Loop Fusion, Loop Distribution and Their Place in the Loop Optimization Pipeline

April 9, 2019

Kit Barton, IBM Canada
Johannes Doerfert, Argonne National Labs
Hal Finkel, Argonne National Labs
Michael Kruse, Argonne National Labs

Agenda

Loop Fusion Current Status

Loop Distribution

Data Dependence Graph

Loop Optimization Pipeline

Next Steps

Loop Fusion

Combine two (or more) loops into a single loop

```
for (int i=0; i < N; ++i) {
  A[i] = i;
}
for (int j=0; j < N; ++j) {
  B[j] = j;
}
```

```
for (int i=0, j=0; i < N && j < N; ++i,++j) {
  A[i] = i;
  B[j] = j;
}
```

Motivation

– Data reuse, parallelism, minimizing bandwidth, …
– Increase scope for loop optimizations

Our Goals

1. Way to learn how to implement a loop optimization in LLVM
2. Starting point for establishing a loop optimization pipeline in LLVM

Requirements

In order for two loops, L_j and L_k to be fused, they must satisfy the following conditions:

1. *L_j and L_k must be adjacent*
2. *L_j and L_k must iterate the same number of times*
3. *L_j and L_k must be control flow equivalent*
4. *There cannot be any negative distance dependencies between L_j and L_k*

Loop Fusion – Current Status

Initial patch for *Basic Loop Fusion:* https://reviews.llvm.org/D55851

Approved, but waiting confirmation of one remaining review comments

Improvements to *basic loop fusion* currently under development

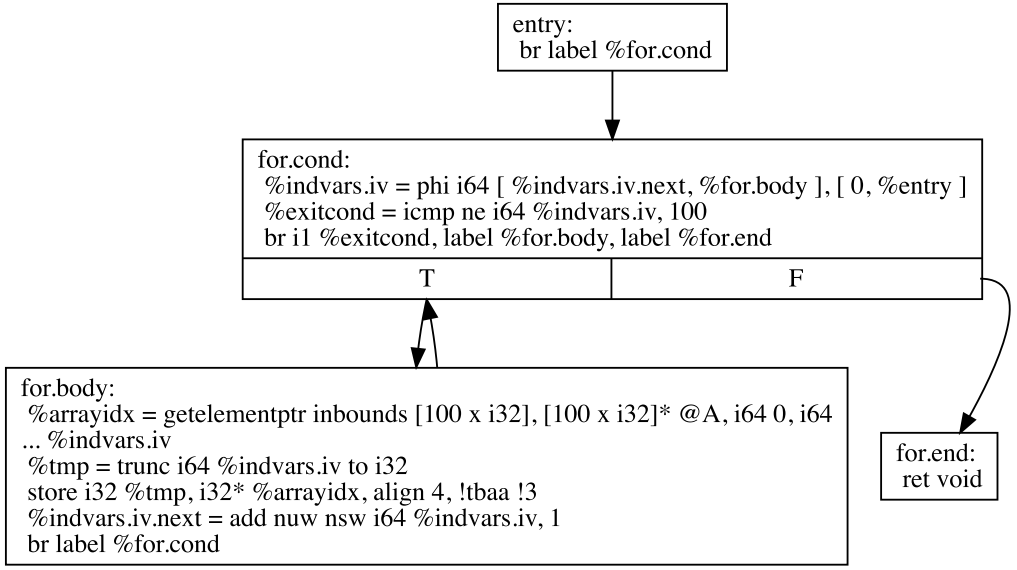
1. Require rotated loops
2. Handling of guarded loops
3. Merging latch blocks during fusion

