yz709-compiler-sup4

Question 1

This question is about function inlining. It is a common optimisation and most C implementations have directives for it. How can it adversely affect computation time? What sort of functions would be ideal candidates for inlining? (*Hint*: remember computer design lectures)

- Inline expansion is a technique where the compiler copies the code from the function definition directly into the code of the calling function rather than creating a separate set of instructions in memory.
- It improves execution time by avoiding call overhead and provides an opportunity for further optimisations. However, since after inline expansion, we reply on the inlined code's internal implementation details, if we ever change the inline callee codes, every place where we have this inline expansion would have to be changed and recompiled, which further increases build and development time, thus affect computation time.
- Very small functions are good candidates for `inline` because after inlining, the function code snippet and the caller function are located closely, with this spatial locality, these two code snippets have a higher chance of staying in the code cache. Since it is more efficient and cost less if we fetch objects from the cache instead of the main memory, the cache boosts efficiency and boosts performance.
- It is even better if the small function is called very often because in that way, it follows the temporal locality of the cache and the same code snippet can be used multiple times without reloading from the main memory.

💡 Comments: 

Question 2

What is the scope of peephole optimisations? What are the other class of optimisations and rank them in difficulty with brief justification to your ranking. Give five examples of peephole optimisations.

- Scope of peephole optimisation: a technique performed late in the compilation process after machine code has been generated, usually uses a window to examine a few adjacent instructions, and see whether they can be replaced by a shorter and faster set of instructions.

```
(1) eliminate redundant load and store
def f(a):
  b = a + 5;
  c = b;
  d = c * 2;
  return d + 2;
# c = b can be eliminated, and write d = b * 2 for the final statement

(2) strength reduction: replace high execution time operators with lower ones
def g(a):
  return a * 5
# the * operator has a much higher cost compared to +,-,<<,>>
# hence replace a * 5 by (a << 2) + a or simply a + a + a + a + a

(3) null sequence deletion
push 0; pop;
# consecutive statements leave the stack and heap unchanged,
# so can be eliminated altogether

(4) combine operations: several operations are replaced by a single one
def f(a):
  b = a + 5;
  d = b * 2;
  return d + 2;
# replace by => def f(a): return (a+5)*2+2;

(5) improve control flow
GOTO L1 ...; L1: GOTO L2
# replace by => GOTO L2 ...; L1: GOTO L2
```

- Classes of optimisation (in ascending order of difficulty)

- (1) Peephole optimisation, (2) local optimisation to a basic block that has no control flow, (3) loop optimisation to the statements of a loop, (4) global optimisation or inter-procedural optimisation which analyse the whole program's source code
- With a wider scope, more analysis is needed and more time and storage requirements to store intermediate information. Within a function block, control flow is more difficult to optimise and analyse compared to normal statements.

💡 Comments: 🟨

Question 3

In C, when you are allocating memory with `malloc` you need to pass the size of the memory you want to be allocated and the routine returns a pointer. When you later want to free this memory you use the free routine and only pass the pointer. How come *libc* knows how much memory to deallocate?

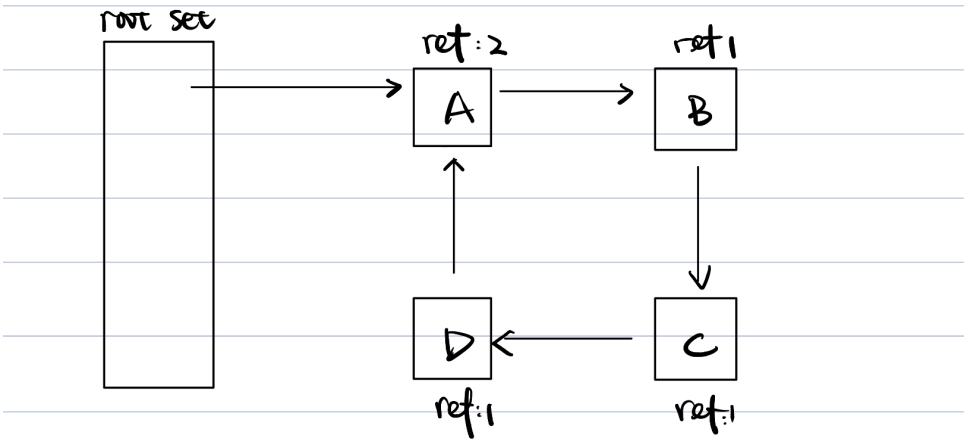
- `malloc` assigns a chunk of memory on the heap and returns a pointer that points to that chunk of memory. The header of the memory chunk stores the size of the chunk, so that when the free routine pass the pointer in, *libc* looks up the header and frees that amount of memory.
- In some implementations, the size is stored in a lookup table elsewhere on the heap, so that `free` would trigger *libc* to look up the table to find the size.

💡 Comments:

Question 5

Cycles in reference counting is a well-known problem. Briefly explain the problem. Suggest a way to solve it without completely changing to a different garbage collection algorithm.

- Problem: if the pointer from the root set to object A is deleted, then the cycle of objects (ABCD) can no longer to reached, but all objects still have a reference count of 1, so they will not be deleted by the algorithm.



- Solution: add another cycle detector mechanism, whenever the reference count of an object decreased to 1, we would run a cycle detector from that object to see whether it will return back to that object, if so, then the object is part of a cycle and it should be deleted.

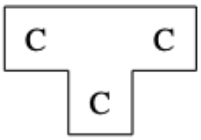
💡 Comments: 🟨

Question 6(Exercise 13.2)

Exercise 13.2

A source-code optimiser is a program that can optimise programs at source-code level, *i.e.*, a program O that reads a program P and outputs another program P', which is equivalent to P, but may be faster.

