Path Invariance Based Partial Loop Un-switching

Ashutosh Nema, Shivarama Rao, Dibyendu Das

AMD
Problem Statement

• Loop un-switching is a well-known compiler optimization technique, it moves a conditional inside a loop outside by duplicating the loop's body and placing a version of it inside each of the if and else clauses of the conditional

• Efficient un-switching is severely inhibited in cases where the condition inside a loop is not invariant, or it is partially invariant (invariant in some of the paths of the loop body but not in all)
Loop Un-switching

- A loop containing a loop-invariant IF statement can be transformed into an IF statement containing two loops by loop un-switching as shown below:

<table>
<thead>
<tr>
<th>Loop (Snip#1)</th>
<th>Un-switched loop (Snip#2)</th>
</tr>
</thead>
</table>
| for (i = 0; i < N; i++)
  if (x)
  a[i] = 0;
else
  b[i] = 0; | if (x)
  for (i = 0; i < N; i++)
  a[i] = 0;
else
  for (i = 0; i < N; i++)
  b[i] = 0; |

In the above example variable ‘x’ is an invariant, hence after loop un-switching it has been hoisted out and guards the two versions of the loop.
Our Approach

• Loop un-switching is inhibited in cases where a condition inside a loop is not invariant or partially invariant (invariant in any of the conditional-paths inside the loop but not invariant in all the paths)

• Loop un-switching can still be applied on the partially invariant condition by generating an improved loop version for the invariant-path while the variant-paths will have the original loop versions
Partial Invariance

• A partial invariant is an expression the value of which remains the same immediately before and immediately after each iteration of a loop along a certain path

\[ P_{\text{Inv}} = \{ I \mid P \}, \text{where 'I' is invariant in path 'P'} \]

• Identification of the partial invariant is the primary task in our work. Note that only a portion of the original loop may be controlled by the property of interest
Partial Invariance

• Partial invariant identification is similar to loop invariant identification. We can apply classical methods for invariant detection on multiple paths in loop body

• Loop analysis is required to identify partial invariants by automatically generating richer relationships among loop variables

• LLVM’s existing invariant analysis framework does not identify partial invariants

• A simple version of path invariance detection is applied, focused on branch conditions at a certain level, where the paths for a conditional branch is the subtree within a loop, starting from either of its branch successors
Partial Invariance

Example: Possible paths for a top level branch in one iteration

```c
for (i = start; i < end; i++ ) {
    if (cond) {
        if (x) {
            s1;
            s2;
        } else {
            s3;
        }
    } else {
        if (y)
            s4;
        s5;
    }
    s6;
}
```
Partial Invariance Detection

• Identify partial invariance by considering following properties within a sub path of a loop:

  1. Condition is explicitly modified in a path implies it is not an invariant path

```c
for (i = start; i < end; i++) {
    if (cond) {
        ....
        cond = <...>
    } else {
        ....
        cond = <...>
        ....
    }
}
```

“cond” is explicitly modified in both if-then & if-else path of conditional branch.

“cond” can’t be a partial invariant
Partial Invariance Detection

• Identify partial invariance by considering following properties within a sub path of a loop:
  
  2. Condition is not explicitly modified in a path, then it might be an invariant path

```plaintext
for (i = start; i < end; i++) {
  if (cond) {
    ....
    cond = <...>
  }
  else {
    ....
    ....
  }
}
```

“cond” is explicitly modified in if-then path.

“cond” is not modified in if-else path.

So “cond” is a partial invariant for if-else path if other conditions are met.
Partial Invariance Detection

• Identify partial invariance by considering following properties within a sub path of a loop:
  3. Varying conditions in loop body in each iteration

  ```java
  for (i = start; i < end; i++) {
    if (cond[i]) {
      ....
    } else {
      ....
    }
  }
  “cond[i]” is varying on an induction variable.
  So “cond[i]” can’t be a partial invariant.
  ```
Partial Invariance Detection

• Identify partial invariance by considering following properties within a sub path of a loop:

  4. Condition dependent on varying variable ‘v’ in loop body can be partial invariant for a path where the variable ‘v’ remains unchanged.

```c
v = <..initial-value..>
for (i = start; i < end; i++) {
    if (cond[v]) {
        ....
        v = <..>
        ....
    }
else {
    ....
}
```

“cond[v]” is varying in the loop body as in the if-then path “v” is varying.

“v” remains unchanged in if-else path.

