Loop Versioning For LICM

Ashutosh Nema

AMD

Agenda

- Background
- Motivation
- Overview

- Design & Implementation
- Example
- Current Status & Results

- Challenges
- Acknowledgement

Background

- Loop invariant code motion (LICM) is an important compiler optimization
- For safety, it considers the memory dependencies arising out of aliasing before moving an invariant out of loop
- may-aliases can make this optimization ineffective
    - Results in possible missed opportunities for LICM

Background

Consider below ‘C’ test:

```
int foo(int *arr1, int *arr2, int len) {
  for (int i = 0; i < len; i++) {
    for (int j = 0; j < len; j++) {
      arr1[i] = arr2[j] + arr1[i];
    }
  }
}
```

Alias analysis is unsure about memory ‘arr1’ & ‘arr2’, as these are input to ‘foo’.
It becomes conservative here and marks ‘arr1’ & ‘arr2’ as may-alias memory.

Access to ‘arr1’ is an inner loop invariant.

This uncertainty about memory aliasing results in missed opportunities by LICM.

Background

- Possible solution for LICM to exploit such missed opportunity is by carrying out better alias analysis
  - The quality of this solution may still not be good enough to capture all cases of interest

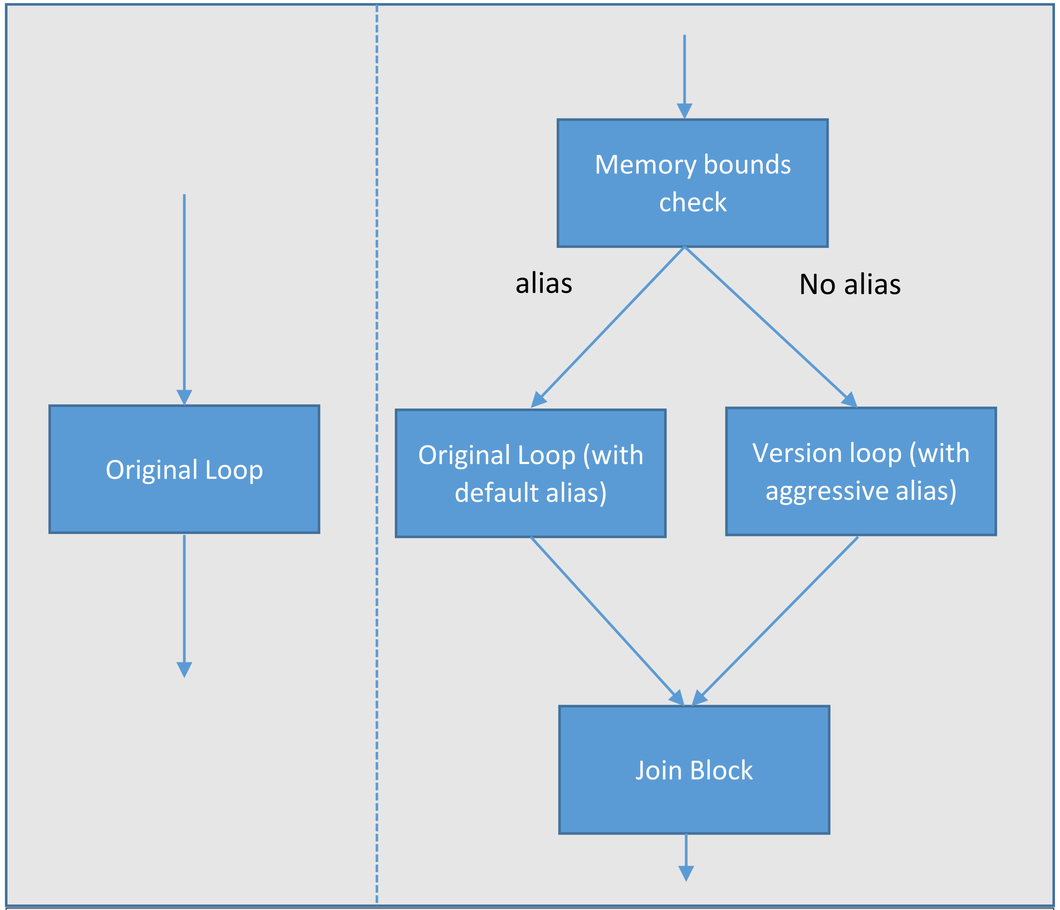
- The alternate solution is LoopVersioning LICM, where the aliasing decision is made at runtime

