AutoFDO: recent improvements

Honza Hubička

SuSE ČR s.r.o
Prague

Joint work with Dhruv Chawla, Petr Hodač, Andi Kleen, Eugene Rozenfeld and Kugan Vivekanandarajah

GNU Cauldron 2025, Porto

What is CFG profile

CFG profile is an annotation of the control flow graph (CFG) by

- expected branch probabilities
- expected basic block execution counts (frequences)

What is CFG profile

CFG profile is an annotation of the control flow graph (CFG) by

- expected branch probabilities
- expected basic block execution counts (frequences)

Callgraph profile is an annotation of the callgraph by

- expected function counts
- expected callsites counts

What is CFG profile

CFG profile is an annotation of the control flow graph (CFG) by

- expected branch probabilities
- expected basic block execution counts (frequences)

Callgraph profile is an annotation of the callgraph by

- expected function counts
- expected callsites counts

Value profile is additional information on

- likely indirect call targets
- order of first executions of functions
- expected alisnments and sizes of string oprerations
- histograms of selected values
  (i.e. is division always by power of 2?)

Zdeněk Dvořák, J. H., Pavel Nejdedlý, Josef Zlomek:

Infrastructure for Profile Driven Optimizations in GCC Compiler, April 2002

`https://www.ucw.cz/~hubicka/papers/proj.pdf`

Instrumentation based profile: `-fprofile-use`

1. Uses data gathered by intrumented binary (via `-fprofile-generate`)
   1. 54% runtme cost.
   2. 90% code size cost.
   3. Need to stream a lot of data at exit (54MB).
   4. Fun with additional runtime in Linux kernel or embedded setups.

   (Measured on compiling clang binary)

Instrumentation based profile: `-fprofile-use`

1. Uses data gathered by intrumented binary (via `-fprofile-generate`)
   1. 54% runtme cost.
   2. 90% code size cost.
   3. Need to stream a lot of data at exit (54MB).
   4. Fun with additional runtime in Linux kernel or embedded setups.

   (Measured on compiling clang binary)
2. Determines profile of single-threaded program precisely

Instrumentation based profile: `-fprofile-use`

1. Uses data gathered by intrumented binary (via `-fprofile-generate`)
   1. 54% runtme cost.
   2. 90% code size cost.
   3. Need to stream a lot of data at exit (54MB).
   4. Fun with additional runtime in Linux kernel or embedded setups.

   (Measured on compiling clang binary)
2. Determines profile of single-threaded program precisely
3. Multi-threaded programs need to deal with race conditions (may have extreme performance impact)
4. Profiles are highly specific to build environemnt (GCC version, library headers etc.)
5. We do not implement path profiles and context sensitive profiles

Ball T, Larus JR. Optimally profiling and tracing programs. ACM TOPLAS. 1994 Jul 1;16(4):1319-60.

Static profile: `-fguess-branch-probability`

Set of heuristics to predict branch outcomes

- Good on predicting branch outcome:
  1. 56% of branches executed are predicted by reliable heuristics with 88.4% success rate out of 88.8%.

Static profile: `-fguess-branch-probability`

Set of heuristics to predict branch outcomes

- Good on predicting branch outcome:
  1. 56% of branches executed are predicted by reliable heuristics with 88.4% success rate out of 88.8%.
  2. 21% of branches executed are predicted by unreliable heuristics with 72% success rate out of 85%

Static profile: `-fguess-branch-probability`

Set of heuristics to predict branch outcomes

- Good on predicting branch outcome:
  1. 56% of branches executed are predicted by reliable heuristics with 88.4% success rate out of 88.8%.
  2. 21% of branches executed are predicted by unreliable heuristics with 72% success rate out of 85%
  3. 17% of branches executed are not predicted.

Static profile: `-fguess-branch-probability`

Set of heuristics to predict branch outcomes

- Good on predicting branch outcome:
  1. 56% of branches executed are predicted by reliable heuristics with 88.4% success rate out of 88.8%.
  2. 21% of branches executed are predicted by unreliable heuristics with 72% success rate out of 85%
  3. 17% of branches executed are not predicted.

  Overall 73% sucess rate out of 86% measured on SPEC2017

Static profile: `-fguess-branch-probability`

Set of heuristics to predict branch outcomes

- Good on predicting branch outcome:
  1. 56% of branches executed are predicted by reliable heuristics with 88.4% success rate out of 88.8%.
  2. 21% of branches executed are predicted by unreliable heuristics with 72% success rate out of 85%
  3. 17% of branches executed are not predicted.