Loop Rotation

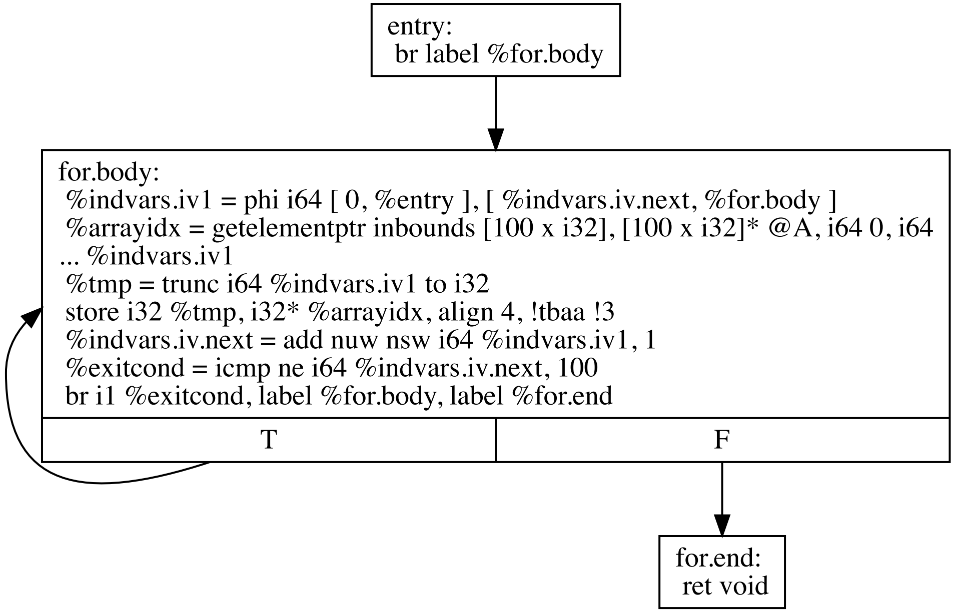
```
int A[100];
void example() {
  for (int i = 0; i < 100; ++i)
    A[i] = i;
}
```

Convert a loop into a do/while style loop

Canonicalize loop latch to have a single successor



CFG for 'example' function



CFG for 'example' function

Motivation

- Canonicalize loop latch to have a single successor
- Makes analysis for loop fusion easier because loop structure is canonical
- Makes mechanics of fusing loops easier because the *latch* and *exiting* blocks are the same

