EEE3096S: Embedded Systems II

LECTURE 5: COMPILER OPTIMISATION

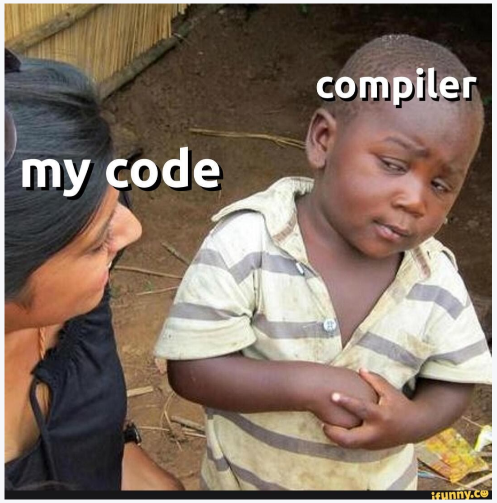
Presented by:

Dr Yaaseen Martin

Electrical Engineering
University of Cape Town

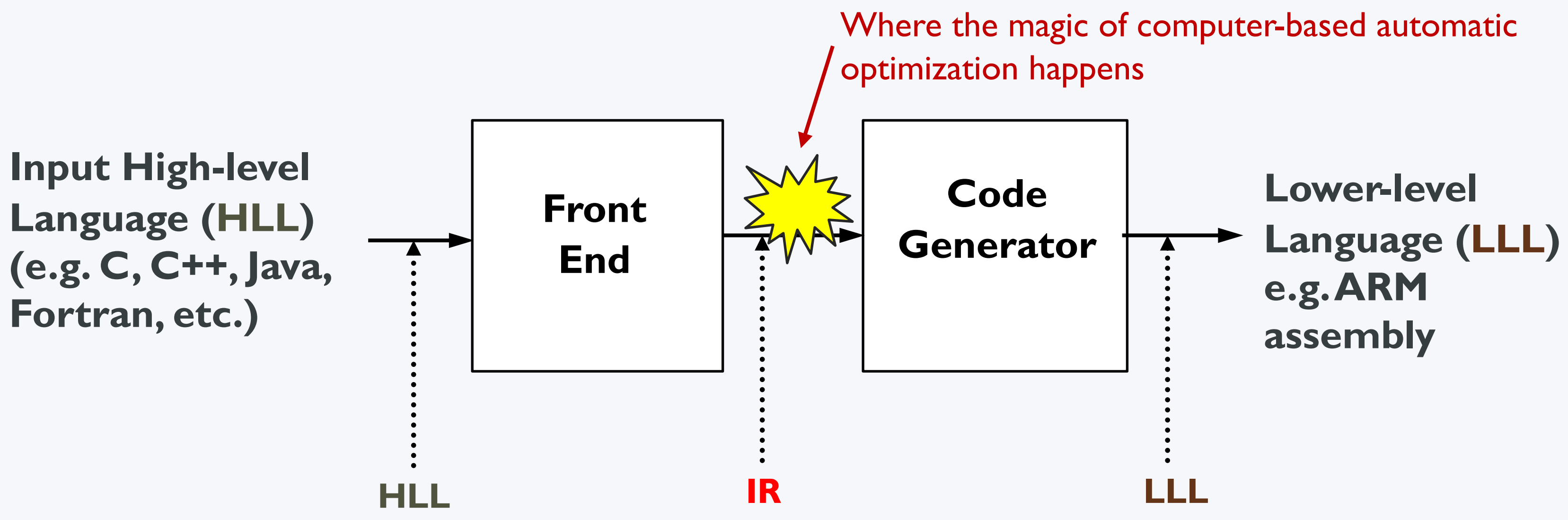
BRIEF BACKGROUND TO COMPILER OPTIMISATION

- Before we simply discuss the GCC optimisation flags, it’s appropriate for you to know a little (tiny bit) about how an optimising compiler, such as GCC, works…



