agenda

- translate the title
- contextual instrumentation implementation (main topic)
- results
- non-server example
- plans & speculations
what is “instrumented...” (aka iFDO)?

---

- 2 builds:
  - instrument certain edges, before IPO:
    - `llvm.instrprof.*` intrinsics
    - identify each edge with an index (0, 1, 2..)
    - lower to counter increments in a global, per-function buffer
  - run the program -> counter values form the profile
    - maybe run it with a bunch of different inputs & merge profiles
  - rebuild: profile ingested at same position in pass pipeline as instrumentation
    - so that counter indices match
  - $ profit! $ :)

- compiler-rt code for e.g. saving the globals to a file

datacenter applications (compiler view)

- - -

- **init**: set up internal architecture
- **steady state**, threads in a pool execute an infinite loop:
  - pick up *work* from some queue
  - execute some synchronous *task* (typically short lived)
- **a task** is the analogue of “classical” *main*
  - just that the entry point is the RPC handler
  - the **most direct** impact of the optimizing compiler: 
    *finish tasks fast*

- we collect profiles via an RPC, too!
a profiling problem

- - -

- we load profiles before IPO
- callee behavior can be dependent on use
  - but! profile has *averages* over all possible callers!
  - => profile quality degrades through inlining
  - poor profile => less profit :(  

a measurable effect: can we estimate dynamic instruction count changes if inline policy changes -> reward signal for MLGO training

see also the CSPGO talk earlier, for the sampling-based profiling approach to this problem
contextual profiles

- - -

● keep distinct counters for different call sites paths
  ○ btw, “paths” starting from where?

● challenge for instrumentation (stemming from current technique):
  ○ we pre-allocate counters statically
  ○ we don’t know the call paths - how many counter versions to allocate?
  ○ (...various alternatives)
key insight

Rather than think of it as a “classical binary”

Why not think of it as a “package of tasks”
  (distinct entry points & call graphs)
do we know the entry points (to the tasks)?

---

- yes (well, the binary owner would know)
- ..or, can detect from behavior in production
  - this is just about determining main entry point functions (~=main RPC handlers)
  - unlikely to change too often
- they are coarse, architectural property of a binary (i.e. fairly stable over time)
implementation
high level

- pass task entry points via an LLVM flag
- main LLVM change: *how instrumentation intrinsics are lowered*
- new instrumentation intrinsic: `llvm.instrprof.callsite`
  - identifies by index a callsite in a function
  - precedes any call site that's not inline asm or intrinsic
- lowering:
  - no global counter arrays
  - entry BB: call to `__llvm_instrprof_{get|start}_context`
    - returns a chunk of memory - the Context
  - counter update: `Context.counters[counter_index] += <step>`
  - `llvm.instrprof.callsite`:
    - save `CS.getCalledValue()` and `&Context.callsites[callsite_index]` in TLS
low(er) level Details

---

ContextNode

<table>
<thead>
<tr>
<th>Header</th>
<th>NrCounters counters</th>
<th>NrCallsites callsites</th>
</tr>
</thead>
</table>

uint64_t Guid

ContextNode* Next

uint32_t NrCounters
uint32_t NrCallsites

-------------
8 + 8 + 4 + 4 = 24 bytes

NrCounters and NrCallsites are function-specific, compile-time constants

passed to __llvm_instprof_{get|start}_context
define void @an_entrypoint(...) {
    call @llvm.instrprof.increment(<an_entrypoint_guid>, .., <nr_counters>, 0)
    ...
    br i1 %smth label %a, label %b

a:
    call @llvm.instrprof.increment(<an_entrypoint_guid>, .., <nr_counters>, 5)
    ...
    call @llvm.instrprof.callsite(<an_entrypoint_guid>, .., <nr_callsites>, 2, %callee_1)
    call void %callee_1
}

