C code Optimization

Sự đúng đắn và tính dễ hiểu của code

Phần cứng đã đạt đến giới hạn của nó, code size quá lớn hoặc code ko đủ nhanh

Làm sao cho code chạy nhanh hơn, kích thước code nhỏ hơn và đỡ tốn điện hơn, thường thì ko thể nào đạt được tất cả các mục tiêu cùng lúc, sẽ phải hy sinh 1 trong các yếu tố đó. Có 2 chỗ mà quá trình optimization xảy ra chính là lập trình viết code và thứ 2 là quá trình optimization của complier

Các pp optimization /*….*/

Những biến được đặt trong thanh ghi thì sẽ truy xuất nhanh hơn các biến đặt trong RAM

Những hàm có chữ inline đằng trước, code của hàm inline sẽ được copy vào nơi gọi cái hàm đó ra. Việc này sẽ được thực hiện bởi trình biên dịch và như vậy sẽ ko có overhead như là khi gọi hàm các hàm bình thường, tuy nhiên code size sẽ lại tăng lên

Instruction scheduling

Cần phải tận dụng được các lợi thế của từng loại processor khác nhau

Lifetime analysis

1 thanh ghi có thể được sử dụng lại cho nhiều biến, miễn là những biến đó ko nằm trong cùng 1 souce code. Khi mà chuyển source code, giá trị của thanh ghi sẽ được push vào stack

Đó là optimize bằng tay

Đây là optimize bằng tool

Bật option về optimize trong trình biên dịch

Dùng tool profiling để biết hàm nào chạy mất bao nhiêu thời gian, hàm nào được gọi nhiều lần.

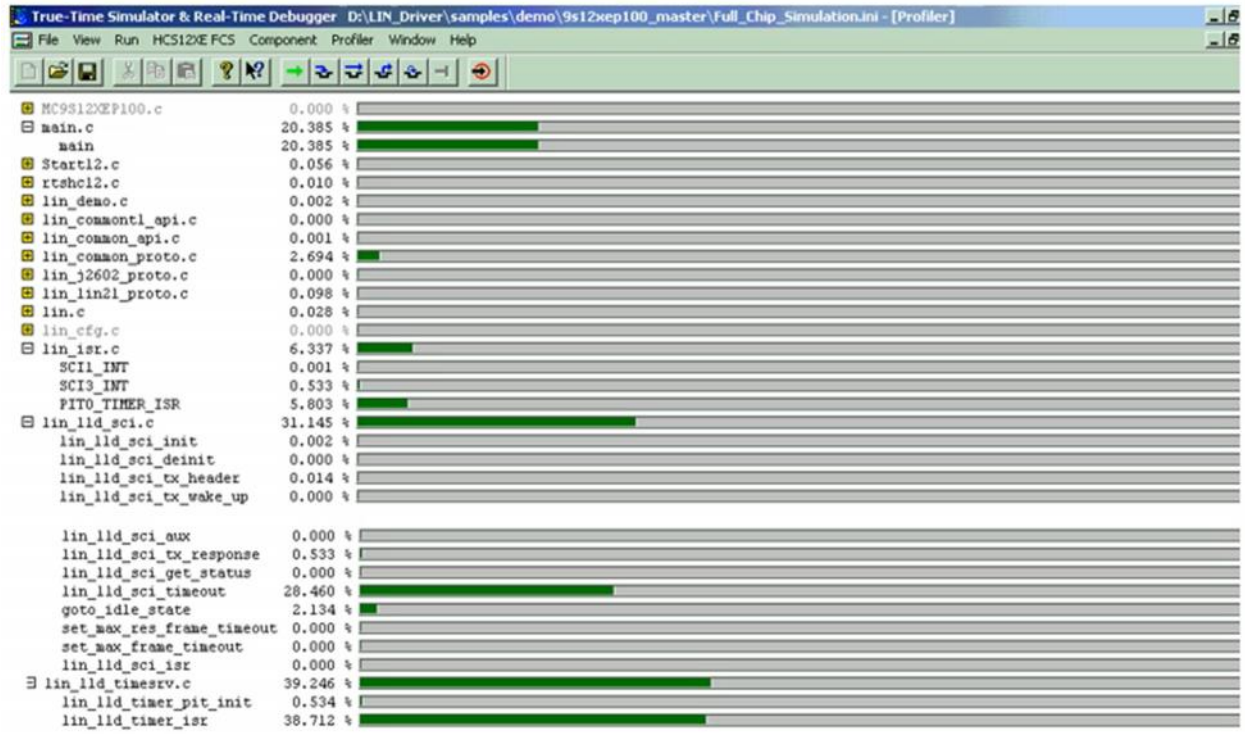
Có 2 loại tool profiling là dựa vào xem code của nó có chèn vào code của chúng ta hay không

Ví dụ intrusive là Gprof, tool này sẽ chèn thêm code ở đầu và cuối hàm để ghi thời gian thực hiện hàm và có thể đếm số lần thực hiện hàm đó

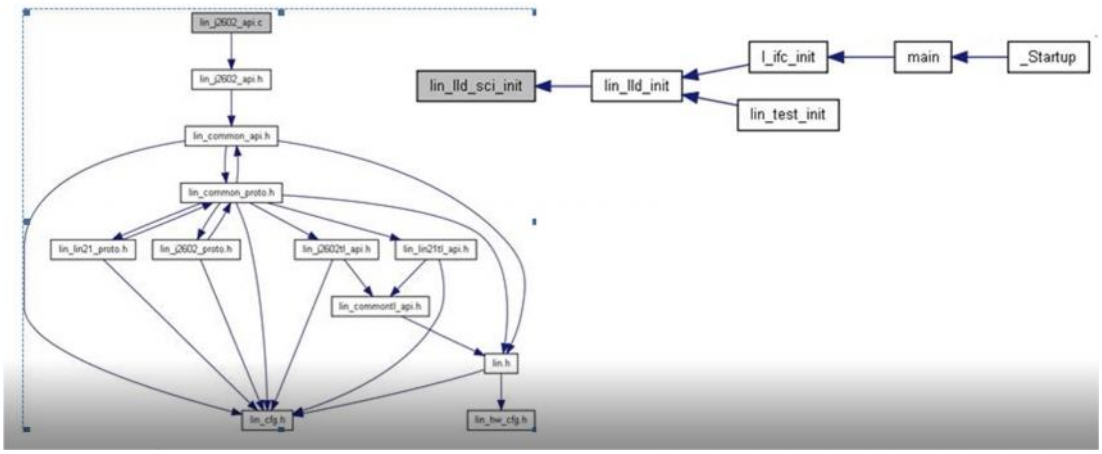
Việc **chèn thêm code sẽ sinh ra overhead** và overhead này có thể khá lớn tuỳ vào chương trình gốc và tăng thời gian chạy của code gốc lên vài lần nên là profiling cũng ko có ý nghĩa mấy

Đối với non-intrusive profiling thì cần có phần cứng riêng biệt

Ví dụ chip của ARM thì có thể tìm hiểu về etm trace module, vì nó phụ thuộc và hardware nên là platform nào cũng oke được



Đối với hệ thống tất cả các code mà ko phải code nào cũng do chúng ta viết ra thì chúng ta có thể sử dụng các tool như phân tích code tĩnh để biết cấu trúc code như thế nào



Rồi phần nào để code nhiều hàm nhất thì như vậy có thể giúp cho quá trình optimization dễ dàng hơn

Cuối cùng là lựa chọn thuật toán sao cho phù hợp

C common defects: các lỗi thường gặp trong C

Allignment là gì, packing là gì, mục đích của allignment là gì

Tiếp theo là về macro, cần tìm hiểu về khái niệm size effect

Phần kiểu nữa.

