có 61 gian đoạn khác nhau, loanAmount có thể hình dung từ $1000 thông qua $100000, thường nhiều mục hơn bạn mong muốn trong một bang tra cứu

Hầu hết các tính toán không dựa vào loaAmount, tuy nhiên, bạn có thể đặt một phần ngốc nghếch vào tính toán(mẫu số của biểu thức lớn hơn) vào bảng mà được đánh chỉ số bỏi interestRate và month.Bạn có thể tính lại loanAmount mỗi lần:

|  |
| --- |
| **Java Example of Precomputing a Complex Computation**  double ComputePayment(  long loanAmount,  int months,  double interestRate  ) {  int interestIndex =  Math.round( ( interestRate - LOWEST\_RATE ) \* GRANULARITY \* 100.00 );  return loanAmount / loanDivisor[ interestIndex ][ months ];  } |

Trong đoạn mẫ này, việc tính toán thừa đã thay bằng các chỉ số mảng và mảng duy nhất. Dưới đây là kết quả việc thay đổi:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** | **Performance Ratio** |
| --- | --- | --- | --- | --- |
| Java | 2.97 | 0.251 | 92% | 10:1 |
| Python | 3.86 | 4.63 | -20% | 1:1 |

Bạn không cần bảng dể tăng sự thi hành, bạn có thể đạt được bằng tính trước một biểu thức. Đoạn mã tương tự như đoạn mã ví dụ trước, tăng kha năng của một số loại tính toán trước khác nhau. Giả sử bạn có đoạn mã tính toán tienf trả cho nhiều khoản vay, như ở đây:

|  |
| --- |
| **Java Example of a Second Complex Computation That Could Be Precomputed**  double ComputePayments(  int months,  double interestRate  ) {  for ( long loanAmount = MIN\_LOAN\_AMOUNT; loanAmount < MAX\_LOAN\_AMOUNT;  loanAmount++ ) {  payment = loanAmount / (  ( 1.0 - Math.pow( 1.0+(interestRate/12.0), - months ) ) /  ( interestRate/12.0 )  );  ...  }  } |

Mặc dù không tính toán trước bảng, bạn có thể tính ttoán trước phần biểu thức phức tạp bên ngoài vòng lặp và sử dụng chúng trong vong lặp, nó sẽ trở nên như thế này:

|  |
| --- |
| **Java Example of Precomputing the Second Complex Computation**  double ComputePayments(  int months,  double interestRate  ) {  long loanAmount;  double divisor = ( 1.0 - Math.pow( 1.0+(interestRate/12.0). - months ) ) /( interestRate/12.0 );  for ( long loanAmount = MIN\_LOAN\_AMOUNT; loanAmount <= MAX\_LOAN\_AMOUNT;  loanAmount++ ) {  payment = loanAmount / divisor;  ...  }  } |

Điều này cũng tương tự như các ví dụ trước về việc đặt một mảng tham khảo và con trỏ ở bên ngoài vong lặp. Kết quả đối với Java trong trường hợp này có thể so sánh với trường hợp tính toán trươc bảng theo cách tối ưu đầu tiên:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** | **Performance Ratio** |
| --- | --- | --- | --- | --- |
| Java | 7.43 | 0.24 | 97% | 30:1 |
| Python | 5.00 | 1.69 | 66% | 3:1 |

Python đã được cải thiện ỏ đây, nhưng không trong cách tối ưu đầu tiên. Nhiều lúc khi một cách tối ưu không đưa ra kết quả mong muốn, một cách tối ưu tương tự sẽ cho ra kết quả

Tối ưu chương trình bằng tính toán trước có vài dạng:

* Tính toán trước kết quả trước khi chương trình thi hành, và rói buộc chúng vào hằng số được gán khi bên dịch
* Tính toán trước kết quả trước khi chương trình thi hành, và mã hóa cứng chúng vào các biến , sử dụng khi đang chạy
* Tính toán trước kết quả trước khi chương trình thi hành, đặt vào trong file được thi hành khi đang chạy
* Tính kết quả một lần, lúc chương trình bắt đầu, và tham khảo chúng mọi lúc cần
* Tính toán nhiều nhất có thể trước khi vong lặp bắt đầu, tối thiểu các coong việc cần làm trong vòng lặp
* Tính toán kết quả vào lần đầu tiên cần, lưu trữ chúng và ta có thể gọi chúng khi cần

