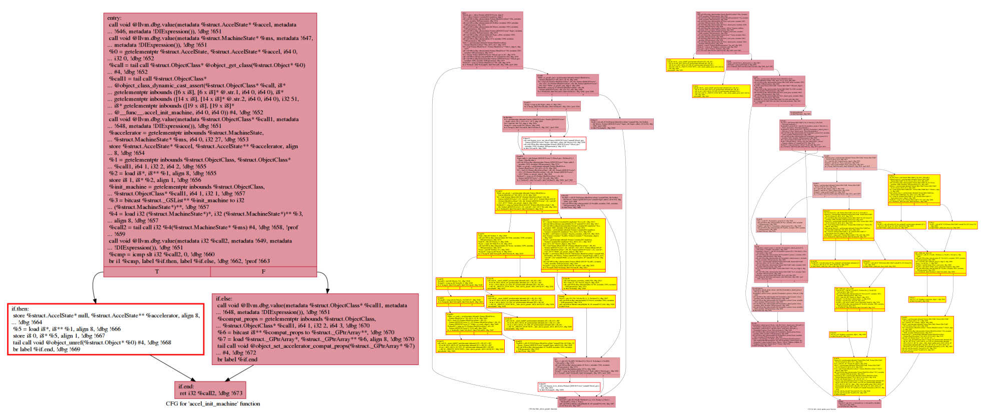
Improving Hot/Cold Splitting in LLVM

Ruijie Fang `ruijief@princeton.edu`

*Joint work with **Aditya Kumar** (Facebook) and **Rodrigo Rocha** (University of Edinburgh)*



Agenda

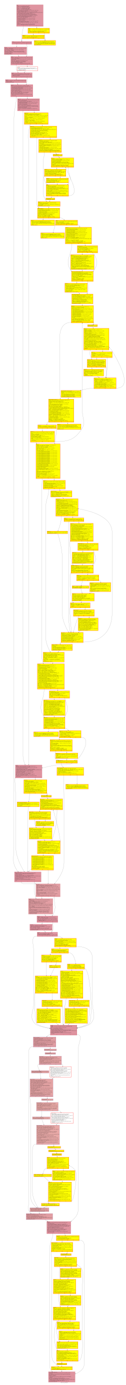
- Hot/Cold Splitting
  - Motivation
- Improvement Ideas and Benchmark Results
  - Outline before inlining.
  - Outlining exception handling blocks.
  - Adding a cold section.
  - Cost model.
- Concluding thoughts

Motivation

Organize code in a hot trace within a large function as closely together as possible to improve icache locality.

More specifically, given profile / analysis info: group hot blocks together and separate out the cold blocks.

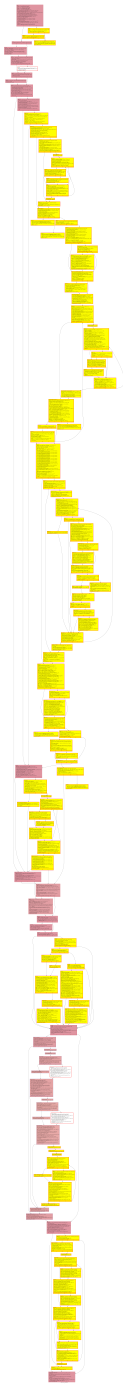
Many different ways to do this --- Today's focus: Hot/cold splitting pass in LLVM's mid-end.



Example. Control-flow graph of `aio_poll` function in qemu.