C code Optimization

Hầu hết con chip mình chọn có kích thước bộ nhớ flash và Ram hoặc vừa đủ hoặc lớn hơn so với kích thước chương trình của mình, yêu cầu đáp ứng của mình, ko chặt chẽ về thời gian đáp ứng. như vậy nên mình cũng chưa optimize code của mình

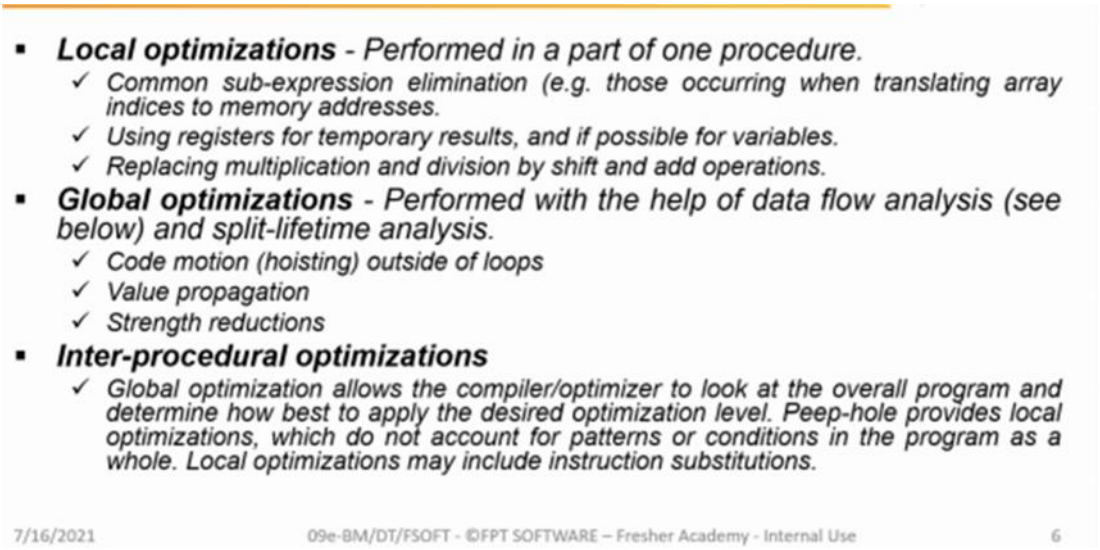
Khi vào dự án, khi kích thước bộ nhớ flash, ram nhỏ ko đáp ứng được thì phải optimization hoặc là có các yêu cầu đặt ra của khách hàng là đáp ứng được function này là bao nhiêu tick, s thì khi đó mình lại ingroup về tốc độ, về speed.

Khách hàng sẽ đưa ra yêu cầu về hàm ngắt để tránh trường hợp là mình bị missing ngắt ví dụ mình đang chạy ngắt này thì ngắt khác đến thì ko đáp ứng kịp. nếu thời gian chạy ngắt quá lâu thì sẽ bị bỏ qua 1 vài ngắt thì khi đó chương trình của mình sẽ bị sai, khi đó khách hàng đưa ra là hàm handle ngắt phải chạy trong bao nhiêu ns, ms, bao nhiêu tick, nếu mình vượt quá thời gian đó thì phải ingroup về speed sao cho cái handle ngắt đó phải nhanh lên, nó ko chạy chậm nữa

Áp dụng các quy tắc, các thuật toán hoặc là áp dụng, phân tích các cái design khác để cho làm mã có thể nhanh hơn, nhỏ hơn và hiệu quả hơn nhưng mà thường thì khi optimization thì các loại optimization sẽ xung đột với nhau ví dụ là mình có thể làm cho mã có thể làm chương trình chạy nhanh hơn nhưng điều đó lại làm tăng kích thước chương trình lên hoặc là nếu mình làm giảm kích thước đi thì có thể làm chương trình của mình chạy chậm đi.

Tối ưu hoá có thể thực hiện ở 1 vài cấp độ, hoặc là ở trong 1 số function, hoặc có thể optimization ở cả chương trình của mình hoặc là mình có thể optimization ở theo cấp độ code chẳng hạn nhưng theo 1 số cấp độ thì mình có thể optimization ở cả trình biên dịch. Complier sẽ optimization.

Ngoài các pp manual thì còn pp dựa vào tool



3 cấp độ chính để optimization

Đầu tiên là local, chỉ cần tối ưu hoá cục bộ, hàm ngắt, hàm handle ko đủ thời gian đáp ứng thì mình sẽ optimization mỗi cái hàm handle thì đây gọi là local optimization

Global optimization là mình phải phân tích, đánh giá chương trình của mình để mình có thể tối ưu hoá cả chương trình của mình, có thể là khi mà giảm kích thước chẳng hạn thì muốn giảm kích thước của cả chương trình đó

Inter-procedural optimization thì pp này sẽ kết hợp cả 2 pp trên, có thể là optimization theo cách cục bộ, từng hàm 1 từng hàm một hoặc là đánh giá flow của chương trình, đánh giá các cái structure của chương trình để mình có thể optimization của cả chương trình đó

Các pp optimization có nhiều pp, mỗi pp có 1 ưu điểm và nhược điểm khác nhau, pp đó có thể là tốc độ code nhanh hơn hoặc là giảm cái code size của mình đi hoặc là 1 số phương pháp có thể làm giảm cả 2, 1 số phương pháp thì nó có thể làm giảm tốc độ nhưng mà nó lại làm tăng code size. 1 số pp thì lại làm giảm code size nhưng mà lại giảm thời gian xử lý tăng lên

Đầu tiên là loại bỏ biểu thức con chung

- ***If the value resulting from the calculation of a sub-expression is used multiple times, perform the calculation once and substitute the result for each individual calculation***

Normal	Optimization

float x = a*min/max + sx;
 	
float y = a*min/max + sy;
	float temp = a*min/max;
	
	float x = temp + sx;
	
	float y = temp + sy;

Thay vì để code tính lại nhiều lần a*min/max thì mình tính ra 1 thằng temp để nó bằng biểu thức chung đó, sau khi tính x y thì lấy thằng temp đó để cho nó với giá trị delta, thực hiện tiếp phần khác nhau. Làm vậy thì chương trình chỉ tính 1 lần biểu thức chung và cái chương trình của mình nhanh lên nhưng mà kích thước của mình cũng tăng lên nhưng ở đây kích thước của mình nó tăng lên ko đáng kể.

