Profile-based Indirect Call Promotion

Ivan Baev, Qualcomm Innovation Center
Outline

- Motivation
- Indirect call promotion transformation and heuristics
- Results
- Related optimizations
Motivation: reduce indirect branch mispredictions

- Object-oriented programs are ubiquitous
  - Virtual function calls usually implemented with indirect branch instructions
- Indirect calls can be common in C programs too
  - 104 static indirect calls in gap benchmark
- Indirect branch is more difficult to predict than conditional branch in hardware
  - It requires prediction of target address instead of prediction of branch direction
  - Branch direction can take only two values: taken or not-taken
  - Indirect branch target prediction can involve N possible target addresses
Motivation: reduce indirect branch mispredictions

UT-Austin/Intel study with Intel Core Duo T2500 processor with a specialized indirect branch predictor [H. Kim et al., ISCA, 2007]
Motivation: impact of profile-based optimizations

- Inlining
- Indirect call promotion
- Code (basic blocks, functions) placement optimizations
- Data (globals, structures) placement optimizations
- Profile-enhanced classical optimizations (if-conversion, partial redundancy elimination, scheduling, register allocation, etc.)
Impact of selected IP and profile-based optimizations

Google study with Open64 compiler on Intel Pentium 4 [X. Li et al., CGO, 2010]
Indirect call promotion (ICP) – definition and opportunities

- ICP replaces an indirect call with:
  - A compare instruction, conditional branch, and direct call to the hottest target
  - The direct call is often inlined

- ICP reduces indirect branch misprediction penalty
- Enhances the impact of inter-procedural optimizations – e.g. inlining or function placement
- Enlarges the scope of optimizations around indirect calls - e.g. loop or global optimizations
Example of indirect call transformation with two targets promoted

```assembly
define void @main(void (i32)* %fp) {
  entry:
    %0 = bitcast void (i32)* %fp to i8*
    %1 = bitcast void (i32)* @foo to i8*
    %2 = icmp eq i8* %0, %1
    br i1 %2, label %if.true, label %if.false

  if.true:
    call void @foo(i32 10)  // direct call to foo
    br label %if.merge

  if.false:
    %3 = bitcast void (i32)* %fp to i8*
    %4 = bitcast void (i32)* @bar to i8*
    %5 = icmp eq i8* %3, %4
    br i1 %5, label %if.true2, label %if.false3

  if.true2:
    call void @bar(i32 10)  // direct call to bar
    br label %if.merge

  if.false3:
    call void %fp(i32 10)
    br label %if.merge

  if.merge:
    ret void
}
```

define void @main(void (i32)* %fp) {
  entry:
    %0 = bitcast void (i32)* %fp to i8*
    %1 = bitcast void (i32)* @foo to i8*
    %2 = icmp eq i8* %0, %1
    br i1 %2, label %if.true, label %if.false

  if.true:
    call void @foo(i32 10)  // direct call to foo
    br label %if.merge

  if.false:
    %3 = bitcast void (i32)* %fp to i8*
    %4 = bitcast void (i32)* @bar to i8*
    %5 = icmp eq i8* %3, %4
    br i1 %5, label %if.true2, label %if.false3

  if.true2:
    call void @bar(i32 10)  // direct call to bar
    br label %if.merge

  if.false3:
    call void %fp(i32 10)
    br label %if.merge

  if.merge:
    ret void
}

%fp may go to functions @foo, @bar, …
ICP design goals

- Provide a general solution as an LLVM transformation pass
- Provide many tuning options for deployment in an LLVM-based compiler depending on customer requirements and workloads

- Clear interfaces to allow development in parallel:
  - Interface with indirect call profiling - through indirect call metadata
    `{"indirect_call_targets", i64 6000, !"foo", i64 3000, !"bar", i64 2500, !"other", i64 500}`
  - Interface with inliner - defer any inlining decisions to Inliner which has a complete view of the application
Indirect call profiling

- For each indirect call/invoke we record the number of times their target functions are invoked
- Instrument at clang level by extending the existing profiling infrastructure
- Extended to value profiling
  - Currently reviewed and upstreamed in several patches
- With early inline and late instrumentation we might instrument at LLVM IR level
Example of indirect call transformation with two targets promoted

```c
#define void @main(void (i32)* %fp) {
    entry:
        call void %fp(i32 10), !prof !1
        ret void
}

!1 = {!"indirect_call_targets", i64 6000, !"foo", i64 3000, !"bar", i64 2500, !"other", i64 500}

define void @main(void (i32)* %fp) {
    entry:
        %0 = bitcast void (i32)* %fp to i8*
        %1 = bitcast void (i32)* @foo to i8*
        %2 = icmp eq i8* %0, %1
        br i1 %2, label %if.true, label %if.false, !prof !0
    if.true:
        call void @foo(i32 10) // direct call to foo
        br label %if.merge
    if.false:
        %3 = bitcast void (i32)* %fp to i8*
        %4 = bitcast void (i32)* @bar to i8*
        %5 = icmp eq i8* %3, %4
        br i1 %5, label %if.true2, label %if.false3, !prof !1
    if.true2:
        call void @bar(i32 10) // direct call to bar
        br label %if.merge
    if.false3:
        call void %fp(i32 10), !prof !2
        br label %if.merge
    if.merge:
        ret void
}

!0 = {!"branch_weights", i32 3000, i32 3000}
!1 = {!"branch_weights", i32 2500, i32 500}
!2 = {!"indirect_call_targets", i64 500, !"other", i64 500}
```
Example of indirect invoke transformation with one target promoted