**Loại bỏ biểu thức con**

Nếu bạn có biểu thức tính toán lại nhiều lần, gán kết quả cho biến và sử dụng biến hơn là tính lại biểu thức. Ví dụ hàm tính cho vay có một biểu thức con phổ biến mà ta có thể loại bỏ:

|  |
| --- |
| **Java Example of a Common Subexpression**  payment = loanAmount / (  ( 1.0 -- Math.pow( 1.0 + ( interestRate / 12.0 ), -months ) ) /  ( interestRate / 12.0 )  ); |

Trong ví dụ này, ban có thể gán interestRate/12.0 bằng một biến mà sau này tham chiếu hai lần chứ không phải tính biểu thức hai lần. Nếu đặt tên bến tốt, cách này tối ưu khả năng đọc và khả năng thi hành cùng lúc. Dưới đây là đoạn mã:

|  |
| --- |
| **Java Example of Eliminating a Common Subexpression**  monthlyInterest = interestRate / 12.0;  payment = loanAmount / (  ( 1.0 -- Math.pow( 1.0 + monthlyInterest, -months ) ) /  monthlyInterest  ); |

Tiết kiệm trong trường hợp này chưa có hiệu quả cao lắm:

| **Language** | **Straight Time** | **Code-Tuned Time** | **Time Savings** |
| --- | --- | --- | --- |
| Java | 2.94 | 2.83 | 4% |
| Python | 3.91 | 3.94 | -1% |

Có vẻ như xuất hiên chương trình Math.pow() là tốn kém và làm lu mờ các khoản tiết kiệm từ loại bỏ biểu thức con. Hoặc biểu thức con đã bị loại bỏ bởi trình biên dịch. Nếu biểu thức con là một phần chi phí lớn của biểu thức, hoặc trình dịch tối ư chưa tôt, cách tối ưu này có thể tác động nhiều hơn.

1. **Hiệu chỉnhchương trình con**

Một trong những công cụ manh mẽ nhất trong hiệu chỉnh code là cách phân tích chương trình con . Một chương trình con nhỏ và tốt sẽ tieets kiệm bộ nhơ vì chúng làm việc ở nhiều nơi. Chúng làm chương trình dễ tối ưu hóa vì bạn có thể cấu trúc lại mã trong một chương trình con và do đó phát triển tất cả các chương trình gọi nó. Chương trình con nhỏ là dễ dàng sửa lại trong một ngôn ngữ cấp thấp. Chương trihf con dài, phức tạp khó có thể tự hiểu,và với ngôn ngữ thấp như hợp ngữ, là không thể

**Viết lại chương trình con inline**

Trong những buổi đầu lập trình máy tính, vài máy đặt lệnh cấm gọi chương trình con. Một cuộc gọi đến chương trình con làm hệ điều hành phải hóan đổi các chương trình, hoán đổi thư mục các chương trình con, hoán đổi các chương trình con riêng biệt, thưc hiên chương trình, hoán đổi chương trình con và lợi gọi vào vị trí cũ. Tấy cả các hoán đổi này đều tốn tài nguyên và làm chương trình chậm

Các máy tính hiện đại có cách làm giảm chi phsi gọi chương trình con, đây là kết qảu việc đặt chương trình con bằng inline:

| **Language** | **Routine Time** | **Inline-Code Time** | **Time Savings** |
| --- | --- | --- | --- |
| C++ | 0.471 | 0.431 | 8% |
| Java | 13.1 | 14.4 | -10% |

Vài trường hợp, bạn cần tiết kiệm vài nano giây bằng cách đưa trực tiếp chương trình con vào chương trình bằng các sử dugj hàm inline trong C++. Nếu bạn làm việc trên ngôn ngữ không có inline, nhưng có tiền xử lí macro, bạn có thể sử dụng macro để đưa mã vào, chuyển ra vào khi cần thiết. Nhưng với các máy tính hiện đại hầu như không cấm gọi các chương trình con. Như bạn thấy, bạn có thể làm giả hiệu suất bằng các sử dụng inline để tối ưu hóa.

