World-readable doc. Recent as of early-March 2018.

ThinLTO for Chrome on Android

gbiv@chromium.org
http://crbug.com/807147
Doc status: WIP

Overview

ThinLTO (Thin Link-Time Optimization) is a feature that enables the linker to perform substantially more optimizations, such as cross-TU inlining and whole-program devirtualization, than usual. In short, it does this by emitting LLVM bitcode to object files instead of machine code, and running LLVM to optimize this bitcode one last time during link steps. In theory, this can help make Chrome leaner and faster.

ThinLTO is already on without optimizations for CFI builds of Chrome, which should be coming soon to an Android near you, and have been enabled for Chrome on Linux for a while.

Current status

Seems profitable, and simple to turn on.

CL: https://chromium-review.googlesource.com/c/chromium/src/+/919865

Summary

- Performance boost of thinlto -O3: ~+FIXME% on Speedometer; 1-7% performance improvement on various metrics in benchmarks that AFDO regressed
- Binary size increase of thinlto -O3 for libchrome.so: ~972KB; -O2 is 784KB
    - If we don't like this, it can apparently be tuned down, at the cost of performance.
- Performance boost of thinlto -O0: ~+1% on Speedometer
- Binary size **decrease** of thinlto -O0 for libchrome.so: ~???KB
- Nop for unofficial builds of Chrome (like AFDO, it's not enabled by default)
- For official builds of Chrome, link time goes through the roof, but that's potentially a good thing anyway

I mostly write off the build time increase because there's already an effort to make LTO with -O0 on by default. While LTO+-O0 doesn't seem to have as much of a build-time impact as -O$N, it's all "go do something else" levels of time, anyway.

It seems Goma always falls back on local actions for ThinLTO links.

ThinLTO Details

There are many details available online about ThinLTO, including very informative talks.

As noted in the summary, ThinLTO causes some build artifacts (e.g. object files) to hold LLVM bitcode instead of machine code. It then becomes the linker's job to invoke LLVM to optimize this and turn it into machine code before the linker can do much with the code. This potentially buys us a substantial amount of visibility in making optimizations. For example, consider:

```
// in a.h
struct Feature {
  virtual ~Feature() = default;
  virtual void RunFeature() = 0;
};

size_t GetTotalNumberOfFeatures();
std::unique_ptr<Feature> GetFooFeature();

// in a.cc
Feature *GetFooFeature() {
  struct MyFeature final : Feature {
    virtual void RunFeature() {
      // TODO(): Implement this feature by the time it's deprecated
    }
  };
  return new MyFeature();
}

size_t GetTotalNumberOfFeatures() { return 42; }

// in b.cc
#include "a.h"
int foo() {
  if (GetTotalNumberOfFeatures() > 42) {
    chrome::feedback_report::Report(
      "Too many features. Please remove three");
  }
  auto *f = GetFooFeature();
  f->RunFeature();
  delete f;
}
```

In a traditional compilation model, a.cc and b.cc would be compiled separately, meaning that `foo` would need to actually do something, since it can't know what `GetTotalNumberOfFeatures()` returns, nor what `GetFooFeature()` wants to hand back. Under

ThinLTO, it's theoretically possible (given an aggressive enough compiler) that foo() gets compiled into a nop, since the compiler has sufficient information to know that:

- `` `GetTotalNumberOfFeatures > 42` `` is always false
- `` `f->RunFeature` `` has a single call target (hence it can be turned into a direct call, then we'll realize that it's a direct call to a nop, so we can remove it)
- `` `f` `` was allocated with `` `new` `` just to be `` `delete`d ``, so we can remove the allocation entirely.

In addition, ThinLTO allows the compiler to perform new optimizations altogether.

ThinLTO has its own optimization level, independent of the optimization level of "regular" compilations. Hence, one may see references to "ThinLTO -O$N" here.

Why not Non-Thin LTO?

ThinLTO was predated by a feature called LTO. As the name implies, ThinLTO tries to be a thinner/lighter version of LTO. The primary difference between LTO and ThinLTO is that ThinLTO has less information available to it than LTO: LTO has a global view of *everything* in the program at all times, whereas ThinLTO cuts the program up based on its dependencies, and processes all of those slices in parallel.

In practice, this makes ThinLTO substantially faster than full LTO, and allows it to support incremental relinking, which cuts link times by almost 2 orders of magnitude on my machine. The downside is that most ThinLTO optimizations need to happen in a world that lacks a global view of the application. (ThinLTO summaries may/may not be a full remedy for this)

Potential issues

The issues we may see here are:

- Drastically increased link times, and (probably) linker memory usage
- Increased Chrome binary size (more inlining/optimization opportunities)
- Bugs from cross-module optimizations, meaning:
    - Existing bugs in Chrome that are exposed by cross-module optimizations (unlikely, especially given that ThinLTO has already been vetted, but not unheard of),
    - Bad interactions with existing infrastructure that assumes object files represent the "final" state of pieces of the binary (see below)

Binary size increase

The reason we're not already using ThinLTO on Android is simple: binary size increase. I'm told there are many knobs we can turn to bring the binary size increase down, while still gaining benefits from ThinLTO.

```
# Baseline: today's "official" build
$ du out/ARMWithAFDO/libchrome.so
42460

# ThinLTO -O3
$ du out/ARMWithAFDOAndThinLTO/libchrome.so
43432

# ThinLTO -O2
$ du out/ARMWithAFDOAndThinLTO/libchrome.so
43244

# ThinLTO -O1
$ du out/ARMWithAFDOAndThinLTO/libchrome.so
43248
```

So we end up with a bump of 784KB (-O2) or 972KB (-O3). This is presumably thanks to our setup that asserts that our profiles are "accurate". Turning just that (sample_profile_is_accurate) off makes Chrome around 5MB larger with ThinLTO, which is roughly the expected ThinLTO tax:

```
$ du out/ARMWithInaccurateAFDOAndThinLTO/libchrome.so
# ThinLTO -O2; -O3 is presumably higher.
47160
```

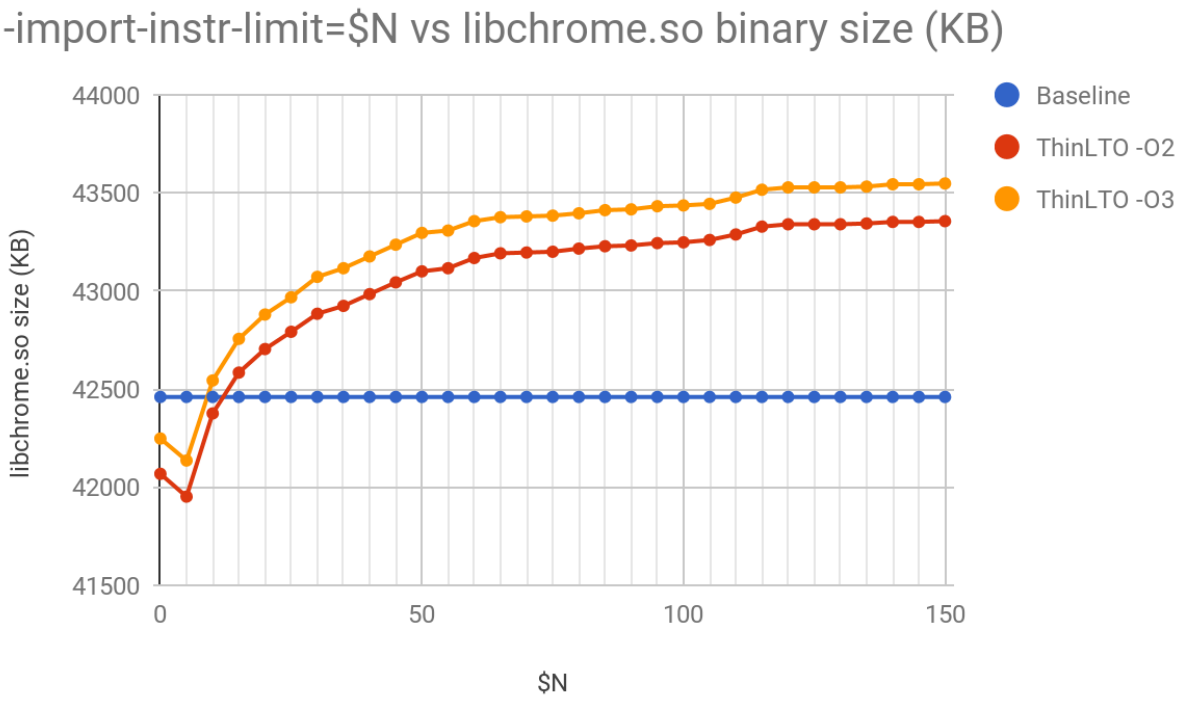
With CFI on, we see the following numbers:

```
# ThinLTO -O3
45584

# ThinLTO -O2
45412

# ThinLTO -O1
45420
```

I was not able to test the 'base' CFI configuration, because clang was segfaulting on my machine. I have not yet looked into why.

