Prof. Dr. Sebastian Hack
Johannes Doerfert, B.Sc.

Compiler Construction WS15/16

Exercise Sheet 10

Exercise 10.1. Schedules

1. Describe the difference between a statement and a statement instance and why this differentiation is useful.
2. Write down the schedule for a 2D loop nest as a 2D matrix that describes a loop interchange.
3. Write down the schedule for a 2D loop nest as a 2D matrix that describes a loop reversal of the inner loop.
4. How can the two schedules be combined? How does the combined scheduling matrix look like and what effect will it have on a 2D loop nest.
5. Describe when a schedule is considered legal with regards to a dependence relation Γ that contains all pairs of statement instances which are dependent on each other.

Exercise 10.2. Dependences

```
   for (int i = 0; i < N; i++)
     for (int j = 2; j < N; j++)
S:     C[j][i] = a * C[j-1][i] + b * C[j-2][i];
   for (int i = 0; i < N; i++)
P:   Out[i] = C[N-1][i];
```

1. Write down the iteration spaces (aka domain) of the statements `S` and `P`. Use a constraint system or visualize it in a 2D coordinate system.
2. Describe the dependences for the example above using distance and direction vectors, the dependence depth as well as dependence polyhedra. Characterize the dependences (RAW/WAR/WAW).
3. Describe a legal and beneficial transformation that could be applied and write down the transformed loop nest. Argue why the transformation is beneficial.
4. Which dimension in the example above (after your transformation) can be vectorized? How would the (pseudo) code look like?

Project task G. Analysis and Optimizations

For this project assignment, we introduce an additional compiler switch namely `--optimize`. This switch performs the same operation as `--compile` and additionally performs optimizations.

Project task G.1. Sparse Conditional Constant Propagation (SCCP)

Implement SCCP as part of your compiler. It is required that it performs the following steps:

1. substitute instructions with constants,
2. substitute conditional branches with unconditional ones,
3. remove dead instructions and
4. remove unreachable blocks.

Remember that these steps should not be tackled separately but all at the same time as it will significantly increase the power of your optimization. Additionally, you can replace the basis of the constant propagation part by a more complex technique, e.g., an interval analysis or pentagon analysis.

In the final evolution of your project, your analyses and optimizations will be manually inspected and executed on our test cases. Your final grade is impacted by the effectiveness of your analyses and optimizations. Hence, you are encouraged to implement other techniques on top of the required ones.

Note: If you do not want to implement SSA construction yourself you can use the LLVM *mem2reg* pass[1] but *no other* LLVM passes.

[1] See `llvm::createPromoteMemoryToRegisterPass()`