Project 4: Loop Cost
CS 5130
Robbie Watling

Introduction
The project calculates the loop cost for a series of loop nests in LLVM. First, loop induction instructions are identified based on “getCanonicalLoopInductionVariable” function that has been trimmed down for this application. For each loop nest, references are examined in the inner most loop and are used to construct reference groups.

We detect if there is reuse between two difference references, either spatial or temporal. To detect group spatial reuse, all subscripts besides the last must be the same. Then in the last subscript of each of the references, the stride must be less than the cache line size. To detect group temporal reuse, a dependence from the function “depends” is examined between two references. The dependence yields a distance. If it is less than a maximum distance of 2 in our implementation there is temporal reuse. If there is group reuse, then the references belong to the same reference group and the first reference is determined to be the leader or representative. The reference groups are then printed after all references are analyzed.

Then the loop cost is calculated for each loop. In each loop, the product of the trip counts is calculated and that is multiplied by the reference group cost to yield the loop cost for a reference group. The reference group cost is calculated by taking the reference group leader and calculating the reference cost.

The reference cost first considers if the reference is invariant across all loop levels. If it is, the reference cost is 1. Otherwise, we calculate the trip count. If the references are consecutive (within the same cache line) then the reference cost becomes the produce of the trip count and the stride, divided by the cache line size. If the are not consecutive, the the reference cost is the trip count. This is then returned and printed to the console.

This approach to calculating the loop cost is largely based off of “LoopCacheAnalysis” in LLVM. Initially, I was hoping to just call several of the functions in this file but there were some translation issues. Most of the implementation was easy to follow, but figuring out the distances was problematic. Therefore, for much of this implementation, the subscript iterations were adapted from the patch in the previous project. Some creative liberty was taken with just including these in the “DependenceInfo” class as it already observes many of the passes necessary to calculate reference information. Ultimately the “LoopCacheAnalysis” was helpful for enhancing my understanding of loop costs and re-implementing the functions from “LoopCacheAnalysis” with the patch helped me keep track of details well.

Function Descriptions
bool hasSpatialReuse(Instruction* Rep, Instruction* Other)
- Returns true if there is group spatial reuse

bool hasTemporalReuse(Instruction* Rep, Instruction* Other)
- Returns true if there is group temporal reuse

bool isCoeffForLoopZeroOrInvariant(const SCEV* Subscript, const Loop &L)
- Returns true if if the coefficient is 0 or invariant

bool checkLoopInvariant(Instruction* Src, const Loop &L)
- Returns true if the reference is invariant across all loop levels

bool checkConsecutive(Instruction* Src, const Loop &L)

- Returns true if the reference falls within the same cache line size upon each access

static bool populateReferenceGroups(MyReferenceGroupsTy &RefGroups, Loop* Loop, DependenceInfo* DI)

- Calculates the reference groups

static CacheCostTy computeLoopCost(const Loop &L, const MyReferenceGroupsTy &RefGroups, const SmallVector<LoopTripCountTy, 3> &TripCounts, DependenceInfo* DI)

- Calculates the overall loop cost given a loop and reference groups. Calls computeRefGroupCost()

static CacheCostTy computeRefGroupCost(const MyRefGroupTy &RG, const Loop &L, DependenceInfo* DI)

- Calculates the cost of a reference group by calling computeRefCost

CacheCostTy computeRefCost(Instruction* Representative, const Loop &L, DependenceInfo* DI)

- Checks for self-reuse and returns the corresponding cost