Size tuning

The primary knobs that seem to exist for tuning the size of a ThinLTO'ed binary seem to be optimization levels (-O0, -O1, …) and function import thresholds. The latter is tunable by passing `-Wl,-mllvm,-import-instr-limit=$N` to the linker, where N is a positive integer. If unspecified, the default value of N is 100. Folklore says that the smaller N is, the smaller your binary will be (but the fewer optimization opportunities we'll have). Numbers agree:



'Baseline' is what we have today, which has no ThinLTO, so $N doesn't affect it.

It's interesting to note that binary size is decreased for $N=0 and $N=5 (and, for ThinLTO -O2, $N=10). The performance implications of going this close to 0 are unclear to me, though worth exploring.

Link times

Tl;dr: For full relinks with ThinLTO, expect to spend 145x as much CPU and 1.5x-2x as much memory as as in a regular build. The CPU cost is parallelizable (across multiple threads, or multiple machines). In practice, ThinLTO on my dev box was able to eat almost 2100% CPU, and took 20x longer (wall time) than a regular link. I'd like to emphasize that this is with ThinLTO's job limit (which is currently set to 8) removed.

For your standard “edit a file, rebuild” cycle, expect to spend 1.5x as long to have a fully-built libchrome.so without goma. With goma, FIXME.

For a full rebuild from scratch, expect to spend around 5% more CPU time. Wall-time delta is presumably heavily machine-dependent, but a full ThinLTO build of libchrome.so took around 16% more wall time for me. Again, this is with no job limit.

Any commands mentioned were run three times: twice as a warm-up, once to collect the final numbers.

libchrome.so

All numbers were a ThinLTO -O3 build, since O3 is intended to spend the most time doing optimizations. An -O0 vs -O3 comparison will also be provided for reference, as soon as I can get LLVM to stop crashing on -O0. :)

For non-component links, `$run_link` is the actual link command (using clang++ as a wrapper) that gets run by `ninja libchrome.so`; the libchrome.so.rsp file that pairs with it was saved from an earlier link.

Full build

On a machine with 48 hardware threads, no Goma:
$ ninja -t clean && /usr/bin/time -v ninja -j55 libchrome.so

Without ThinLTO:

User time (seconds): 107,124.38
System time (seconds): 6,356.50
Percent of CPU this job got: 4633%
Elapsed (wall clock) time (h:mm:ss or m:ss): 40:49.17
Maximum resident set size (kbytes): 5,339,684

With ThinLTO:

User time (seconds): 111,465.00
System time (seconds): 7,798.45
Percent of CPU this job got: 4196%
Elapsed (wall clock) time (h:mm:ss or m:ss): 47:21.94
Maximum resident set size (kbytes): 7,472,232

Non-incremental

$ rm libchrome.so; /usr/bin/time -v $run_link

Without ThinLTO:

User time (seconds): 54.84
System time (seconds): 9.79
Percent of CPU this job got: 281%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:22.92
Maximum resident set size (kbytes): 5,340,188

With ThinLTO:

User time (seconds): 7737.96
System time (seconds): 1543.48
Percent of CPU this job got: 2074%
Elapsed (wall clock) time (h:mm:ss or m:ss): 7:27.49
Maximum resident set size (kbytes): 7,823,932

Non-incremental; thread limited

ThinLTO in Chrome is currently limited by default in terms of the number of parallel jobs it can run. The numbers here describe what we get when we have the limit.

$ rm libchrome.so; /usr/bin/time -v $run_link
Without ThinLTO:

Identical to “non-incremental”

With ThinLTO:

User time (seconds): 4866.90
System time (seconds): 133.63
Percent of CPU this job got: 737%
Elapsed (wall clock) time (h:mm:ss or m:ss): 11:17.66
Maximum resident set size (kbytes): 6,896,256

It’s odd to me that we see such a huge drop in total CPU time (and system time) here. I’m assuming some portion of it is that my machine is a dual-socket, and the 8-job link appeared to sit entirely on one physical CPU. I don’t know enough about back-of-the-napkin comp arch to speculate about whether that’s the main factor in the CPU time increase. :)

Incremental; nop

$ rm libchrome.so; /usr/bin/time -v $run_link

Without ThinLTO:

User time (seconds): 54.84
System time (seconds): 9.79
Percent of CPU this job got: 281%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:22.92
Maximum resident set size (kbytes): 5,340,188

With ThinLTO:
User time (seconds): 108.30
System time (seconds): 9.14
Percent of CPU this job got: 221%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:53.08
Maximum resident set size (kbytes): 6,630,884

Incremental with a small change

The idea of this was to get the total time of a rebuild of Chrome with a small change. Because of this, ninja was used. $touch_file was chosen at random; `sed`'s job is replacing "((attributes & READ_ONLY) == 0)" with "((attributes & $RANDOM) == 0)" in a function body.

```
$ touch_file=$CHROME/v8/src/property.cc
$ git -C $CHROME/v8 checkout -- $touch_file
$ sed -i "s/READ_ONLY/$RANDOM/" $touch_file
$ /usr/bin/time -v ninja libchrome.so
```

Without ThinLTO or goma:
User time (seconds): 77.30
System time (seconds): 16.13
Percent of CPU this job got: 204%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:45.73
Maximum resident set size (kbytes): 5,337,916

With ThinLTO or goma:
User time (seconds): 151.89
System time (seconds): 17.64
Percent of CPU this job got: 216%
Elapsed (wall clock) time (h:mm:ss or m:ss): 1:18.41
Maximum resident set size (kbytes): 6,634,792

ThinLTO O0 vs O3

No cache used. Just to get a feel for how long O3 takes on top of O0.

O3:
User time (seconds): 7982.27
System time (seconds): 1610.04

Percent of CPU this job got: 2073%
Elapsed (wall clock) time (h:mm:ss or m:ss): 7:42.71
Maximum resident set size (kbytes): 5,791,236

O0:

FIXME

Component builds

This I did a bit differently:

```
$ targets=( $(find ./ -name \*.so -maxdepth 1) )
$ rm -rf thinlto-cache ${targets[@]}
$ /usr/bin/time -v ninja -j1000 ${targets[@]}
```

With ThinLTO:

User time (seconds): 3802.70
System time (seconds): 140.75
Percent of CPU this job got: 887%
Elapsed (wall clock) time (h:mm:ss or m:ss): 7:24.24
Maximum resident set size (kbytes): 1,619,432

Without ThinLTO:

User time (seconds):
System time (seconds):
Percent of CPU this job got:
Elapsed (wall clock) time (h:mm:ss or m:ss):
Maximum resident set size (kbytes):

Goma

Goma falls back to local builds for these links, so no immediate problem seems to exist here. This may change in the future (b/73648878).

Making these work in Goma may/may not be straightforward, depending on whether it has a sane model for “single job which can flood the current machine with threads.” That’s a problem we can deal with later (and in an internal doc).

Infrastructure [WIP]

The key pieces we’re concerned about here are:

- Stack unwinding information, which might affect sampling profilers and crash dumps
- Orderfile generation

[[Section is left unfinished until perf numbers are gathered]]

Performance

Configurations

ThinLTO has four optimization modes: -O0, -O1, -O2, and -O3, which roughly equate to the different optimization levels offered by clang. I tested neither -O0 nor -O1, because -O0 is nonfunctional on my machine (at the time of writing), and -O1 produces a binary that's nearly the same size (+/-4KB) as -O2.