  Overall 73% sucess rate out of 86% measured on SPEC2017
- Good on identifying hot paths in funtions
  (basic block reordering)

Static profile: `-fguess-branch-probability`

Set of heuristics to predict branch outcomes

- Good on predicting branch outcome:
  1. 56% of branches executed are predicted by reliable heuristics with 88.4% success rate out of 88.8%.
  2. 21% of branches executed are predicted by unreliable heuristics with 72% success rate out of 85%
  3. 17% of branches executed are not predicted.

  Overall 73% sucess rate out of 86% measured on SPEC2017
- Good on identifying hot paths in funtions
  (basic block reordering)
- Good on identifying relative frequences of basic block
  (spill code placement)

Static profile: `-fguess-branch-probability`

Set of heuristics to predict branch outcomes

- Good on predicting branch outcome:
  1. 56% of branches executed are predicted by reliable heuristics with 88.4% success rate out of 88.8%.
  2. 21% of branches executed are predicted by unreliable heuristics with 72% success rate out of 85%
  3. 17% of branches executed are not predicted.

  Overall 73% sucess rate out of 86% measured on SPEC2017
- Good on identifying hot paths in funtions
  (basic block reordering)
- Good on identifying relative frequences of basic block
  (spill code placement)
- Unable to determine value profiles

Static profile: `-fguess-branch-probability`

Set of heuristics to predict branch outcomes

- Good on predicting branch outcome:
  1. 56% of branches executed are predicted by reliable heuristics with 88.4% success rate out of 88.8%.
  2. 21% of branches executed are predicted by unreliable heuristics with 72% success rate out of 85%
  3. 17% of branches executed are not predicted.

  Overall 73% sucess rate out of 86% measured on SPEC2017
- Good on identifying hot paths in funtions
  (basic block reordering)
- Good on identifying relative frequences of basic block
  (spill code placement)
- Unable to determine value profiles
- Unable to determine loop iteration counts
  (vectorization, unrolling, . . .)

Static profile: `-fguess-branch-probability`

Set of heuristics to predict branch outcomes

- Good on predicting branch outcome:
  1. 56% of branches executed are predicted by reliable heuristics with 88.4% success rate out of 88.8%.
  2. 21% of branches executed are predicted by unreliable heuristics with 72% success rate out of 85%
  3. 17% of branches executed are not predicted.

  Overall 73% sucess rate out of 86% measured on SPEC2017
- Good on identifying hot paths in funtions (basic block reordering)
- Good on identifying relative frequences of basic block (spill code placement)
- Unable to determine value profiles
- Unable to determine loop iteration counts (vectorization, unrolling, . . .)
- No inter-procedural profiles at all

Ball T, Larus JR. Branch prediction for free. ACM SIGPLAN Notices. 1993 Jun 1;28(6):300-13.

Auto-fdo: `-fauto-profile`

1. Uses `perf` to record low-overhead profile. Requires LBR (x86-64) or BRBE (aarch64) support
2. Debug info is used to infer approximate CFG profile out of perf profile
3. Small runtime overhead (useful in production setups)
4. May be easier to set up (i.e. for profiling kernel)
5. Profiles are less sensitive to build environment and can be resused in slightly different setup (i.e. shipped with the source codes)

History

1. 2014: Contributed by Google in 2014 (Dehao Chen)

Application	FDO	AutoFDO	Ratio
400.perlbench	15.27%	14.99%	98.17%
401.bzip	1.35%	1.00%	74.07%
403.gcc	7.73%	7.52%	97.28%
429.mcf	0.04%	2.75%	100.00%
445.gobmk	3.67%	3.23%	88.01%
456.hmmer	-0.73%	1.90%	100.00%
458.sjeng	6.19%	6.03%	97.42%
462.libquantum	-10.41%	-0.61%	100.00%
464.h264ref	1.61%	-1.75%	0.00%
471.omnetpp	4.03%	1.31%	32.51%
473.astar	8.86%	10.12%	114.20%
483.xalancbmk	14.44%	11.98%	82.96%
mean	4.40%	4.87%	112.33%

