Name: ***Solutions***

100 points total. Each point should take roughly a minute of your time to answer.

1) a) [3] Name three ways in which hoisting loop invariant code out of the loop can violate program correctness.

divides can be hoisted out of a loop, leading to errors which wouldn't have occurred before

an assignment can be hoisted out of the loop, overwriting a value before it has been read (turning an antidependence into a flow dependence); a similar change reverses output dependences.

a computation can be moved before its operands have been computed.

b) [3] Even if loop invariant code motion doesn't violate correctness, it can hinder transparent source-level debugging (i.e., examining and manipulating the program as if no optimizations had been performed). Explain how. How can a significant amount of work be hoisted out of loops without hindering debugging? [Hint: some code space cost can be imposed.]

Statements can be reordered, potentially changing the values of variables in ways that make delivering a clear source-level picture of the program state difficult.

The first iteration of a loop can be peeled off, with CSE optimizing the remaining iterations of the loop.

c) [2] How could profiling information help improve the quality of loop invariant code motion?

Profiling could tell how many iterations of a loop are typical (if < 1, then don't do loop invariant code motion).

Profiling could tell whether some branch within a loop was executed often enough that hoisting code out of the branch would save time.

2) a) [2] Define what a live range is, precisely.

A live range is a region of the control flow graph covered by a connected maximal def/use graph.

b) [3] When constructing the interference graph as part of live variables analysis, what special work has to be done at merges?

Interferences among all the variables in at least one but not all of the merging live variables sets must be introduced.

c) [2] What was the key contribution of the Chow & Hennessy Priority-Based Coloring algorithm?

Live range splitting.

d) [4] In Wall's link-time register allocator, how do the compiler-generated register actions make link-time processing faster?

The compiler pre-computes the effect on the generated instructions of allocating a given variable to a register. The linker only has to quickly run through these actions to implement allocating a variable to a register; no link-time analysis is needed.

3) a) [4] Explain how doing register allocation before instruction scheduling can hurt the quality of the generated code.

Two distinct variables can be assigned the same register, which prevents instructions manipulating those variables to be reordered.

b) [4] Explain how doing instruction scheduling before register allocation can also create problems.

Instruction scheduling can increase the demand for registers by moving defs far away from uses, introducing additional spill code.

The register moves and spill code inserted by the register allocator won't be scheduled well.

c) [2] How can pointer analysis help improve instruction scheduling?

Loads and stores through unaliased pointers can be reordered to improve scheduling.

d) [2] The Gibbons & Muchnick list scheduling algorithm takes into account flow, anti-, and output dependences, but not control dependences. Why not?

It's a local scheduler. There are no control dependences within a basic block.

4) a) [4] Specify as an equation how `gprof` computes T_f, the hierarchical time spent in a function `f`, given S_f, the time spent within f itself independently of `f`'s callees, and w_e^r, the number of times function `r` called function `e`. (You may ignore recursion.)

$T_f = S_f + \Sigma_{c \text{ called by } f} T_c * (w_c^f / w_c)$, where $w_c = \Sigma_{f \text{ calling } c} w_c^f$

b) [3] What assumption is `gprof` making in this equation?

gprof is assuming that each invocation of a function takes the same amount of time (hierarchically).

c) [5] How would you augment the information monitored by `gprof` to avoid this simplification, without introducing any compiler analyses?

I'd change the run-time monitoring code to directly increment the time of all procedures on the call stack at each sample.

d) [3] What advantage does the approach exemplified by Ball & Larus's QPT have over the approach exemplified by `gprof`?

It does accurate rather than statistical profiling. It can profile fine-grained things like the # of times a variable is referenced.

5) a) [3] Give three reasons why you might want to debug optimized code (assuming there were no bugs in the optimizer).

Optimized code can have different timing properties than unoptimized code.

Optimized code might do different things than unoptimized code, due to (intentional) underspecification of the language (e.g., order of evaluation of operands).

Optimized code might run a lot faster.

Don't want to have to recompile to debug, or to keep around two versions of a program.

b) [5] What tables does the compiler need to generate for the debugger in order to support transparent source-level debugging in the presence of inlining?

The compiler generates a tree of scope descriptors, the root descriptor for the procedure being compiled and nested descriptors for each inlined procedure. The compiler also generates a map from p.c. to scope descriptor.

6) a) [2] What is required of a language's run-time environment in order to use a compacting or copying garbage collector?

Pointers must be able to be identified unambiguously.

b) [5] The key implementation difference between generational & non-generational collectors is that the generational ones require a write barrier. What is the purpose of the write barrier? Describe two techniques for implementing a write barrier.

The write barrier enables the collector to catch all updates to old space, so that pointers from old space to new space can be identified efficiently and used as roots when collecting new space.

Several strategies for write barriers exist: hardware-level page protection and software-level card marking or sequential store buffers or remembered sets.

c) [4] The Ephemeral Collector implemented by Moon for Symbolics LISP machines used 8 generations rather than the 2 generations used by Ungar's Generation Scavenging system. Why might you want more than 2 generations?

Collecting intermediate generations is probably faster than doing a full collection of the entire heap, so intermediate generations act as a way to delay doing a full collection longer, by collecting intermediate-lifetime data structures at an intermediate frequency.

7) a) [3] C has functions as values (i.e. function pointers), but can get away without implementing closures. Why?

C has no nested, lexically-scoped functions.

b) [4] Cecil heap-allocates closures, but this is not sufficient to support fully first-class functions (upward funargs). Why not?

The lexically-enclosing environment must also be heap-allocated, to allow functions to be returned (or otherwise outlive) their enclosing function.

8) a) [4] Typically, statically-typed languages such as C++ and Modula-3 implement dynamic dispatching with indexed table lookups, while dynamically-typed languages such as Smalltalk and Cecil use other, non-table-driven lookup techniques. Why is indexed table lookup less feasible in dynamically-typed languages?

The static type information is guaranteeing that no messages will be sent to objects that don't understand them. This allows only the messages understood by a class to be given table entries. Otherwise, either every class would have to have an entry for every message, or extra run-time checking would be required to do a bounds check and a message name clash check on each table lookup.

b) [3] A virtual function call and a run-time class test take roughly the same number of instructions to implement. Why might you still want to insert run-time class tests to check common receiver classes, short-circuiting the general virtual function call?

The target methods for the common cases can be inlined if tested for explicitly, and the cost of the test can be amortized over many sends through code duplication, and the hardware supports branch prediction better than indirect call prediction.

c) [4] What information useful for optimization can be derived by examining the entire class hierarchy of a program?

If the compiler can determine that some variable holds an instance of class C or any of C's subclasses, through CHA the compiler can determine which methods are implemented for that variable. Messages sent to the variable can be statically bound if only one method is inherited by all these classes. This achieves the effect of non-virtual functions in C++, and more.

9) [2] Write a Cecil method specialized on lists to iterate over the list and return the `i`'th element for some argument `i`. You may assume the list has at least `i` elements.

```
method foo(l@:list[`T], i:int):T {
    let var which:int := 0;
    l.do(&(x:T){
        if(which = i, { ^ x });
        which := which + 1;
    });
    error("list is too short") }
```