50006 - Compilers - (Prof Kelly) Lecture 6

Oliver Killane

07/02/22

Introduction to Optimisation

Lecture Recording

Lecture recording is available here

GCC -O

- **Without Optimisations**
  Program is slower, however easier to debug as generated code represents a simple translation.
- **With Optimisation**
  Various optimisations such as loop hoisting (moving loop invariant code out of the loop), constant propagation, inlining and smart register allocation make the program execute more quickly, but at the cost of the debugger.

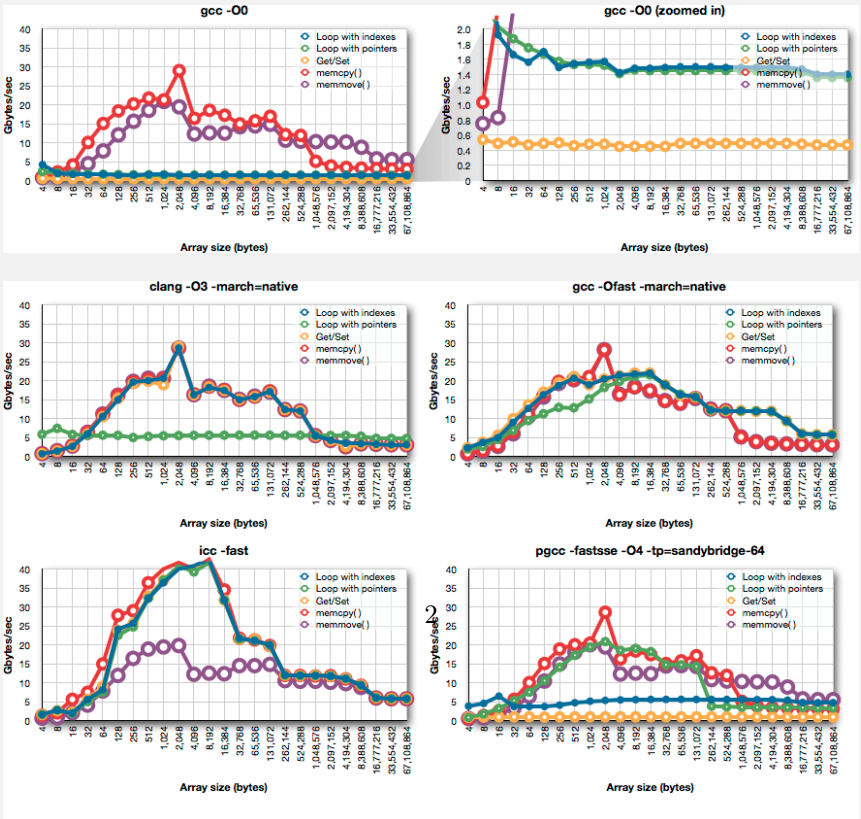
  Debuggers now much less useful as statements are reordered or completely changed from the source, & degugger tools such as setting variable values, jumping to lines and breakpoints will often not have the intended effect.

Example: Copying Arrays

When copying an array of values the compiler can optimise in several areas (index calculations, highly optimised copying from libraries such as **memcpy**) including in the most advanced case using vectorised instructions which can use a single instruction on multiply sections of data in an array (SIMD).

An example of this is here:

```
1  /* Loop with array indexes */
2  for ( size_t i = 0; i < length; ++i )
3          dst[i] = src[i];
4
5  /* Loop with pointers */
6  const int* s = src;
7  int* d = dst;
8  const int*const dend = dst + n;
9  while ( d != dend )
10         *d++ = *s++;
11
12 /* Loop with get/set function calls */
13 int get( const int*const src, const size_t index ) { return src
       ↪ [index]; }
14 int set( int*const dst, const size_t index, const int value ) {
       ↪  dst[index] = value; }
15
16 for ( size_t i = 0; i < n; ++i )
17         set( dst, i, get( src, i ) );
18
19 /* Memcpy (two regions are disjoint) */
20 memcpy( (void*)dst, (void*)src, n * sizeof(int) );
21
22 /* Memmove (for when regions may overlap) */
23 memmove( (void*)dst, (void*)src, n * sizeof(int) );
```



Definition: High Level Optimisation

Using high-level information encoded in the program (types, function analysis).

