Computer Science Ph.D. Qualifier Examination
Spring 2010
Programming Languages and Compilers

Notes:

1. You must answer all questions.
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.

1. On the CUDA programming model:
   a) Discuss when loop unrolling can be beneficial or harmful.
   b) Show at least two examples/reasons when/why loop unrolling can be useful in the CUDA programming model.
   c) Nvidia describes their execution model as SIMT (single instruction, multiple thread) instead of SIMD (single instruction, multiple data). Discuss what characteristics of the CUDA programming model suggest the name of SIMT.

2. SSA
   a) What are the pros and cons of using the SSA representation for compiler IR?
   b) Contrast the algorithm for common subexpression elimination in SSA and non-SSA form.
   c) Give an example of redundancy which dataflow analysis for CSE can not catch but which exists – how will you remedy this using SSA based representation?

3. Shape Analysis
   a) What does shape analysis tell you about a program?
   b) Give a pseudo-code example where shape analysis provides benefits above flow-sensitive points-to analysis.

4. Work Stealing
   a) The Cilk-5 language runtime uses a “provably good” work stealing algorithm to schedule threads. What is the heuristic behind its approach and what is the intuition behind its effectiveness.
   b) How do you think this algorithm will perform as chips scale to 100s or 1000s of cores? How would you augment it?

5. Data-flow analysis can be used to detect unsound program that have un-initialized variables. An un-initialized variable is the one which is used at least one program point before it is defined. Since garbage values can reside in un-initialized variables, it is desirable to detect them. An un-initialized variable can
propagate such values to other variables’ definitions that are derived from them and this can continue transitive. We call such a set of definitions as **unsound**. Our goal in this question is to devise a dataflow framework for detecting such variables assuming no prior reaching definition analysis has been done.

5.1 Define the problem of detecting all **un-initialized** and **unsound** definitions at the entry as well as exits of basic blocks through data-flow analysis. First work through examples and then define the data-flow equations to formulate it.

5.2 Devise a bit-vector algorithm based on the above data-flow equations.

5.3 Through a small example, show how the algorithm finds unsound as well as uninitialized variables.

5.4 Comment on how to minimize false positives or negatives in the presence of aliased variables.

6. Consider the following loop (note: there are no anti- nor output dependencies):

Assume there is one unit of each type (A, L, M, Branch) in the multiop (instruction 5 is of type Branch) and loop iterates for more than three times.

(a) Find the ResMII and RecMII for the loop

(b) Show the final software pipelined code. It is sufficient to write the number 1-5 to indicate the instruction. **Indicate where each of the “kernel,” “prolog,” and “epilog” begins and ends.** You may not need all rows in the table. Instruction 6 must go at the end of the kernel.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>L</th>
<th>M</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>