CS 577 Modern Language Processors
Take-home Final Exam Due 5pm, Friday, March 19, 2004

This exam has 6 questions, each worth the same amount. **Answer any 5 questions, and only 5 questions.** Please write your answer to each question on a separate sheet of paper. Turn your exam in at my office or have it placed in my departmental mailbox.

The exam is open-book, open-notes; you can use any reference materials you wish. However, you must work independently, and you cannot share reference materials with other students.

1. **Interpreter Performance**

Consider the following Java function and its associated byte-code.

```
static void foo () {
  int[] arr = new int[100000];
  int k = 0;
  int j = 0;
  for (int i = 0; i < arr.length; i++)
    k+=arr[i]-j-j;
}
```

```
static void foo();
   Code:
    0:   bipush  100000
    2:   newarray int
    4:   astore_0
    5:   iconst_0
    6:   istore_1
    7:   iconst_0
    8:   istore_2
    9:   iconst_0
    10:  istore_3
    11:  iload_3
    12:  aload_0
    13:  arraylength
    14:  if_icmpge  33
    17:  iload_1
    18:  aload_0
    19:  iload_3
    20:  iaload
    21:  iload_2
    22:  isub
    23:  iload_2
    24:  isub
    25:  iadd
    26:  istore_1
    27:  iinc    3, 1
    30:  goto    11
    33:  return
```

Suppose we are running an interpreter using threaded code, in the style of Figure 2 of the assigned paper by Ertl and Gregg, and we wish to improve its performance by introducing (static) superinstructions, as described in that paper. Here are three possible superinstructions:

```
(a) iload_2_isub ≡
  iload_2
  isub

(b) iload_3_aload_0 ≡
  iload_3
  aload_0

(c) iconst_0_istore_1 ≡
  iconst_0
  istore_1
```

For *each* of these possibilities, explain whether or not using the superinstruction would be likely to improve performance, and why. In particular, explain the impact (if any) of each proposed superinstruction on branch-target-buffer prediction accuracy.

2. **Object Layout**

Consider the following Java code:

```
class A {
  int x,y;
  A(int x, int y) { this.x = x; this.y = y; }
}

void foo() {
  A[] arr = new A[10];
  for (int i = 0; i < arr.length; i++)
    arr[i] = new A(i,10*i);
}
```

(a) Draw a picture showing the memory layout of `arr` and its elements (including all headers) at the point just before `foo` returns, as it would appear in the AIX/Power-PC implementation of the Jalapeño system, according to the assigned paper by Alpern, *et al.*

(b) Under Linux/IA-32, reads from page 0 cause pointer faults whereas reads from very high memory addresses do not. In light of this, how should this memory layout be changed under the Linux/IA-32 implementation of Jalepeño (or Jikes, as it is now called).

3. **Garbage collection**

State whether each of the following assertions is true or false, and explain why.

(a) A pure reference counting collector is suitable for Java.

(b) Copying collectors are suitable to be the basis of conservative collectors.

(c) A copying collector is not suitable for an application that allocates heavily but has very little live data at any given point in execution.

(d) Generational collectors require some form of write barrier.

4. **Program Representations**

Consider the following code, written in the intermediate language (IL) described in lecture 12 (also in Appel's textbook).

```
     a <- 5
     b <- 10
     i <- 1
     goto L4
L1:  if i < a goto L2
     c <- i * 2
     goto L3
L2:  c <- i * 3
L3:  a <- a + c
     i <- i + 1
L4:  if i < b goto L1
     d <- a + b
```

(a) Identify the basic blocks and draw a control flow diagram using the basic blocks as the nodes.

(b) Number the basic blocks and draw a dominator tree for the procedure.

(c) Identify the dominance frontier of each block that contains one or more assignments.

(d) Put the procedure into SSA form (displayed as a control flow diagram).

(e) Put the procedure into Reference-Safe SSA form, as described in Section 2 of the assigned paper by Amme, *et al.*

5. **Optimizations**

Many program transformations used by compilers can be broadly classified into two categories.

1. *Control flow elimination* removes jumps, tests, and associated bookkeeping code, typically by replicating code. It typically makes the program faster, but larger (although subsequent application of other optimizations may shrink the program again). One example of such an optimization is loop unrolling, which usually increases program size, but removes the overhead of performing the loop test.

2. *Redundancy elimination* removes redundant computations. It typically make the program both smaller and faster. One example is hoisting invariant code out of loops.

Consider the performance results for the Swift compiler described in Table 2 of the assigned paper by Scales, et al. Based on these results, which category of transformations is more important for Swift? (Note: not all the optimizations described in this table fit into either category; you'll have to decide which ones do.)

6. **Verification**

Consider the five conditions that must be verified by the JVM before it begins executing bytecode, as described in section 2 (p. 4) of the assigned paper by Leroy.

(a) Which of these conditions are necessary to ensure run-time type safety? Explain.

(b) Of these safety-critical conditions, which would become irrelevant if SafeTSA were used instead of bytcode for transmitting Java programs? Explain. (Note: You may need to make some assumptions about SafeTSA — just be sure to state what these are.)