Example: Function Inlining

Replace the call site with a copy of the body of the function.

- Avoids call/return overhead.
- Creates further oportunities to optimise (e.g part of function result not used, or removal of call allowing for better register allocation).
- Can require static analysis (e.g call to a virtual or overloaded function requires information about types)

Definition: Low-Level Optimisations

Using low-level information (instruction types, the ISA, the order of instructions in the IR, etc) to optimise the output.

Note that we can loose information from high level representations as we get lower level.

Example: Instruction Scheduling

In pipelined architectures instruction order can impact the speed of processing, (e.g instructions immediately after a conditional branch may be waiting after a pipeline stall).

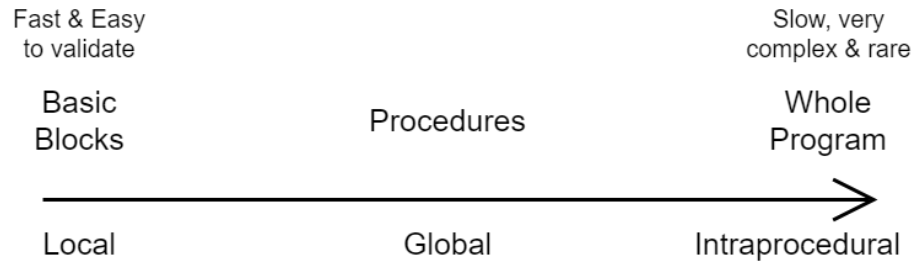
- Re-Order instructions to better allow parallel processing.
- Requires some dependency analysis (e.g switching order of storing to an array - check indexes, do $A[i]$ and $A[j]$ refer to the same location in a given context).

The spectrum of Optimisations

Definition: Peephole Optimisation

Scan through the assembly in order, looking for obvious cases to optimise.

- Can catch some of the worst cases (e.g store follwed by load of the same location).
- Very easy to implement (at smallest just consider two adjacent instructions).
- *Phase ordering problem* in what order should the optimisations be applied to get the best result?



Local	Optimisation on the level of basic blocks (single entry adn exit points) (e.g expressions)	Runs quickly and easy to validate.
Global	Optimisation on the scale of a whole procedure	Can have worse than $O(n)$ (Linear) complexity given n represents instructions, basic blocks or variables.
Intraprocedural	Optimisation over the whole program	Rare and hard to avoid excessive compile times.

Loop Optimisations

- **Loop Invariant Code Motion**
  An instruction is loop-invariant if its operands (inputs) only arrive from outside the loop (moving it out of the loop does not change the semantics).

  Hence the loop iteration has no effect on the value, so we can move this computation from within the loop (being redundantly run with the same inputs many times) to the loop preheader (compute value just before loop and then use).

  More generally **strength reduction** is replacing a compelx operation, with a smaller simpler one.

```c
int a = 7;
for (int b = 0; b < 4; b++) {
    int c = a + 3 * 2; // invariant code
    printf("%d", c);
}
```

```c
int a = 7;
int c = a + 3 * 2; // code hoisted out of loop
for (int b = 0; b < 4; b++) {
    printf("%d", c);
}
```

- **Strength Reduction**
  An **induction variable** is one which increases/decreases by a **loop invariant** amount each loop iteration.