INSIDE AN OPTIMIZING COMPILER

This is a very simplified view of a compiler structure…

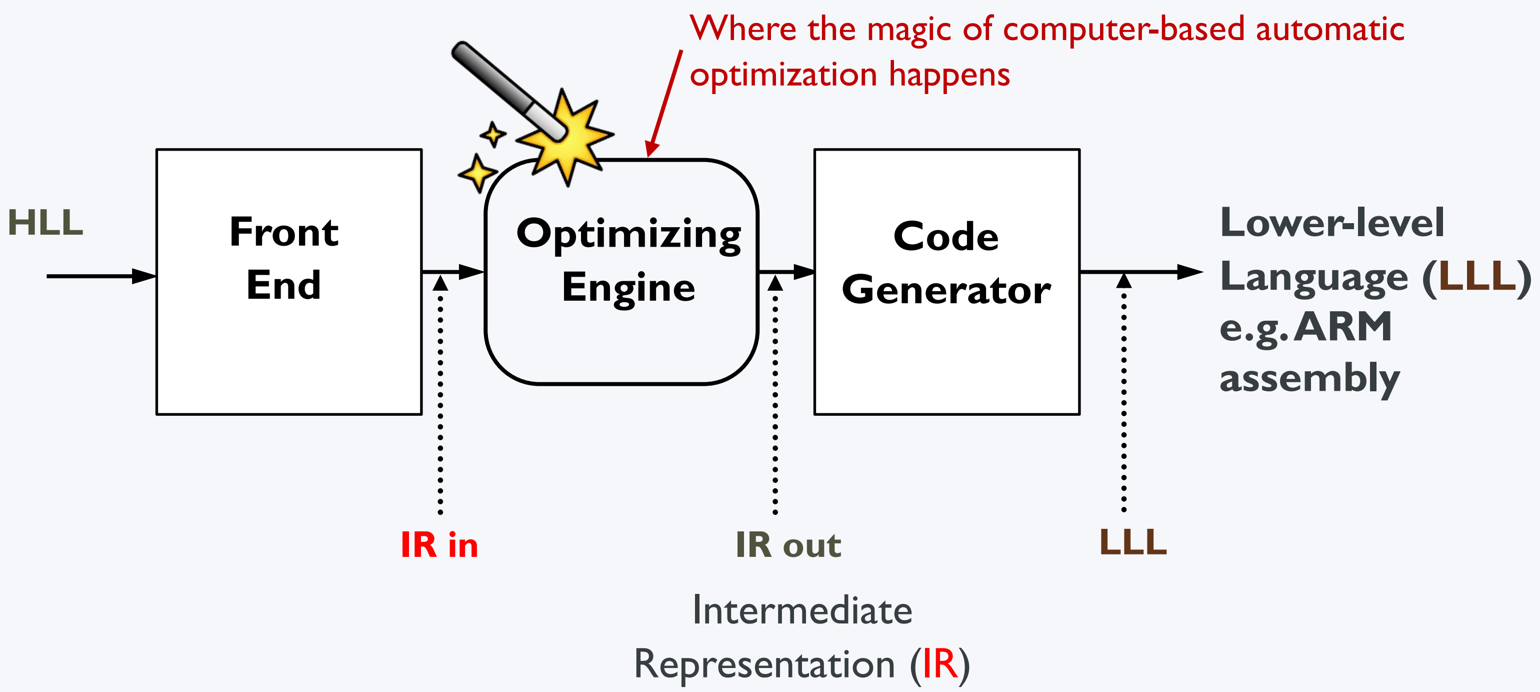

Where the magic of computer-based automatic optimization happens
Input High-level Language (HLL) (e.g. C, C++, Java, Fortran, etc.)
Front End
Code Generator
Lower-level Language (LLL) e.g. ARM assembly
HLL
IR
LLL

Intermediate Representation (IR) (similar to assembly but may be a bit more high-level and interconnected*)

The 'IR' may be quite a unique language, typically not planned to be human readable, but rather planned around capturing more of the semantic characteristics and associations to other program elements that will make it easier to generate optimize code (I'm personally quite fascinated by these IR languages and they may be designed around the platform and application you are planning to support.)