@define void @a_callee() {
    call @llvm.instrprof.increment(<a_callee_guid>, .., <nr_counters>, 0)
define void @an_entrypoint(...) {
  %ctx = __llvm_instrprof_start_context(1234, ...)
  ... 
  br i1 %smth label %a, label %b 
  a:
  %ctx.counters[5] += 1 
  ... 
  _tls.expected_cs = %callee_1
  _tls.callsite_info = <gep, %ctx.callsites, 2>
  call void %callee_1 
  ... 
} 

@define void @a_callee() {
  %ctx = __llvm_instrprof_get_context(5678, 
    @a_callee, nr_counters, nr_callsites) 
  ... 
}
define void @entrypoint(...) {
    call @foo  // entrypoint id: 2
}

@define void @foo() {
    call @bar  // entrypoint id: 0
}

@define void @bar() {
    call @foo  // entrypoint id: 5
}

A function will have a different context depending on position in call graph
entry points are special

---

```c
struct ContextRoot {
    ContextNode *FirstNode;
    Arena *FirstMemBlock;
    Arena *CurrentMem;
    __sanitizer::StaticSpinMutex Lock;
};
```

Set up and zero-initialized on LLVM side

Parameter to `__llvm_instrprof_start_context`

Lock.tryLock() failed? get a “ScratchContext” instead!
ScratchContext

- - -

- also what `__llvm_instrprof_get_context` returns if:
  - expected callee doesn’t match
  - there’s TLS data absent (== we’re outside any entry point)

- understood by compiler-rt:
  - if function uses ScratchContext, all callees will also use that
  - detectable because TLS will contain a ScratchContext interior pointer

(also what `__llvm_instrprof_start_context` returns if it can’t take the lock)
special consideration: signal handlers

- at any random point in execution, a signal handler may be called
  - it will promptly call `__llvm_instprof_get_context`! now what?
- it will discover:
  - either no call info on TLS (the handler in runtime “consumes” that info); or
  - expected callee is not itself; or
  - the call info is pointing in ScratchContext

  => return ScratchContext
other special considerations
---

- **recursion**
  - not doing anything special - just chain recursive activations
  - “it’s OK” - RPCs should finish fast

- **tail calls**
  - currently doing nothing special
  - could get smart and keep a “bubble” of contexts
profile format

---

- 2 step:
  - raw: dump the Arenas (plus a small header)
  - post-process to LLVM Bitstream
results
setup

- - -

- search binary
  - not IO bound, low/no contention -> longer running RPCs
  - 2 workloads, “large” and “small”
- reusing current instrumented profiling collection tooling

- runtime mem overhead: 120+25 + N_THREADS * 1MB (<=ScratchContext size)
- regular iFDO profile: 204MB (zipped: 65MB)
- final ctx profile: 46MB (zipped: 12MB)
## Profile Characteristics

<table>
<thead>
<tr>
<th>Workload</th>
<th>Raw Profile</th>
<th># Contexts</th>
<th># Counters</th>
<th># Non-0 Counters</th>
<th>Max Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large</td>
<td>120MB</td>
<td>1,118,987</td>
<td>9,376,222</td>
<td>3,201,863</td>
<td>89</td>
</tr>
<tr>
<td>Small</td>
<td>25MB</td>
<td>279,725</td>
<td>2,082,311</td>
<td>774,796</td>
<td>62</td>
</tr>
</tbody>
</table>
## Binary Size

---

<table>
<thead>
<tr>
<th>section</th>
<th>IFDO</th>
<th>Contextual</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>449MB</td>
<td>645MB</td>
</tr>
<tr>
<td>_<em>llvm_prf</em>*</td>
<td>213MB (35MB are names, so “critical” is 117MB)</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>964MB</td>
<td>960MB</td>
</tr>
</tbody>
</table>