Constant propagation

- ***Replace variables that rely on an unchanging value with the value itself***

Normal	Optimization

x2 = 5;
 	
x3 = x1 + x2;
	x3 = x1 + 5;

Pp này làm tăng tốc độ hay giảm kích thước, làm giảm kích thước.

Khi mà code thường là sử dụng thằngg optimize luôn nhưng mà trong 1 chương trình lớn thì giả sử đây là 1 hàm ngắt, đoạn này ko thực hiện cộng nữa mà thực hiện 1 process ngắt thì hàm ngắt đấy giả sử của stm module channel 5 chẳng hạn thì thay vì mình gọi đến hàm process ngắt thì mình truyền vào 2 tham số 0 và 5 thì trong chương trình của mình để dễ hiểu hơn thì mình có thể define là modul = 0 và channel là 5 để mình truyền vào function đó thì khi mà viết như vậy thì code sẽ clear hơn và nó sẽ dễ hiểu hơn. Khi mà 1 người vào ví dụ như là bạn để lại cho người khác, nơi phần code đó mà người khác đọc

code đã hiểu luôn là họ đã tạo ra 1 biến là channel và module = 0 và channel = 5, truyền vào hàm process ngắt thì như vậy đọc đến đấy là người ta đã hiểu luôn chứ ngta ko phải tìm lại là process ngắt có parram ntn, param 1 là gì, param 2 là gì, tại sao là 0 và tại sao là 5. Trong các chương trình lớn người ta code kiểu normal rất nhiều, nên khi optimize thì ko define cái x2 nữa mà sử dụng luôn 5 hay ko define 1 biến là channel = 5 và module = 0 mà mình truyền trực tiếp 0 và 5 vào hàm

(truyền số vào hàm) kích thước của code sẽ giảm đi.

Nhiều trường hợp cũng dùng define 1 mac, nhưng nếu define mà ko sử dụng thì có 1 số lỗi complier warning. Khi nào ko sử dụng thì rào nó lại để cho nó ko bị lỗi miss rar hoặc complier warning

Copy propagation

Normal	Optimization

x2 = x1;
 	
x3 = x1 + x2;
 	
x2 = 3;
	x3 = x1 + x1;
	
	x2 = 3;

Thay vì sử dụng truy cập đến 2 biến thì sử dụng 2 thằng x1 mà sẽ ko dùng 2 biến để truy cập cùng 1 biểu thức nữa, như vậy mình chỉ cần truy cập đến 1 biên thôi và tính biểu thức này dựa trên những biến đó, line code cũng giảm đí, kích thước chương trình giảm đi

Pp này sẽ làm giảm tốc độ xử lý của mình

- **Dead code elimination**
  - ✓ *Code that never gets executed can be removed from the object file to reduce stored size and runtime footprint*
- **Global register allocation**
  - ✓ *Variables that do not overlap in scope may be placed in registers, rather than remaining in RAM. Accessing values stored in registers is faster than accessing values in RAM*
- **Inline calls**
  - ✓ *A function that is fairly small can have its machine instructions substituted at the point of each call to the function, instead of generating an actual call. This trades space (the size of the function code) for speed (no function call overhead).*

Dead code: xoá bỏ các code ko bao giờ thực thi, vòng for, while ko bao giở nhảy vào được thì xoá đi

1 function ko bao giờ được gọi thì xoá đi

Pp thứ 2, trong cái cấu trúc MCU, các giá trị nằm trong thanh ghi (CPU) bao giờ cũng truy cập nhanh hơn các giá trị lưu trữ trong RAM, sử dụng trực tiếp thanh ghi để lưu trực tiếp giá trị và sử dụng các giá trị đó. 1 biến mà call đi nhiều lần thì thay vì define biến đó trên RAM thì sử dụng trực tiếp thanh ghi thì khi đó tốc độ truy cập của mình sẽ nhanh hơn do việc truy cập vào trong thanh ghi sẽ nhanh hơn là RAM

Pp sử dụng các hàm inline thì khi mà mình gọi hàm khác thì chương trình thực hiện các process ntn, nó phải thế nào để cho nó mới nhảy đến cái function call?

Khi mình gọi hàm thì chương trình sẽ lưu lại các loại địa chỉ gồm các biến mà ta thực hiện lên vùng nhớ tạm và sau đấy chuyển sang hàm mà mình gọi để nó thực thi. Thực thi xong thì quay về địa chỉ đó để nó làm tiếp.

Khi mình gọi hàm thì chương trình sẽ lưu lại các thông tin, các biến vào trong stack và 1 cái nữa cũng đưa trong stack là các return pointer là cái địa chỉ mà chương trình nó đang đứng, đang chạy đến, khi mà gọi hàm đó thì nó sẽ nhảy đến địa chỉ của hàm sau đó thực hiện hàm xong nó quay lại, lấy lại return pointer là nó biết là sau khi hàm này gọi xong thì nó quay lại cái vị trí nào của chương trình mà nó phải nhảy đến hàm đó. (đó là return pointer).

Return pointer là cái mà đã nói đến trong overflow, các hacker thường lợi dụng việc overflow để return, replace lại cái, thay đổi cái giá trị của return pointer đi thì khi đó thay vì chương trình quay lại vị trí lúc đầu thì nó nhảy tới chương trình của hacker muốn nó nhảy đến. đó là các thông tin mà chương trình của mình cần lưu lại trong stack. Lưu xong thì chương trình của mình jump đến, nhảy đến địa chỉ của hàm. Sau khi nhảy đến địa chỉ của hàm xong thì nó thực hiện chương trình của hàm, nó phải quay lại, lấy lại thông tin trong stack ra, nó biết quay lại cái return pointer của nó là gì và quay lại cái vị trí đó và nó dựa vào thông tin trước đó nó chạy tiếp chương trình, khi mà làm như vậy thì chương trình của mình mất 1 vài process, 1 vài clock để nó có thể lưu lại chương trình, jump trở lại, nó lấy lại thông tin đã chạy trước đó thì thay vì vậy mình không sử dụng call function nữa mà mình sử dụng lại inline call.

Hàm inline gần giống như define macro, nó sẽ sử dụng thay thế luôn cái đoạn call đó bằng đoạn code của hàm inline call thì khi đó chương trình chỉ cần chạy 1 mạch từ đầu đến cuối chứ ko cần jump đến các function define riêng nữa.