INSIDE AN OPTIMIZING COMPILER

The optimizing engine of a compiler…


Where the magic of computer-based automatic optimization happens
HLL
Front End
Optimizing Engine
Code Generator
Lower-level Language (LLL) e.g. ARM assembly
IR in
IR out
LLL
Intermediate Representation (IR)

The **optimizing engine** is where the incoming IR is reworked, maintaining data dependences and correct operation, etc., to produce a better optimized IR, possibly using the same or different IR language. This is then fed into the code generator that translates the revised IR into lower-level / assembly code (note that this lower-level come is sometimes just C, and this is fed into a more basic, non-optimizing compiler).

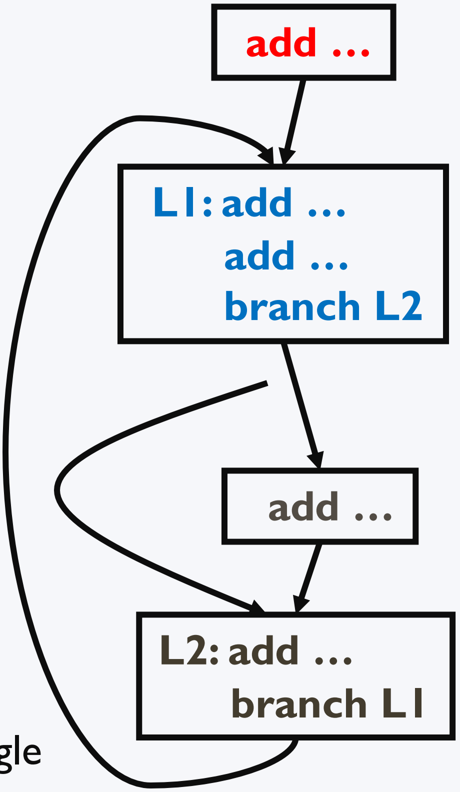
CONTROL FLOWS (TYPICAL USED WITH IR)

The way the compiler 'sees' or works on you program is typically through the use of **control graphs** or control flows. The IR syntax may incorporates the control graph structure (not show in the simple IR illustrated on left).

Example IR:

```
    add ...
L1: add ...
    add ...
    branch L2
    add ...
L2: add ...
    branch L1
```

Basic Blocks*:



*__Basic Block:__ a group of sequential instructions with a single entry point and a single exit point.

THREE PERFORMANCE OPTIMISATION INVARIANTS

Invariant: a quantity or property that remains unchanged when a transformation is applied

The three main performance optimisation invariants that an optimising compiler should maintain are the following:

- Thou shalt Preserve Correctness 😌
  - (the speed of an incorrect program is likely still worse than a slow one that works correctly)
- Thou shalt Improve Performance On Average 😎
  - Optimised may be worse than original if unlucky on some pieces but overall it should be better otherwise it isn't an optimization
- Thou shalt Be Worth It 🥹
  - If the optimised code becomes more fragile, or difficult to debug, one needs consider if it is worth it either as a user or as the compiler developer.

OVERARCHING APPROACH: MINIMISE CPI AND NOI

- The thing to keep in mind is:

  execution_time = NOI * CPI

  where: CPI = cycles per instruction*
  NOI = number of instructions
- Fewer cycles per instruction
  - Sequence instructions to avoid dependencies and pipelining (e.g. avoid having the next instruction blocked due to waiting for a result from the previous instruction)
  - Improve cache/memory behaviour (e.g. improving locality, relative addressing, use registers or stack more)
- Fewer instructions
  - Make better use of the available instruction on the target processor (e.g. specialized instructions)

OPTIMISING APPROACHES

- Efficient mapping of program to the architecture
  - Get rid of minor inefficiencies
  - Ensure efficient code selection and ordering
  - Optimize the register allocation (e.g. sometimes it might be better to have more memory accesses in parts if it means other parts, e.g. a loop, has less)
- Allow options for programmer to select best overall algorithm (don’t optimize out what the programmer is trying to do)

RESPONSIBILITY OF THE PROGRAMMER?

