C code Optimization

Objectives

- Code optimization overview
- Classification of optimization types
- Optimization methods and terms
- Document analysis tools
- Common algorithms

Code optimization overview

- Code optimization involves the application of rules and algorithms to program code with the goal of making it faster, smaller, more efficient, and so on
- Often these types of optimizations conflict with each other: for instance, faster code usually ends up larger, not smaller
- Optimizations can be performed at several levels (e.g. source code, intermediate representations), and by various parties, such as the developer or the compiler/optimizer

Classification of code optimization

- ***Local optimizations*** - Performed in a part of one procedure.
  - ✓ Common sub-expression elimination (e.g. those occurring when translating array indices to memory addresses).
  - ✓ Using registers for temporary results, and if possible for variables.
  - ✓ Replacing multiplication and division by shift and add operations.
- ***Global optimizations*** - Performed with the help of data flow analysis (see below) and split-lifetime analysis.
  - ✓ Code motion (hoisting) outside of loops
  - ✓ Value propagation
  - ✓ Strength reductions
- ***Inter-procedural optimizations***
  - ✓ Global optimization allows the compiler/optimizer to look at the overall program and determine how best to apply the desired optimization level. Peep-hole provides local optimizations, which do not account for patterns or conditions in the program as a whole. Local optimizations may include instruction substitutions.

Optimization methods

❑ Common sub-expression elimination

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

Optimization methods (cont)

❑ Constant propagation

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

Normal	Optimization

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

Optimization methods (cont)

❑ Copy propagation

Replace multiple variables that use the same calculated value with one variable

Normal	**Optimization**

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

Optimization methods (cont)

- **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).

Optimization methods (cont)

❑ Instruction scheduling

Instructions for a specific processor may be generated, resulting in more efficient code for that processor but possible compatibility or efficiency problems on other processors. This optimization may be better applied to embedded systems, where the CPU type is known at build time

❑ Lifetime analysis

A register can be reused for multiple variables, as long as those variables do not overlap in scope

Optimization methods (cont)

❑ Loop invariant expressions (code motion)

Values that do not change during execution of a loop can be moved out of the loop, speeding up loop execution.

Normal	Optimization

for (int i = 0; i < length; i++)
 	
    x[i] += pi + cos(y);
	double temp = pi + cos(y);
	
	for (int i = 0; i < length; i ++)
	
	    x[i] += temp;

Optimization methods (cont)

❑ Loop unrolling

Statements within a loop that rely on sequential indices or accesses can be repeated more than once in the body of the loop. This results in checking the loop conditional less often.

Normal	Optimization

double temp = pi + cos(y);
 	
for (int i = 0; i < length; i++)
 	
    x[i] *= temp;
	double temp = pi + cos(y);
	
	for (int i = 0; i < length; i += 2) {
	
	    x[i] *= temp;
	
	    x[i+1] *= temp;
	
	}

Optimization methods (cont)

❑ 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

Optimization methods (cont)

- Steps of code size optimization

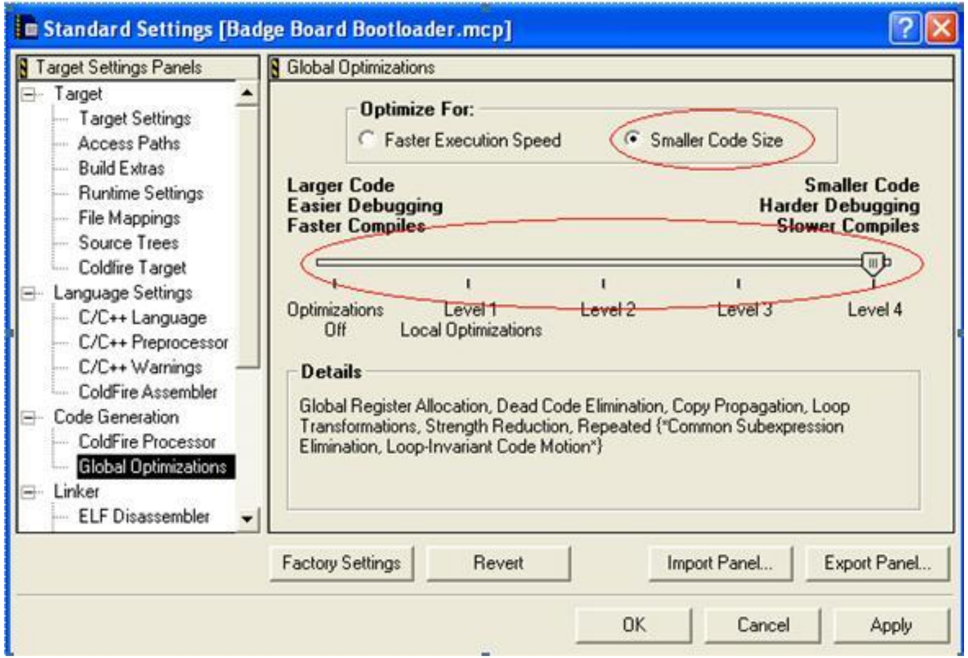
Step 1. Manual optimization

- ✓ 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.