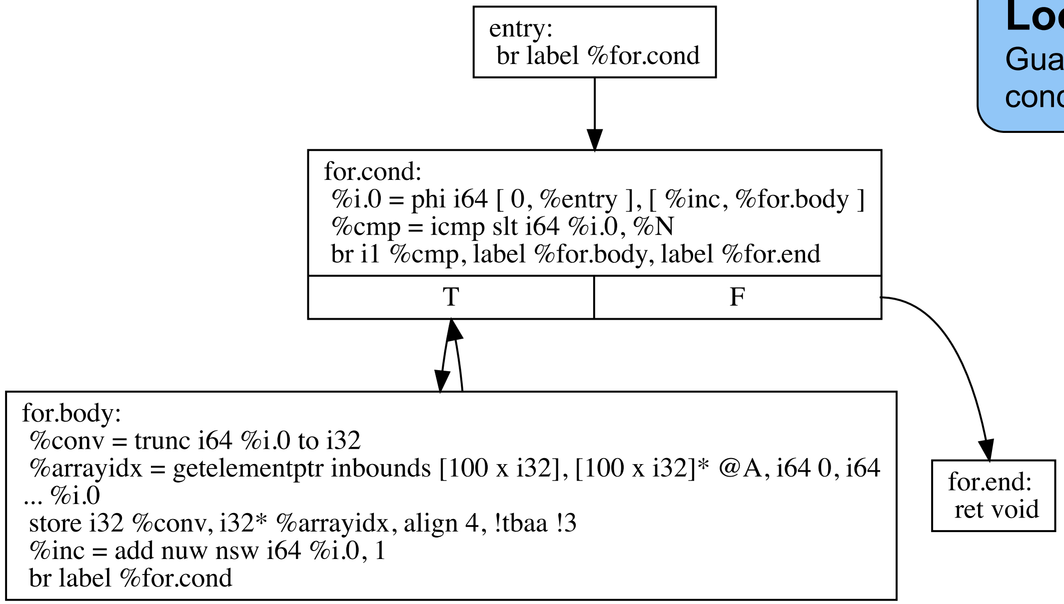
https://reviews.llvm.org/D22630

Loop Rotation – Guarded Loops

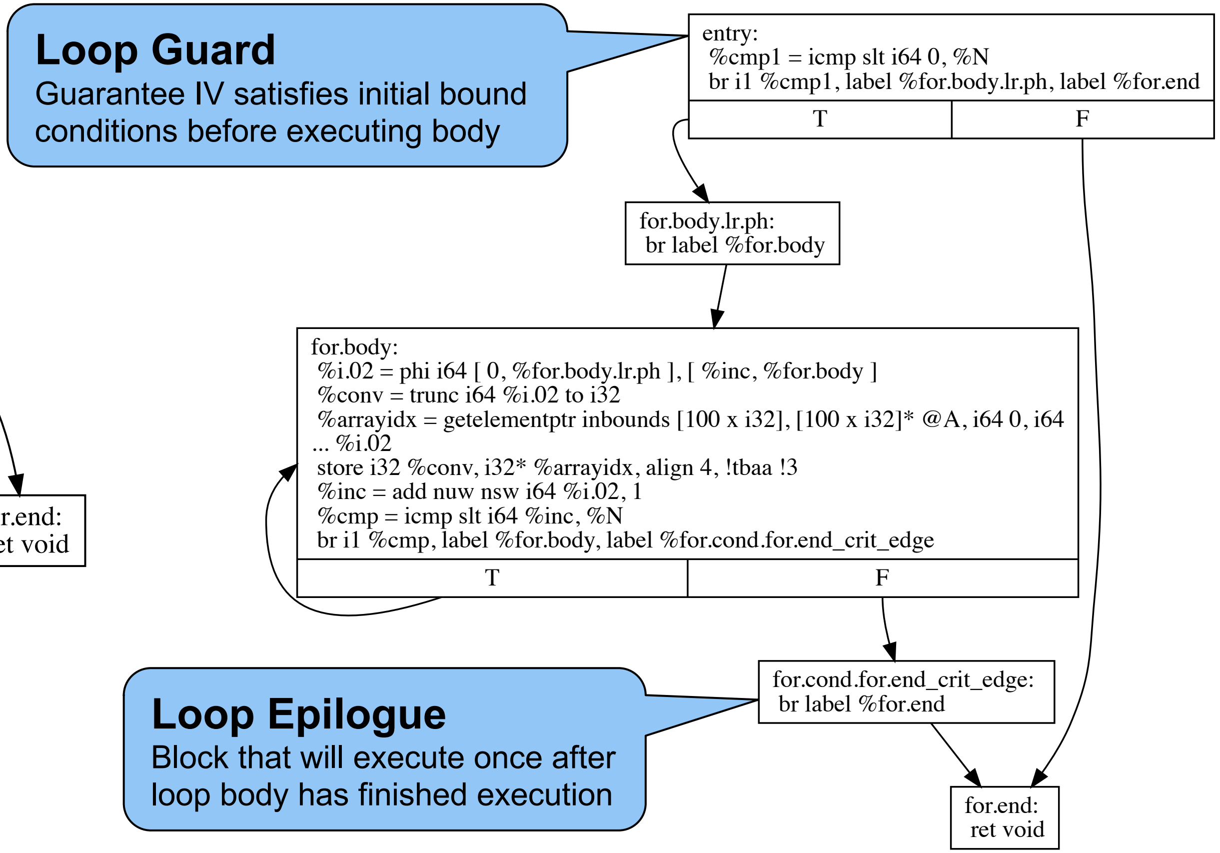
```
int A[100];
void example(long N) {
  for (int i = 0; i < N; ++i)
    A[i] = i;
}
```

When compiler cannot prove loop body will execute at least once, it inserts a *guard*

It also inserts a loop epilogue, that will be executed once after the body is finished executing