I did some preliminary performance testing on -O3, which costs around 200KB more than -O2. There were few clear performance differences between -O2 and -O3. Thus, I don't believe -O3 is a viable path.

Hence, I primarily tested three configurations:

- The standard official build configuration: "AFDO"
- The standard official build configuration with ThinLTO at -O2: "AFDOThinLTOO2"
- The standard official build configuration with ThinLTO at -O2, but with `-Wl,-mllvm,-import-instr-limit=25`: "AFDOThinLTOO2Threshold25". 25 was chosen arbitrarily, and can be tweaked depending on the tradeoffs we want to make. As mentioned in the size tuning section, values of 5 and 10 are probably very interesting.

The libchrome.so size for each configuration
"AFDO": 42,460KB
"AFDOThinLTOO2": 43,244KB (+784KB)
"AFDOThinLTOO2Threshold25": 42,788KB (+328KB)

Experimental setup

All experimentation was done on ToT with the ThinLTO CL applied.
I used a Marlin (Pixel XL) device, which was running Android's internal master branch.
I had a fan blowing on the Marlin device, which kept the thermals within acceptable limits (at least, the "cooldown" step of telemetry never seemed to happen with the fan applied).

Configurations:

“AFDO”: args.gn. This is our standard official build configuration.
“AFDOThinLTOO2”: args.gn. This is AFDO + ThinLTO at -O2.
“AFDOThinLTOO2Threshold25”: This is AFDOThinLTOO2, except we also passed “-Wl,-mllvm,-import-instr-limit=25” to the linker. This argument makes ThinLTO eat less binary size, potentially at the cost of performance. 25 was arbitrary; the default is 100.

All benchmarks were run via telemetry. All telemetry invocations were of the form:

```
ANDROID_SERIAL=$MARLIN ./tools/perf/run_benchmark --browser=android-chrome \
    “$benchmark” --output-dir=”$benchmark” --pageset-repeat=5 \
    --results-label=”$trial_name”
```

All benchmarks were run back-to-back, with no time for the phone to cool down. Browsers were run in the order:

- AFDO
- AFDOThinLTOO2
- AFDOThinLTOO2Threshold25

The set of benchmarks I used was the union of “everything I used to vet AFDO’s performance” and “every benchmark AFDO had regressions on (that resulted in a bug filed against me).” Namely:

- loading.mobile (**)
- rasterize_and_record_micro.top_25
- smoothness.gpu_rasterization.tough_path_rendering_cases
- smoothness.key_mobile_sites_smooth
- smoothness.tough_canvas_cases
- smoothness.tough_filters_cases
- smoothness.tough_path_rendering_cases
- speedometer
- speedometer2
- start_with_url.cold.startup_pages
- system_health.common_mobile
- thread_times.key_silk_cases
- thread_times.tough_scrolling_cases

(**) - loading.mobile was modified to remove the “3g connectivity” mode. I found this adds substantial noise, and does not seem to provide much (any?) signal for ThinLTO. The way I disabled it just gives us the effect of --pageset-repeat=10 for this benchmark.

...Together, I imagine this will give us good coverage, at least for the “should we seriously consider this” question. I ignored blink benchmarks, since the consensus seems to be that they’re too micro-benchmarky for the kind of change we’re trying to perform here.

Numbers

Raw telemetry results (Googler-only for the moment; sorry): results.tar.bz2. Please ignore the 'Value' column. I don't know what caused it to appear there.

Please note that all pagesets were repeated 5 times (via --pageset-repeat; exact command is in the Experimental setup section).

I plan to aggregate the telemetry results into a more easily grokable format, and hopefully will be able to put some of that here.

In lieu of that, a summary of each benchmark result follows. If it's not explicitly stated, I'm comparing the AFDOThinLTOO2Threshold25 and AFDO configs, since I think that's the most interesting case for us. In each summary, the metrics I focused on are selected on based on what sounds good. If someone knows what specifically to care about, please let me know. Otherwise, I'm not going to waste your time going through every site's performance on every metric.

In all cases, the first image in each section dictates the column order. Be careful around start_with_url.cold.startup_pages: it flips the order on you for reasons unknown to me. I ran all of these in precisely the same order.

I abbreviate AFDOThinLTOO2Threshold25 to Threshold25 for my sanity. :)

loading.mobile

Name ▲	AFDOThinLTOO2Threshold25	Value	AFDO	AFDOThinLTOO2
benchmark_total_duration	(missing)	6,317,921.999 ms	(missing)	(missing)
▶ cpuTimeToFirstMeaningfulPaint	-5.613%	(missing)	734.395 ms	-4.816%
▶ timeToFirstContentfulPaint	-2.676%	(missing)	662.480 ms	-3.161%
▶ timeToFirstCpuIdle	-3.581%	(missing)	1,898.842 ms	-3.398%
▶ timeToFirstMeaningfulPaint	-3.874%	(missing)	911.426 ms	-3.198%
▶ timeToFirstPaint	-3.142%	(missing)	640.234 ms	-3.362%
▶ timeToInteractive	-3.581%	(missing)	1,898.842 ms	-3.398%
▶ timeToOnload	-4.226%	(missing)	1,342.363 ms	-2.792%

Drilling down into cpuTimeToFirstMeaningfulPaint, there are three reported regressions (two for ~4%, one at ~12%). I glanced at timeToFirstContentfulPaint; no real outliers seemed to exist.

rasterize_and_record_micro.top_25

Name	AFDOThinLTOO2Threshold25	Value	AFDO	AFDOThinLTOO2
benchmark_total_duration	(missing)	4,361,132.964 ms	(missing)	(missing)
▶ pixels_rasterized	±0.000%	(missing)	2,926,933.333	±0.000%
▶ pixels_rasterized_non_solid	±0.000%	(missing)	2,619,733.333	±0.000%
▶ pixels_recorded	±0.000%	(missing)	9,142,115.708	±0.000%
▶ rasterize_time	-5.602%	(missing)	11.239 ms	-5.850%
▶ record_time	-9.192%	(missing)	0.516 ms	-8.950%
▶ record_time_caching_disabled	-4.735%	(missing)	19.559 ms	-4.955%
▶ record_time_construction_disabled	-5.898%	(missing)	5.804 ms	-5.855%
▶ record_time_painting_disabled	-4.175%	(missing)	6.881 ms	-2.864%
▶ record_time_partial_invalidation_ms	-7.586%	(missing)	0.533 ms	-7.653%
▶ record_time_subsequence_caching_dis…	-5.446%	(missing)	5.386 ms	-4.312%
▶ viewport_picture_size	+0.104%	(missing)	137.4 KiB	-0.077%

Looking at rasterize_time and record_time, record_time has a single 0.5% regression (googleplus.html). This regression (0.006ms) is well within 1stdev (0.065ms) of AFDO's result.

smoothness.gpu_rasterization.tough_path_rendering_cases

Name	AFDOThinLTOO2Threshold25	Value	AFDO	AFDOThinLTOO2
benchmark_total_duration	(missing)	622,178.689 ms	(missing)	(missing)
first_gesture_scroll_update_latency	(empty)	(missing)	(empty)	(empty)
frame_time_discrepancy	+2.458%	(missing)	1,095.103 ms	-7.088%
frame_times	-2.188%	(missing)	44.235 ms	-3.959%
mean_frame_time	-2.819%	(missing)	91.840 ms	-3.850%
mean_pixels_approximated	+NaN%	(missing)	0.000%	+NaN%
mean_pixels_checkerboarded	+NaN%	(missing)	0.000%	+NaN%
percentage_smooth	+10.281%	(missing)	15.135	+17.585%
queueing_durations	-15.271%	(missing)	0.968 ms	-1.848%