Optimization methods (cont)

- Steps of code size optimization (cont)

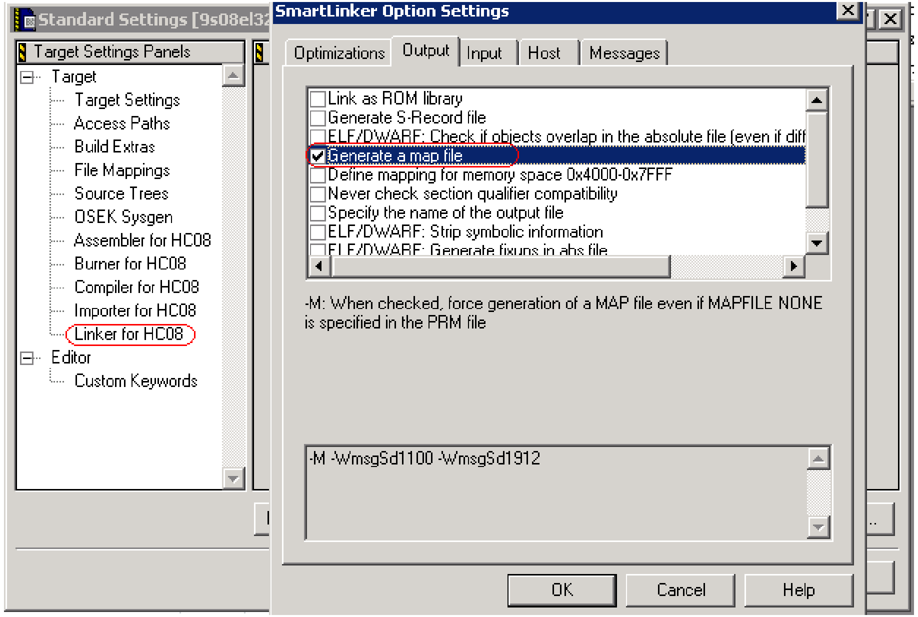
Step 2. Using compiler options (tools)



Optimization methods (cont)

- Steps of code size optimization (Practice)

Step 1. **Analyze** code distribution in memory using Linker **MAP file**



Optimization methods (cont)

- Steps of code size optimization (Practice)

Step 2. Analyze the linker MAP file and identify the objects most memory consuming

```
*************************************************************************
MODULE STATISTIC
  Name                                     Data    Code   Const
-------------------------------------------------------------------------
  Start08.c.o                                 0       7       0
  main.c.o                                    5     113       0
  mc9s08el32.c.o                            128       0       0
  RTSHC08.C.o (ansiis.lib)                    0     206       0
  lin_cfg.c.o                               164       0     273
  lin_common_api.c.o                          0     864       0
  lin_common_proto.c.o                        0    3405       0
  lin_j2602_proto.c.o                         0     438       0
  lin_lin21_proto.c.o                         0     639       0
  lin_commontl_api.c.o                        0     624       0
  lin.c.o                                     0     279       0
  lin_lld_slic.c.o                            4     124       0
  slic_isr.c.o                                2     333       2
  other                                      64      16       8
```

Optimization methods (cont)