← The blocks shown in yellow are cold blocks! (not executed by the specific trace we're analyzing). 609 lines of assembly compiled by `clang -O3` on a x86-64 computer. Equivalently `nm --print-size` tells us it occupies 2428 bytes (largest function in the object file).

```
0000000000004112 0000000000000005 T aio_context_use_g_source
0000000000000000 0000000000000012 T aio_poll_disabled
0000000000000165 0000000000000015 r .L.str.4
0000000000000032 0000000000000017 r .L.str.1
0000000000004080 0000000000000018 T aio_context_destroy
0000000000000278 0000000000000022 r .L.str.6
0000000000000085 0000000000000023 r .L.str.2
0000000000000368 0000000000000025 r .L.str.8
0000000000000434 0000000000000027 r .L__PRETTY_FUNCTION__.rcu_read_unlock
0000000000004048 0000000000000029 T aio_context_setup
0000000000000000 0000000000000032 r .L.str
0000000000000049 0000000000000036 r .L__PRETTY_FUNCTION__.aio_poll
0000000000004128 0000000000000037 T aio_context_set_poll_params
0000000000000238 0000000000000040 r .L.str.5
0000000000000393 0000000000000041 r .L.str.9
0000000000000800 0000000000000045 T aio_set_fd_poll
0000000000000461 0000000000000047 r .L.str.10
0000000000000628 0000000000000047 r .L.str.13
0000000000000579 0000000000000049 r .L.str.12
0000000000000108 0000000000000057 r .L.str.3
0000000000000180 0000000000000058 r .L__PRETTY_FUNCTION__.run_poll_handlers
0000000000000928 0000000000000066 T aio_set_event_notifier_poll
0000000000000300 0000000000000068 r .L.str.7
0000000000000508 0000000000000071 r .L.str.11
0000000000000016 0000000000000074 T aio_add_ready_handler
0000000000000848 0000000000000074 T aio_set_event_notifier
0000000000001264 0000000000000104 T aio_dispatch
0000000000001008 0000000000000106 T aio_prepare
0000000000001120 0000000000000139 T aio_pending
0000000000001376 0000000000000238 t aio_free_deleted_handlers
0000000000004176 0000000000000437 t aio_dispatch_handler
0000000000000096 0000000000000698 T aio_set_fd_handler
0000000000001616 0000000000002428 T aio_poll
```



Example. Control-flow graph of `aio_poll` function in qemu.

← **After hot/cold splitting**: Extract yellow-colored blocks into 6 individual functions with cold attributes. The `aio_poll` function is now 192 lines of assembly and 731 bytes.

```
0000000000001760 0000000000000130 t aio_poll.cold.4
0000000000000208 0000000000000159 T aio_dispatch
0000000000000032 0000000000000164 T aio_pending
0000000000001072 0000000000000194 t aio_poll.cold.1
0000000000000816 0000000000000201 t aio_dispatch.cold.1
0000000000003344 0000000000000211 t aio_context_use_g_source.cold.1
0000000000000128 0000000000000230 T aio_context_destroy
0000000000000448 0000000000000237 t aio_set_fd_handler.cold.1
0000000000000368 0000000000000325 t aio_dispatch_handler
0000000000001392 0000000000000355 t aio_poll.cold.3
0000000000000112 0000000000000594 T aio_set_fd_handler
0000000000000800 0000000000000731 T aio_poll
0000000000001904 0000000000001367 t aio_poll.cold.5
```

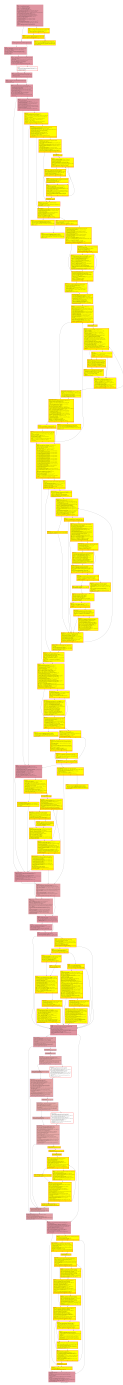
Hot/cold splitting

- An optimization pass for **instruction cache locality** and **code size** in mid-end
- Takes in {profile info|static analysis info}, determine cold blocks using cost model, extract cold regions using `CodeExtractor`
- Contributed by Aditya Kumar in 2019
- Significant improvements made by Vedant Kumar, Aditya Kumar, and others

Hot/cold splitting

- **This talk:** Ideas for improving HCS and results/insights obtained from benchmarking these ideas on open-source codebases.
- Three codebases:
  - firefox
  - Z3 SMT solver+quantifier-free linear arithmetic (`QF_LIA`) as background theory
  - qemu
- What ideas worked, what didn't work, and what workloads are[n't] worth applying HCS to.