Motivation

- LoopVersioning LICM is a step to exploit those missed opportunities where memory aliasing (may-alias) makes LICM optimization ineffective

Overview

- LoopVersioningLICM creates two versions of a targeted loop L – one with aggressive alias and the other with conservative (default) alias
- Aggressive alias version of L (AL) has all the memory accesses marked as no-alias
- Conservative alias version of L (CL) carries the default conservative alias
- The two versions of loop L is preceded by a runtime alias check
    - Uses bound checks for all unique memory accessed in loop
- If the runtime check asserts there is 'noalias' then AL gets executed, else CL gets executed



A Loop before and after LoopVersioningLICM

Design & Implementation

Legality

- Loop Structure Legality
- Instruction Legality
- Memory Legality

Costing

- Invariant Benefit

Transform

- Memory Bounds Check
- LoopVersioning

Legality

During legality consider following:

- loop structure
  - ensures the layout of loop is adequate for LoopVersioningLICM
- memory accesses
  - ensures memory dependencies in the loop are proper for LoopVersioning
- loop instruction
  - ensures the instructions in loop are good for LoopVersioning

```
bool LoopVersioningLICM::loopStructureLegality() {
  // 1. Loop should have a pre header.
  // 2. Loop should be inner most.
  // 3. Loop should not have multiple back edge.
  // 4. Loop should have single exit block.
  // 5. Loop depth should be under LoopDepthThreshold.
  // 6. Loop should have a trip count
}

bool LoopVersioningLICM::loopMemoryLegality() {
  // 1. Check LoopAccessAnalysis for memory safety.
  // 2. Check memory alias dependencies
}

bool LoopVersioningLICM::loopInstructionsLegality() {
  // 1. Loop should not be read only.
  // 2. Loop should have possible invariant instructions.
  // 3. Safely handle call instructions.
  // 4. Make sure loop should not have possiblity of exception.
}
```

Costing

During costing, consider the following for a loop (L):

Identify all possible invariants in loop (L) and make sure the their percentage is above an ***InvariantThreshold***. If its less, then do not consider loop (L) for LoopVersioning LICM

Default InvariantThreshold is 25%, and it can be overwritten by a command line option

Total address/pointers for memcheck should be below ***RuntimeMemoryCheck*** Threshold. Default is 8, and it can be overwritten by a command line option

```
bool LoopVersioningLICM::isBeneficialForVersioning() {
  // 1. Compare possible invariant percentage with invariant threshold.
  //    If its less then ignore this loop.
  // 2. Total address/pointers for memcheck should below
  //    RuntimeMemoryCheckThreshold.
}
```