A source-code optimiser is like a compiler, except the source and target languages are the same. Hence, we can describe a source-code optimizer for C written in C with the diagram



Assume that you additionally have the following components:

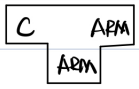
- A compiler, written in ARM code, from C to ARM code.
- A machine that can execute ARM code.
- Some unspecified program P written in C.

Now do the following:

a) Describe the above components as diagrams.

b) Show, using Bratman diagrams, the steps required to optimise P to P' and then execute P'.

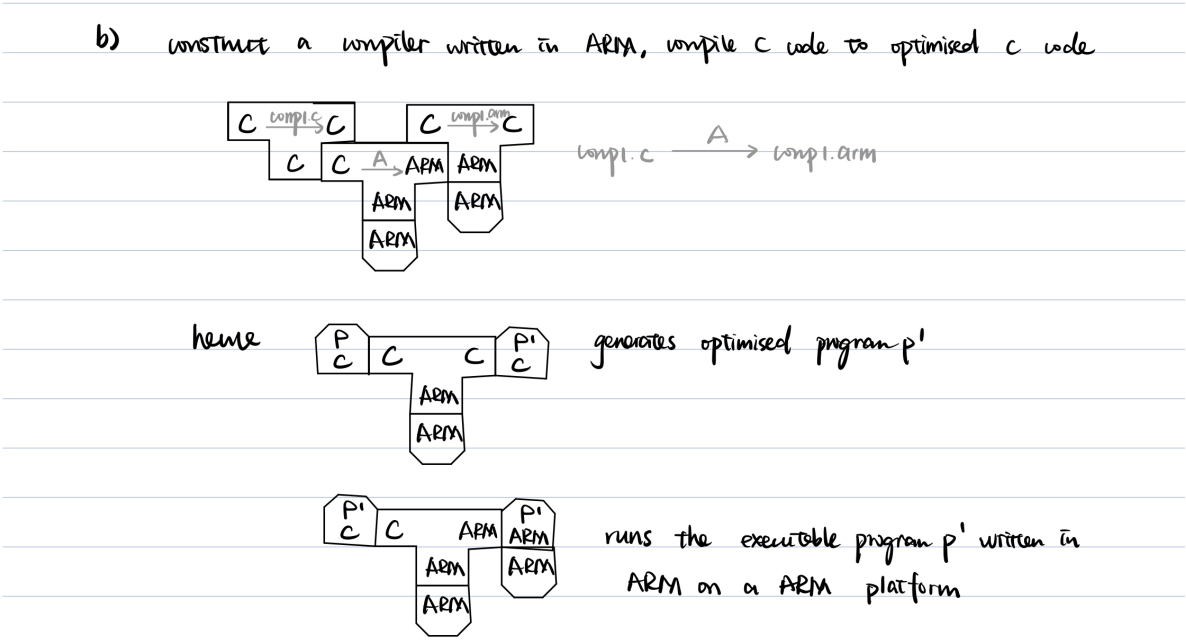
a)



A compiler written in ARM, from C to ARM

A machine execute ARM

A program P written in C



Comments:

Question 7(y2017p3q3)

(a) Explain why some programming languages require automatic memory management ("garbage collection") for program execution. [4 marks]

- By allowing programmers to implicitly allocate new storage dynamically, with no need to worry about reclaiming space no longer used, memory could easily be exhausted without some methods of reclaiming and recycling the spaces.
- Since it is hard for the programmer to know when an object in the heap could be safely deleted, automatic garbage collection is required, which also eliminates the problems that programmers might forget to free an object and cause a memory leak or attempt to access memory for an object that has already been freed.

(b) At a given point in the execution of a program, what can be considered as garbage? How can garbage be located in memory? [4 marks]

- Garbages are those objects that will no longer to accessed in the following execution of a program, the resources will be wasted or cause memory leaks if we do not recycle the memory spaces back. For instance, if we make some temporary array objects and copy the contents of it to another memory location (i.e., deep copy), if we no longer need the temporary array anymore, it is considered garbage.
- We can start from the root set, do a depth-first-search to find all objects that are reachable from the root set, unreachable objects are garbage.

(c) Suppose a programmer is implementing garbage collection using reference counting. Discuss whether or not they need to consider the possibility of a reference count overflowing when incremented. [4 marks]

- The worst case is when all objects in the heap and all entries from the root set point to a particular object (including a self pointer), so if there are 2^n entries in the heap and the root set, and we set the reference counter with n bits, then there is no possibility that the reference count will overflow.
- But if the reference counter has less than n bits, then we need to consider the possibility. However, among practical implementations, they never handle reference counter overflow by ensuring the capacity of the reference counters.

(d) Suppose we are writing a compiler for an ML-like language. We want to employ the equation

```
(map f) o (map g) = map (f o g)
```

as a left-to-right rewrite rule for optimisation. The symbol `o` represents function composition — for any value `v` the expression `(f o g) v` evaluates to the value of `f(g v)`.

Discuss the merits of this idea. Is it always correct? [8 marks]

- No, if we have a list of values, and if each function `f` and `g` cause some side effects in the storage or print out some values in the console, then the evaluation order of the list of values matters.
  - For $(map\ f) \circ (map\ g)$, we would apply `g` to every value of the list, then apply `f` to the updated list of values; But for $map\ (f \circ g)$, we would apply `g` and then `f` to every value of the list, hence each value is evaluated fully before operating on the next one.

- But this idea also has some advantages: we only need to apply the function `map` on each value in the list once, which saves computation time and improves efficiency.

> 💡 Comments:

Question 4

After reviewing lecture notes on garbage collection, read Bacon et al. (2004) up to and including section 4.