D. Chen D, D.X. Li, T. Moseley, AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. CGO 2016.

History

1. 2014: Contributed by Google in 2014 (Dehao Chen)
2. 2015: Bitrotting
   (Google switched to LLVM; I gave up looking for compatible setup)

History

1. 2014: Contributed by Google in 2014 (Dehao Chen)
2. 2015: Bitrotting
   (Google switched to LLVM; I gave up looking for compatible setup)
3. 2016: Andi Kleen contributed autoprofiledbootstrap and testsuite support; main problem was limited hardware support and `create_gcov` requiring specific old version of perf

History

1. 2014: Contributed by Google in 2014 (Dehao Chen)
2. 2015: Bitrotting
   (Google switched to LLVM; I gave up looking for compatible setup)
3. 2016: Andi Kleen contributed autoprofiledbootstrap and testsuite support; main problem was limited hardware support and `create_gcov` requiring specific old version of perf
4. 2017: Rewrite of profile representation with AutoFDO in mind (me); continued testsuite work (Andi Kleen)

History

1. 2014: Contributed by Google in 2014 (Dehao Chen)
2. 2015: Bitrotting
   (Google switched to LLVM; I gave up looking for compatible setup)
3. 2016: Andi Kleen contributed autoprofiledbootstrap and testsuite support; main problem was limited hardware support and `create_gcov` requiring specific old version of perf
4. 2017: Rewrite of profile representation with AutoFDO in mind (me); continued testsuite work (Andi Kleen)
5. 2018: Bugfixes by Bin Cheng

History

1. 2014: Contributed by Google in 2014 (Dehao Chen)
2. 2015: Bitrotting
   (Google switched to LLVM; I gave up looking for compatible setup)
3. 2016: Andi Kleen contributed autoprofiledbootstrap and testsuite support; main problem was limited hardware support and `create_gcov` requiring specific old version of perf
4. 2017: Rewrite of profile representation with AutoFDO in mind (me); continued testsuite work (Andi Kleen)
5. 2018: Bugfixes by Bin Cheng
6. 2019–2020: Bitrotting again

History

1. 2014: Contributed by Google in 2014 (Dehao Chen)
2. 2015: Bitrotting
   (Google switched to LLVM; I gave up looking for compatible setup)
3. 2016: Andi Kleen contributed autoprofiledbootstrap and testsuite support; main problem was limited hardware support and `create_gcov` requiring specific old version of perf
4. 2017: Rewrite of profile representation with AutoFDO in mind (me); continued testsuite work (Andi Kleen)
5. 2018: Bugfixes by Bin Cheng
6. 2019–2020: Bitrotting again
7. 2021–2024: Eugene Rozenfield started fixing autoFDO, worked on aarch64 support; appointed as a maintainer

History

1. 2014: Contributed by Google in 2014 (Dehao Chen)
2. 2015: Bitrotting
   (Google switched to LLVM; I gave up looking for compatible setup)
3. 2016: Andi Kleen contributed autoprofiledbootstrap and testsuite support; main problem was limited hardware support and `create_gcov` requiring specific old version of perf
4. 2017: Rewrite of profile representation with AutoFDO in mind (me); continued testsuite work (Andi Kleen)
5. 2018: Bugfixes by Bin Cheng
6. 2019–2020: Bitrotting again
7. 2021–2024: Eugene Rozenfield started fixing autoFDO, worked on aarch64 support; appointed as a maintainer
8. 2025: I have noticed that my machine supports AutoFDO and Dhruv Chawla, Kugan Vivekanandarajah started working on aarch64 improvements

Using AutoFDO: Compile and train

Example (Test program)

```
[[gnu::used]] int a[N];
[[gnu::noipa]] void test()
{
  for (int i = 0; i < N; i++)
    a[i]++;
}
int main()
{
   for (int i = 0; i < M; i++)
     test();
   return 1;
}
```

Using AutoFDO: Compile and train

Example (Test program)

```
[[gnu::used]] int a[N];
[[gnu::noipa]] void test()
{
  for (int i = 0; i < N; i++)
    a[i]++;
}
int main()
{
   for (int i = 0; i < M; i++)
     test();
   return 1;
}
```

Compile and train

```
$ gcc -O2 test.c -g -DN=1000 -DM=1000000
$ perf record -e ex_ret_brn_tkn:Pu -b -c 100003 -- ./a.out
```

I additionally used `-fno-tree-vectorize -fno-unroll-loops` to simplify the assembly.

