Ivan Baev, Qualcomm Innovation Center

Profile-based Indirect Call Promotion

Outline

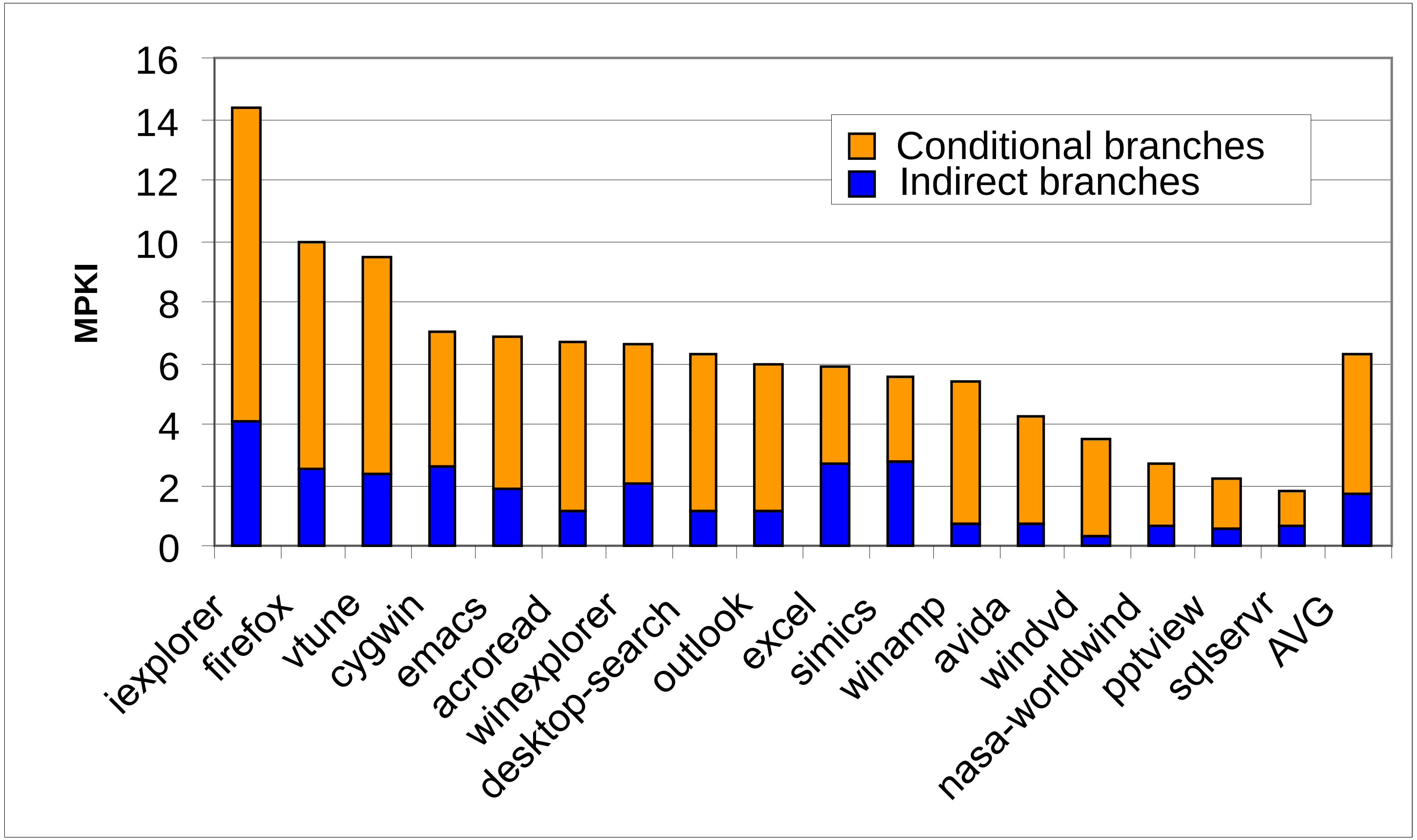
- Motivation
- Indirect call promotion transformation and heuristics
- Results
- Related optimizations