Nếu 1 function được complier ra thì vùng code của function đó được đặt riêng ở 1 chỗ khác biệt thì vùng code của inline call nó được đặt luôn ở cái đoạn nó gọi function nên là chương trình của mình ko cần mất thời gian cất thông tin vào stack nữa hoặc là lấy thông tin từ stack nữa, cũng ko cần mất thời gian để nó jump từ đoạn chương trình đang chạy jump đến đoạn code của hàm nó gọi nữa, nó chỉ cần chạy 1 mạch thôi

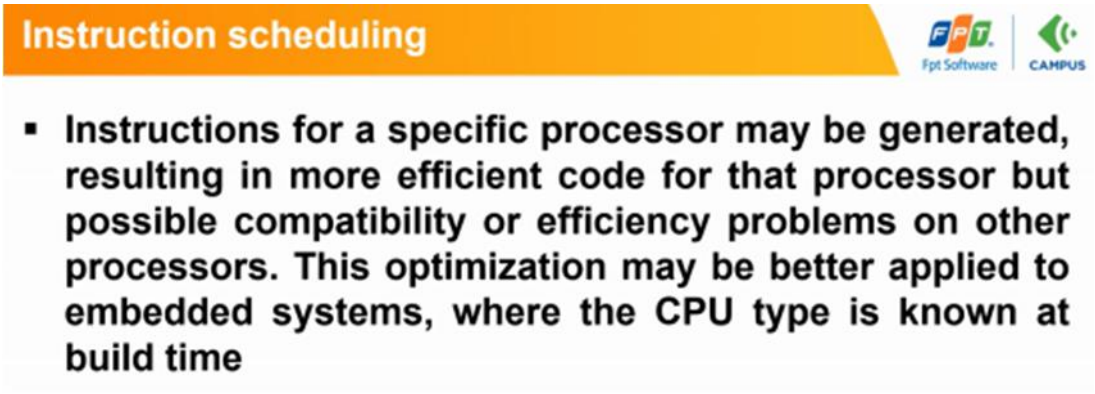
Khi mà làm như vậy thì tốc độ xử lý của mình nó nhanh lên, làm tăng tốc độ xử lý, giảm thời gian chạy nhưng mà nếu hàm inline call mình call 5 lần 10 lần 100 lần chẳng hạn thì tương ứng đoạn code đó được replace lại vào cái.., replace 100 lần vào cái chỗ mà nó gọi hàm inline thì khi mà làm như vậy thì kích thước của code sẽ tăng lên.

Variable that do no overlap

Tức là 1 function có 5 biến, 1 biến được sử dụng ở đoạn đầu function biến thứ 2 ở giữa function, lúc đó sử dụng 2 biến đó cùng 1 register tại vì 2 biến đó ko bị overlap ở cái đoạn nào. Nó không sử dụng chung ở đoạn nào.

Ví dụ đoạn trên sử dụng x để tính 1 cái gì đó, ở đoạn dưới này lại tạo ra 1 biến y để tính cái gì đó thì thay vì sử dụng 2 biến x và biến y nó ko bị chung (ko bị overlap) thì mình sử dụng luôn 1 register cho cả 2 biến.

Nhưng nếu xen kẽ nhau thì dùng cả 2 register



Phần này sẽ dựa vào 1 bộ vi xử lý cụ thể để đưa ra 1 cách include hiệu quả hơn, tăng tốc độ xử lý chương trình dựa trên các cái ví dụ như 1 CPU đó có nhiều core thì mình có thể xử lý từng đoạn code một, lập lịch cho từng đoạn code một, chạy trên từng core khác nhau, multi thread hoặc là multi proccess. Thì pp tối ưu hoá này sẽ làm tăng tốc độ xử lý chương trình dựa trên kiến trúc của phần cứng nên nó chỉ tối ưu hoá cho hệ thống nhúng cho nó thôi

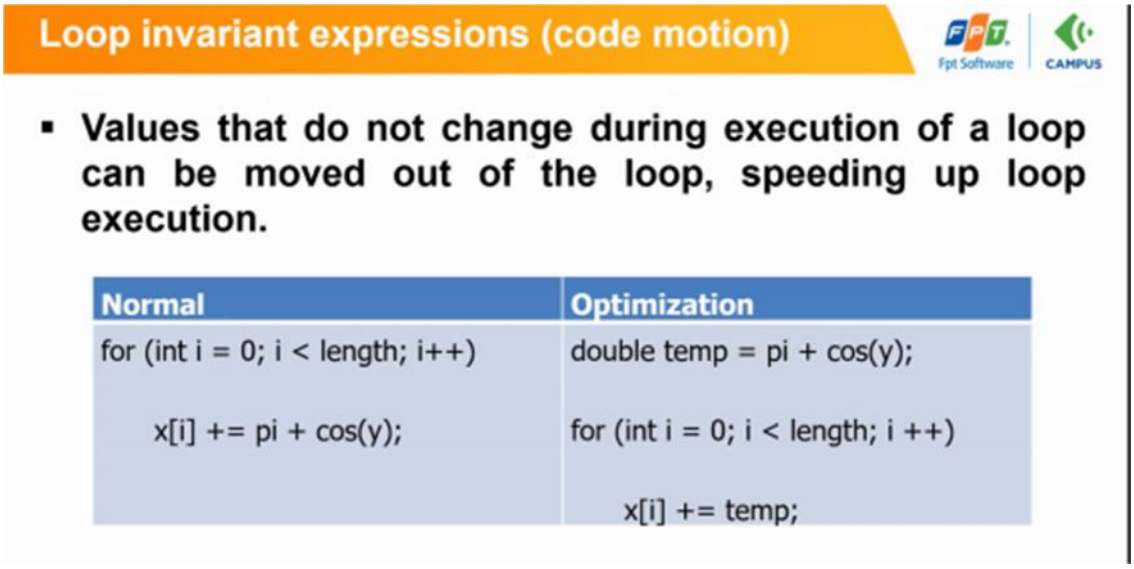
Nếu mà áp dụng vào hệ thống nhúng khác thì mình lại phải đưa ra 1 cái, phân tích, làm lại cái pp này để tối ưu hoá. Ví dụ mình tối ưu hoá dựa vào core thì dựa vào cái CPU A

có 3 core thì mình tối ưu hoá dựa vào 3 core đó, có thể ingroup, có thể lập lịch để cho nó chạy trên 3 core đó.

1 cpu khác chỉ có 2 core hay 1 core chẳng hạn thì có thể chia ra các task hoặc là chia ra proccess, chia ra lập lịch như thế nào như là windows. Còn nếu có 2 core thì chia ra các core như là thằng kia nhưng là với trên 2 core thôi thì pp này sẽ làm tăng tốc độ xử lý của mình lên rất nhanh. Nhưng mà pp này nó lại ko áp dụng được với nhiều nền tảng khác nhau, phân tích lại từ đầu.

Pp này liên quan đến vấn đề tương thích, hiệu quả xử lý của CPU với các bộ vi xử lý khác nhau.