*Inherent .text overhead due to callsites.*
## Profile Collection Performance

<table>
<thead>
<tr>
<th>Metric</th>
<th>iFDO</th>
<th>Contextual</th>
<th>Improv. Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;TOTAL&quot;: QPS:</td>
<td>3,456.95</td>
<td>18,399.70</td>
<td>5.32</td>
</tr>
<tr>
<td>&quot;SMALL&quot;: avg_cpu_kcycles:</td>
<td>49,112.96</td>
<td>10,699.07</td>
<td>4.59</td>
</tr>
<tr>
<td>&quot;SMALL&quot;: 99.9%ile_server_latency_usec:</td>
<td>1,152,165.00</td>
<td>166,625.00</td>
<td>6.91</td>
</tr>
<tr>
<td>&quot;SMALL&quot;: avg_round_trip_latency_usec:</td>
<td>54,970.23</td>
<td>11,297.49</td>
<td>4.87</td>
</tr>
<tr>
<td>&quot;LARGE&quot;: avg_cpu_kcycles:</td>
<td>91,293.63</td>
<td>13,937.04</td>
<td>6.55</td>
</tr>
<tr>
<td>&quot;LARGE&quot;: 99.9%ile_server_latency_usec:</td>
<td>634,050.00</td>
<td>75,521.00</td>
<td>8.40</td>
</tr>
<tr>
<td>&quot;LARGE&quot;: avg_round_trip_latency_usec:</td>
<td>65,588.55</td>
<td>10,403.51</td>
<td>6.30</td>
</tr>
</tbody>
</table>

- **(absence of) shared writes**
  - anecdote: if the null context were shared => 20x slowdown compared to normal iFDO! (yes, without any concurrency control)
- **steady-state overhead is just around callsites**
  - i.e. decaying occurrence of any (bump-)allocation
Distribution of nr of counters per ctx (y: log scale)
Distribution of nr of callsites per ctx (y: log scale)
Distribution of max indirect calls in a ctx (y: log scale)
Distribution of number of contexts by depth (linear)
a non-server use
### opt

- collected the IR of PassBuilder.cpp (~8MB)
- `opt --passes='default<O2>'`

<table>
<thead>
<tr>
<th>metric</th>
<th>iFDO</th>
<th>Contextual</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>12.51s*</td>
<td>30.29s*</td>
</tr>
<tr>
<td>profile size</td>
<td>19MB</td>
<td>138MB</td>
</tr>
<tr>
<td>#counters</td>
<td></td>
<td>66M</td>
</tr>
<tr>
<td>#contexts</td>
<td></td>
<td>3.5M</td>
</tr>
<tr>
<td>runtime mem usage</td>
<td></td>
<td>868MB</td>
</tr>
</tbody>
</table>

*(for reference: ~9s non-instr)*
Distribution of nr of counters per context (y: log scale)
Distribution of nr of callsites per ctx (y: log scale)
Distribution of max indirect calls in a ctx (y: log scale)
Distribution of nr of contexts by depth (linear)
some differences

- - -

- max counter values (relative to entry):
  - large: 4K
  - small: 16K
  - opt: 24M

- it’s why opt’s profile is relatively small
  - spends more time in loops
plans & speculations
profile ingestion (use)

- - -

- interplay with ThinLTO
- ThinLTO ingestion builds on existing “Workload Definitions” (PR #74545)
  - ingest all of a graph into one module
- post-link opt leverages ModuleInliner:
  - do all IPO first, and then function simplification
  - ICP, Inliner awareness about ctx profiling
  - this can be relaxed, piecemeal, for passes in the function simplification pipeline, as necessary
possible commonalities with CSPGO

---

- let’s first iterate a bit, risk of “too early abstractions”

- Realistic goal (*I think*):
  - a common “ContextualProfileAnalysis”
  - or at least an abstraction
  - goal is to make “contextual awareness” for a pass a technique-independent change
in closing

---

- RFC -> after EuroLLVM
- “showcase” [PR #86036](https://github.com/llvm/llvm-project/blob/master/PR86036)
- the “task-based”, “pass the entry points” approach may be more general
  - main isn’t what it used to be
  - lots of other programs are event-driven (browsers, phone apps.)
  - focus analysis, optimizations… (“optimizing, but to what end?”)