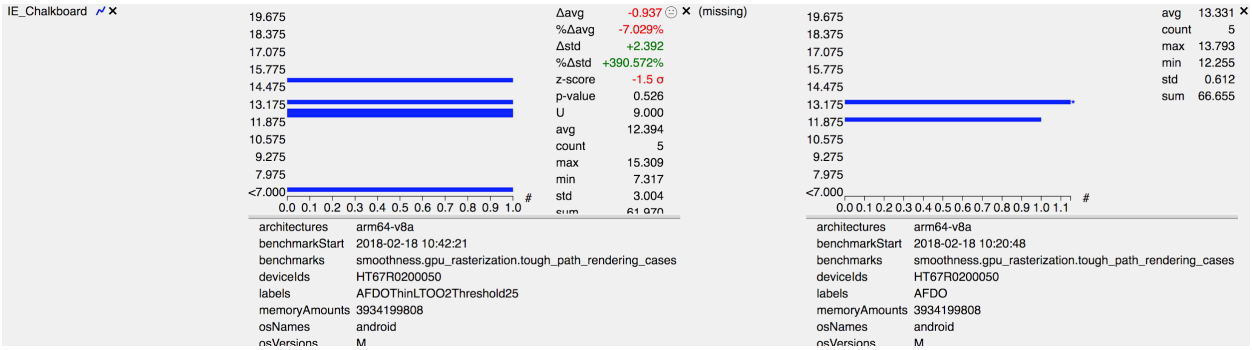
Frame_time_discrepancy has regressions of IE_Chalkboard by 6.3% (or +203.533ms; stdev of 508ms) and MotionMark_Canvas_Stroke_Shapes by 9.0% (+50ms; stdev of 150ms). In general, frame_time_discrepancy seems very noisy, so I'm going to ignore it.

Percentage_smooth is a more interesting story:

▼ percentage_smooth	+10.281%	(missing)	15.135	+17.585%
GUIMark_Vector_Chart_Test	+15.823%	(missing)	46.792	+18.487%
IE_Chalkboard	-7.029%	(missing)	13.331	+6.324%
MotionMark_Canvas_Fill_Shapes	+∞%	(missing)	0.000	+∞%
MotionMark_Canvas_Stroke_Shapes	-100.000%	(missing)	0.417	+192.000%

...MotionMark_*'s percentage_smooth metrics are all pretty inconsistent. Their most common value is '0' across the board.

IE_Chalkboard is better, but still quite noisy. AFDO's stdev here is 0.612, AFDOThinLTOO2's is 1.226, and Threshold25's is 3.004 (!). IE_Chalkboard seems to be a loss for Threshold25 because there's a single outlier score of 7.006:



GUIMark_Vector_Chart_Test is a very clear win for ThinLTO.

smoothness.key_mobile_sites_smooth

Name ▲	AFDOThinLTOO2Threshold25	Value	AFDO	AFDOThinLTOO2
benchmark_total_duration	(missing)	4,730,159.723 ms	(missing)	(missing)
first_gesture_scroll_update_latency	+2.570%	(missing)	24.341 ms	+0.200%
frame_time_discrepancy	+0.294%	(missing)	75.418 ms	+1.182%
frame_times	+0.056%	(missing)	17.456 ms	+0.075%
input_event_latency	-0.806%	(missing)	17.308 ms	-1.009%
input_event_latency_discrepancy	-0.139%	(missing)	218.487 ms	-1.644%
main_thread_scroll_latency	-6.983%	(missing)	33.660 ms	-7.961%
main_thread_scroll_latency_discrepancy	-3.211%	(missing)	666.564 ms	-7.675%
mean_frame_time	-0.008%	(missing)	17.503 ms	-0.027%
mean_input_event_latency	-1.000%	(missing)	17.418 ms	-1.162%
mean_main_thread_scroll_latency	-2.566%	(missing)	39.376 ms	-2.566%
mean_pixels_approximated	+3.736%	(missing)	0.123%	-15.165%
mean_pixels_checkerboarded	+2.306%	(missing)	1.430%	-11.435%
percentage_smooth	+0.482%	(missing)	57.440	+0.232%
queueing_durations	-5.045%	(missing)	0.734 ms	-5.896%

percentage_smooth: Most differences are within 2% either way. Notable (> 2% on average) potential regressions are:

- wowwiki.com (-2.7% total smoothness on average for Threshold25), though the stdev is 3.7% smoothness.
- m.youtube.com/watch?v=... (-3% total smoothness on average for Threshold25), though stdev is 5.4% smoothness.

mean_pixels_*: most metrics are 0% here, so the few non-zero metrics are disproportionately weighted. There's also a lot of noise (boingboing.net sees a 732% regression, thanks to a single outlier).

smoothness.tough_canvas_cases

Name ▲	AFDOThinLTOO2Threshold25	Value	AFDO	AFDOThinLTOO2
benchmark_total_duration	(missing)	4,116,509.333 ms	(missing)	(missing)
first_gesture_scroll_update_latency	(empty)	(missing)	(empty)	(empty)
frame_time_discrepancy	-1.139%	(missing)	246.059 ms	+1.809%
frame_times	-2.041%	(missing)	31.080 ms	-2.844%
input_event_latency	(empty)	(missing)	(empty)	(empty)
input_event_latency_discrepancy	(empty)	(missing)	(empty)	(empty)
main_thread_scroll_latency	(empty)	(missing)	(empty)	(empty)
main_thread_scroll_latency_discrepancy	(empty)	(missing)	(empty)	(empty)
mean_frame_time	-2.390%	(missing)	109.951 ms	-2.726%
mean_input_event_latency	(empty)	(missing)	(empty)	(empty)
mean_main_thread_scroll_latency	(empty)	(missing)	(empty)	(empty)
mean_pixels_approximated	+NaN%	(missing)	0.000%	+NaN%
mean_pixels_checkerboarded	+NaN%	(missing)	0.000%	+NaN%
percentage_smooth	+5.378%	(missing)	24.920	+3.298%
queueing_durations	-4.059%	(missing)	2.968 ms	-2.670%

frame_time_discrepancy: The stdev for many of these is > 10% of the average score. Ignoring.

frame_times: Lots of high-confidence improvements for both ThinLTOs. canvas2d_balls_common/bouncing_balls.html saw a 4% (61ms) regression with both ThinLTOs; stdev was around 70-80ms, so this is probably significant. No other regressions > 1% for Threshold25.

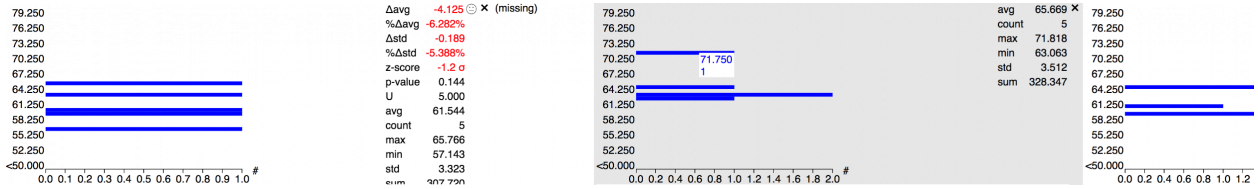
percentage_smooth:

../../../chrome/test/data/perf/canvas_b…		+NaN%	(missing)	0.000	+NaN%
http://geoapis.appspot.com/agdnZW9…	(empty)		(missing)	(empty)	(empty)
http://hakim.se/experiments/html5/ma…		-50.164%	(missing)	0.530	-24.833%
http://ie.microsoft.com/testdrive/Grap…		+31.408%	(missing)	3.317	+37.034%
http://ie.microsoft.com/testdrive/Grap…		+12.442%	(missing)	25.431	+12.035%
http://ie.microsoft.com/testdrive/Grap…		+2.400%	(missing)	41.438	+7.875%
http://ie.microsoft.com/testdrive/Perfo…		+9.984%	(missing)	70.820	+5.212%
http://ie.microsoft.com/testdrive/Perfo…		-0.759%	(missing)	64.872	+0.172%
http://ie.microsoft.com/testdrive/Perfo…		+5.982%	(missing)	63.737	-0.585%
http://ie.microsoft.com/testdrive/Perfo…		+49.222%	(missing)	0.978	-14.695%
http://jarrodoverson.com/static/demos…		+NaN%	(missing)	0.000	+NaN%
http://mix10k.visitmix.com/Entry/Detai…		+17.820%	(missing)	5.067	+11.356%
http://runway.countlessprojects.com/p…		-6.282%	(missing)	65.669	-4.760%
http://spielzeugz.de/html5/liquid-partic…		+24.668%	(missing)	0.530	-25.497%
http://themaninblue.com/experiment/…		+213.994%	(missing)	3.568	+253.026%
http://www.chiptune.com/starfield/star…		+5.914%	(missing)	61.957	+11.996%
http://www.craftymind.com/factory/gui…		+17.651%	(missing)	51.778	+5.505%
http://www.effectgames.com/demos/c…		+2.894%	(missing)	58.314	+1.783%
http://www.kevs3d.co.uk/dev/canvask…		+6.291%	(missing)	61.780	+1.740%
http://www.megidish.net/awjs/		+NaN%	(missing)	0.000	+NaN%
http://www.smashcat.org/av/canvas_t…		-3.601%	(missing)	57.687	-6.826%
tough_canvas_cases/canvas_toBlob.…		+3.282%	(missing)	71.011	-1.183%
tough_canvas_cases/canvas-animati…		+11.232%	(missing)	42.858	+12.779%
tough_canvas_cases/canvas-font-cyc…		+NaN%	(missing)	0.000	+NaN%
tough_canvas_cases/canvas2d_balls…		+NaN%	(missing)	0.000	+NaN%
tough_canvas_cases/canvas2d_balls…		+0.613%	(missing)	71.024	-4.256%

(Note that some test-cases are cut off due to just being NaN%)

themaninblue's 213% improvement is hilarious. And it doesn't appear to be a single outlier driving it up: the maximum value that AFDO gets is 6.225, whereas both ThinLTOs hit > 19. In any case, the stdev here is up to 5.7pts, so…

The only meaningful regressions I can see here are countlessprojects.com and smashcat.org. Countlessprojects has a single outlier driving it up for AFDO (excuse the cropping).



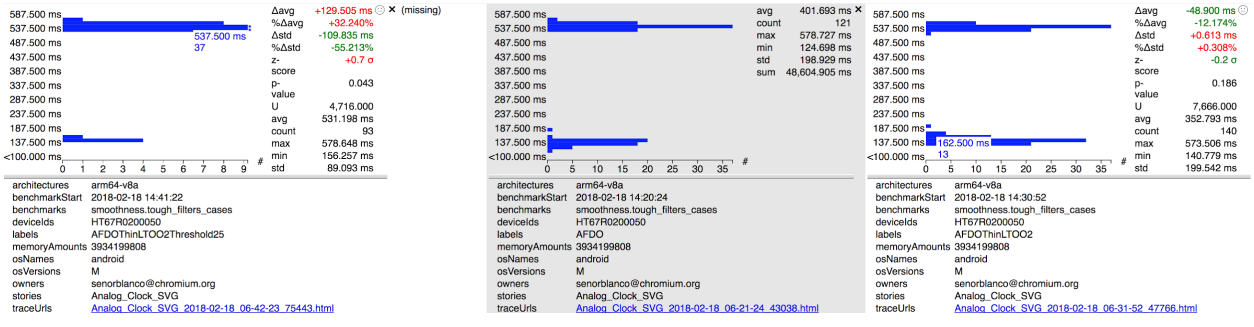
Smoothness.tough_filters_cases

Name ▲	AFDOThinLTOO2Threshold25	Value	AFDO	AFDOThinLTOO2
benchmark_total_duration	(missing)	604,670.842 ms	(missing)	(missing)
▶ first_gesture_scroll_update_latency	(empty)	(missing)	(empty)	(empty)
▶ frame_time_discrepancy	-36.619%	(missing)	1,211.398 ms	-30.757%
▶ frame_times	+0.939%	(missing)	68.817 ms	-1.085%
▶ mean_frame_time	+13.304%	(missing)	228.745 ms	+2.378%
▶ mean_pixels_approximated	+NaN%	(missing)	0.000%	+NaN%
▶ mean_pixels_checkerboarded	+NaN%	(missing)	0.000%	+NaN%
▶ percentage_smooth	-7.188%	(missing)	18.698	-5.681%
▶ queueing_durations	-4.113%	(missing)	6.268 ms	-7.361%

Digging into frame_times: Analog_Clock_SVG seems to be hurt the most here:

▼ frame_times	+0.939%	(missing)	68.817 ms	-1.085%
Analog_Clock_SVG	+32.240%	(missing)	401.693 ms	-12.174%
Filter_Terrain_SVG	-0.698%	(missing)	219.385 ms	-0.249%
IE_PirateMark	-0.958%	(missing)	21.344 ms	-1.166%
MotionMark_Focus	+2.862%	(missing)	255.041 ms	+3.493%

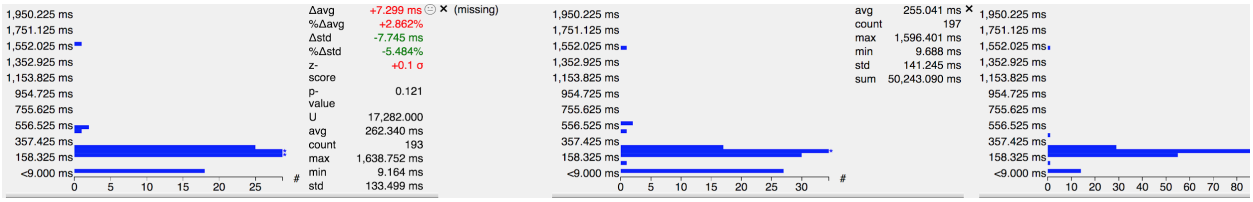
However, the individual metrics seem quite noisy:



Note that the upper bars are inconsistent in number; Threshold25 has 88 metrics > 500ms, AFDO has 75, and AFDOThinLTOO2 has 69. I'll retry this with --pageset-repeat=20 and see if I can get a better grip on what's going on.

Filter_Terrain_SVG isn't obviously noisy, so we've got that going for us, which is nice.

MotionMark_Focus at least looks somewhat more consistent with its outlier:



...So I'm assuming that has a chance of being significant, but I suppose we'll see when the higher pageset run comes back...

smoothness.tough_path_rendering_cases

Name ▲	AFDOThinLTOO2Threshold25	Value	AFDO	AFDOThinLTOO2
benchmark_total_duration	(missing)	626,378.681 ms	(missing)	(missing)
▶ first_gesture_scroll_update_latency	(empty)	(missing)	(empty)	(empty)
▶ frame_time_discrepancy	+0.856%	(missing)	1,177.602 ms	-11.080%
▶ frame_times	-4.985%	(missing)	60.731 ms	-4.190%
▶ mean_frame_time	-6.169%	(missing)	120.744 ms	-5.333%
▶ mean_pixels_approximated	+NaN%	(missing)	0.000%	+NaN%
▶ mean_pixels_checkerboarded	+NaN%	(missing)	0.000%	+NaN%
▶ percentage_smooth	+25.076%	(missing)	12.673	+17.595%
▶ queueing_durations	-14.061%	(missing)	1.300 ms	+1.851%

frame_times is entirely composed of statistically significant improvements for Threshold25 and AFDOThinLTOO2. All of them are in the range 3.5%-8%. Mean_frame_time also sees 0 regressions, though all the faces are ':|' instead of ':)'.