  Once we detect **induction variables** we can use less costly instructions to compute their value.

```c
int a = 7;
for (int b = 0; b < 4; b++) {
    a += 4;
    printf("added 4 to a");
}
```

```c
int a = 23; // 7 + 4 * 4
for (int b = 0; b < 4; b++) {
    printf("added 4 to a");
}
```

Or a far, far more complex example:

```c
int a = 9;
int b = 7;
while (some_predicate(a)) {
    b = some_function(b);
    for (int i = 0; i < b; b++) {
        a++
    }
}
```

```c
int a = 9;
int b = 7;
while (some_predicate(a)) {
    b = some_function(b);
    a += b
}
```

- **Control Variable Selection**
  Replace the loop control variable with one of the induction variables that is used. The exit condition for the loop must be changed to work with the variable chosen.

```c
int* a = malloc(15 * sizeof(int));
for (int i = 0; i < 15; i++) {
    a[i] = 5;
}
```

```c
int* a = malloc(15 * sizeof(int));
int* end = a + 15;
for (; a != end; a++) {
    *a = 5;
}
```

- **Dead Code Elimination**
  Code that does not produce a used result can be eliminated.

  many other optimisations result in dead code (e.g inlining a function where not all the function's returned values or optional arguments are used.)

Intermediate Representation

An **intermediate representation** (**IR**) of the program used in code synthesis and optimisation.

- Must represent all primitive operations needed for execution.

- Be easy to analyse and manipulate.
- Independent of the target instruction set (allowing for many different ISAs to be targeted by backends using the **IR**).
- Uses temporary variables to store intermediate values to allow for some optimisations and register assignment (e.g via graph colouring).
- Still an area of discussion & active research, many different approaches including multiple IRs for different stages are used.

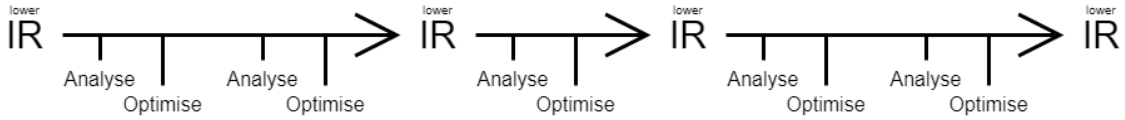
Definition: Lowering Representation

Taking high-level features and converting them into lower-level representations.

For example taking arrays and converting them into pointer arithmetic/address calculation.

When lowering you loose high-level information (e.g that values are part of an array), but can optimise the lower level representation (optimise address calculations).

Optimising Compiler synthesis can be described as several stages of analysis, optimisation and lowering of **IR**.



Data Flow Analysis for Live Ranges

Lecture Recording

Lecture recording is available here

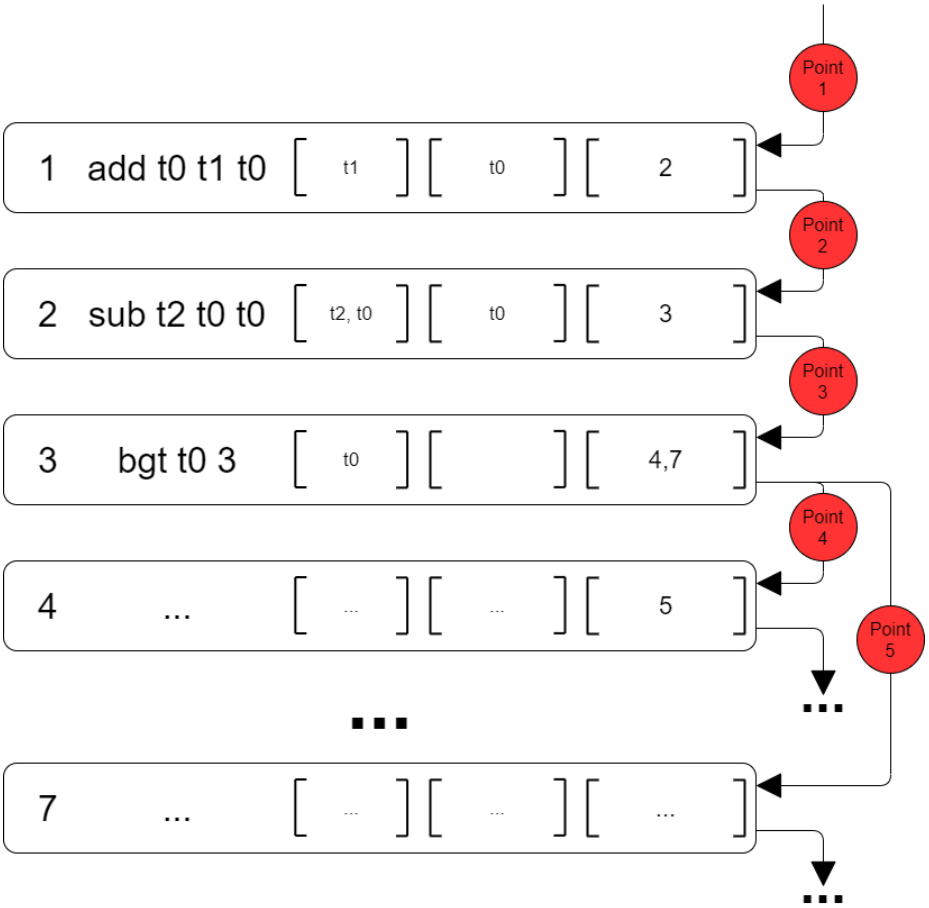
A live range is the range of instructions for which a temporary value must be maintained.