CFG for 'example' function



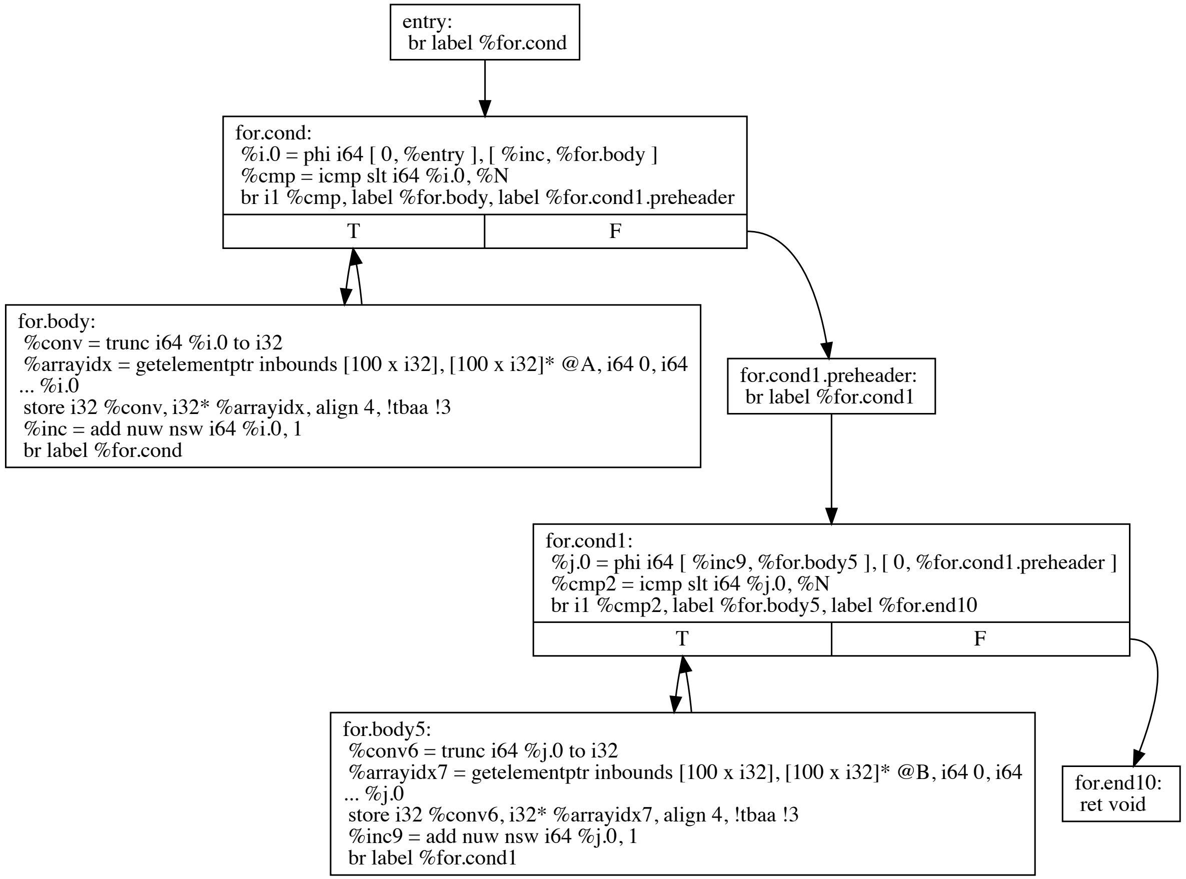
CFG for 'example' function

Loop Rotation – complications for loop fusion

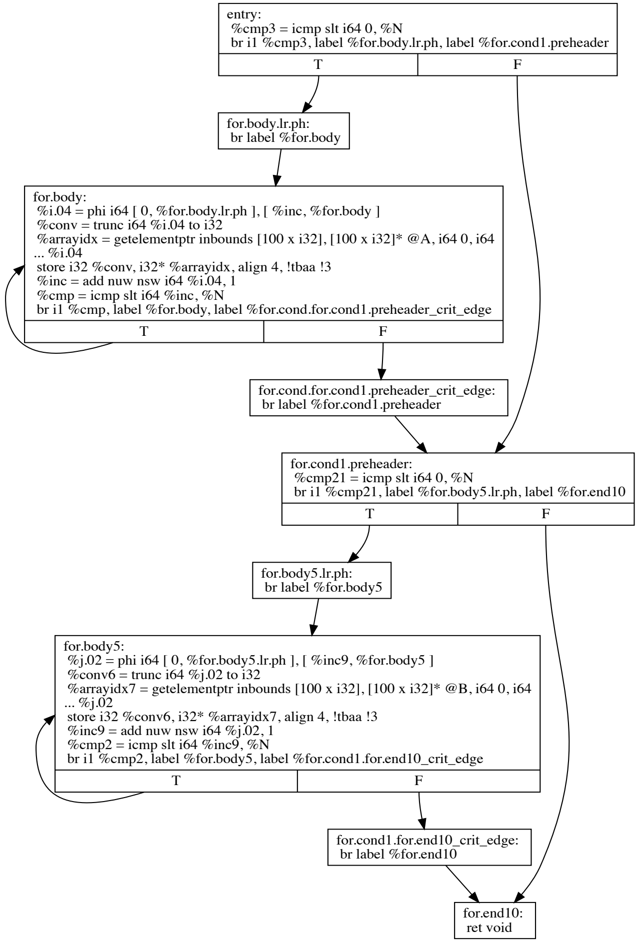
```
int A[100];
int B[100];
void example(long N) {
  for (long i = 0; i < N; ++i)
    A[i] = i;
  for (long j = 0; j < N; ++j)
    B[j] = j;
}
```