Using AutoFDO: Compile and train

Compile and train

```
$ gcc -O2 test.c -g -DN=1000 -DM=1000000 \
        -fno-tree-vectorize -fno-unroll-loops
$ perf record -e ex_ret_brn_tkn:Pu -b -c 100003 -- ./a.out
```

1. `-e ex_ret_brn_tkn:Pu` enables recording of retired taken branches in userland on AMD Zen 3, 4 and 5.
   1. Use `-e br_inst_retired.near_taken:pu` for Intel cores
   2. Do not specify `-e` for Aarch 64
2. `-b` enables branch stack sampling (LBR or BRBE). Each sample captures a sequences of 32 branches.
3. `-c` enables sampling count. It is better to be prime.

Using AutoFDO: verify data (optional)

Example (Test program)

```
[[gnu::used]] int a[N];
[[gnu::noipa]] void test()
{
  for (int i = 0; i < N; i++)
    a[i]++;
}
```

N=1000, M=1000000, sample each 100003

`perf report` of function `test`

```
                    74          mov     $0x404040,%eax
                 94339 10:      addl    $0x1,(%rax)
                                add     $0x4,%rax
                                cmp     $0x404fe0,%rax
                 94339          jne     10
                   106          ret
```

Using AutoFDO: Produce GCC readable profile

Compile AutoFDO tools from
`https://github.com/google/autofdo`
(Good luck!)

Using AutoFDO: Produce GCC readable profile

Compile AutoFDO tools from
`https://github.com/google/autofdo`
(Good luck!)

Create GCC readable profile

```
$ create_gcov --binary a.out --gcov_version 2 perf.data \
  --gcov test.gcov
[WARNING:/home/jh/autofdo/third_party/perf_data_converter
/src/quipper/perf_reader.cc:1322] Skipping 264 bytes of
metadata: HEADER_CPU_TOPOLOGY
[WARNING:/home/jh/autofdo/third_party/perf_data_converter
/src/quipper/perf_reader.cc:1069] Skipping unsupported
event PERF_RECORD_ID_INDEX
WARNING: Logging before InitGoogleLogging() is written to
STDERR
```

...

Using AutoFDO: Produce GCC readable profile

Compile AutoFDO tools from
`https://github.com/google/autofdo`
(Good luck!)

Create GCC readable profile

```
$ create_gcov --binary a.out --gcov_version 2 perf.data \
  --gcov test.gcov
[WARNING:/home/jh/autofdo/third_party/perf_data_converter
/src/quipper/perf_reader.cc:1322] Skipping 264 bytes of
metadata: HEADER_CPU_TOPOLOGY
[WARNING:/home/jh/autofdo/third_party/perf_data_converter
/src/quipper/perf_reader.cc:1069] Skipping unsupported
event PERF_RECORD_ID_INDEX
WARNING: Logging before InitGoogleLogging() is written to
STDERR
```

. . . Do not panic; the tools are chatty

Using AutoFDO: Dump GCC readable profile

Dump GCC readable profile

```
$ dump_gcov test.gcov
test total:274729 head:74
  1: 74
  2: 74
  2.1: 91492
  3: 91492
  4: 105
main total:215 head:0
  1: 0
  2: 0
  2.1: 71
  3: 73  test:74
  4: 0
  5: 0
```

Example (Test program)

```
   [[gnu::used]] int a[N];
   [[gnu::noipa]] void test()
1  {
2    for (int i = 0; i < N; i++)
3      a[i]++;
4  }
   int main()
1  {
2    for (int i = 0; i < M; i++)
3      test();
4    return 1;
5  }
```

Line numbers are represented as **relative-line.discriminator**

Using AutoFDO: Dump GCC readable profile

Dump GCC readable profile

```
$ dump_gcov test2.gcov
main total:275074 head:0
  1: 0
  2: 0
  2.1: 83
  4: 0
  5: 0
  3: test total:274908
    2.1: 91636
    3: 91636
```

Example (Test program)

```
   [[gnu::used]] int a[N];
   static void test()
1  {
2    for (int i = 0; i < N; i++)
3      a[i]++;
4  }
   int main()
1  {
2    for (int i = 0; i < M; i++)
3      test();
4    return 1;
5  }
```

Inline functions are recorded as separate instances (gaining simple context sensitivity)