1. **Chuyển sang hợp ngữ**

Kiến thức thông thường không nên để lại những thứ không được nhắc đến, là lơi khuyên khi thi hành gặp trở ngai, bạn nên viết lại code theo hợp ngữ. Viết lại mã trong hợp ngữ đồng thời tăng cả tốc độ và kích thước mã. Dưới đây là cách tiếp cận điển hình đẻ tối ưu hóa bằng hợp ngữ:

* Viết toàn bộ chương trình bằng ngôn ngữ bậc cao
* Kiểm thử toàn bộ ứng dụng đảm bảo luôn chạy đúng
* Nếu cần tăng hiệu suất sau đó, kiểm tra thông tin ứng dụng để xác định những điêm nóng. Thương thì 5% chương trình thường tốn 50% thời gian chay, bạn có thể xác định những mảnh nhỏ của chương trình là điểm nóng
* Viết lại vài mảnh nhỏ thành hợp ngữ

Cho dù bạn đi theo con đường này hay không dựa vào bạn có thoải mái với hợp ngx hay không, hợp ngữ phù hợp với vấn đề như thế nào và độ tuyệt vọng.

Tôi có sự khẳng định này đầu tiên với chương trình DES ở vài chương trước. Tôi cố gắng tối ưu theo mọi cách từng nghe nhưng chươn trnhf vẫn chạy chậm bằng một nửa so với đích. Viết lại một phần chương trình bằng hợp ngữ là lựa chon duy nhất còn lại. Là người mới học hợp n gữ, tât cả nhưng gì tôi có thể làm là chuyển thẳng từ ngôn ngữ bậc cao sang hợp ngữ. Nhưng tôi đã cải thiện được 50%, dù ở mức thô sơ

Giả sư bạn có đoạn mã chuyển dãy nhị phân thành phần tử trong ASCII, đây là đoạn mã delphi:

|  |
| --- |
| **Delphi Example of Code That's Better Suited to Assembler**  procedure HexExpand(  var source: ByteArray;  var target: WordArray;  byteCount: word  );  var  index: integer;  lowerByte: byte;  upperByte: byte;  targetIndex: integer;  begin  targetIndex := 1;  for index := 1 to byteCount do begin  target[ targetIndex ] := ( (source[ index ] and $F0) shr 4 ) + $41;  target[ targetIndex+1 ] := (source[ index ] and $0f) + $41;  targetIndex := targetIndex + 2;  end;  end; |

Mặc dù khó thấy những điểm béo trong đoạn mã, nó chưa rất nhiều thao tác bit, vốn không phải điểm mạnh của Delphi. Thao tác bit là sở trường của hợp ngữ, vì vậy dù đây là chương trình tốt, thì vẫn có thể viết lại mã. Dưới đây là mã hợp ngữ:

|  |
| --- |
| **Example of a Routine Recoded in Assembler**  procedure HexExpand(  var source;  var target;  byteCount : Integer  );  label  EXPAND;  asm  MOV ECX,byteCount // load number of bytes to expand  MOV ESI,source // source offset  MOV EDI,target // target offset  XOR EAX,EAX // zero out array offset  EXPAND:  MOV EBX,EAX // array offset  MOV DL,[ESI+EBX] // get source byte  MOV DH,DL // copy source byte  AND DH,$F // get msbs  ADD DH,$41 // add 65 to make upper case  SHR DL,4 // move lsbs into position  AND DL,$F // get lsbs  ADD DL,$41 // add 65 to make upper case  SHL BX,1 // double offset for target array offset  MOV [EDI+EBX],DX // put target word  INC EAX // increment array offset  LOOP EXPAND // repeat until finished  end; |