❑ Steps of speed optimization

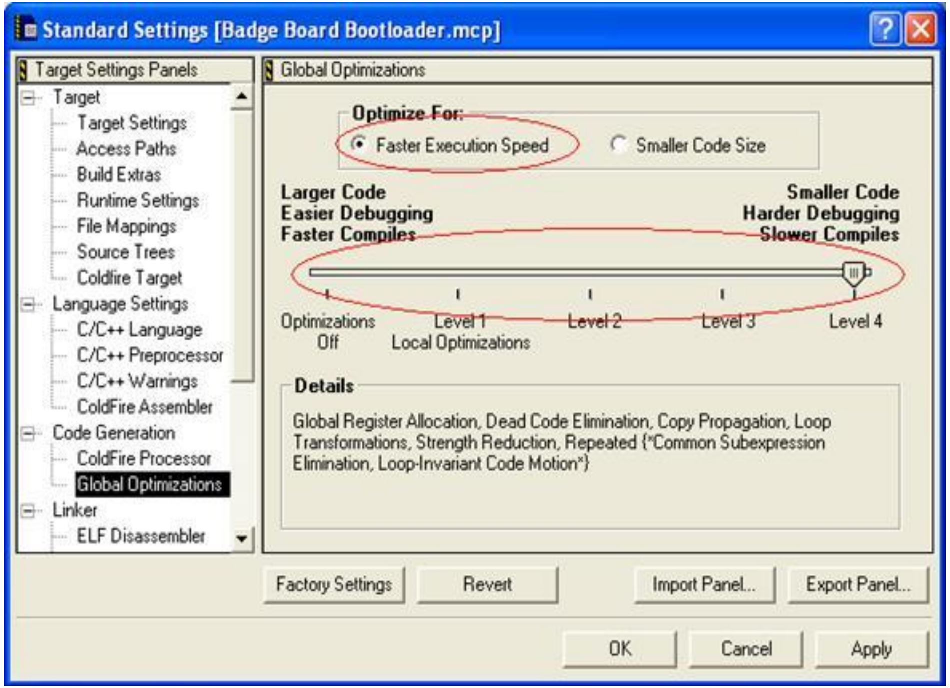
Step 1. Manual optimization

- ✓ Dead code elimination;
- ✓ Common sub-expression elimination;
- ✓ Constant propagation;
- ✓ Copy propagation: Replace multiple variables that use the same calculated value with one variable;
- ✓ Global register allocation;
- ✓ Inline calls;
- ✓ Instruction scheduling;
- ✓ Loop invariant expressions (code motion);
- ✓ Loop transformations;
- ✓ Loop unrolling;
- ✓ Strength reduction;
- ✓ Using a fast algorithm;
- ✓ Writing in assembly language

Optimization methods (cont)

- Steps of speed optimization (cont)

Step 2. Using compiler options (tools)



Optimization methods (cont)

❑ Steps of speed optimization (Practice)

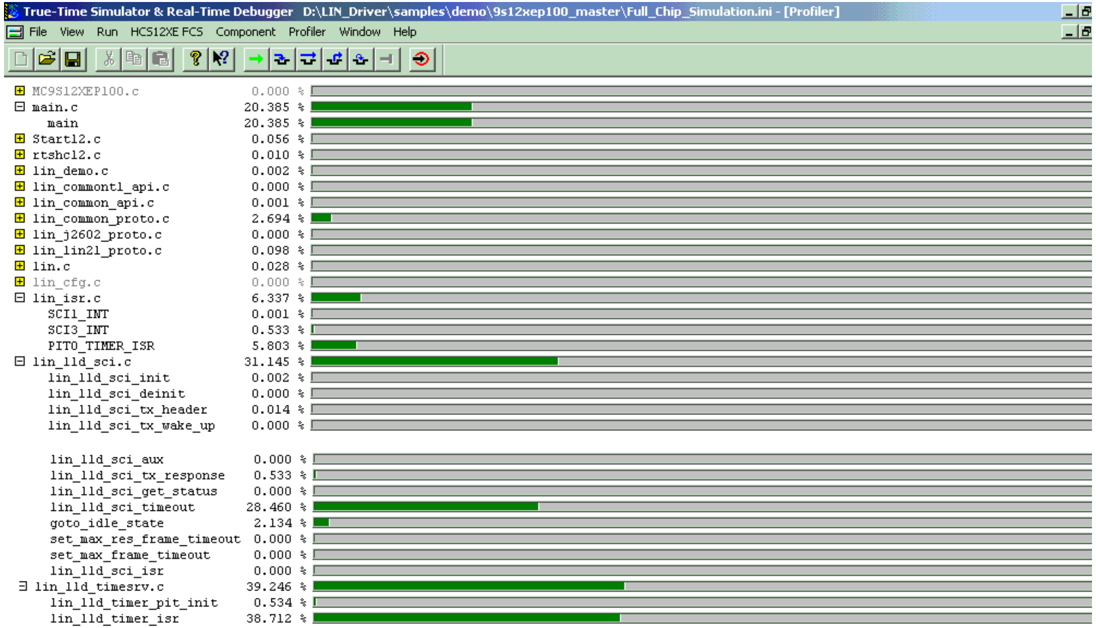
Step 1. Profile the code

- ❑ Profiling is the process of analyzing software to determine how much time, on average, an executable spends on a particular amount of code.
- ❑ While profiling can reveal a lot of useful information about your code. Profiling identifies bottlenecks -- areas of code that hold back the entire performance of the system. Optimization attempts to make those bottleneck faster.
- ❑ Most code loosely follows an 80/20 pattern of execution -- 80% of execution time is spent executing 20% of the code

Optimization methods (cont)

❑ Steps of speed optimization (Practice)

Step 1. Profile the code (cont)