Loop guards cause loop preheaders to be no longer control flow equivalent

Loop epilogue cause loops to no longer be adjacent



CFG for 'example' function



CFG for 'example' function

Loop rotate summary

We want to focus on fusing rotated loops only

- Work on a canonical form of loops makes the implementation for fusion simpler

In some cases, loop rotate is creating a guard block

- Necessary because it is creating a do loop
- If it cannot prove the loop should execute at least once iteration, it needs a guard at the beginning
- Guard block will make loop preheaders not control flow equivalent, meaning they cannot be fused

When a guard block is created, a loop epilogue is also created

- Provides a location to sink statements that only need to be run once, after the loop body finishes
- Epilogue block makes loops not adjacent (temporary limitation of fusion)

Running SimplifyCFG after loop rotate will cleanup (empty) epilogue block, but not (valid) guard

Possible solutions for loop fusion

Make guard block and epilogue block part of the canonical structure of loops

- In cases where no guard is necessary, add a "trivial" guard block (*i.e.*, `if (1) { }`)
- Modify LoopSimplify to always ensure a guard block is present
  - Add similar interfaces for other components to allow getLoopGuard, *etc*.
- Modify Loop Fusion to use guard block for control flow equivalence checks and adjacency check

Modify LoopFusion to handle cases for guarded loops and non-guarded loops separately

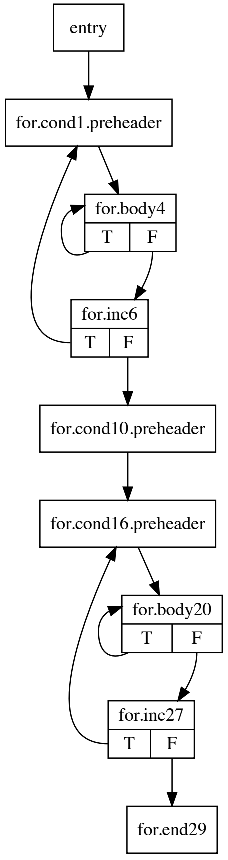
- If a guard is present:
  1. Use the guard block for the control flow equivalence checks
  2. Use guard block successors for adjacency check (if successor of a loop guard is another loop guard, loops are adjacent)
  3. Check if guard blocks have same branch and can be merged
- If no guard is present:
  1. Use preheader for control flow equivalence checks (done today)
  2. Use exit block to check for adjacency (if exit block of loop is another loop preheader/guard, loops are adjacent)

Loop Fusion – merging latches

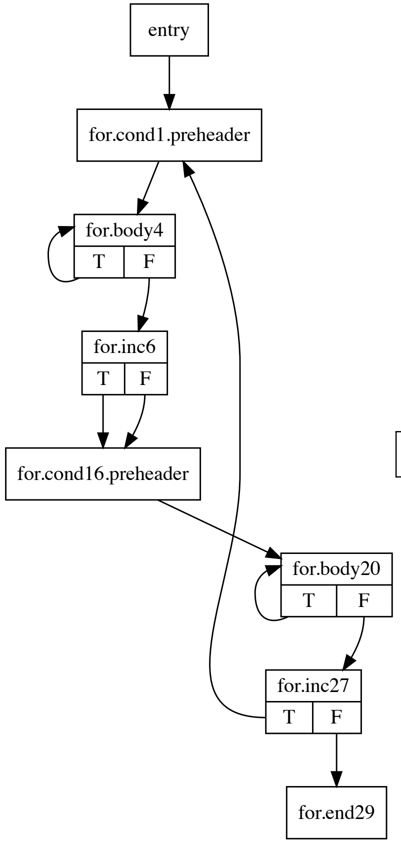
```
int A[100][100];
int B[100][100];
void example() {
  for (long i = 0; i < 100; ++i)
    for (long j = 0; j < 100; ++j)
      A[i][j] = i+j;
  for (long i = 0; i < 100; ++i)
    for (long j = 0; j < 100; ++j)
      B[i][j] = i-j;
}
```

Current implementation of fusion simply changes edges in CFG when fusing:

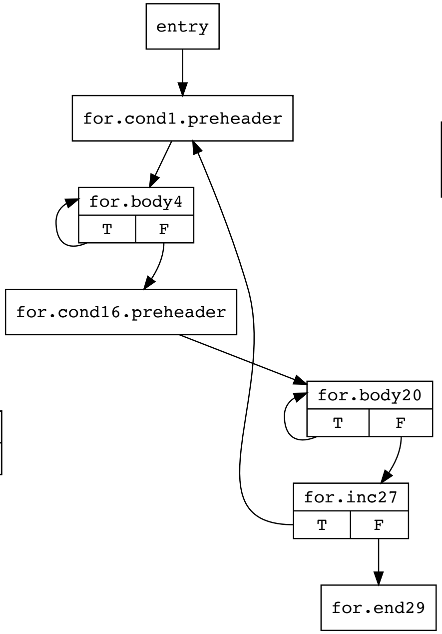
- This prevents fusion in nested loops because inner loops are not adjacent after fusing outer loops
- This can be improved by actually merging the blocks from the two loops
- Allows for subsequent fusion of nested loops because inner loops are now control flow equivalent
- **Only implemented for rotated loops!**



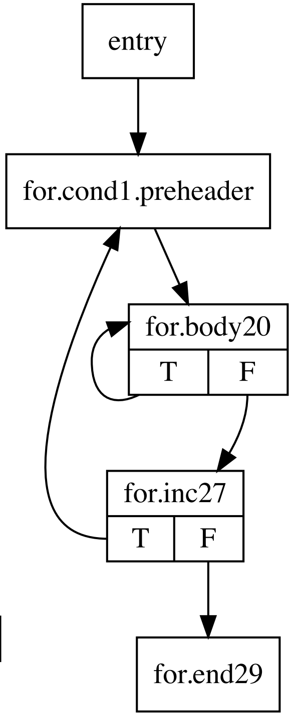
CFG for 'example' function



CFG for 'example' function



CFG for 'example' function



CFG for 'example' function

Number of Loops Fused

SPEC 2017 Basic Fusion

Benchmark	Candidates for Fusion	Loops Fused
perlbench_r	487	0
gcc_r	2019	2
namd_r	1393	0
parest_r	9385	1
povray_r	294	0
lbm_r	10	1
omnetpp_r	198	0
x264_r	797	0
blender_r	4190	6
deepsjeng_r	99	13
imagick_r	2710	2
nab_r	73	0
xz_r	70	0

SPEC 2017 Basic Fusion + Merge Latches

Benchmark	Candidates for Fusion	Loops Fused
perlbench_r	479	0
gcc_r	1930	1
namd_r	1393	0
parest_r	9349	1
povray_r	287	0
lbm_r	10	1
omnetpp_r	195	0
x264_r	765	0
blender_r	4079	6
deepsjeng_r	99	13
imagick_r	2700	2
nab_r	73	0
xz_r	69	0

SPEC 2017 Basic Fusion + Merge Latches + Guard Handling

Benchmark	Candidates for Fusion	Loops Fused
perlbench_r	480	0
gcc_r	1937	4
namd_r	1393	0
parest_r	9349	1
povray_r	286	0
lbm_r	10	1
omnetpp_r	195	0
x264_r	765	1
blender_r	4078	5
deepsjeng_r	99	14
imagick_r	2700	3
nab_r	74	0
xz_r	69	1

Loop Distribution

Separate a loop nest into two (or more) loop nests

```
for (int i=0; i < N; ++i) {
  A[i] = B[i] * r;
  for (int j=1; j < N; ++j) {
    C[i][j] = D[i][j-1] / A[i];
    E[i][j] = E[i][j] * A[i];
  }
}
```

```
for (int i=0; i < N; ++i)
  A[i] = B[i] * r;
for (int i=0; i < N; ++i)
  for (int j=1; j < N; ++j) {
    C[i][j] = D[i][j-1] / A[i];
    E[i][j] = E[i][j] * A[i];
  }
```

Motivation

- Data reuse, parallelism, minimizing bandwidth, …
- Improve effectiveness of subsequent loop optimizations
  - Creating perfect nests, removing loop carried dependencies, *etc*

Our Goals

1. Continue to improve and strengthen loop optimizations in LLVM
2. Create common infrastructure and utilities that can be used for many loop optimizations