== Basic Blocks Before ==

entry:
  invoke void @_ZN11EtherAppReqD1Ev(%class.EtherAppReq * %this)
to label %invoke.cont unwind label %lpad, !prof !6

l6 = !{"indirect_call_targets", i64 39458265, !"_ZN11EtherAppReqD2Ev", i64 39458265}

== Basic Blocks After ==

entry:
  %0 = bitcast (%class.EtherAppReq*) @_ZN11EtherAppReqD1Ev to i8*
  %1 = bitcast (%class.EtherAppReq*) @_ZN11EtherAppReqD2Ev to i8*
  %2 = icmp eq i8* %0, %1
  br i1 %2, label %if.true, label %if.false

if.true:
  invoke void @_ZN11EtherAppReqD2Ev(%class.EtherAppReq * %this)
to label %if.merge unwind label %lpad

if.false:
  invoke void @_ZN11EtherAppReqD1Ev(%class.EtherAppReq * %this)
to label %if.merge unwind label %lpad, !prof !7

if.merge:
  br label %invoke.cont

l7 = !{"indirect_call_targets", i64 0}
ICP heuristics

- Which call sites to consider?
- For a given call site, which targets to consider for promotion?
- Should we add inline hints to promoted targets?
- Should we consider other profile information?
Call site hotness heuristic

- We should consider all indirect call sites for promotion if there is no concern for size expansion.

- Option `callHotnessThreshold` to filter out cold indirect calls.
  
  Cold indirect call count < `callHotnessThreshold` * (Sum of indirect call counts)
  
  `callHotnessThreshold` = 0.001 by default.
Call target hotness heuristic

- Promote the most frequent target if
  \[ \text{target count} > \text{targetHotnessThreshold} \times (\text{call site count}) \]
  \[ \text{targetHotnessThreshold} = 40\% \text{ by default} \]

- Promote the second most frequent target if
  the most frequent target is promoted &&
  \[ \text{target count} > \text{target2HotnessThreshold} \times (\text{call site count}) \]
  \[ \text{target2HotnessThreshold} = 30\% \text{ by default} \]

- Option enable-second-target to allow promotion of the second target
Inline hints and inline heuristic

- Clang adds inline hint to a direct call if its profile count is > 30% of the most frequent call count.

- Add inline hint to the promoted target if
  
  \[
  \text{target count} > \text{inlineHintThreshold} \times (\text{Sum of call sites counts})
  \]

  inlineHintThreshold = 1% by default

- Inliner gives a small bonus to a call with inline hint
  - A direct call coming from ICP needs to overcome the overhead of compare and conditional branch instructions
  - Sophisticated profile-based inliner will likely take this into account
### ICP impact on SPEC2000/2006

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of static indirect calls considered/promoted</th>
<th>Speedup (%)</th>
<th>Code size increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eon (C++)</td>
<td>28/28</td>
<td>9</td>
<td>0.6</td>
</tr>
<tr>
<td>h264ref (C)</td>
<td>33/33</td>
<td>6</td>
<td>0.2</td>
</tr>
<tr>
<td>namd (C++)</td>
<td>12/12</td>
<td>2</td>
<td>6.6</td>
</tr>
<tr>
<td>omnetpp (C++)</td>
<td>37/37</td>
<td>3</td>
<td>0.3</td>
</tr>
<tr>
<td>povray (C++)</td>
<td>7/6</td>
<td>4</td>
<td>0.2</td>
</tr>
<tr>
<td>sjeng (C)</td>
<td>1/1</td>
<td>2</td>
<td>0.0</td>
</tr>
</tbody>
</table>

QC Snapdragon 3.7 LLVM compiler  
QC A57-based device in AArch64 mode, indirect predictor with path history  

4 second most frequent targets promoted in eon for 4% improvement
ICP enables other optimizations - future work

- Better inlining
- Function placement
  - IC profiling allows complete information for indirect call nodes in the application call graph
- ThinLTO, AutoFDO – advanced link-time frameworks
  - ICP allows better partitioning of call graph and optimizations on hot partitions

- Investigate interaction with indirect branch target prediction hardware and other micro-architectural features
- Consider function entry and basic block profile information
Acknowledgements

Betul Buyukkurt (QuIC), David Li (Google), Teresa Johnson (Google)

Questions?