So for initial value of “v” in if-else path cond[v] is partial invariant, if other conditions are met.
Partial Invariance Detection

• Identify partial invariance by considering following properties within a sub path of a loop:

  5. Function call can modify the condition value, i.e. a function modifying a global value which belongs to condition or condition value passed to a function call.

```c
for (i = start; i < end; i++) {
    if (cond) {
        ....
        foo(); // Can modify "cond".
        ....
    } else {
        ....
        bar(&cond); // Can modify "cond".
        ....
    }
}
```

  If “cond” is global then call to function “foo” can modify “cond”.

  Function call to “bar” can modify “cond” as it is passed as an argument.

  “cond” is not a partial invariant if call to foo & bar is not proven safe.
Partial Invariance Detection

• Identify partial invariance by considering following properties within a sub path of a loop:

  6. Control flow guarantees that at runtime loop will only execute the non-modifying (invariant) path and it will never execute the modifying (variant) path.

\[
\text{for } (i = \text{start}; i < \text{end}; i++) \{ \\
\quad \text{if } (\text{cond}) \{ \\
\quad \quad \text{L1:} \\
\quad \quad \quad \text{...} \\
\quad \quad \quad \text{cond} = <..> \\
\quad \quad \quad \text{...} \\
\quad \quad \} \\
\quad \text{else} \{ \\
\quad \quad \text{...} \\
\quad \quad \text{...} \\
\quad \quad \text{goto L1;} \\
\quad \} \\
\}
\]

The if-then and if-else paths overlap and “cond” is varying.

Control flow does not guarantee “cond” invariance in one of the paths

“cond” cannot be a partial invariant.
Partial Invariance Detection

• Identify partial invariance by considering following properties within a sub path of a loop:

  7. Non explicit modification to the condition may change paths during execution and loop can enter from a non-modifying (invariant) path in one iteration to a modifying (variant) path in the next.

```c
for (i = start; i < end; i++) {
    if (cond) {
        ...
        cond = <...>
    }
    else {
        ...
        *ptr = <...>
    }
}
```

“cond” is varying in if-then path.

“cond” is not varying is if-else path, but there is a store.

If “ptr” does not alias with “cond” then “cond” is partial invariant in if-else path.
Else its variant in both paths.
Partial Invariance Detection (Summary)

- Identify partial invariance by considering following properties within a sub path of a loop:
  1. Condition is explicitly modified in path implies it is not an invariant path.
  2. Condition is not explicitly modified in a path, then it might be an invariant path.
  3. Condition in loop body may depend on an induction/reduction variable and it may switch the path at runtime.
  4. Condition dependent on varying variable ‘v’ in loop body can be partially invariant for a path where the variable ‘v’ remains unchanged.
  5. Function call can modify the condition value.
  6. Control flow guarantees that at runtime loop will only execute the non-modifying (invariant) path and it will never execute the modifying (variant) path.
  7. Non explicit modification to the condition may change paths during execution and loop can enter from a non-modifying (invariant) path to modifying (variant) path.
Partial Loop Un-switching

• Partial loop un-switching is an efficient technique to un-switch partial loop-invariant conditions

• It identifies partial-invariant condition for a path and it moves the conditional from inside the loop to outside of it, by duplicating the loop's body, and placing a version of it inside each of the if and else clauses of the conditional

• The variant path will have the full loop with all conditions, whereas the partial invariant path will have the improved version
Partial Loop Un-switching

• Example:

<table>
<thead>
<tr>
<th>Original Loop</th>
<th>Loop with partial un-switched version</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = 0; i &lt; N; i++)</td>
<td>LoopExecutionAssurance = (0 &lt; N);</td>
</tr>
<tr>
<td>if (X)</td>
<td>if (LoopExecutionAssurance) {</td>
</tr>
<tr>
<td>a[i] = 0;</td>
<td>if (X) {</td>
</tr>
<tr>
<td>} else {</td>
<td>for (i = 0; i &lt; N; i++)</td>
</tr>
<tr>
<td>X = b[i]; // X is modified</td>
<td>a[i] = 0;</td>
</tr>
<tr>
<td>b[i] = &lt;...&gt;</td>
<td>} else {</td>
</tr>
<tr>
<td>}</td>
<td>// Original loop version</td>
</tr>
<tr>
<td></td>
<td>for (i = 0; i &lt; N; i++)</td>
</tr>
<tr>
<td></td>
<td>if (X)</td>
</tr>
<tr>
<td></td>
<td>a[i] = 0;</td>
</tr>
<tr>
<td></td>
<td>else {</td>
</tr>
<tr>
<td></td>
<td>X = b[i]; // X is modified</td>
</tr>
<tr>
<td></td>
<td>b[i] = &lt;...&gt;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

In the above example ‘X’ is modified in if-else path & remains invariant in if-then path of the loop body. It is a candidate for partial un-switching.