Viết lại mã bằng hợp ngữ đen lại lợi, tiết kiệm 41% thời gian, Đó là kết quả để suy ra nhận định: doạn mã với ngôn ngữ thích hợp hơn với thao tác bít- ví dụ C++- sẽ đạt được ít hơn. Dưới đây là kết quả:

| **Language** | **High-Level Time** | **Assembler Time** | **Time Savings** |
| --- | --- | --- | --- |
| C++ | 4.25 | 3.02 | 29% |
| Delphi | 5.18 | 3.04 | 41% |

Chương trình con hợp ngữ cho thấy, viết lại mã bằng hợp ngữ không tạo ra chương trình con lớn, ngớ ngẩn. Các chương trình này là thường ít, giống như cái vừa rồi. Đôi khi mã hợp ngữ ngỏ gọn như mã ngôn gnwx bậc cao của nó.

Môt chiến lược tương đối dễ dàng và hiệu quả cho việc viết lại mã bằng hợp ngữ, là bắt đầu với trình biên dịch mà tạo ra mã hợp ngẽ như một phần phụ của trình biên dich.

Xuất ra mã hợp ngữ cho chương trình con, và lưu trính trong một file mã nguồn riêng. Sử dụng trình biên dich hợp ngữ làm cơ sở, tự tai tối ưu hóa mã, kiểm tra tính dúng đắn và cải tiến ở mỗi bước. ột số trình biên dịch đưa ra câu lệnh cấp cao như câu lênh trong hợp ngữ, nếu có, bạn có thể giữ chúng trong mã hợp ngữ theo tài liệu hướng dẫn.

**Checklist: Code-Tuning Techniques**

Cải thiện tốc độ và kích cỡ:

* Chú thích bảng tìm kiếm cho vấn đề logic phức tạp
* Vòng lặp Jam
* Sử dụng số nguyên thay cho biến dấu phẩy đông
* Khởi chạy dữ liệu tại thời điểm biên dịch
* Sử dụng hằng số đúng kiểu
* Tính toán trước kết ủa
* Loại bỏ các biểu thức con phổ biến
* Dịch các chương trình con quan trọng ra hợp ngữ

Chỉ tăng tốc độ:

* Ngừng kiểm thử nếu biết câu trả lời
* Kiểm thử câu lệnh case và chuỗi if-then-else bằng tần số
* Đối chiếu sự thực hiện của vài cấu trúc logic giống nhau
* Sử dụng đánh giá lười
* Không đổi vòng lặp mà chứa kiểm thử if
* Mở vòng lặp
* Tối thiểu công viecj trong vòng lặp
* Sử dụng lính canh trong tìm kiếm vòng lặp
* Đổi mảng nhiều chiều thành một chiều
* Tối thiểu tham chiếu mảng
* Bổ sung chỉ số cho kiểu dữ liệu
* Cache thường xuyên sử dụng giá trị
* Sử dụng đồng nhất thức đại số
* Giảm khối lươgj biểu thức logic và toán học
* Cẩn thận với chương trình con hệ thống
* Viết lại chương trình con bằng inline

Từ khóa:

* Kết quả tối ưu hóa là khác nhau với các ngôn ngữ, môi trường, trình biên dich khác nhau. Nếu không có sự đo lường mỗi cách tối ưu hóa, bạn sẽ không có nhận định cho dù nó giúp hoặc làm hỏng chương trình bạn
* Cách tối ưu hóa đầu tiên chưa phải tốt nhất, dù tìm ra một cách, hay tìm một cách tốt hơn
* Hiệu chỉnh code giống như năng lượng hạt nhân, nó giống như một chủ đề gây tranh cãi và cảm tính. Vài người nghĩ nó có hại cho việc đọc và khả năng duy trì mà họ sẽ không làm tất cả . Những người khác nghĩ với biện phái bảo vệ thích hợp, đó là có lợi. Nếu bạn qyết định áp dụng các ki thuật trong chương này, áp dụng chúng cẩn thận