We have a single percentage_smooth regression for Threshold25: MotionMark_Canvas_Stroke_Shapes. The stdev is near 1.0, and the regression is 0.042. Similarly, IE_Chalkboard's readings -- where we do better with Threshold25 and worse with AFDOThinLTOO2 -- could be in the noise.

speedometer

Name ▲	AFDOThinLTOO2Threshold25	Value	AFDO	AFDOThinLTOO2
AngularJS-TodoMVC	-0.161%	(missing)	3,224.080 ms	-1.675%
BackboneJS-TodoMVC	-4.272%	(missing)	2,004.333 ms	-5.237%
benchmark_total_duration	(missing)	438,371.759 ms	(missing)	(missing)
EmberJS-TodoMVC	-1.925%	(missing)	4,339.220 ms	-2.823%
FlightJS-TodoMVC	-1.939%	(missing)	1,540.100 ms	-2.087%
jQuery-TodoMVC	-5.662%	(missing)	2,906.900 ms	-8.870%
React-TodoMVC	-0.119%	(missing)	4,581.207 ms	-2.911%
RunsPerMinute	+2.305%	(missing)	21.430	+4.246%
Total	-2.198%	(missing)	19,593.613 ms	-4.031%
VanillaJS-TodoMVC	-5.658%	(missing)	997.773 ms	-8.518%

No comment, so have a cat instead.

speedometer2

Name	AFDOThinLTOO2Threshold25	Value	AFDO	AFDOThinLTOO2
Angular2-TypeScript-TodoMVC	-6.535%	(missing)	759.267 ms	-7.883%
AngularJS-TodoMVC	-3.967%	(missing)	3,350.067 ms	-3.336%
BackboneJS-TodoMVC	-4.693%	(missing)	979.653 ms	-5.221%
benchmark_total_duration	(missing)	788,307.348 ms	(missing)	(missing)
Elm-TodoMVC	-0.321%	(missing)	2,522.107 ms	-5.095%
EmberJS-Debug-TodoMVC	-3.544%	(missing)	10,882.740 ms	-3.869%
EmberJS-TodoMVC	-1.388%	(missing)	2,774.753 ms	-3.827%
Flight-TodoMVC	-3.682%	(missing)	1,425.707 ms	-3.670%
Inferno-TodoMVC	-5.041%	(missing)	8,197.867 ms	-6.116%
jQuery-TodoMVC	-6.528%	(missing)	3,977.840 ms	-8.616%
Preact-TodoMVC	-9.055%	(missing)	234.053 ms	-9.642%
React-Redux-TodoMVC	-3.321%	(missing)	3,690.313 ms	-3.019%
React-TodoMVC	-3.611%	(missing)	1,813.107 ms	-5.219%
RunsPerMinute	+5.143%	(missing)	11.921	+6.244%
Total	-4.103%	(missing)	43,754.707 ms	-5.057%
Vanilla-ES2015-Babel-Webpack-TodoM…	-6.066%	(missing)	965.540 ms	-5.825%
Vanilla-ES2015-TodoMVC	-6.993%	(missing)	984.513 ms	-7.104%
VanillaJS-TodoMVC	-5.204%	(missing)	834.653 ms	-5.851%
VueJS-TodoMVC	-7.922%	(missing)	362.527 ms	-9.417%

No comment.

start_with_url.cold.startup_pages

The column order here is different than every other benchmark. I have no clue why, since I ran this in the same (browser) order as everything else.

Also, please note that this benchmark is likely to be heavily influenced by the orderfile, which might not be what we'd generate for ThinLTO builds. In prior experiments (for AFDO), I found that the orderfile materially impacts the performance of benchmarks, so I did leave it enabled.

Name	Value	AFDO	AFDOThinLTOO2	AFDOThinLTOO2Threshold25
benchmark_total_duration	363,243.768 ms	(missing)	(missing)	(missing)
▼foreground_tab_load_complete	(missing)	3,055.510 ms	-1.628%	+0.818%
about:blank	(missing)	2,162.779 ms	-0.075%	+0.175%
http://bbc.co.uk	(missing)	3,948.241 ms	-2.479%	+1.170%
foreground_tab_request_start	(missing)	2,026.502 ms	-0.011%	+0.282%
▼messageloop_start_time	(missing)	1,664.490 ms	+0.107%	+0.054%
about:blank	(missing)	1,680.625 ms	-0.563%	-0.622%
http://bbc.co.uk	(missing)	1,648.354 ms	+0.791%	+0.743%

foreground_tab_request_start results are for bbc.co.uk.

All improvements/regressions were within ½ a stdev of AFDO (except for AFDOThinLTOO2's foreground_tab_load_complete for bbc.co.uk, which was around ⅔ * stdev(AFDO))

system_health.common_mobile

(Please note that this benchmark was manually run 5 times, since it doesn't seem to respect --pageset-repeat)

Name	AFDOThinLTOO2Threshold25	Value	AFDO	AFDOThinLTOO2
benchmark_total_duration	(missing)	2,778,609.822 ms	(missing)	(missing)
▶ browser_accessibility_events	(empty)	(missing)	(empty)	(empty)
▶ clock_sync_latency_linux_clock_monot…	-1.169%	(missing)	4.251 ms	+0.871%
▶ cpu_time_percentage	-2.113%	(missing)	52.682%	-2.161%
▶ cpuPercentage:all_processes:all_threa…	-2.091%	(missing)	44.638%	-2.508%
▶ cpuPercentage:all_processes:all_threa…	-1.080%	(missing)	73.931%	-1.266%
▶ cpuPercentage:browser_process:all_thr…	-1.783%	(missing)	15.311%	-2.170%
▶ cpuPercentage:browser_process:CrBro…	-2.181%	(missing)	7.089%	-2.569%
▶ cpuPercentage:gpu_process:all_thread…	-0.744%	(missing)	5.140%	-1.747%
▶ cpuPercentage:renderer_processes:all_…	-2.572%	(missing)	24.188%	-2.884%
▶ cpuPercentage:renderer_processes:Cr…	-2.352%	(missing)	16.970%	-2.707%
▶ cpuTime:all_processes:all_threads:all_s…	-2.911%	(missing)	19,221.558 ms	-2.928%
▶ cpuTime:all_processes:all_threads:Loa…	-2.955%	(missing)	9,170.771 ms	-3.013%
▶ cpuTime:browser_process:all_threads:a…	-2.498%	(missing)	6,173.942 ms	-2.315%
▶ cpuTime:browser_process:CrBrowserM…	-3.285%	(missing)	2,762.213 ms	-2.995%
▶ cpuTime:gpu_process:all_threads:all_st…	-1.403%	(missing)	2,131.782 ms	-1.733%
▶ cpuTime:renderer_processes:all_thread…	-3.440%	(missing)	10,915.834 ms	-3.508%
▶ cpuTime:renderer_processes:CrRender…	-3.234%	(missing)	7,772.344 ms	-3.399%
▶ cpuTimeToFirstMeaningfulPaint	-5.130%	(missing)	672.869 ms	-3.681%
▶ interactive:500ms_window:renderer_eqt	-1.584%	(missing)	305.278 ms	-4.066%
▶ interactive:500ms_window:renderer_eqt…	-19.972%	(missing)	26.170 ms	-33.333%
▶ peak_event_rate	-13.962%	(missing)	21,539	-13.034%
▶ peak_event_size_rate	-13.792%	(missing)	4.5 MiB	-13.043%
▶ render_accessibility_events	(empty)	(missing)	(empty)	(empty)
▶ render_accessibility_locations	(empty)	(missing)	(empty)	(empty)
▶ timeToFirstContentfulPaint	-2.787%	(missing)	650.896 ms	-1.817%
▶ timeToFirstCpuIdle	-2.520%	(missing)	1,223.085 ms	-2.374%
▶ timeToFirstMeaningfulPaint	-3.638%	(missing)	841.379 ms	-2.912%
▶ timeToFirstPaint	-3.285%	(missing)	631.373 ms	-2.075%
▶ timeToInteractive	-2.520%	(missing)	1,223.085 ms	-2.374%
▶ timeToOnload	-3.291%	(missing)	1,937.379 ms	-3.604%
▶ total:500ms_window:renderer_eqt	-1.956%	(missing)	432.300 ms	-4.514%
▶ total:500ms_window:renderer_eqt_cpu	-2.491%	(missing)	86.777 ms	-0.554%
▶ trace_import_duration	+2.319%	(missing)	3,271.785 ms	+2.888%
▶ trace_size	+0.153%	(missing)	15.0 MiB	+0.084%