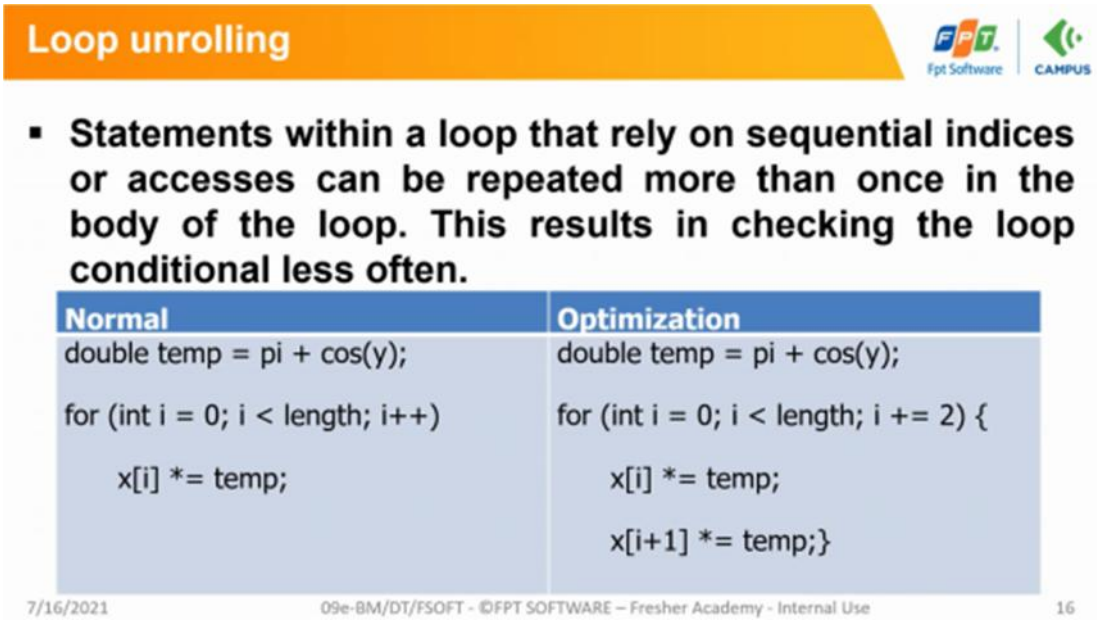
Chính là phần overlap nghĩa là 1 biến mà ko sử dụng, các biến khác nhau thì nó sử dụng ở các vùng khác nhau, nó ko bị overlap vào nhau thì mình có thể sử dụng cùng 1 thanh ghi hoặc là sử dụng cùng 1 biến cho các biến đó thì nó cũng làm giảm kích thước của code đi. Sử dụng thanh ghi thì nó vừa làm giảm kích thước, vừa tăng tốc độ xử lý của chương trình. Cùng 1 thanh ghi có thể sử dụng cho nhiều biến khác nhau. Pp này sẽ phân tích, đưa ra các scope mà biến đó tồn tại và biến khác có thẻ tồn tại trong vùng đó ko? Nếu nó ko bị overlap với nhau thì hoàn toàn có thể sử dụng được cùng 1 thanh ghi cho nhiều biến khác nhau



ở đây, optimize về tốc độ, tại vì ở đây, biểu thức pi + cos(y) nó ko thay đổi ở trong vòng for nên là tính trước và lưu vào trong 1 biến tạm và bên trong chỉ lấy giá trị của 1 biến tạm đó thôi mình sẽ đỡ được tính toán mỗi lần lặp lại thì tính lại x[i] = pi + cos (y)

trong vòng lặp kia mình đã thực hiện quá nhiều biểu thức tính đi tính lại pi + cos(y) cho nên là để tăng tốc độ xử lý của chương trình của mình lên thì mình sẽ tính biểu thức kia ở bên ngoài

cuối cùng để tăng tốc độ xử lý vòng lặp này lên, vòng lặp này chỉ cộng 2 biến vào với nhau. Lúc đó ko phải thực hiện lại việc pi+ cos(y) nữa



Cũng là tăng tốc độ

Ví dụ length là 10 lần thì vòng for 10 lần

Thay vào đó bên kia là lặp 5 lần, tăng tốc độ xử lý. Giảm vòng lặp cũng là cách để tăng tốc độ xử lý.

Strength reduction

- Certain operations and their corresponding machine code instructions require more time to execute than simpler, possibly less efficient counterparts.

Normal	Optimization
x/4	x >> 2
x*2	x << 1

Hầu hết các thanh ghi của mình được tạo ra để thao tác với bit cho nên là các toán tử về bit sẽ được thực hiện nhanh hơn so với các phép chia, phéo cộng, phép nhân.

Các phép chia phép nhân bao giờ cũng chậm hơn phép dịch bit -> tăng tốc độ xử lý để thay thế

Section 3

Code size optimization and *Speed optimization*

CÙNG 1 PP mà đưa vào vùng code này có thể hiệu quả, đưa vào vùng code kia thì không hoặc là cùng 1 vùng code thì pp này hiệu quả, pp kia ko hiệu quả bằng thì mình sẽ cần làm gì và phân tích như thế nào để đưa ra các phương án optimize code cho phù hợp

Code size optimization

- Step 1. Manual optimization
  - ✓ Dead store elimination;
  - ✓ Dead code elimination;
  - ✓ Lifetime analysis: A register can be reused for multiple variables, as long as those variables do not overlap in scope;
  - ✓ Constant propagation: Replace variables that rely on an unchanging value with the value itself;
  - ✓ Copy propagation: Replace multiple variables that use the same calculated value with one variable;
  - ✓ Not use inline calls.

Đây là 1 số pp manual để giảm kích thước của code đi

Ko sử dụng inline call để giảm kích thước bộ nhớ đi

Optimize manual rồi mới để complier optimize

Đôi khi optimize bằng tool cũng có thể chạy sai, ví dụ như sử dụng các biến volatile, các complier sẽ optimize biến đó đi nếu biến đó nó sử dụng volatile.

Volatile

Khi mà chạy 1 đoạn code. 1 chương trình thì chương tình có có 2 phương án, lưu biến đó 1 lần trên RAM sau đó thực hiện luôn các phép toán với biến đó trên RAM, 2 là lưu biến đó có vùng nhớ riêng trên bộ nhớ thì mỗi lần tính toán thì chương trình sẽ note lại 1 lần cái biến đó và thực hiện nó thay vì cứ sử dụng nó trên RAM.