Why improvements?



Recall Example: Control-flow graph of `aio_poll` function in qemu.

$\leftarrow$ **After hot/cold splitting**: Extract yellow-colored blocks into 6 individual functions with cold attributes. The `aio_poll` function is now 192 lines of assembly and 731 bytes.

```
0000000000001760 0000000000000130 t aio_poll.cold.4
0000000000000208 0000000000000159 T aio_dispatch
0000000000000032 0000000000000164 T aio_pending
0000000000001072 0000000000000194 t aio_poll.cold.1
0000000000000816 0000000000000201 t aio_dispatch.cold.1
0000000000003344 0000000000000211 t aio_context_use_g_source.cold.1
0000000000000128 0000000000000230 T aio_context_destroy
0000000000000448 0000000000000237 t aio_set_fd_handler.cold.1
0000000000000368 0000000000000325 t aio_dispatch_handler
0000000000001392 0000000000000355 t aio_poll.cold.3
0000000000000112 0000000000000594 T aio_set_fd_handler
0000000000000800 0000000000000731 T aio_poll
0000000000001904 0000000000001367 t aio_poll.cold.5
```

Caveat! Sum of size of `aio_poll` + extracted cold blocks is 2979 bytes (> 2428 bytes before optimization)

Improvement Ideas

Two considerations:

- (More) code size reduction.
- Performance: icache / branch miss rate, pagefaults

Improvement Ideas

1. Detect and determine cold blocks.
    - Rearranging order of optimization passes: calling HCS early before every inliner pass, using HCS together w/ other passes
2. Splitting more cold blocks.
    - Splitting Itanium-style EH blocks that are marked cold
3. Where to put the cold blocks.
    - Putting cold functions in a separate cold section.

Bottom line:

- No code size blowup + perf improvement, or
- Code size reduction + no perf regression.

Turns out…

Different ideas have different effect across different codebases.

Experimental Setup

- Ubuntu 20.04LTS / Intel E5-1607 v3 @3.1GHz / 32GB RAM / 32K L1 cache, 256K L2 cache, 10240K L3 cache.
- Frequency scaling disabled.

For `firefox`: With `-Os` or `-O3`, workload uses `talos-test perf-reftests` benchmark and uses PGO information from the same benchmark.

For `z3`: No PGO, compiled with `-O3`, HCS uses only static analysis info. Workload from `SMTLIB2` benchmark suit's `QF_LIA/CAV2009/45vars`.

Idea 1: Outline before inlining.

- Schedule HCS early in the new PassManager's PGO optimization pipeline, before the stock ModuleInliner pass.
- Outline code every time before inliner is called.
- More regions split, *slight* perf gain, but code size blowup.

firefox, talos-test perf-reftest (-O2 6 runs, performance).

	Time (mean)	Time (median)	Regions Detected	Regions Split
-O2 Baseline	1015.8s	1015.0s		
-O2 PGO+Vanilla HCS	961.3s	961.0s	152048	69444
-O2 PGO+Inliner HCS	959.444s	959.0s	157166	74166
-O2 D59715	964.447s	953.472s		

Idea 2: Outlining exception handling blocks.

- C++ `catch` blocks are marked cold by default. However, can't extract them without complications because EH handling isn't regular control flow
- *Experimental*: Before we start, words of caution
    - The method we use is *destructive*: Transforms EH regions (while not guaranteeing splitting).
    - Not the best approach, but *an* approach; in general, quite difficult to do in mid-end.
    - Not an expert on EH, and full discussion of EH handling is beyond scope.