Using AutoFDO: Reading profile back to GCC

Executing GCC

```
$ gcc -O2 -DN=1000 -DM=1000000 test.c \
   -fauto-profile=test.gcov -Wauto-profile
```

1. `-fauto-profile` specifies profile
2. `-Wauto-profile` enables warnings about profile mismatches (new in GCC 16)
3. You may look into dumps `-fdump-ipa-afdo_offline` (new in GCC 16) and `-fdump-ipa-afdo-blocks-details`

GCC processing: AFDO offline pass

AFDO offline pass is a new pass in GCC 16. It does the following

1. Reads afdo profile into memory
2. Strips symbol suffixes introduced by later optimizations (`isra`, `constprop`, `part`, `cold`)
3. Removes instance of functions not defined in current unit and offlines inline instance corresponding to cross-module inlinig
4. Merges profile of duplicate instances of the same name; offlines functions if necessary.

GCC processing: AFDO inlining

Auto-FDO inlines all early-inlinable functions inlined during the train run which have enough samples in them.

Dump GCC readable profile

```
main total:275074 head:0
  1: 0
  2: 0
  2.1: 83
  4: 0
  5: 0
  3: test total:274908
    2.1: 91636
    3: 91636
```

```
$ gcc -O2 -DN=1000 -DM=1000000 \
   -fauto-profile=test2.gcov test2.c -opt-info
test2.c:10:6: optimized: Inlining using auto-profile
test/2 into main/3.
test2.c:4:21: optimized: loop vectorized using 16 byte vectors
```

GCC processing: AFDO offline pass; lookup

Profile verification: `a-test.c.019i.afdo_offline`

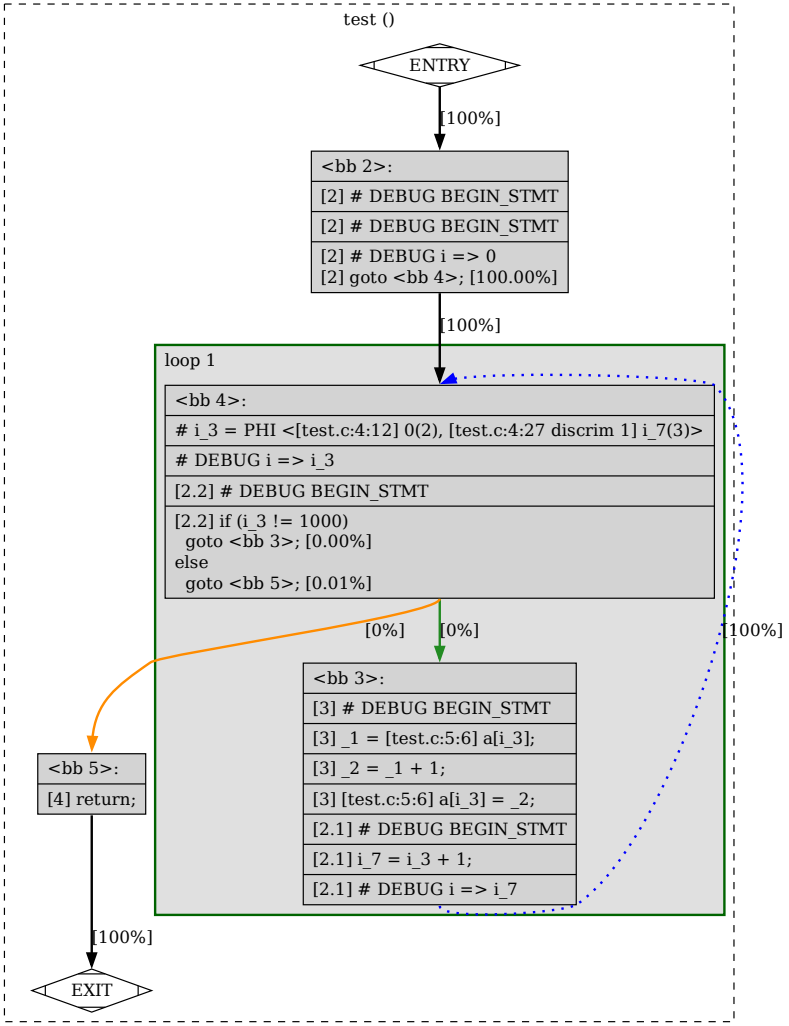
```
Matching gimple function test/2 with auto profile: test
 basic block 2
      2              74 # DEBUG BEGIN_STMT
      2              74 # DEBUG BEGIN_STMT
      2              74 i = 0;
 basic block 3
      3           91492 # DEBUG BEGIN_STMT
      3           91492 _1 = a[i];
      3           91492 _2 = _1 + 1;
      3           91492 a[i] = _2;
      2.1         91492 # DEBUG BEGIN_STMT
      2.1         91492 i = i + 1;
 basic block 4
      2.2      no info  # DEBUG BEGIN_STMT
      2.2      no info  if (i <= 999)
 basic block 5
      4             105 return;
```