Current implementation of loop distribution in LLVM

Provides mechanics of distributing *inner* loops

Focuses entirely on distributing loops for vectorization

- Uses the LoopAccessInfo classes, which were developed for loop vectorization
- Only tries to distribute inner loops

Added in the pass pipeline but not enabled by default

Heuristic to determine how to distribute loops

We are creating heuristics that can be used to determine how to distribute a loop nest

Heuristics are based on the heuristics used in IBM's XL Compilers

Use two key data structures not currently available in LLVM:

- Data Dependence Graph
- *Affinity* Graph

Data Dependence Graph

Directed multigraph that represents data dependencies between statements

- Nodes correspond to either a single statement or a group of statements
- Edges represent data dependencies between nodes

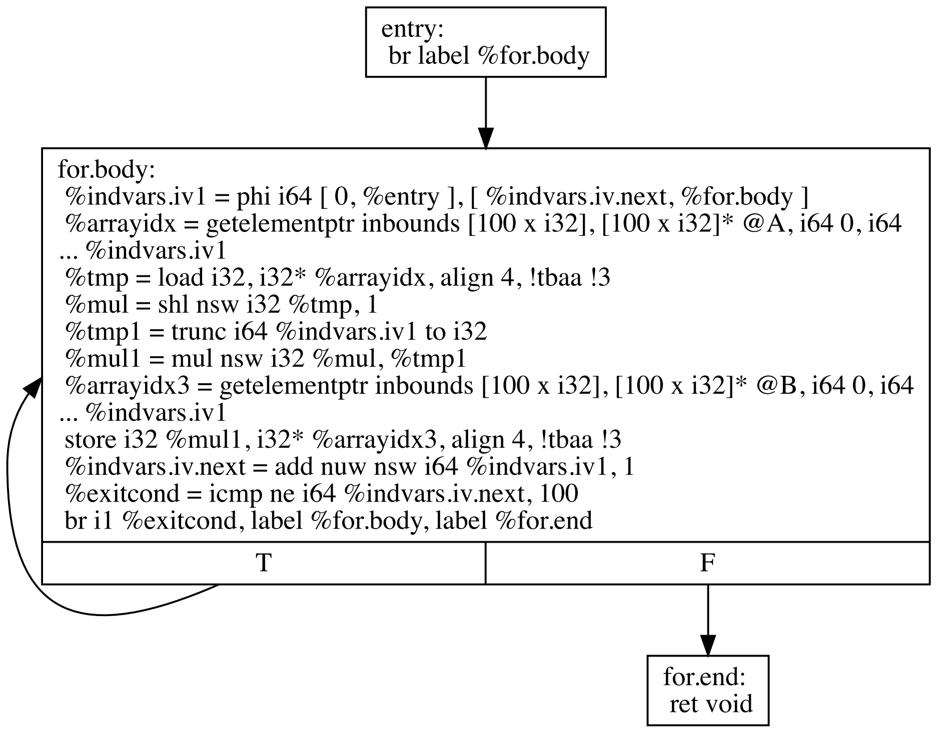
A directed edge from node N_i to N_j represents a data dependency from N_i to N_j

Edges can have attributes that represent the type of data dependence and a distance/direction vector

Data Dependence Graph Example - WIP

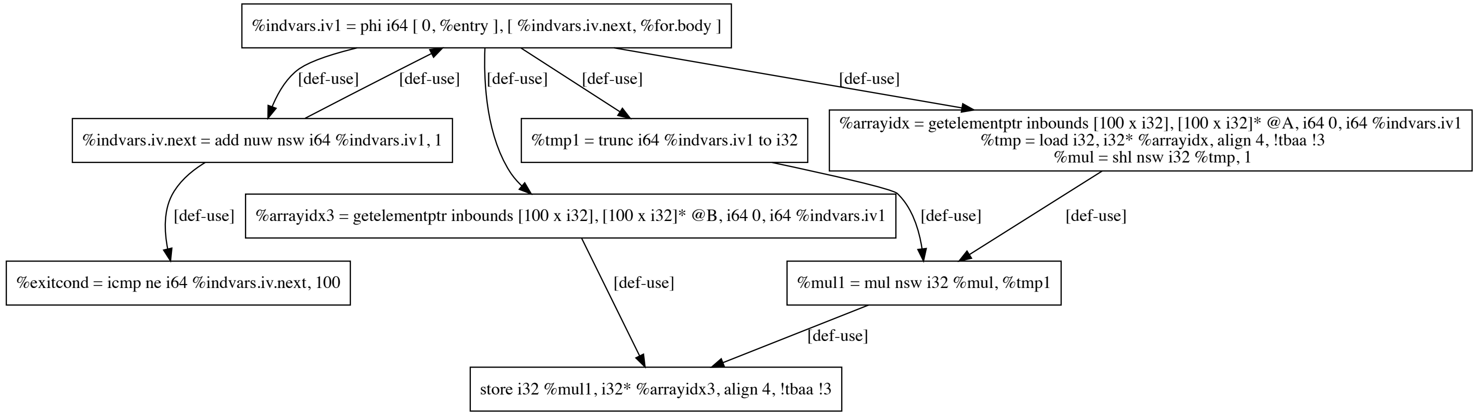
```
int A[100];
int B[100];
void example() {
  for (int i = 0; i < 100; ++i)
    B[i] = A[i] * 2 * i;
}
```