Optimization methods (cont)

❑ Steps of speed optimization (Practice)

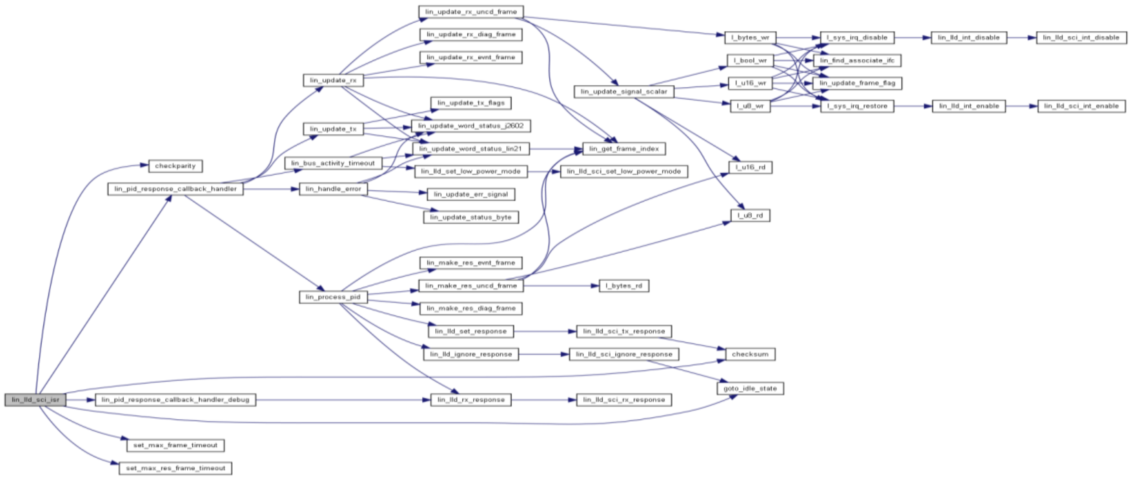
Step 2. Optimize for the speed

❑ **Inline Functions/Loop Unrolling/Strenght reduction**

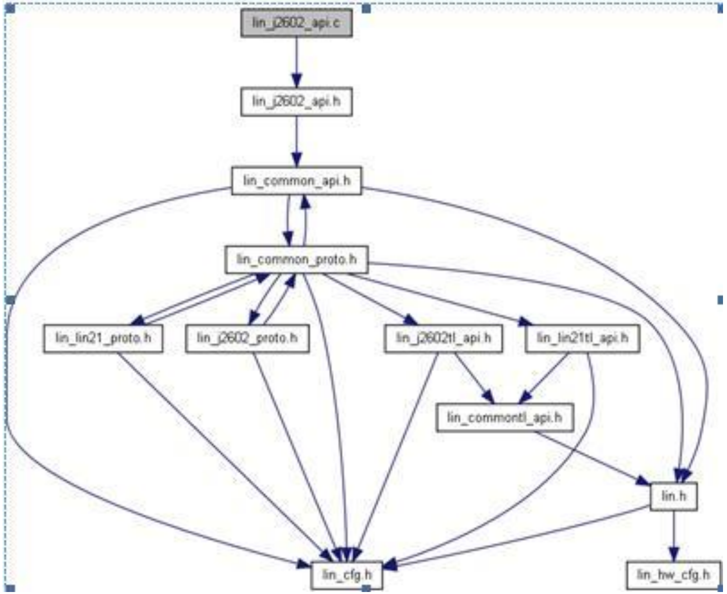
http://www.eventhelix.com/realtimemantra/Basics/OptimizingCAndCPPCode.htm

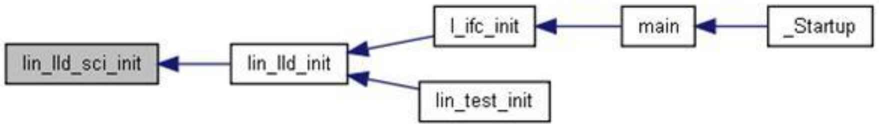
❑ **Minimize Interrupt Service Routine Overhead by keeping the ISR Simple**

Interrupt service routines (ISRs) improve performance and ease maintenance. Because ISRs operate asynchronously, they are inherently difficult to debug; so keep their tasks to a minimum.
Try to move any data processing or housekeeping tasks out of an ISR and into the main program. Then, the ISR will only grab data, say, from hardware, place it in a buffer, raise a data-ready flag and re-enable the interrupt.



Document analysis tools (Doxygen)





Common Algorithms

- Searching and sorting
  - Binary search
  - Bubble sort
  - Selection sort
  - Insertion sort
- Strings
  - Brute-force algorithm

Thank you

Q&A