GCC processing: AFDO pass

AFDO pass estimates the CFG profile from data available

Profile inference: `a-test.c.074i.afdo`, step 1 (lookup)

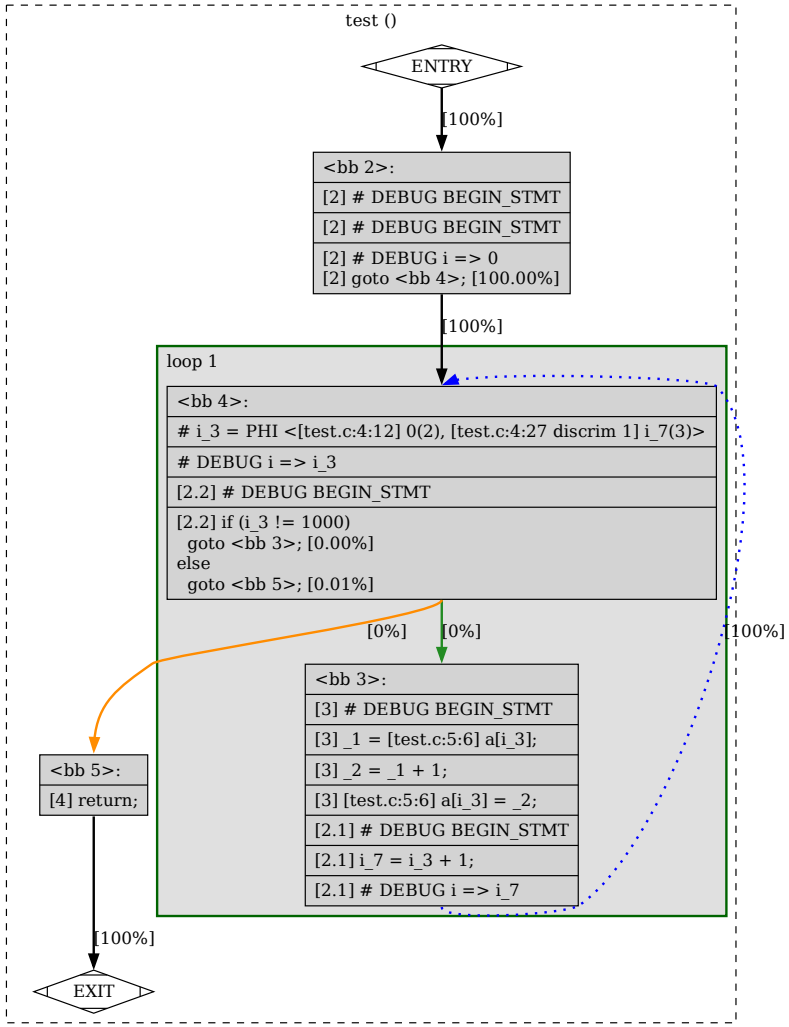
```
Annotating BB profile of test/2

test total:274729 head:74
  2: 74
  2.1: 91492
  3: 91492
  4: 105

 Looking up AFDO count of bb 2
  count 74 in stmt: # DEBUG BEGIN_STMT
  count 74 in stmt: # DEBUG BEGIN_STMT
  count 74 in stmt: # DEBUG i => 0
 Annotated bb 2 with count 74, scaled to 910643478152
 Looking up AFDO count of bb 3
  count 91492 in stmt: # DEBUG BEGIN_STMT
  count 91492 in stmt: _1 = a[i_3];
  count 91492 in stmt: _2 = _1 + 1;
  count 91492 in stmt: a[i_3] = _2;
  count 91492 in stmt: # DEBUG BEGIN_STMT
  count 91492 in stmt: i_7 = i_3 + 1;
  count 91492 in stmt: # DEBUG i => i_7
 Annotated bb 3 with count 91492, scaled to
 1125899906798416
 Looking up AFDO count of bb 4
 Looking up AFDO count of bb 5
```