CFG for example



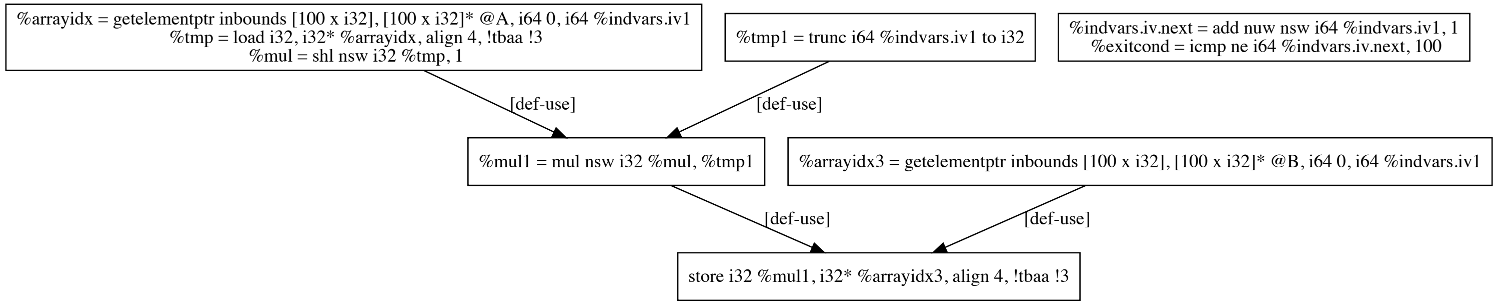
CFG for 'example' function

Initial DDG



DDG for 'example.for.body'

Simplified DDG



DDG for 'example.for.body'

Affinity Graph

Undirected weighted graph

- Nodes correspond to strongly connected components in the DDG
- Edges represent *affinity* between the nodes

Nodes in the graph correspond to strongly connected components in the DDG

- Nodes also have characteristics that are relevant to loop distribution including:
  - –Platform specific metrics such as register requirements, functional unit requirements and prefetchable data streams
  - –Also indicates self dependence for parallelization and vectorization

Edges represent affinity between the nodes

- Currently only measure of affinity is data reuse

How to distribute loops

Use greedy algorithm to distribute nodes in the affinity graph

Nodes are gathered in increasing order of desirability

Decision about grouping nodes is based on:

1. affinity graph
2. data dependencies
3. desirability based on node attributes

   If grouping nodes together exceeds platform-specific threshold or adds data dependencies then nodes should not be grouped together

Placing loop fusion and distribution in the loop opt pipeline

Fuse early, primarily to create opportunities for other loop optimizations

- Use loop rotate to create (guarded) do loops
- Rely on Loop Simplify to put loops in canonical form

Run loop distribution later, after fusion

- Do we want to rely on distribution “undoing” decisions made by fusion?

However, since both fusion and distribution can be run for multiple criteria, maybe want to run them multiple times

1. Run early as an optimization-enable pass
2. Run late to make platform-specific optimization decisions (with target-specific overrides)

Next Steps

Loop Fusion

- Converge on direction for loop rotation and guarded loops
- Move intervening code from between loops to make them adjacent
- Improving dependence analysis
- Placing loop fusion in the pipeline and enable by default

Loop Distribution

- Post patches for DDG
- Post patches for initial loop distribution
- Discuss interface for affinity graph