Summary of Benchmark Results

Natalie Popescu

VANILLA Unmodified Rustc

NOBC Rustc modified to *not* insert slice bounds checks

SAFELIB Rustc modified with a safe implementation of `memcpy`

NOBC+SL Rustc with both of the above modifications

1 Expectations

1. **NOBC** performs at least as well as **VANILLA**
2. **LTO=thin** performs at least as well as **LTO=off** (assuming it exposes more optimization opportunities)
3. **VANILLA** performs at least as well as **SAFELIB** (due to **SAFELIB**'s hacky implementation)

2 Observations

After compiling each crates' built-in benchmarks with each Rustc variation, we run each compiled benchmark at least 32 times–in random order–on one of two types of hardware (see Section 2.1). The tables in Section 2.3 provide an overview of how our modifications empirically match up with our expectations. We start out by counting how many benchmarks match up with our expectations in Tables 1 and 2. We then attempt to quantify just how unexpected the unexpected results really are in Tables 3 and 4. To do this we essentially calculate an average value per crate:

$$\frac{\Sigma \text{ all \% performance differences in the unexpected direction}}{\#\text{ unexpected benchmarks}}$$

Note that Tables 1 and 2 look at *expected* results and Tables 3 and 4 look at *unexpected* results.

2.1 Cloudlab Machines

Wisconsin Cluster: c220g2

CPU Two Intel E5-2660 v3 10-core CPUs at 2.60 GHz (Haswell EP)

RAM 160GB ECC Memory (10x 16 GB DDR4 2133 MHz dual rank RDIMMs)

Disk One Intel DC S3500 480 GB 6G SATA SSDs

Disk Two 1.2 TB 10K RPM 6G SAS SFF HDDs

NIC Dual-port Intel X520 10Gb NIC (PCIe v3.0, 8 lanes)

NIC Onboard Intel i350 1Gb.

Wisconsin Cluster: c220g5

CPU Two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz

RAM 192GB ECC DDR4-2666 Memory

Disk One 1 TB 7200 RPM 6G SAS HDs

Disk One Intel DC S3500 480 GB 6G SATA SSD

NIC Dual-port Intel X520-DA2 10Gb NIC (PCIe v3.0, 8 lanes)

NIC Onboard Intel i350 1Gb

2.2 Embedding Bitcode and LTO

In evaluating any performance impact that Link Time Optimizations (LTO) have on our above modifications, we hope to better understand the directions we should further explore when developing an effective transformation pass. LLVM's optimization passes generally run on individual modules, but LTOs have the advantage of analyzing whole programs as a single executable, enabling program-wide optimizations.

Here we clarify the purpose and intended behavior of two LTO-related flags we pass to Rustc: the first is the "-C embed-bitcode" flag which takes a boolean "yes" or "no" value. A value of "yes" causes Rustc to embed the generated LLVM bitcode into object files, which is necessary for LTO. A value of "no" is preferable if you will not perform LTO due to smaller generated file sizes and faster compilation times.

The value of the "-C embed-bitcode" flag implicitly sets the default value of the "-C LTO" flag. Specifying "-C embed-bitcode=no" implies a default of "-C LTO=off", and cannot be used with any other LTO value. Similarly, specifying "-C embed-bitcode=yes" implies a default of "-C LTO=thin", with some exceptions. When bitcode is embedded, it is also possible to specify a value of "-C LTO=fat". The high-level difference between **LTO=thin** and **LTO=fat** is that **LTO=thin** tries to achieve an **LTO=fat**-level of performance gains with compile times closer to that of **LTO=off**. All of these defaults can, of course, be passed explicitly: we do this to avoid any confusion.

2.3 Results

Crate	**#1 [LTO=off]**	**#1 [LTO=thin]**	**#3 [LTO=off]**	**#3 [LTO=thin]**
KDFs	X	66.$\overline{6}$%	X	33.$\overline{3}$%
PAKEs	50%	X	X	-
RustQIP	45%	65%	45%	55%
arrayvec	63.$\overline{63}$%	36.$\overline{36}$%	45.$\overline{45}$%	72.$\overline{72}$%
average	50%	66.$\overline{6}$%	33.$\overline{3}$%	16.$\overline{6}$%
base-x-rs	50%	66.$\overline{6}$%	66.$\overline{6}$%	16.$\overline{6}$%
bayesic_rs	33.$\overline{3}$%	66.$\overline{6}$%	X	66.$\overline{6}$%
bdays	33.$\overline{3}$%	X	X	33.$\overline{3}$%
boyer-moore-magiclen	66.$\overline{6}$%	45.8$\overline{3}$%	50%	66.$\overline{6}$%
bravery_router	88.$\overline{8}$%	44.$\overline{4}$%	X	77.$\overline{7}$%
bucket_queue	X	60%	50%	80%
chess_perft	72.$\overline{2}$%	27.$\overline{7}$%	72.$\overline{2}$%	31.48%
coinaddress	-	-	X	X
color-thief-rs	X	50%	X	50%
crc-any	60%	30%	40%	80%
directories-rs	-	-	X	50%
exrs	61.54%	7.69%	X	76.92%
fair-baccarat	-	-	X	X
fbleau	-	-	X	50%