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

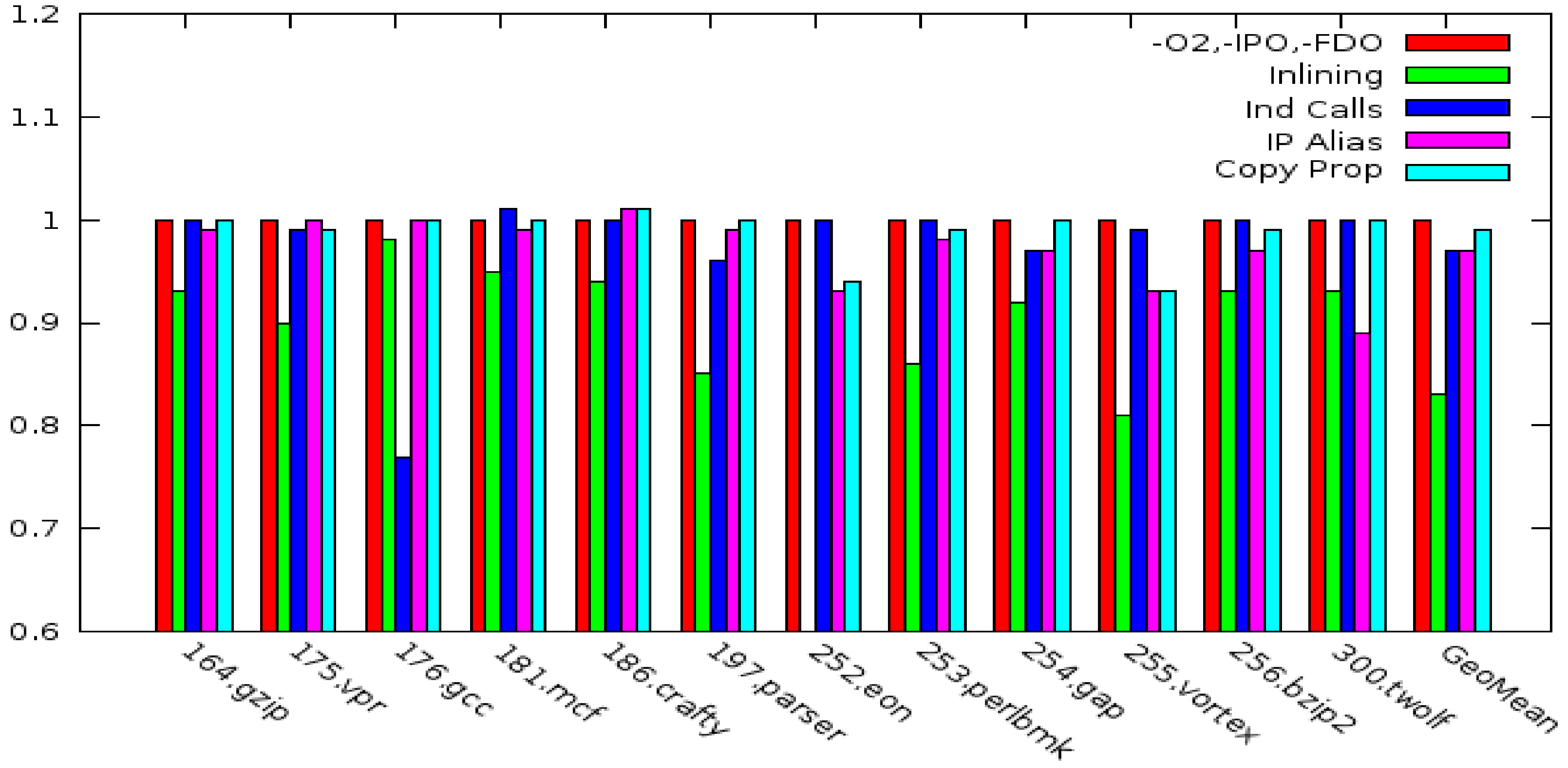


UT-Austin/Intel study with Intel Core Duo T2500 processor with a specialized indirect branch predictor [H. Kim et al., ISCA, 2007]

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

```
define void @main(void (i32)* %fp) {
entry:
  call void %fp(i32 10)
  ret void
}
```

%fp may go to functions @foo, @bar, …

```
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
}
```

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

```
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 Block Before ==

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

!6 = !{!"indirect_call_targets", i64 39458265, !"_ZN11EtherAppReqD2Ev",
i64 39458265}
```

== Basic Blocks After ==

```
entry:
  %0 = bitcast void (%class.EtherAppReq*)* @_ZN11EtherAppReqD1Ev to i8*
  %1 = bitcast void (%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

!7 = !{!"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
  - target count > targetHotnessThreshold * (call site count)
    - targetHotnessThreshold = 40% by default

- Promote the second most frequent target if
  - the most frequent target is promoted &&
  - target count > target2HotnessThreshold * (call site count)
    - target2HotnessThreshold = 30% 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

  target count > inlineHintThreshold * (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

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

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?