The partial un-switched version has modified code in if-then path and the original loop in if-else path.

This opens up further optimization opportunities for the new versioned loop.
Implementation Framework

Partial Invariance Analysis

- Goal is to identify partial loop invariant condition for a path in the loop body.
- Implement an analysis utility in LLVM to identify partial invariants.

Partial loop un-switching

- Goal is to transform by generating a un-switched version for partial invariant cases.
- Extend Loop un-switch to handle the partial invariant case.
LLVM’s Existing Loop Un-switching

STEP#1: For the given loop it creates versions by hoisting the condition outside.

```c
for (i = start; i < end; i++) {
    if (cond) {
        ....
        x = cond;
    }
    else {
        ....
        y = cond;
    }
}
```

```c
if (cond) {
    for (i = start; i < end; i++) {
        if (cond) { // Version#1
            ....
            x = cond; // Where "cond" can assumed
                      // to be 1 in condition.
        } //
        else { // In this version "cond" will
            ....
            y = cond; // be replaced by constant 1 in
                      // the if condition.
        } //
    }
}
else { // Version#2
    for (i = start; i < end; i++) {
        if (cond) { // Where "cond" can assumed
            ....
            x = cond; // to be 0 in if condition.
        } //
        else { // In this version "cond" will
            ....
            y = cond; // be replaced by constant 0 in
                      // the if condition.
        } //
    }
}
```
LLVM’s Existing Loop Un-switching

STEP#2: Replace condition with true or false value in appropriate versions.

```c
if (cond) {
    for (i = start; i < end; i++) {
        if (cond) {
            // Version#1
            .... // Where "cond" can assumed
            x = cond; // to be 1 in condition.
        }
        else {
            // In this version "cond" will
            .... // be replaced by constant 1 in
            y = cond; // the if condition.
        }
    }
} else {
    for (i = start; i < end; i++) {
        if (cond) {
            // Version#2
            .... // Where "cond" can assumed
            x = cond; // to be 0 in if condition.
        }
        else {
            // In this version "cond" will
            .... // be replaced by constant 0 in
            y = cond; // the if condition.
        }
    }
}

if (cond) {
    for (i = start; i < end; i++) {
        .... // After condition specialization
        x = cond; // After branch eliminations.
    }
} else {
    for (i = start; i < end; i++) {
        .... // After condition specialization
        y = cond; // After branch eliminations.
    }
}
```
Partial Invariant extension to Un-switching

• STEP#1 remains identical.

• STEP#2: Replace condition with true or false value in appropriate partial invariant version & retain original loop in variant version.
Partial Invariance: Implementation Details

1) From header block identify the branch instruction and its dependent instructions i.e.
   a) Memory accesses it is dependent on.
   b) PHINode it is dependent on.

2) Check safety of the branch condition and ensure it is not modified in the header block.

3) For a branch at a certain level identify subtree within the loop where condition remains invariant by applying classical invariant detection techniques. For each successor block:
   a) Return if it is a loop exit block.
   b) For each instruction in block:
      i. If there is a write to memory then the written address should not be (or alias with) the condition
      ii. If there is a write to memory then the written address should not be (or alias with) the condition’s dependent memory accesses
      iii. Condition depending on a PHINode should not be affected by the write to memory in the path
      iv. If there is a CallInst then it should be a safe call (i.e. no indirect calls, called function does not access memory or read only call)
      v. For branch instructions all successor blocks repeat step “3:a”
Loop Un-switch: Implementation Details

- Introduced “TryPartialLoopUnswitch” method to “LoopUnswitch” class, which gets called after all loop invariants get hoisted out.
- It expects loop to have simple form i.e. it should have a pre-header, latch block, exit blocks.
- It queries partial invariance analysis to check the presence of partial invariance.
- If the loop has a partial invariant then it gets hoisted out and a un-switched loop version gets generated for the partial invariant path, for the variant path the original loop is retained.
Results

• 1.07x uplift measured with SPEC CPU2006 omnetpp.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>Partial Loop Un-Switching</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU2006 omnetpp</td>
<td>254s</td>
<td>238s</td>
</tr>
</tbody>
</table>

• No Impacts on other SPEC CPU2006 benchmarks.
Questions
• Thank you!