thread_times.key_silk_cases

Name	AFDOThinLTOO2Threshold25	Value	AFDO	AFDOThinLTOO2
benchmark_total_duration	(missing)	3,565,833.445 ms	(missing)	(missing)
mean_frame_time_renderer_compositor	-0.526%	(missing)	28.147 ms	-0.363%
tasks_per_frame_browser	+0.770%	(missing)	7.413	+0.764%
tasks_per_frame_GPU	+0.662%	(missing)	6.522	+0.819%
tasks_per_frame_GPU_transfer	+NaN%	(missing)	0.000	+NaN%
tasks_per_frame_IO	-0.342%	(missing)	18.625	-0.116%
tasks_per_frame_other	-1.411%	(missing)	0.335	+0.431%
tasks_per_frame_raster	+0.153%	(missing)	5.203	+0.146%
tasks_per_frame_renderer_compositor	+2.149%	(missing)	6.288	+2.126%
tasks_per_frame_renderer_main	+3.888%	(missing)	2.205	+3.620%
tasks_per_frame_total_all	+0.560%	(missing)	46.591	+0.668%
tasks_per_frame_total_fast_path	+0.442%	(missing)	38.848	+0.572%
thread_browser_cpu_time_per_frame	-3.609%	(missing)	2.779 ms	-3.255%
thread_GPU_cpu_time_per_frame	-0.543%	(missing)	6.146 ms	-0.983%
thread_GPU_transfer_cpu_time_per_fr…	+NaN%	(missing)	0.000 ms	+NaN%
thread_IO_cpu_time_per_frame	-2.217%	(missing)	2.744 ms	-2.642%
thread_other_cpu_time_per_frame	+1.212%	(missing)	0.366 ms	+0.491%
thread_raster_cpu_time_per_frame	-2.393%	(missing)	1.811 ms	-2.308%
thread_renderer_compositor_cpu_time…	-5.839%	(missing)	3.129 ms	-5.856%
thread_renderer_main_cpu_time_per_fr…	-6.130%	(missing)	4.177 ms	-4.917%
thread_total_all_cpu_time_per_frame	-3.178%	(missing)	21.153 ms	-3.083%
thread_total_fast_path_cpu_time_per_fr…	-2.549%	(missing)	14.799 ms	-2.748%

Thread_browser_cpu_time_per_frame sees many statistically significant improvements for both ThinLTO variants. There were two individual regressions reported, but each was well within ¼ a stdev of the benchmark.

Thread_other_cpu_time_per_frame is interesting. It shows intermixed regressions/improvements, but also only represents 1.7% of the time we spend on each frame, on average:

▼thread_other_cpu_time_per_frame	+1.212%	(missing)	0.366 ms	+0.491%
font_wipe.html	-2.524%	(missing)	0.720 ms	-0.990%
http://groupcloned.com/test/plain/list-r…	+1.326%	(missing)	0.182 ms	+0.184%
http://groupcloned.com/test/plain/stick…	-8.370%	(missing)	0.038 ms	-7.860%
http://jsbin.com/UVIgUTa/38/quiet	+4.993%	(missing)	0.022 ms	-0.463%
http://jsfiddle.net/3yDKh/15/show/	-9.082%	(missing)	0.004 ms	+0.739%
http://jsfiddle.net/3yDKh/16/show/	+27.955%	(missing)	0.004 ms	+17.458%
http://jsfiddle.net/bNp2h/3/show/	-65.657%	(missing)	0.106 ms	-26.698%
http://jsfiddle.net/cKB9D/7/show/	-3.447%	(missing)	0.103 ms	-61.203%
http://jsfiddle.net/jx5De/14/show/	-34.302%	(missing)	0.096 ms	-18.885%
http://jsfiddle.net/R8DX9/4/show/	-0.500%	(missing)	0.004 ms	+3.000%
http://jsfiddle.net/rF9Gh/7/show/	+20.328%	(missing)	0.003 ms	+5.944%
http://jsfiddle.net/TLXLu/3/show/	-9.127%	(missing)	0.129 ms	-70.540%
http://jsfiddle.net/ugkd4/10/show/	+6.511%	(missing)	0.207 ms	-2.075%
http://jsfiddle.net/vBQHH/11/show/	+94.021%	(missing)	0.039 ms	+165.447%
http://jsfiddle.net/xLuvC/1/show/	-2.855%	(missing)	0.108 ms	+32.655%
http://mobile-news.sandbox.google.c…	+4.310%	(missing)	0.296 ms	+2.962%
http://mobile-news.sandbox.google.c…	-7.163%	(missing)	0.678 ms	-3.083%
http://wiltzius.github.io/shape-shifter/	-2.146%	(missing)	0.548 ms	-3.554%
http://www.google.com/#q=google	+3.065%	(missing)	2.278 ms	+4.483%
https://www.google.com/search?hl=e…	+5.932%	(missing)	2.157 ms	+3.927%
inbox_app.html?stress_hidey_bars	+12.577%	(missing)	0.315 ms	+7.848%
inbox_app.html?toggle_drawer	+2.887%	(missing)	0.924 ms	+0.353%
infinite_scrolling.html	-2.111%	(missing)	0.244 ms	-3.977%
list_animation_simple.html	-27.399%	(missing)	0.005 ms	-33.211%
pushState.html	-4.478%	(missing)	0.178 ms	-6.075%
silk_finance.html	+0.019%	(missing)	0.140 ms	-0.841%

Thread_total_all_cpu_time_per_frame shows many statistically significant improvements for both ThinLTO variants:

▼thread_total_all_cpu_time_per_frame	-3.178%	(missing)	21.153 ms	-3.083%
font_wipe.html	-2.092%	(missing)	19.062 ms	-2.868%
http://groupcloned.com/test/plain/list-r…	-4.049%	(missing)	18.293 ms	-4.578%
http://groupcloned.com/test/plain/stick…	-3.846%	(missing)	16.401 ms	-3.371%
http://jsbin.com/UVIgUTa/38/quiet	-5.873%	(missing)	15.099 ms	-7.261%
http://jsfiddle.net/3yDKh/15/show/	-2.746%	(missing)	15.087 ms	-3.981%
http://jsfiddle.net/3yDKh/16/show/	-5.730%	(missing)	14.383 ms	-5.769%
http://jsfiddle.net/bNp2h/3/show/	-4.059%	(missing)	23.269 ms	-3.690%
http://jsfiddle.net/cKB9D/7/show/	-2.539%	(missing)	20.410 ms	-3.402%
http://jsfiddle.net/jx5De/14/show/	-3.171%	(missing)	17.442 ms	-3.648%
http://jsfiddle.net/R8DX9/4/show/	-1.571%	(missing)	16.527 ms	-3.008%
http://jsfiddle.net/rF9Gh/7/show/	+0.584%	(missing)	16.873 ms	-0.281%
http://jsfiddle.net/TLXLu/3/show/	-4.526%	(missing)	20.173 ms	-1.323%
http://jsfiddle.net/ugkd4/10/show/	-3.071%	(missing)	11.172 ms	-3.377%
http://jsfiddle.net/vBQHH/11/show/	-1.620%	(missing)	23.322 ms	-2.006%
http://jsfiddle.net/xLuvC/1/show/	-2.507%	(missing)	57.968 ms	-2.604%
http://mobile-news.sandbox.google.c…	-2.915%	(missing)	20.849 ms	-3.211%
http://mobile-news.sandbox.google.c…	-7.970%	(missing)	35.501 ms	-4.494%
http://wiltzius.github.io/shape-shifter/	-3.580%	(missing)	21.070 ms	-3.717%
http://www.google.com/#q=google	-0.922%	(missing)	32.730 ms	-0.859%
https://www.google.com/search?hl=e…	-1.305%	(missing)	31.801 ms	-0.589%
inbox_app.html?stress_hidey_bars	-1.085%	(missing)	16.282 ms	-1.507%
inbox_app.html?toggle_drawer	-1.791%	(missing)	17.504 ms	-2.339%
infinite_scrolling.html	-4.072%	(missing)	20.226 ms	-4.132%
list_animation_simple.html	-4.399%	(missing)	19.306 ms	-4.892%
pushState.html	-4.876%	(missing)	13.555 ms	-4.353%
silk_finance.html	-3.727%	(missing)	15.681 ms	-3.781%