Transformation

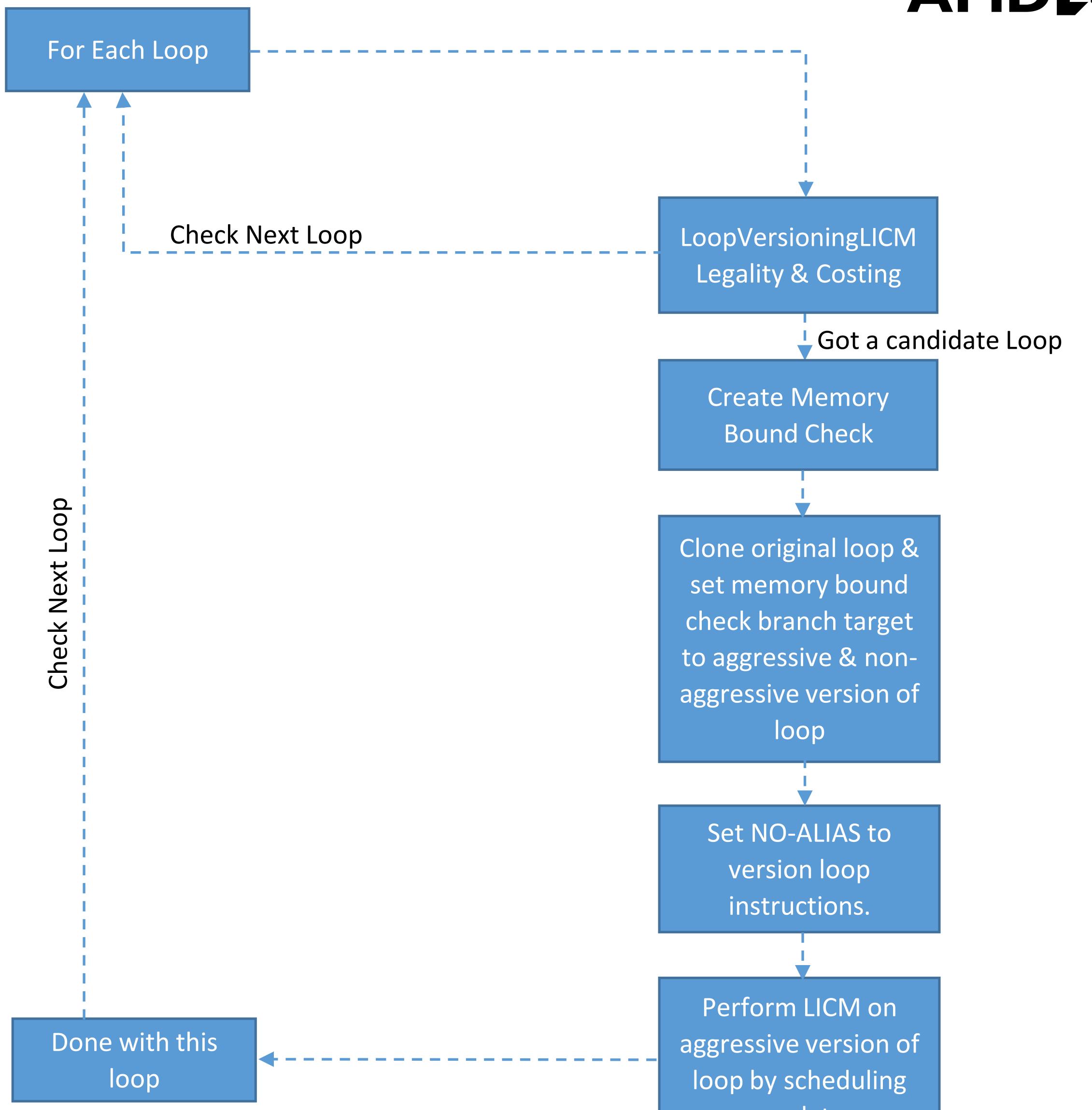
- Implemented as loop Pass
- Files Added: lib/Transforms/Scalar/LoopVersioningLICM.cpp
- Option to enable this feature: -loop-versioning-licm

```
bool LoopVersioningLICM::runOnLoop(Loop *L, LPPassManager &LPM) {
  if (isLegalForVersioning()) {
    // 1. Create runtime bound check & a new version of Loop, by cloning original.
    // 2. Update PHI nodes for the values those used outside.
    // 3. Set metedata in both loop, for later identification.
    // 4. Set no-alias to instructions of aggressive alias version of loop.
  }
}

bool LoopVersioningLICM::isLegalForVersioning() {
  // 1. loopStructureLegality()
  // 2. loopInstructionsLegality()
  // 3. loopMemoryLegality()
}
```

Implementation Details

- Perform loop Legality and Costing check and confirm that the loop is a candidate for loop multi-versioning
- If the loop is a candidate for versioning then create a memory bounds check, by considering all the unique memory accesses in the loop body
- Clone the original loop and set all memory access as no-alias in the new loop
- Set original and versioned loops as branch targets of runtime check result
- Perform loop invariant code motion on newly generated aggressive alias version of loop by scheduling LICM pass later