EH Outlining

Itanium-style EH handling in LLVM follows roughly the following structure:

```
invoke-***
|
lpad-***
|
catch.dispatch
|         |
catch     |
|         |
catch.fallthrough
|
resume
```

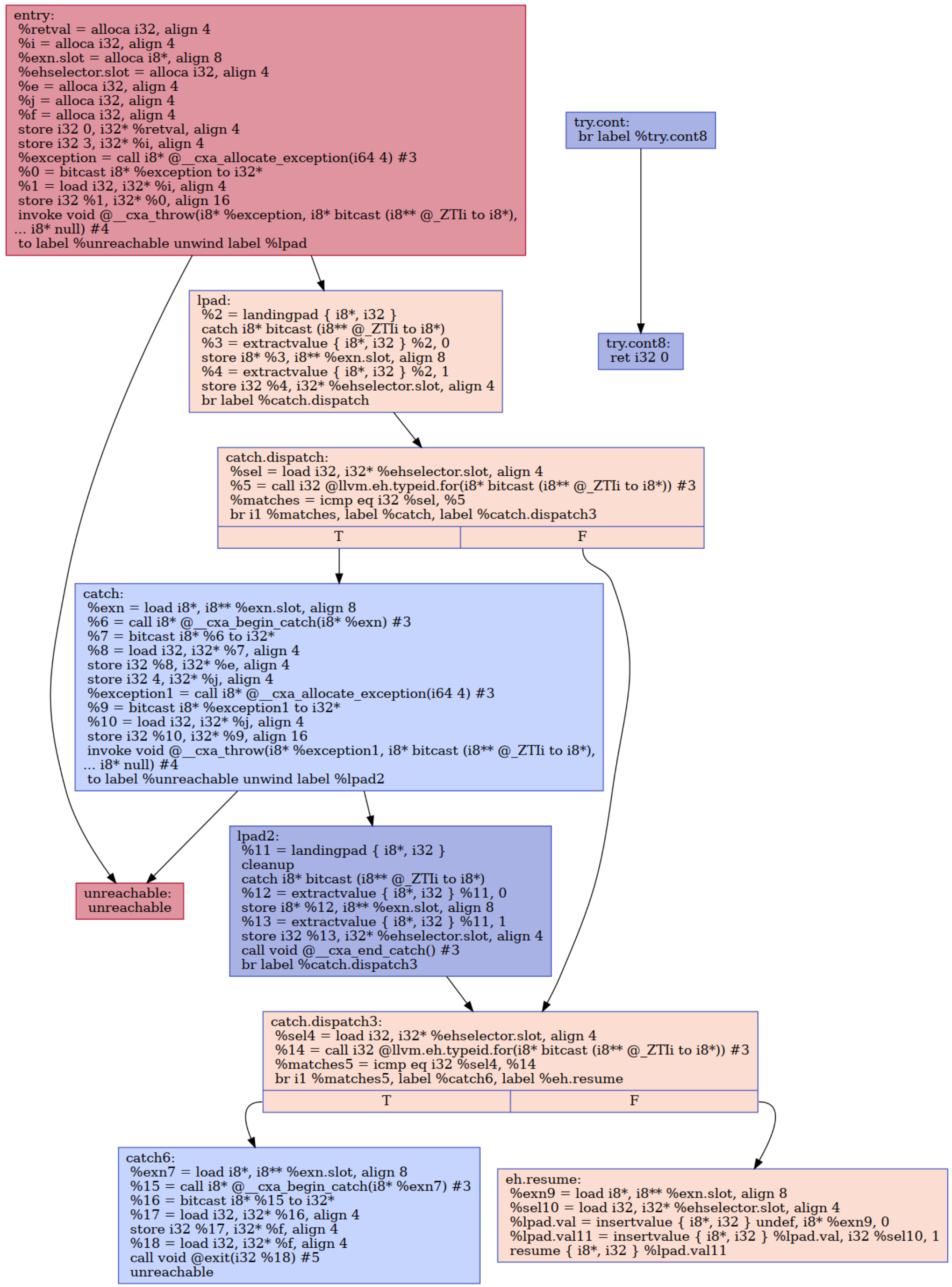
EH outlining difficulties

1. Cannot extract the block containing the invoke (otherwise hot branch might be extracted)
2. Cannot extract the entire landing pad block, since the first instruction after the unwind edge into the lpad block must be the landingpad instruction.
3. Nothing above `catch.dispatch` maybe extracted: `catch.dispatch` contains calls to `eh.typeid.for` intrinsic but it is function-specific. As such, CodeExtractor cannot extract these calls.

