Computer Science Ph.D. Qualifier Examination

Spring 2011

Programming Languages and Compilers

Total : 100 points

Instructions

1. **You must answer each of the questions Q1, Q2, Q3 and Q4. In addition, choose any two out of the questions Q5, Q6 and Q7. Thus you will answer six questions in total.**
2. **Answers must be crisp, to the point and contain details of sufficient magnitude. Clear algorithms and pseudo-code must accompany answers wherever necessary to bring out the generality of the answer.**
3. **State assumptions as necessary**
4. **Questions that are open ended require discussion and answers should factor that in. Answers must bring out critical issues clearly in such cases.**
5. **Feel free to draw figures, flow graphs, etc. to bring out details in your answers.**
6. **This is a closed internet, closed book, closed notes examination. Georgia Tech honor code will be enforced.**

Q1 : Path based data-flow analysis **19 points (must answer)**

Infeasible paths in control flow graphs (CFGs) can cause heavy imprecision in dataflow analysis. Traditional data-flow analysis considers all paths in a CFG as feasible. That is it assumes that at runtime any one of those paths could be taken – whereas it is possible there exists a path in control flow graph which will never be taken; we call such paths as infeasible paths and that can cause imprecision in analysis.

(a) Show through example CFGs how reaching definition analysis and liveness analysis can become imprecise due to the assumption that all paths in CFG are feasible.

(b) Devise new techniques for improving the precision of reaching definition and liveness by formulating the problem as a path based data-flow problem.

(c) Comment on the fix-point reached and the precision vs. complexity of the technique. Show how the dataflow facts found improve using the technique over traditional methods.

Q2 : Partial deadness **19 points (must answer)**

A value is partially dead at a program point if there exists at least one path on which there is no use of that value before its redefinition. The goal of partial deadness is to first detect such values and then move their computation to remove the partial deadness.

(a) Develop a CFG example to show the cases of partial deadness – note that the definition is strict in terms of deadness – that is values that are totally dead are not to be included.

(b) Show how partial deadness can be removed by moving the computation. Consider placement issues. Also, comment on the safety issues of determining the partial deadness in terms of actual vs. found.

(c) Develop a data-flow analysis framework to detect partial deadness considering safety and precision issues in the presence of pointers.

(d) Develop a code motion framework which removes partial deadness (as a best effort) without introducing any redundancy. Hint : Use the concept of available expressions – you are moving the complete quad : t1 = x + y to remove deadness with respect to the use of t1

Q3. Parallelizing a loop. Consider the following loop (assume no aliasing of arrays). **19 points (must answer**

```
for i=1 to n do
    A[i] = A[i] / W[i];      /* (s1) */
        for j=i to n do
            X[i,j] = Y[i,j]*Y[i,j];    /* (s2) */
            Z[j] = Z[j] + X[i,j];      /* (s3) */
```

(a) Draw the program dependence graph (PDG) that relates the three statements, *s1*, *s2*, and *s3*.
(b) From the PDG, explain what opportunities for parallelism exist and give pseudocode for a parallel version of the loop. For your parallel pseudocode, assume a shared memory parallel programming model with a "**parfor**" (parallel-for) loop construct as well as basic synchronization primitives, such as locks or barriers.
(c) Give a high-level description of a compiler algorithm that can carry out this parallelization. That is, explain what program analysis and compiler transformations are necessary to successfully parallelize the loop.

Q4. Iteration spaces. Consider the following initialization loop. **19 points (must answer)**

```
for i=0 to 5 do
    for j=i to 7 do
        Z[i+1,j+1] = 0;
```

(a) Express the iteration space of this loop in the standard linear inequality form, $\mathbf{A}\cdot\mathbf{x} + \mathbf{b} \geq 0$, where x is the column vector $(\mathbf{i}, \mathbf{j})^T$ representing the iteration variables.
(b) Express the array reference, Z[i+1,j+1], in the affine form, $\mathbf{F}\cdot\mathbf{x} + \mathbf{f}$.

(c) Suppose the array is stored in column-major order. Rewrite the loop to improve its spatial locality.

Using your answers to parts (a) and (b), explain how a compiler can analyze the loop nest to produce your answer to (c).

Choose any two out of the following three questions (Q5, Q6, Q7)

Q5. Dependence Analysis **12 points**

(a) Show through examples how GCD test and Banerjee Inequalities are necessary but not sufficient conditions , ie, how they produce may dependence information and show code examples where in fact it is a case of independence ie, may dependence predicted by them is a false dependence.

(b) Show through examples how array dependence analysis system can be tightened by adding additional constraints to make it more accurate. You should use GCD and Banerjee Test as the underlying solvers for solving a system of dependence equations. That is illustrate that with additional information and analyses, more accurate (precise) answers can be sought. Show additional analyses that should be triggered in setting up the dependence system and weigh in on their usefulness.

Q6. Memory models **12 points**

In heterogeneous architectures, defining the memory space usage is one of the critical design decisions. For example, Intel's Larrabee programming model employs hybrid memory spaces. The memory space is broken into three sub-spaces: shared (shared by both CPUs and GPUs), a private space for CPUs and another private memory space for GPUs. Recently, NVIDA released that CUDA 4.0 that has a unified memory space for CPUs and GPUs. Discuss the pros and cons of unified memory spaces and Intel Larrabee's hybrid model. Discuss what type of compiler analysis and optimization may be necessary to decide which variables of a loop go into which memory space (shared, private for CPU and private for GPU); the loop is being accelerated on GPU.

Q7. Register allocation and instruction scheduling **12 points**

Phase ordering (which phase occurs before which one) of register allocation and instruction scheduling is an open issue in compiler research. Show an example where register allocation preceding instruction scheduling is helpful to generate better code; show another example where reverse is true. Propose an integrated scheme in which spilling and instruction scheduling decisions could be taken simultaneously. You may consider Briggs's type graph coloring register allocator and critical path based instruction scheduler.