Design: Loop Versioning for invariant code motion

Example

Consider below case:

```
3   for(; i < itr; i++) {
4     for(; j < itr; j++) {
5       var1[j] = itr + i;
6       var2[i] = var1[j] + var2[i];
7     }
8   }
```

```
3   for(; i < itr; i++) {
      # load var2[i] to register
      %8 = load i32* %var2[i]
4     for(; j < itr; j++) {
5       var1[j] = itr + i;
6       %8 = var1[j] + %8;
7     }
      # Store register value to var2[i]
      store i32 %8, i32* %var2[i]
8   }
```

Line #6 has load & store for ‘var2[i]’, and it’s a inner loop invariant.
Alias dependencies between ‘var1’ & ‘var2’ restrict LICM to perform invariant code motion.

LoopVersioningLICM helps here to move invariant out of loop.

Post LoopVersioningLICM

```
for.body.3.lver.memcheck:                     ; preds = %for.cond.1.preheader
  %add = add nsw i32 %i.027, %itr
  %arrayidx5 = getelementptr inbounds i32, i32* %var2, i32 %i.027
  %scevgep = getelementptr i32, i32* %var1, i32 %j.028
  %bound0 = icmp ule i32* %scevgep, %arrayidx5
  %bound1 = icmp ule i32* %arrayidx5, %scevgep31
  %memcheck.conflict = and i1 %bound0, %bound1
  br i1 %memcheck.conflict, label %for.body.3.lver.orig.preheader, label
%for.body.3.ph
```

for.body.3.lver.orig:

for.body.3:

Post LoopVersioningLICM

```
for.body.3.lver.memcheck:                    ; preds = %for.cond.1.preheader
br i1 %memcheck.conflict, label %for.body.3.lver.orig.preheader, label
%for.body.3.ph
```

```
for.body.3:
```

```
for.body.3.lver.orig:
  %j.125.lver.orig = phi i32 [ %inc.lver.orig, %for.body.3.lver.orig ], [ %j.028, %for.body.3.lver.orig.preheader ]
  %arrayidx.lver.orig = getelementptr inbounds i32, i32* %var1, i32 %j.125.lver.orig
  store i32 %add, i32* %arrayidx.lver.orig, align 4, !tbaa !1
  %1 = load i32, i32* %arrayidx5, align 4, !tbaa !1
  %add6.lver.orig = add nsw i32 %1, %add
  store i32 %add6.lver.orig, i32* %arrayidx5, align 4, !tbaa !1
  %inc.lver.orig = add nsw i32 %j.125.lver.orig, 1
  %exitcond.lver.orig = icmp eq i32 %inc.lver.orig, %itr
  br i1 %exitcond.lver.orig, label %for.inc.8.loopexit, label %for.body.3.lver.orig
```

Post LoopVersioningLICM

```
for.body.3.lver.memcheck:                        ; preds = %for.cond.1.preheader
br i1 %memcheck.conflict, label %for.body.3.lver.orig.preheader, label
%for.body.3.ph
```

for.body.3.lver.orig:

```
for.body.3:
  %add635 = phi i32 [ %arrayidx5.promoted, %for.body.3.ph ], [ %add6, %for.body.3 ]
  %j.125 = phi i32 [ %j.028, %for.body.3.ph ], [ %inc, %for.body.3 ]
  %arrayidx = getelementptr inbounds i32, i32* %var1, i32 %j.125
  store i32 %add, i32* %arrayidx, align 4, !tbaa !1, !alias.scope !10, !noalias !10
  %add6 = add nsw i32 %add635, %add
  %inc = add nsw i32 %j.125, 1
  %exitcond = icmp eq i32 %inc, %itr
  br i1 %exitcond, label %for.cond.1.for.inc.8_crit_edge.loopexit34, label %for.body.3

for.cond.1.for.inc.8_crit_edge.loopexit34:       ; preds = %for.body.3
  %add6.lcssa = phi i32 [ %add6, %for.body.3 ]
  store i32 %add6.lcssa, i32* %arrayidx5, align 4, !tbaa !1, !alias.scope !6, !noalias !9
  br label %for.inc.8
```