- Do not:
  - Write messy code that is woefully inefficient so that the compiler can optimise it
  - Write difficult-to-read code
- Do:
  - Write readable and maintainable code (very NB!)
  - Use procedures if possible (optimisers are better with this), or recursion
  - Eliminate optimsation blockers (i.e. things the optimiser has difficulty understanding and likely won't be able to optimise, even though the larger problem may be obvious to a human)
  - Focus on Inner Loops (this is usually the crux of the problem and you can rely on the compiler to use things like loop unrolling to optimize the looping)
  - Put some effort into manually optimising the code — particularly parts that are executed repeatedly

SOME OPTIMISATIONS TO KNOW ABOUT

This isn’t a compilers course, so you won’t be expected to know much about code optimization techniques. However, as an embedded systems engineer, there are two things you *should* know about:

1. Small Function In-lining (SFI)
2. Loop Unrolling (LUR)

These two essential ingredients I will briefly explain before we close this section; these are among the many strategies used in GCC’s optimization level 2 (invoked with the –O2 flag discussed later)

SMALL FUNCTION IN-LINING (SFI)

Example scenario:

1. Original code

```
#define DC 3

int addDC (int z) {
  int m = DC;
  return z + m;
}

main(){
  …
  x = addDC(x);
  …
}
```

2. In-lining of simple functions

```
main(){
 …
 {
  int tmp1 = x;
  int m = 3;
  int addDC_r = tmp1 + m;
  x = addDC_r;
 }
 …
}
```

3. Further Optimizing

i.e. dead-code elimination and constant folding. Remove unnecessary assignments.

```
main(){
  …
  x = x + 3;
  …
}
```

Main Benefits:

- **Code size:** Can be decreased, and reduce calling overhead, if small procedure body are simply brought into the calling procedure (i.e. 2nd step above)
- **Performance:** eliminates call/return overhead, and can expose the potential further for optimizations (see 3rd step above). <u>But</u>, it can also eat up more instruction memory (trading space for speed)

LOOP UNROLLING (LUR)

Example scenario:

1. Original code

```
j = 0;
while (j < 100) {
    a[j] = b[j+1];
    j += 1;
}
```

2. Loop unrolling

```
j = 0;
while (j < 99) {
    a[j] = b[j+1];
    a[j+1] = b[j+2];
    j += 2;
}
```

And obviously note this also, the 2nd version has ½ the iterations.

Some processors, like ARM, allows the memory address to be added and incremented in one instruction, these instructions are independent so this is better than filling up the pipeline with memory reads and writes, that do not depend on each other. This might speed up the loop something like 2x (depending on the pipelining, width of memory bus to cache etc.)

Main Benefits:

- **Reduce looping overhead:** fewer adds to update j, and also fewer loop condition tests
- Allows **more aggressive instruction scheduling**, i.e. more instructions for scheduler to move around

GCC OPTIMISATION FLAGS

There are a great many options and optimization settings that you can set in GCC, a full list can be seen at:
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

But these are the essential flags that are most commonly used:

Flag	**Description**
-g:	Include debug information, no optimization
-O0:	Default, no optimization
-O1:	Do optimizations that don't take too long. *Including (look these up if you're interested!):* CP (constant propagation), CF (constant folding), CSE (common sub-expression elimination), DCE (dead code elimination), LICM (loop invariant code motion), ISF (in-lining of small functions)
-O2:	Take longer optimizing, more aggressive scheduling
-O3:	Make space/speed trade-offs: loop unrolling, more in-lining
-Os:	Optimize program size

To use these, simply use e.g.: `gcc -O1 main.c -o myprog`