Nếu 1 biến mà mình define bình thường thì complier có thể optimize đi thì nó có thể load lần đầu tiên đi và lần tiếp theo đó sử dụng các biến lưu trên RAM, ví dụ như là khi mà biến x ta define bằng 1 thì lúc chạy chương trình nó load 1 lần vào bộ nhớ và nó load vào RAM là x = 1 và sau đó thực hiện 1 phép cộng nữa là x = 2 thì nó vẫn lưu giá trị x = 2 bằng a, thực hiện phép toán lần thứ 3, nó lại lấy giá trị x = a đó nó tính toán. Thay vì tính toán nó lại load 1 lần trên bộ nhớ thì trình biên dịch nó lại load lại thì khi đó nó đã lưu lại giá trị tính toán lúc cũ rồi, nó sẽ optimize đi, sẽ ko load lại nữa.

Nhưng biến volatile thì lại khác, khi define 1 biến volatile thì đoạn code đó sẽ ko bị optimize đi, nó sẽ ko tính toán trên RAM nữa mà mỗi lần tính toán nó sẽ load lại 1 lần

X ban đầu được define = 1 nó sẽ được load trên RAM, cộng 1 thêm lần nữa, nó sẽ cộng 1 lên, tính lần nữa thì nó lại load lên load từ bộ nhớ load lên Ram cộng lại. thì khi mà làm như vậy thì code của mình code của mình bị optimize phần x đi, thí dụ x + 1, x+3 thì nó thực hiện trên RAM, nhưng nếu nó define 1 biến volatile thì nó ko optimize đoạn code đó đi nữa, bắt buộc là nó phải load lại giá trị được lưu trên bộ nhớ. Có thể là giá trị đấy bị tác động đi. Có thể là 1 thread khác, 1 core khác, 1 chương trình khác nó thay đổi vùng nhớ đó đi.

Nếu mà mình để nó optimize đi nó tính toán trên RAM, tính toán dựa trên giá trị cũ đã lưu lại thì nó sẽ bị sai

Để volatile để biết đoạn code này ko optimize được, mỗi lần truy cập lại nó phải load lại vùng nhớ đó, nó tính toán thì khi đó khi mà thread khác thay đổi giá trị đi mà đã được thay đổi đó nó load lên nó tính toán thì nó sẽ ko bị sai. Hoặc là 1 ví dụ cơ bản nhất là khi mà sử dụng DMA (của micro controller là direct memory access) chẳng hạn

DMA hoạt động bằng cách truy cập trực tiếp vào bộ nhớ ko cần qua bộ đệm, ko phụ thuộc vào CPU, nó hoạt động riêng rẽ, thực hiện copy từ thanh ghi đến bộ nhớ, từ thanh

ghi tới thanh ghi. Thì khi mà copy mình sử dụng DMA để copy 1 giá trị từ thanh ghi đến bộ nhớ chẳng hạn thì bộ nhớ đó là biến của mình

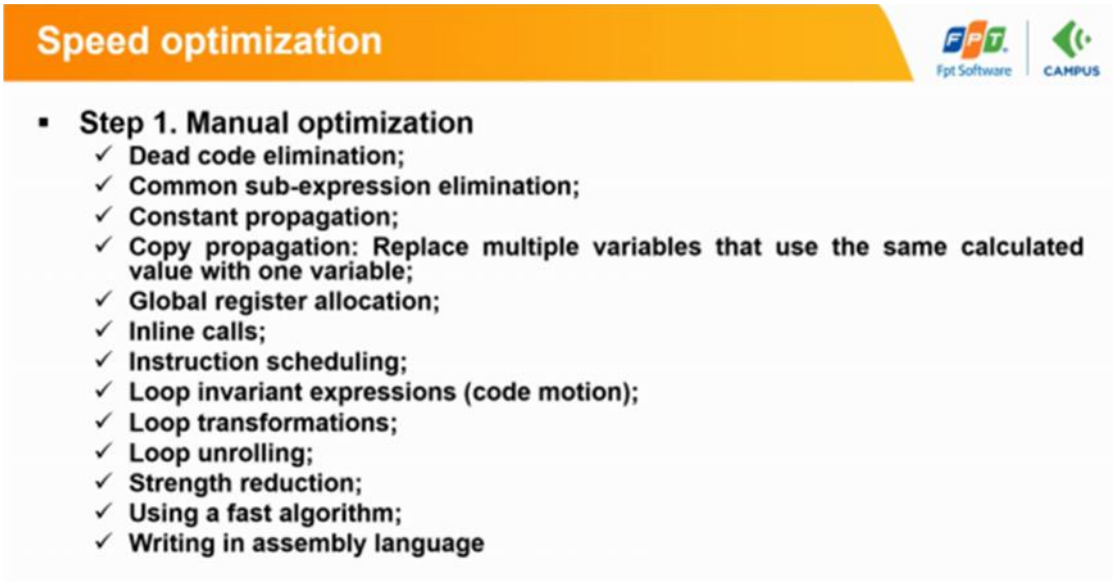
CPU của mình nếu mà complier hoạt động dùng giá trị trên Ram chứ ko dùng giá trị trên bộ nhớ, dùng giá trị biến trên RAM thì khi mà DMA nó tác động và làm thay đổi biến đó đi rồi, lẽ ra mình phải cần phải lấy giá trị mà DMA đã copy cho mình, lúc đấy mình cần lấy giá trị đó mà mình sử dụng nhưng mà mình ko sử dụng thằng đấy mà mình lại sử dụng thằng đã lưu lại trên RAM, đã được tính toán trước trên RAM thì khi mà làm như vậy thì nó sẽ bị sai. Complier optimize đi thì sẽ bị sai

Cho nên là để tránh việc đó xảy ra thì 1 là mình sẽ không sử dụng complier để optimize code và mình đưa về off nhưng ít ai sử dụng như vậy và đôi khi mình sử dụng biến volatile này để biết được là biến volatile này ko optimize đi nữa mà mỗi lần nó tính toán nó sẽ cần truy cập vào bộ nhớ này để nó lấy dữ liệu ra và sau đó nó tính toán dựa trên dữ liệu đó.