Post LoopVersioningLICM

```
for.body.3.lver.memcheck:                    ; preds = %for.cond.1.preheader
  %add = add nsw i32 %i.027, %itr
  %arrayidx5 = getelementptr inbounds i32, i32* %var2, i32 %i.027
  %scevgep = getelementptr i32, i32* %var1, i32 %j.028
  %bound0 = icmp ule i32* %scevgep, %arrayidx5
  %bound1 = icmp ule i32* %arrayidx5, %scevgep31
  %memcheck.conflict = and i1 %bound0, %bound1
  br i1 %memcheck.conflict, label %for.body.3.lver.orig.preheader, label
%for.body.3.ph
```

```
for.body.3.lver.orig:
  %j.125.lver.orig = phi i32 [ %inc.lver.orig, %for.body.3.lver.orig ], [
%j.028, %for.body.3.lver.orig.preheader ]
  %arrayidx.lver.orig = getelementptr inbounds i32, i32* %var1, i32
%j.125.lver.orig
  store i32 %add, i32* %arrayidx.lver.orig, align 4, !tbaa !1
  %1 = load i32, i32* %arrayidx5, align 4, !tbaa !1
  %add6.lver.orig = add nsw i32 %1, %add
  store i32 %add6.lver.orig, i32* %arrayidx5, align 4, !tbaa !1
  %inc.lver.orig = add nsw i32 %j.125.lver.orig, 1
  %exitcond.lver.orig = icmp eq i32 %inc.lver.orig, %itr
  br i1 %exitcond.lver.orig, label %for.inc.8.loopexit, label
%for.body.3.lver.orig
```

```
for.body.3:
  %add635 = phi i32 [ %arrayidx5.promoted, %for.body.3.ph ], [ %add6,
%for.body.3 ]
  %j.125 = phi i32 [ %j.028, %for.body.3.ph ], [ %inc, %for.body.3 ]
  %arrayidx = getelementptr inbounds i32, i32* %var1, i32 %j.125
  store i32 %add, i32* %arrayidx, align 4, !tbaa !1, !alias.scope !10, !noalias !10
  %add6 = add nsw i32 %add635, %add
  %inc = add nsw i32 %j.125, 1
  %exitcond = icmp eq i32 %inc, %itr
  br i1 %exitcond, label %for.cond.1.for.inc.8_crit_edge.loopexit34, label
%for.body.3
for.cond.1.for.inc.8_crit_edge.loopexit34:       ; preds = %for.body.3
  %add6.lcssa = phi i32 [ %add6, %for.body.3 ]
  store i32 %add6.lcssa, i32* %arrayidx5, align 4, !tbaa !1, !alias.scope !6,
!noalias !9
```

Current Status & Results

Current Status:

- Its mostly completed, and under review

Results:

- Tested this with regular benchmarks & functional tests
  - No regressions
- Written test cases and in some tests observed good performance gains.

Challenges

Code Bloat:

To control code bloat LoopVersioningLICM takes some measures in Cost Analysis. It checks if possible invariants in a loop is above the ***InvariantThreshold***(default 25%). Versioning is done only if the threshold is not breached

Runtime Checks:

LoopVersioningLICM defines a limit for the number of runtime memory checks. We ensure that the generated checks should be under that limit. We only consider accesses 'read & write' and 'write & write' for runtime checks

Maximum possible checks for 'n' address are: $(\binom{n}{2} * 3) + \left(\binom{n}{2} - 1\right)$

Challenges

Repeated Runtime Checks:

Passes like loop-versioning-licm, vectorizer and loop distribution generates runtime checks, some parts of these checks are repeated. At this point we do not have any solution in-place to control these repeated checks.

Possible solution: Metadata can be used to control these repeated checks.

Acknowledgement

- Adam Nemet for submitting LoopVersioning utility
- llvm-dev community for reviewing the proposal
- Dibyendu Das, Shivarama Rao and Anupama Rasale for working on this.

Questions ?

Thank You !