The Threshold25 regression for jsfiddle.net/rF9Gh/7/show's is < 1/10th of a stdev away from AFDO.

thread_times.tough_scrolling_cases

Name	AFDOThinLTOO2Threshold25	Value	AFDO	AFDOThinLTOO2
benchmark_total_duration	(missing)	5,972,702.671 ms	(missing)	(missing)
mean_frame_time_renderer_compositor	-0.084%	(missing)	17.565 ms	-0.041%
tasks_per_frame_browser	-0.041%	(missing)	8.781	+0.070%
tasks_per_frame_GPU	+2.926%	(missing)	15.548	+2.661%
tasks_per_frame_GPU_transfer	+NaN%	(missing)	0.000	+NaN%
tasks_per_frame_IO	+2.285%	(missing)	31.215	+2.104%
tasks_per_frame_other	+1.120%	(missing)	0.267	+0.746%
tasks_per_frame_raster	+3.668%	(missing)	7.635	+3.267%
tasks_per_frame_renderer_compositor	+1.593%	(missing)	5.880	+1.846%
tasks_per_frame_renderer_main	+2.482%	(missing)	1.524	+2.197%
tasks_per_frame_total_all	+2.229%	(missing)	70.850	+2.075%
tasks_per_frame_total_fast_path	+2.048%	(missing)	61.424	+1.930%
thread_browser_cpu_time_per_frame	-3.580%	(missing)	3.542 ms	-3.308%
thread_GPU_cpu_time_per_frame	+0.864%	(missing)	5.723 ms	+0.411%
thread_GPU_transfer_cpu_time_per_fr…	+NaN%	(missing)	0.000 ms	+NaN%
thread_IO_cpu_time_per_frame	-0.476%	(missing)	3.745 ms	-0.929%
thread_other_cpu_time_per_frame	-0.701%	(missing)	0.095 ms	-1.469%
thread_raster_cpu_time_per_frame	+1.054%	(missing)	5.481 ms	+0.655%
thread_renderer_compositor_cpu_time…	-6.375%	(missing)	2.682 ms	-5.974%
thread_renderer_main_cpu_time_per_fr…	-5.581%	(missing)	4.236 ms	-4.301%
thread_total_all_cpu_time_per_frame	-1.746%	(missing)	25.504 ms	-1.711%
thread_total_fast_path_cpu_time_per_fr…	-1.696%	(missing)	15.691 ms	-1.840%

Immediately, it's visible that browser/render_compositor are significantly (both statistically and in terms of absolute amount) sped up by either ThinLTO variant.

In general, though, ThinLTO seems to really dislike the text_constant_full_page_raster_* pages:

▼thread_raster_cpu_time_per_frame	+1.054%	(missing)	5.481 ms	+0.655%
canvas_05000_pixels_per_second	-1.830%	(missing)	1.047 ms	-3.939%
canvas_10000_pixels_per_second	-2.637%	(missing)	1.845 ms	-3.734%
canvas_15000_pixels_per_second	-2.401%	(missing)	2.623 ms	-2.985%
canvas_20000_pixels_per_second	-4.063%	(missing)	3.524 ms	-4.700%
canvas_30000_pixels_per_second	-4.103%	(missing)	5.160 ms	-4.480%
canvas_40000_pixels_per_second	-3.787%	(missing)	6.798 ms	-5.418%
canvas_50000_pixels_per_second	-3.434%	(missing)	8.183 ms	-2.362%
canvas_60000_pixels_per_second	-2.904%	(missing)	9.250 ms	-3.088%
canvas_75000_pixels_per_second	+2.104%	(missing)	9.242 ms	+1.255%
canvas_90000_pixels_per_second	+1.524%	(missing)	9.403 ms	+0.967%
text_05000_pixels_per_second	-5.847%	(missing)	1.136 ms	-3.886%
text_10000_pixels_per_second	-2.634%	(missing)	1.933 ms	-3.715%
text_15000_pixels_per_second	-4.304%	(missing)	2.795 ms	-3.945%
text_20000_pixels_per_second	-4.247%	(missing)	3.680 ms	-4.328%
text_30000_pixels_per_second	-3.547%	(missing)	5.287 ms	-3.707%
text_40000_pixels_per_second	-3.150%	(missing)	6.802 ms	-3.081%
text_50000_pixels_per_second	-3.171%	(missing)	8.356 ms	-2.940%
text_60000_pixels_per_second	-4.694%	(missing)	9.562 ms	-3.872%
text_75000_pixels_per_second	-2.332%	(missing)	10.042 ms	-0.917%
text_90000_pixels_per_second	+1.321%	(missing)	9.953 ms	+3.154%
text_constant_full_page_raster_0500…	+4.055%	(missing)	5.754 ms	+4.445%
text_constant_full_page_raster_1000…	+2.040%	(missing)	6.449 ms	+1.859%
text_constant_full_page_raster_1500…	+4.131%	(missing)	6.714 ms	+2.568%
text_constant_full_page_raster_2000…	+8.258%	(missing)	6.102 ms	+8.419%
text_constant_full_page_raster_3000…	+11.278%	(missing)	4.981 ms	+9.060%
text_constant_full_page_raster_4000…	+15.797%	(missing)	4.002 ms	+13.108%
text_constant_full_page_raster_5000…	+21.215%	(missing)	3.523 ms	+16.592%
text_constant_full_page_raster_6000…	+18.975%	(missing)	3.109 ms	+22.631%
text_constant_full_page_raster_7500…	+45.579%	(missing)	2.083 ms	+35.092%
text_constant_full_page_raster_9000…	+137.200%	(missing)	0.774 ms	+110.215%
text_hover_05000_pixels_per_second	-3.825%	(missing)	1.578 ms	-0.736%
text_hover_10000_pixels_per_second	-3.973%	(missing)	1.924 ms	-3.127%
text_hover_15000_pixels_per_second	-3.586%	(missing)	2.749 ms	-3.253%
text_hover_20000_pixels_per_second	-6.246%	(missing)	3.750 ms	-5.384%
text_hover_30000_pixels_per_second	-3.586%	(missing)	5.332 ms	-3.776%
text_hover_40000_pixels_per_second	-3.583%	(missing)	6.765 ms	-3.158%
text_hover_50000_pixels_per_second	-2.300%	(missing)	8.217 ms	-2.670%
text_hover_60000_pixels_per_second	-0.148%	(missing)	9.185 ms	-1.580%
text_hover_75000_pixels_per_second	+2.840%	(missing)	9.716 ms	+0.609%
text_hover_90000_pixels_per_second	+0.193%	(missing)	9.928 ms	+0.196%

The +137% and +110% are both pretty accurate. To put them into context, however, I present the thread_total_all_cpu_time_per_frame for these stories:

text_90000_pixels_per_second	-0.594%	(missing)	31.301 ms	+0.181%
text_constant_full_page_raster_0500…	-0.102%	(missing)	31.837 ms	+0.534%
text_constant_full_page_raster_1000…	-1.202%	(missing)	33.536 ms	-1.045%
text_constant_full_page_raster_1500…	+0.176%	(missing)	34.172 ms	-0.937%
text_constant_full_page_raster_2000…	+1.097%	(missing)	32.900 ms	+1.710%
text_constant_full_page_raster_3000…	+1.321%	(missing)	30.149 ms	+0.481%
text_constant_full_page_raster_4000…	+1.704%	(missing)	27.645 ms	+1.692%
text_constant_full_page_raster_5000…	+2.110%	(missing)	27.007 ms	+0.964%
text_constant_full_page_raster_6000…	+0.093%	(missing)	26.908 ms	+1.441%
text_constant_full_page_raster_7500…	+1.891%	(missing)	26.339 ms	+1.247%
text_constant_full_page_raster_9000…	-1.850%	(missing)	26.072 ms	-1.026%

So, while a 45%+ regression certainly seems bad, it appears that ThinLTO improves other pieces enough to (almost) make up for it.