EH outlining

Only opportunity left: Start extracting SESE region from `catch.dispatch`.
Idea: Extract the calls to `typeid.for` intrinsic to a block further up in the control flow graph, and since we have rather normal control flow, we can do so safely and store the resultant values in some variable.

However, Since there might be nested catch blocks, we cannot simply extract their calls to `eh.typeid.for` to an arbitrary block that precedes them. (Otherwise we need to create phi nodes) Consider the following example of nested throws...



CFG for 'main' function

An experimental solution

- For every call instruction to `eh.typeid.for` in every `catch.dispatch` block, move them to the highest post-landingpad block that dominates the current `catch.dispatch` block.
- Safe --- since the destination block we moved to is within the EH region and dominates `catch.dispatch`
- Also some (but not all) ability to extract nested catch blocks.

Evaluation

On Firefox, `-Os`, with PGO-enabled:

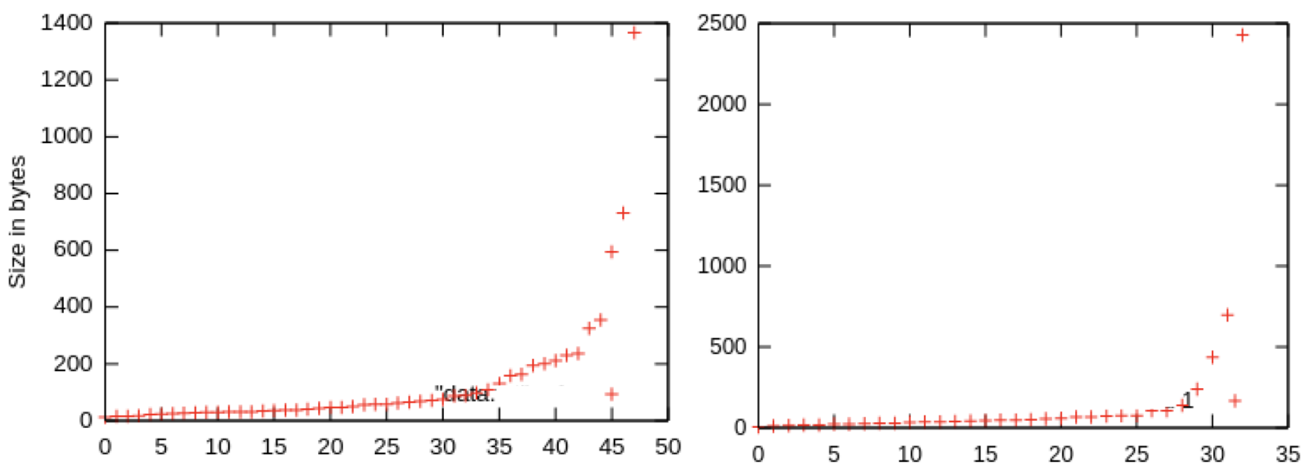
	Opt Level	Size (incl. dynamic libraries)
delta=0 HCS	-Os	2.188262032 GB
EH outlining HCS	-Os	2.187481424GB

	Time (mean)	Time (median)
-O2 Baseline	1015.8s	1015.0s
-O2 PGO+Vanilla HCS	961.3s	961.0s

- *Slight* code-size reduction while vanilla HCS already helps w/ performance

Idea 3: Adding a cold section.

Instead of putting extracted cold functions in the same binary section, keep all cold functions in a different section. → More compact, smaller section size for hot functions.



Left: Function size in `aio-posix.c` w/ HCS, right: without HCS

No significant performance gains on a `qemu` workload...

Setup: `qemu-x86_64-wholesystem`, measure time spent booting Ubuntu 16.04 image and running byte-unixbench benchmarks: pipe, spawn, context1, syscall, dhry2, each for 50,000 iterations.