GCC processing: AFDO pass

Profile inference: `a-test.c.074i.afdo`, step 2 (kihroff laws)

```
Annotated edge 2->4 with count 910643478152 (auto FDO)
Annotated edge 3->4 with count 1125899906798416 (auto FDO)
Annotated edge 5->1 with count 910643478152 (auto FDO)
Annotated edge 0->2 with count 910643478152 (auto FDO)
Annotated edge 4->3 with count 1125899906798416 (auto FDO)
Annotating bb 4 with count 1126810550276568 (auto FDO)
Annotated edge 4->5 with count 910643478152 (auto FDO)
```



GCC processing: AFDO pass

Profile inference: `a-test.c.074i.afdo`, step 3 (scaling guessed profile)

```
Starting connected component in bb 1
  visiting bb 1 with count 10737416 (estimated locally)
    Annotated pred edge to 5 with count 910643478152 (auto FDO)
    bb 1 in count 10737416 (estimated locally) should be 910643478152 (auto FDO)
    adding scale 84810.3005371093750000, weight 910643478153
    accounting scale 84810.3005371093750000, weight 910643478153
  Scaling by 84810.3004150390625000
    bb 1 count updated 10737416 (estimated locally) -> 910643476480 (guessed)
```

GCC processing: AFDO pass

Profile inference: `a-test.c.074i.afdo`, final profile

```
;;   basic block 2, loop depth 0, count 910643478152 (auto FDO, freq 1.0000), maybe hot
;;    pred:       ENTRY [always]  count:910643478152 (auto FDO, freq 1.0000)
;;    succ:       4 [always]  count:910643478152 (auto FDO, freq 1.0000)

;;   basic block 3, loop depth 1, count 1125899906798416 (auto FDO, freq 1236.3784), maybe hot
;;    pred:       4 [99.9% (auto FDO)]  count:1125899910412526 (auto FDO, freq 1236.3784)
;;    succ:       4 [always]  count:1125899906798416 (auto FDO, freq 1236.3784)

;;   basic block 4, loop depth 1, count 1126810550276568 (auto FDO, freq 1237.3784), maybe hot
;;    prev block 3, next block 5, flags: (NEW, VISITED)
;;    pred:       2 [always]  count:910643478152 (auto FDO, freq 1.0000)
;;                3 [always]  count:1125899906798416 (auto FDO, freq 1236.3784)
;;    succ:       3 [99.9% (auto FDO)]  count:1125899910412526 (auto FDO, freq 1236.3784)
;;                5 [0.1% (auto FDO)]  count:910639864042 (auto FDO, freq 1.0000)

;;   basic block 5, loop depth 0, count 910643478152 (auto FDO, freq 1.0000), maybe hot
;;    prev block 4, next block 1, flags: (NEW, VISITED)
;;    pred:       4 [0.1% (auto FDO)]  count:910639864043 (auto FDO, freq 1.0000)
;;    succ:       EXIT [always]  count:910643478153 (auto FDO, freq 1.0000)
```

Comparing AFDO and GCC profile

Compiling with instrumentation and comparing profiles

```
$ gcc -O2 -DN=1000 -DM=1000000 test.c \
        -fprofile-generate
$ ./a.out
$ gcc -O2 -DN=1000 -DM=1000000 test.c \
        -fauto-profile=test.gcov -fprofile-use \
        -fdump-ipa-profile
```

a-test.c.077i.profile

```
test/2 bb 0 fdo 1000000 afdo 910643478152 (auto FDO) (hot) scaled 809427 diff -190573, -19.06%
 preds
 succs 2
test/2 bb 2 fdo 1000000 afdo 910643478152 (auto FDO) (hot) scaled 809427 diff -190573, -19.06%
 preds 0
 succs 4
test/2 bb 3 fdo 1000000000 afdo 1125899906798416 (auto FDO) (very hot) scaled 1000758536 diff 758536, +0.08%
 preds 4
 succs 4
test/2 bb 4 fdo 1001000000 afdo 1126810550276568 (auto FDO) (very hot) scaled 1001567964 diff 567964, +0.06%
 preds 2 3
 succs 3 5
test/2 bb 5 fdo 1000000 afdo 910643478152 (auto FDO) (hot) scaled 809427 diff -190573, -19.06%
 preds 4
 succs 1
test/2 bb 1 fdo 1000000 afdo 910643476480 (guessed) (hot) scaled 809427 diff -190573, -19.06%
 preds 5
 succs
```