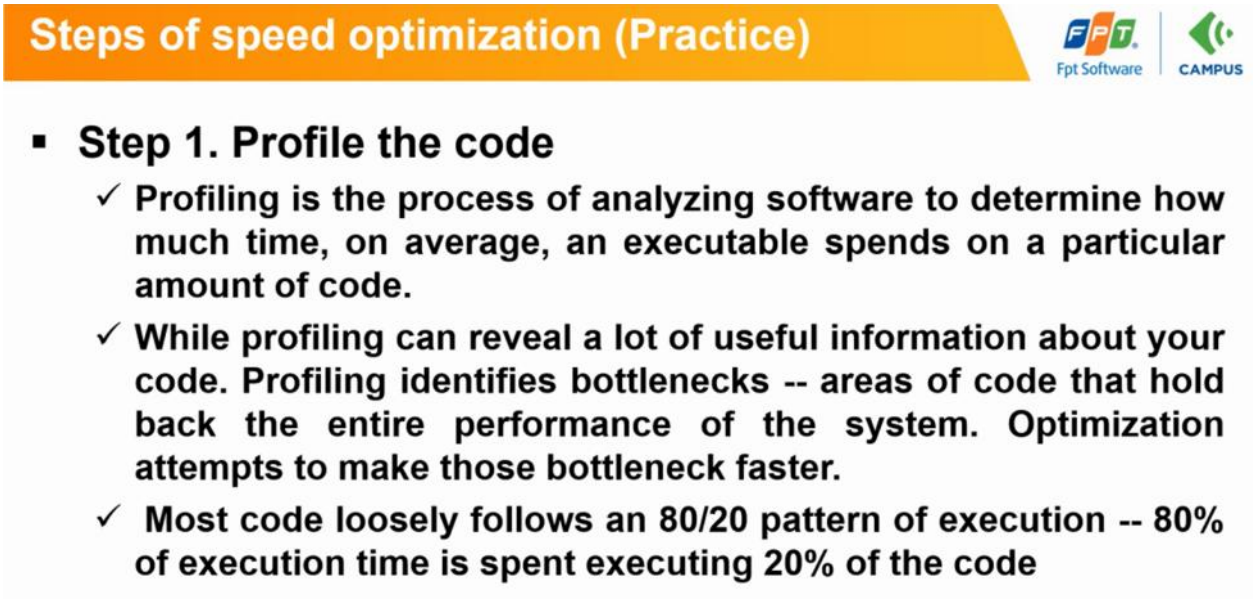
Trong 1 biểu thức sử dụng biến đó thì nó sẽ load lại cái biến ở trong bộ nhớ. Cái thứ 2 là nó ko load lại trong bộ nhớ nữa mà nó sử dụng trực tiếp giá trị mà nó đã lưu trên RAM ví dụ biểu thức trên tính ra x+1 = 2 rồi, x = x+1 thì x = 2; xuống biểu thức tiếp theo sử dụng y = x+3 chẳng hạn thì nó sẽ lấy luôn giá trị mà nó tính toán ở biểu thức trên đã lưu lại đó và nó tính luôn thì như vậy nó sẽ nhanh hơn thay vì truy cập vào bộ nhớ để lấy ra giá trị mà đã tính toán, đã lưu lại ở đó rồi. còn volatile thì chỉ có 1 phương pháp thôi là bắt buộc nó phải sử dụng là load lại trên bộ nhớ, ko được load lại trên RAM.

Mình dùng volatile sẽ làm giảm tốc độ của chương trình đi nhưng mà complier cũng sẽ biết là đoạn đó ko optimize được nó sẽ giữ lại các biến đó mà mỗi lần chạy code thì nó sẽ phải load lại cái vùng nhớ của biến đó tránh yếu tố nhiều yếu tố nó tác động lên cái biến ở 1 thread khác, core khác, DMA cũng tác động vào biến đó được.

Khi mà làm về bộ nhớ hay các chương trình sử dụng DMA thì mình sẽ thấy rất nhiều biến mà mình define là volatile



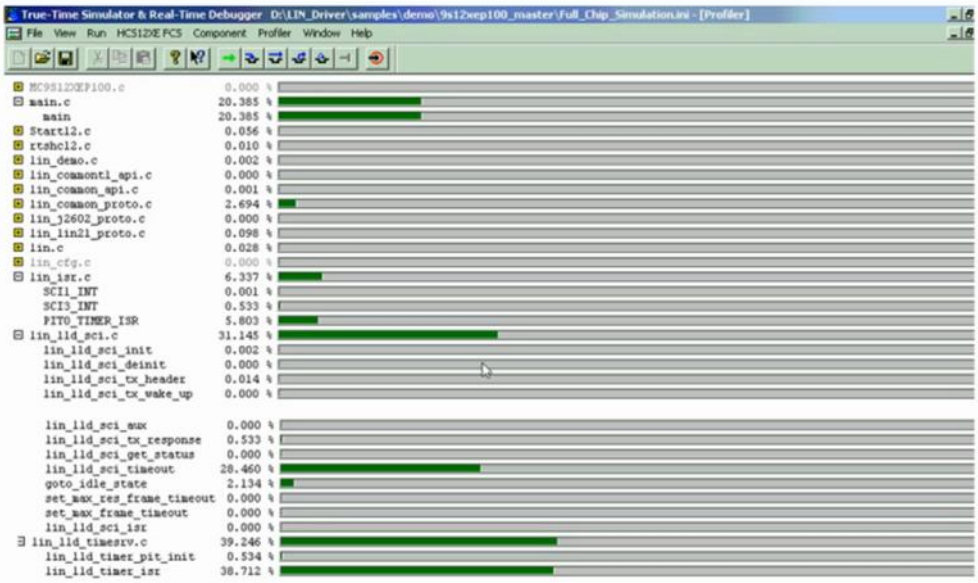
Inline làm tăng tốc độ nhưng cũng làm tăng kích thước. nếu optimize về tốc độ thì sẽ sử dụng inline, còn về kích thước thì hạn chế inline



Profiling là 1 quá trình phân tích phần mềm để phân tích thời gian 1 hàm nó thực thi trong 1 lượng chương trình của mình

Trong profiling có rất nhiều thông tin hữu ích về cái chương trình của mình, ví dụ nó đưa ra được là 1 cái chương trình của mình, đoạn code nào bị thắt lại hoặc là 1 đoạn nào nó bị vòng lặp quá lâu, chạy quá lâu hoặc cái vùng nào làm trật toàn bộ chương trình của mình.

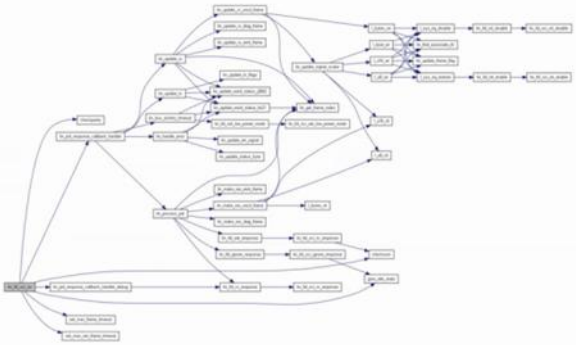
- **Step 1. Profile the code (cont)**



Steps of speed optimization (Practice)