	Time (mean across 6 runs)	icache miss rate	branch miss rate
-O2 Vanilla HCS	38.3379s (stddev: ±.13%)	1.952%	1.692%
-O2 HCS+Cold Sections	38.4339s (stddev: ±.18%)	1.936%	3.118%
-O2 PGO baseline	38.66s (stddev: ±.12%)	1.912%	3.150%

(Insignificant!)

But on the Z3 workload...

Setup: Everything compiled with `-O3` only. Compare vanilla Z3, Z3+HCS, and Z3+HCS+cold section, on SMTLib2 `QF_LIA/CAV2009` benchmark's 45-variable SMT instances (which are randomly generated conjunctions of $\mathcal{LA}(\mathbb{Z})$ inequalities).

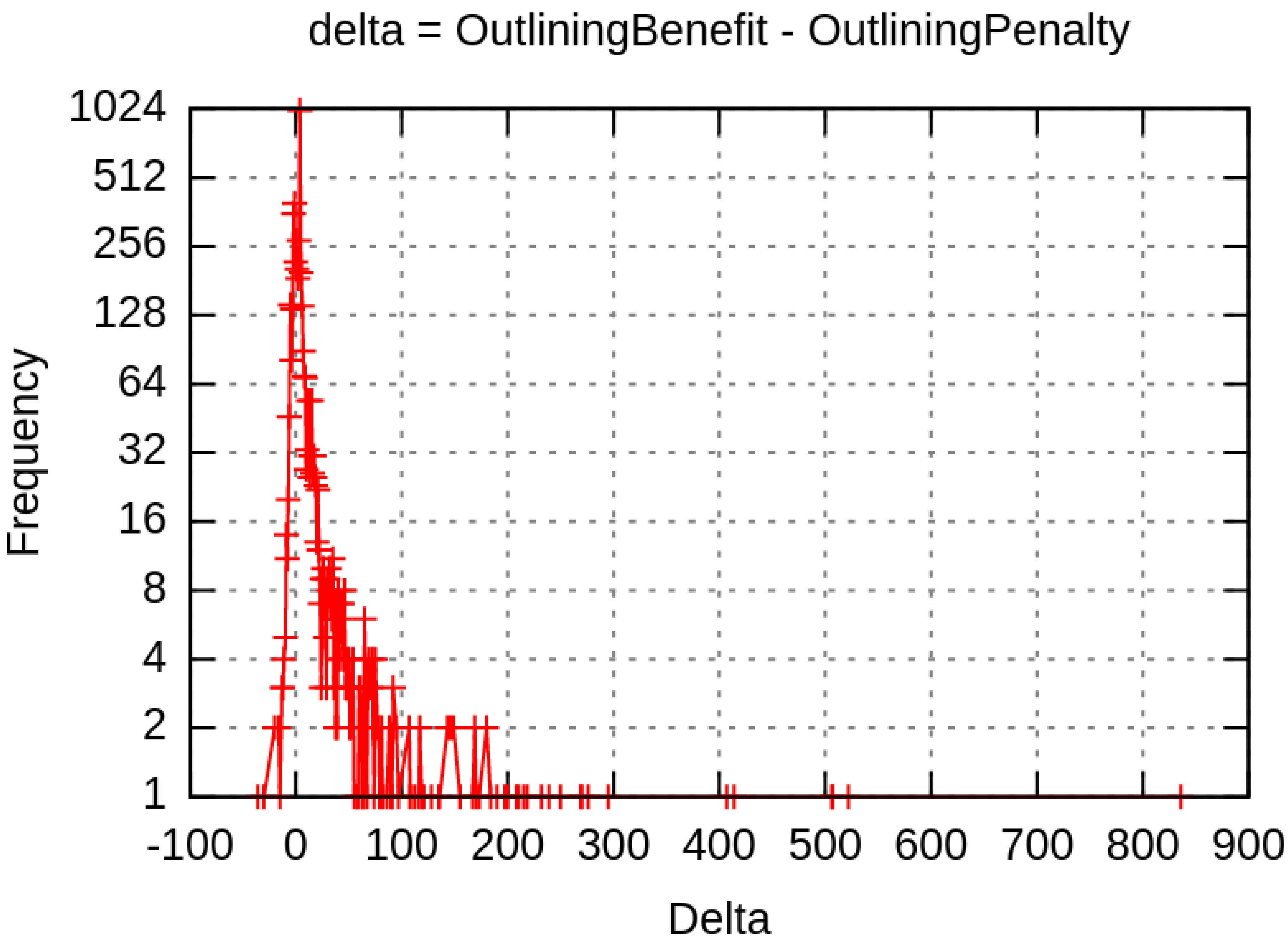
	Mean time (s)	Branch misse rate	icache misses	Pagefaults
Vanilla Z3	21.840±0.223	1.53%	18260045±0.19%	92494±4.63%
Z3+HCS	21.974±0.157	1.55%	22645047±0.40%	98606±4, 74%
Z3+HCS+ColdSec	21.590±0.132	1.49%	16709557±0.20%	90075±4.15%