A live range starts at a definition, and ends when either the variable is used, or immediately if the value is never used.

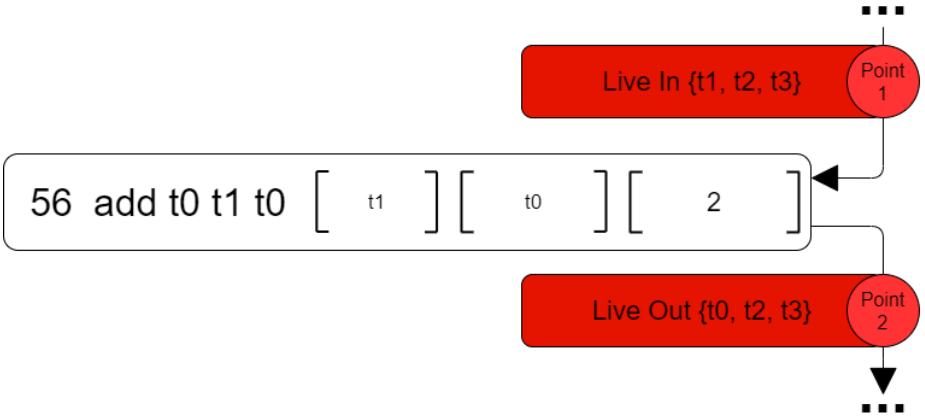
Control Flow Graph

We can generate a graph of simple **IR** (ideally 3 code) instructions in the program:

Live Range Definitions



- A **point** is any location between adjacent nodes.
- A path is a sequence of points traversing through the control flow graph.
- A variable can be *live* at a point if it may be used along some path that goes through the point.



- **Live Out** Live immediately after the node if it is live before any successors.

$$LiveOut(n) = \bigcup_{s \in succ(n)} LiveIn(s)$$

- **Live In** If it is live after the node (unless overwritten by the node) or it is used by the node.

$$LiveIn(n) = uses(n) \cup (LiveOut(n) - defines(n))$$

Iterative Method to get Live Ranges

```
for node in CFG {
    LiveIn(node) = {}
    LiveOut(node) = {}
}

repeat {
    for node in CFG {
        LiveIn(node) = uses(node) set-or (LiveOut(node) - defines(node))
        LiveOut(node) = set-or of LiveIn(all successors to node)
    }
} until LiveIn and LiveOut do not change
```

Once the iteration stops, the assignments will hold as predicates, meaning the definitions for *LiveIn* and *LiveOut* will be true.

Given there are limited nodes, and the size of the *LiveIn* and *LiveOut* can only increase, we know it will terminate.

Improvements to the Iterative Method

To reduce required iterations it is best to update the nodes from last → first, as generally the program will be connected to go from first instruction to last (cycles can exist which affects this). As the definitions are dependent on successor nodes.

Hence we consider it a *backwards analysis*.

The time complexity is dependent on the number of instructions and temporaries/registers.