Fpt Software CAMPUS

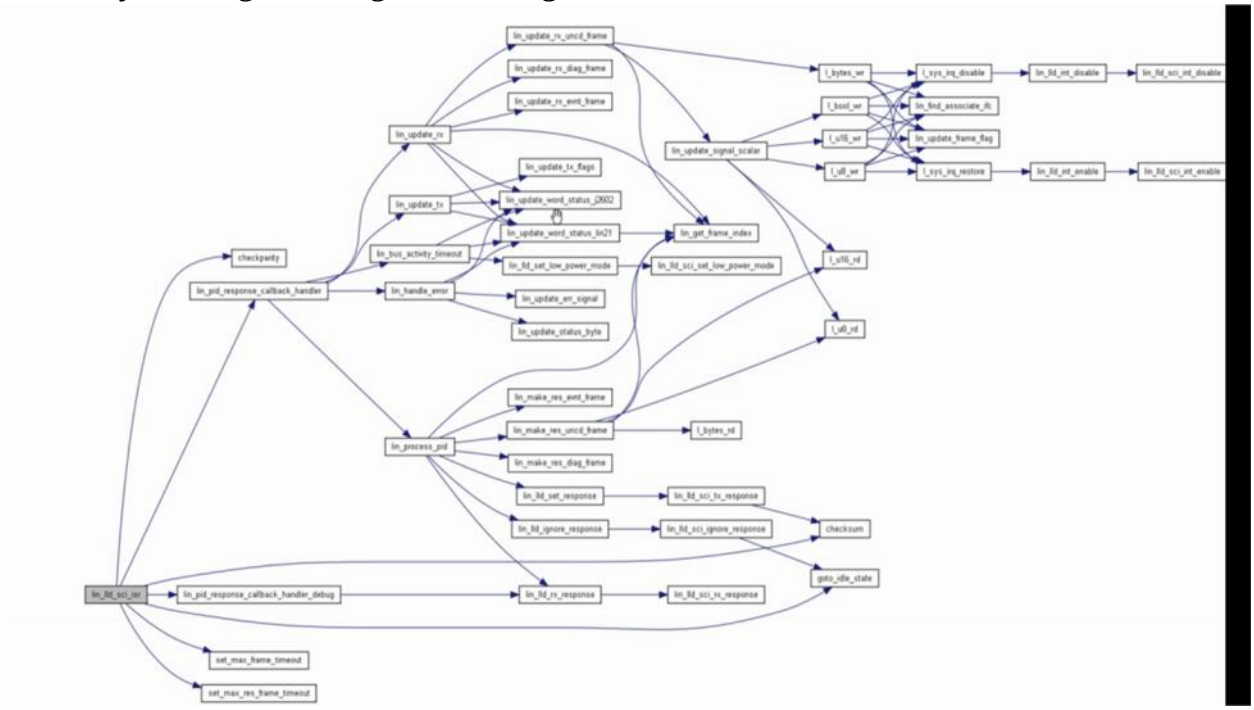
- **Step 2. Optimize for the speed**
  - ✓ **Inline Functions/Loop Unrolling/Strenght reduction**
  - ✓ **http://www.eventhelix.com/realtimemantra/Basics/OptimizingCAndCPPCode.htm**
  - ✓ **Minimize Interrupt Service Routine Overhead by keeping the ISR Simple**



Phân tích, tăng tốc độ xử lý dựa trên các phương pháp đã nói hoặc là sử dụng pulling là kiểm tra đi kiểm tra lại trạng thái của biến thì thay vì sử dụng pp này thì các hệ thống nhúng sử dụng ngắt. thứ nhất là làm giảm cái năng lượng mà nó tiêu hao, tăng tốc độ xử lý lên ví dụ thời gian đó xử lý cái khác ko cần pulling ko cần quay lại kiểm tra lại trạng thái nữa mà trạng thái thay đổi dựa trên ngắt. khi nào trạng thái thay đổi thì ngắt sẽ đếm và notify cho mình. Thì khi đó hạn chế được các vòng lặp, lúc mà check trạng thái đi thì đó cũng là 1 pp để optimize code ngoài ra để optimize code mình có thể sử dụng nhiều

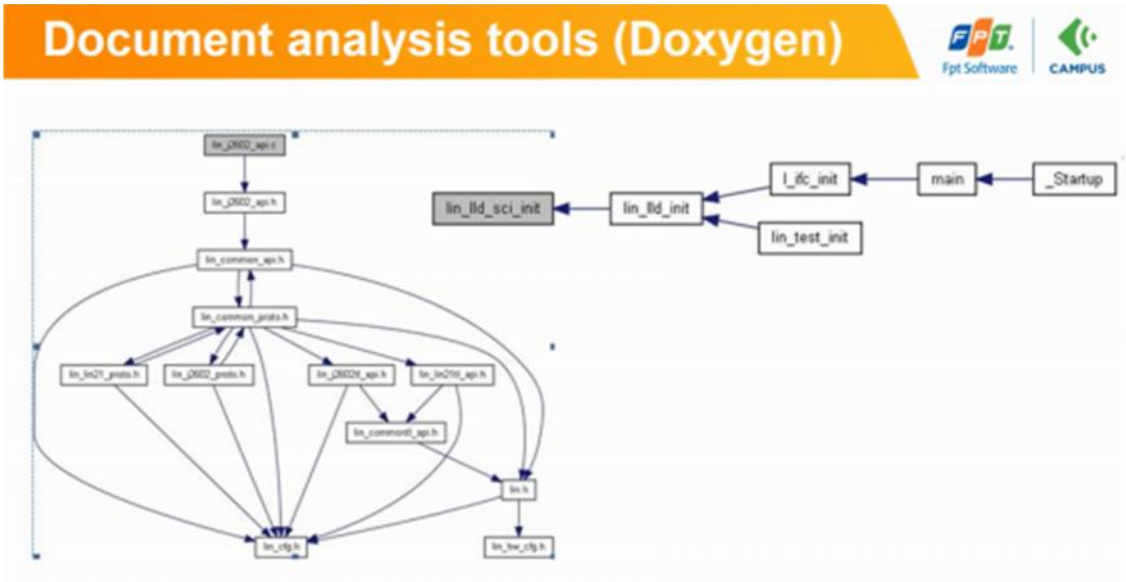
document khác nhau để phân tích ví dụ ở đây là phân tích thằng ISR, cần optimize thằng này, cần các document để xem khi mà nó chạy thì nó chạy như thế nào.

Thì ở đây là thằng ISR nó gọi đến thằng …



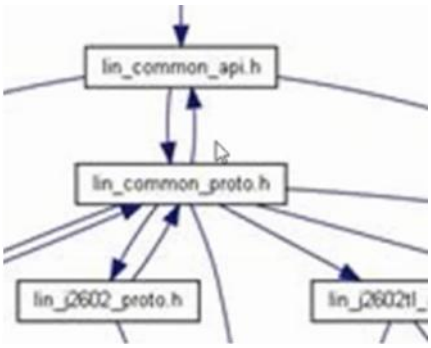
Biết được flow của ISR để biết là reduce ở đoạn nào, giảm code ở đoạn nào, cần dùng pp nào để chạy code.

Để document ra đc 1 file như thế này thì mình cần có tool Doxygen



Là 1 tool dựa trên source code của mình để nó generate document cần thiết để mình phân tích các source code đó

Gen ra các flow, gen ra các cái như là file này include cái nào, đằng sau nó là các flow như là



Include lẫn nhau thì nó ko cần thiết

Viết 1 đoạn code mà 2 register overlap