(10 runs) ~1-2% faster than vanilla HCS/no HCS, ~4% less branch misses, ~9% less pagefaults, ~26% less icache misses

Idea 4: Tuning cost model

- For each cold region, the cost model HCS uses calculates a benefit score and a penalty score, and if their difference is positive, then it tries to split it.
- On firefox and qemu, we found basic blocks mostly come with small benefit-penalty differences, and decision around these small blocks manifest in code size differences.

Idea 4: Tuning cost model



Idea 4: Tuning cost model

	Opt Level	Size (incl. dynamic libraries)
D59715	-Os	2.184796592 GB
delta=5 HCS	-Os	2.206931464 GB
delta=-2 HCS	-O3	2.270277648 GB
delta=0 HCS	-O3	2.247788640 GB
D59715	-O3	2.243288440 GB
delta=2 HCS	-O3	2.259242024 GB
delta=5 HCS	-O3	2.270277648 GB
baseline	-O3	2.299546240 GB

Idea 4: Tuning cost model

- Calls for more fine-grained cost analysis. Brought by Vedant Kumar's patch (https://reviews.llvm.org/D59715, merged)
- Even with D59715 applied, still might have code size issues
  - Z3: 26.585Mb with HCS (5276 cold funcs) vs. 25.765Mb baseline
  - Less-aggressive splitting might help in this case

Concluding thoughts

Findings

- Not "plug-n-play": Not uniformly applicable across all applications, and results vary for different workloads. (sub-par results for postgresql/qemu)
- Performance-wise, HCS effective on software with large code sizes when everything can't fit neatly into icache (e.g. Firefox), or on ad-hoc workloads that have many branches and are cache-sensitive.
- Even for workloads in which using HCS is beneficial, requires some parameter tuning to get the best effect (e.g. tuning cost model / EH outlining).

Concluding thoughts

- *"But only 24 hours a day..."* So many opportunities, so little time.
- "Data-driven"-approach: Use insights from benchmarking open-source codebases to drive improvements
- Explore impact of different HCS parameters and tuning cost model on different code bases
- Using HCS with other passes, and impact of optimization ordering when scheduling HCS with other passes: inliner, MergeFunctions, machine function splitter

Challenges

1. Working with large open-source codebases: Involved compilation process in many applications (e.g. qemu, z3prover) not friendly to LLVM profiling by default.
2. Finding representative benchmarks that model real-world workloads (e.g. benchmarking Firefox vs. real-life web browsing).
3. Obtaining granular, explainable insights into how HCS affects the size/performance of final binary (e.g. looking at function call traces). ⟵ Often ad-hoc, time-consuming, laborious process.

Thank you & Feedback

Acknowledgements: This project was supported by a Google Summer of Code 2020 stipend. Thanks to Aditya Kumar, Rodrigo Rocha for mentoring me during GSoC 2020, and many other LLVM contributors for valuable feedback during patch reviews.

Slides at: **`tr5.org/~ruijie/hcs.pdf`**
More info: `https://tr5.org/~ruijie/gsoc20_hcs/index.xhtml`