flatten-overlapping...	X	X	X	-
frappe	30%	40%	80%	60%
gift	X	-	-	X
horned-owl	75%	25%	X	50%
implicit3d	18.75%	56.25%	75%	50%
ixlist	57.14%	57.14%	28.57%	42.86%
lazy-static-include	22.$\overline{2}$%	33.$\overline{3}$%	55.$\overline{5}$%	66.$\overline{6}$%
lebe	30%	70%	X	40%
lehmer	60%	20%	60%	80%
matrixmultiply	40.63%	37.5%	X	21.88%
metric	75%	X	X	25%
nblast-rs	88.$\overline{8}$%	11.$\overline{1}$%	X	44.$\overline{4}$%
optional	41.05%	31.58%	57.89%	72.63%
outils	83.$\overline{3}$%	50%	X	83.$\overline{3}$%
partition	70%	40%	70%	63.$\overline{3}$%
ripb	35%	10%	90%	60%
ropey	55.84%	76.62%	90.$\overline{90}$%	31.17%
rtriangulate	X	72.$\overline{72}$%	X	9.$\overline{09}$%
rumq	25%	50%	X	25%
rust-boomphf	66.$\overline{6}$%	66.$\overline{6}$%	X	66.$\overline{6}$%
rust-btoi	55.$\overline{5}$%	77.$\overline{7}$%	77.$\overline{7}$%	88.$\overline{8}$%
rust-obstack	40%	60%	X	20%
rust-url	X	X	X	X
simhash-rs	X	X	X	X
simple-irc-rs	-	-	X	50%
sliding_puzzle_rust	61.54%	53.85%	84.62%	7.69%
soa-derive	20%	X	60%	50%
sstable	-	50%	X	50%
ta-rs	73.$\overline{3}$%	40%	26.$\overline{6}$%	46.$\overline{6}$%
ticket	33.$\overline{3}$%	33.$\overline{3}$%	X	66.$\overline{6}$%
time-parse	50%	50%	50%	X
ucg	-	53.85%	X	53.85%
ulid-rs	40%	40%	40%	X
whatlang-rs	50%	X	X	50%
woodpecker	54.84%	87.1%	90.32%	58.06%
xoroshiro	47.62%	42.86%	66.$\overline{6}$%	66.$\overline{6}$%
xoshiro	X	X	X	-
zip-rs	X	-	-	X

Table 1: Percent of benchmarks that line up with expectations #1 and #3 (X if 100%, - if 0%, otherwise explicit).

Crate	**#2 [VANILLA]**	**#2 [NOBC]**	**#2 [NOBC+SL]**	**#2 [SAFELIB]**
KDFs	X	66.$\overline{6}$%	X	X
PAKEs	-	50%	50%	X
RustQIP	X	X	X	90%
arrayvec	81.$\overline{81}$%	54.$\overline{54}$%	63.$\overline{63}$%	72.$\overline{72}$%
average	33.$\overline{3}$%	33.$\overline{3}$%	50%	50%
base-x-rs	66.$\overline{6}$%	66.$\overline{6}$%	66.$\overline{6}$%	66.$\overline{6}$%
bayesic_rs	X	X	X	X
bdays	66.$\overline{6}$%	66.$\overline{6}$%	66.$\overline{6}$%	66.$\overline{6}$%

boyer-moore-magiclen	58.$\overline{3}$%	45.8$\overline{3}$%	54.1$\overline{6}$%	58.$\overline{3}$%
bravery_router	77.$\overline{7}$%	55.$\overline{5}$%	X	X
bucket_queue	30%	-	50%	50%
chess_perft	72.$\overline{2}$%	22.$\overline{2}$%	48.15%	18.52%
coinaddress	-	X	X	X
color-thief-rs	X	-	-	X
crc-any	40%	20%	40%	30%
directories-rs	75%	X	X	X
exrs	23.08%	-	X	X
fair-baccarat	-	-	X	-
fbleau	X	X	X	X
flatten-overlapping...	-	X	X	X
frappe	90%	90%	X	X
gift	X	-	-	X
horned-owl	X	X	X	X
implicit3d	75%	81.25%	81.25%	81.25%
ixlist	85.71%	71.43%	71.43%	71.43%
lazy-static-include	66.$\overline{6}$%	X	33.$\overline{3}$%	77.$\overline{7}$%
lebe	X	X	X	X
lehmer	80%	X	X	X
matrixmultiply	12.5%	6.25%	X	X
metric	-	-	-	-
nblast-rs	88.$\overline{8}$%	88.$\overline{8}$%	X	X
optional	48.42%	46.32%	47.37%	48.42%
outils	50%	25%	33.$\overline{3}$%	66.$\overline{6}$%
partition	90%	76.$\overline{6}$%	66.$\overline{6}$%	80%
ripb	90%	85%	95%	95%
ropey	77.92%	67.53%	92.21%	94.81%
rtriangulate	72.$\overline{72}$%	-	90.$\overline{90}$%	X
rumq	50%	75%	X	X
rust-boomphf	33.$\overline{3}$%	33.$\overline{3}$%	X	X
rust-btoi	X	88.$\overline{8}$%	88.$\overline{8}$%	88.$\overline{8}$%
rust-obstack	60%	80%	X	X
rust-url	-	-	-	-
simhash-rs	X	X	X	X
simple-irc-rs	50%	X	X	X
sliding_puzzle_rust	92.31%	92.31%	X	X
soa-derive	60%	60%	70%	60%
sstable	X	X	X	X
ta-rs	60%	53.$\overline{3}$%	60%	60%
ticket	X	66.$\overline{6}$%	X	X
time-parse	X	X	X	X
ucg	69.23%	69.23%	84.62%	76.92%
ulid-rs	X	80%	60%	X
whatlang-rs	X	X	X	X
woodpecker	67.74%	87.1%	96.77%	93.55%
xoroshiro	61.9%	66.$\overline{6}$%	71.43%	71.43%
xoshiro	X	X	X	X
zip-rs	-	-	-	-

Table 2: Percent of benchmarks that line up with expectation #2, per rustc version (X if 100%, - if 0%, otherwise explicit).

3 Extrapolating the Expected Effects of Our Pass