Benchmarking AutoFDO and AMD EPYC 9755

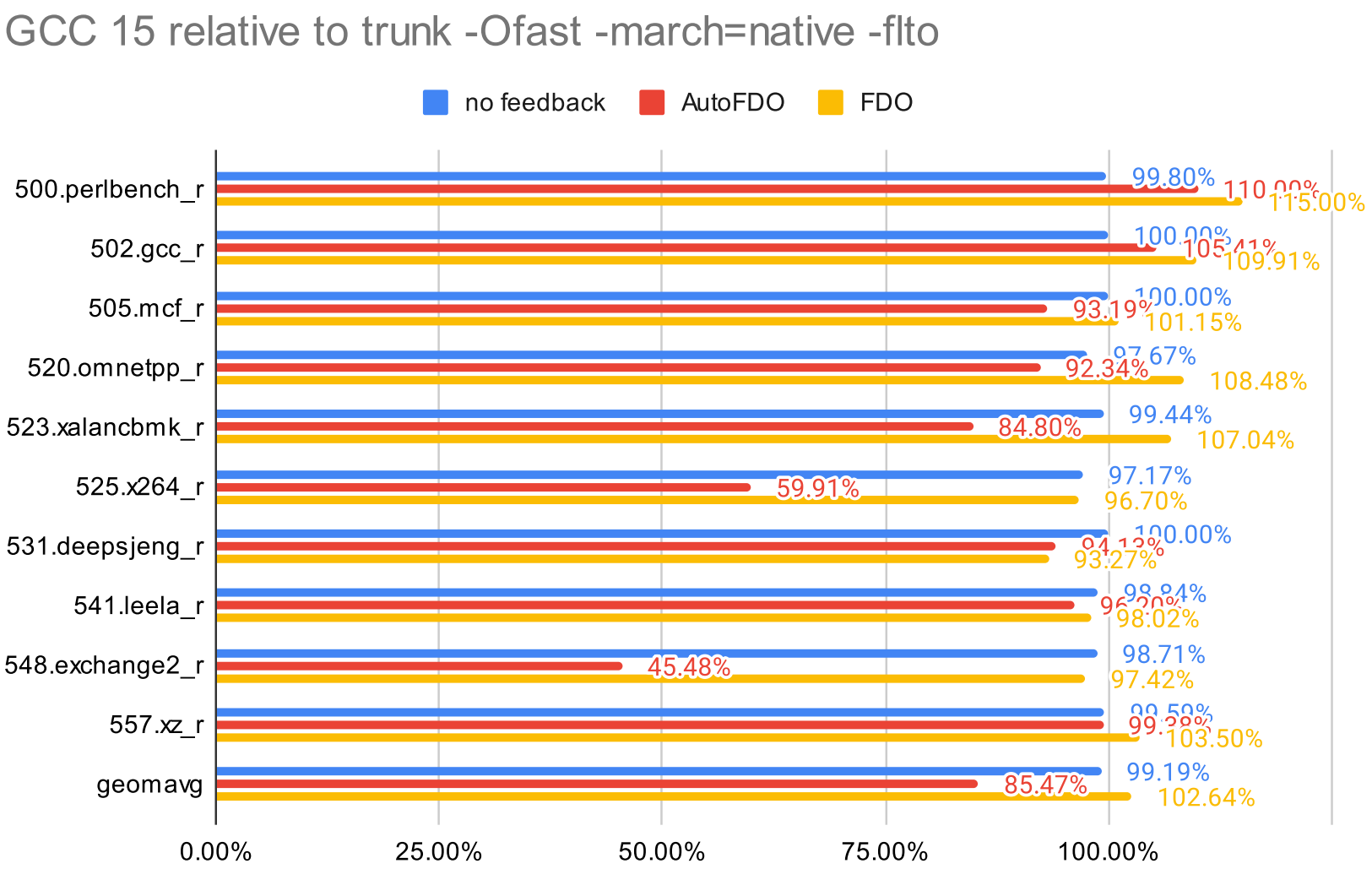
I modified SPEC scripts to allow `-train_with=refrate` which uses reference data set for training. This makes training to run longer and reduces training noise.

SPEC 2017 configuration

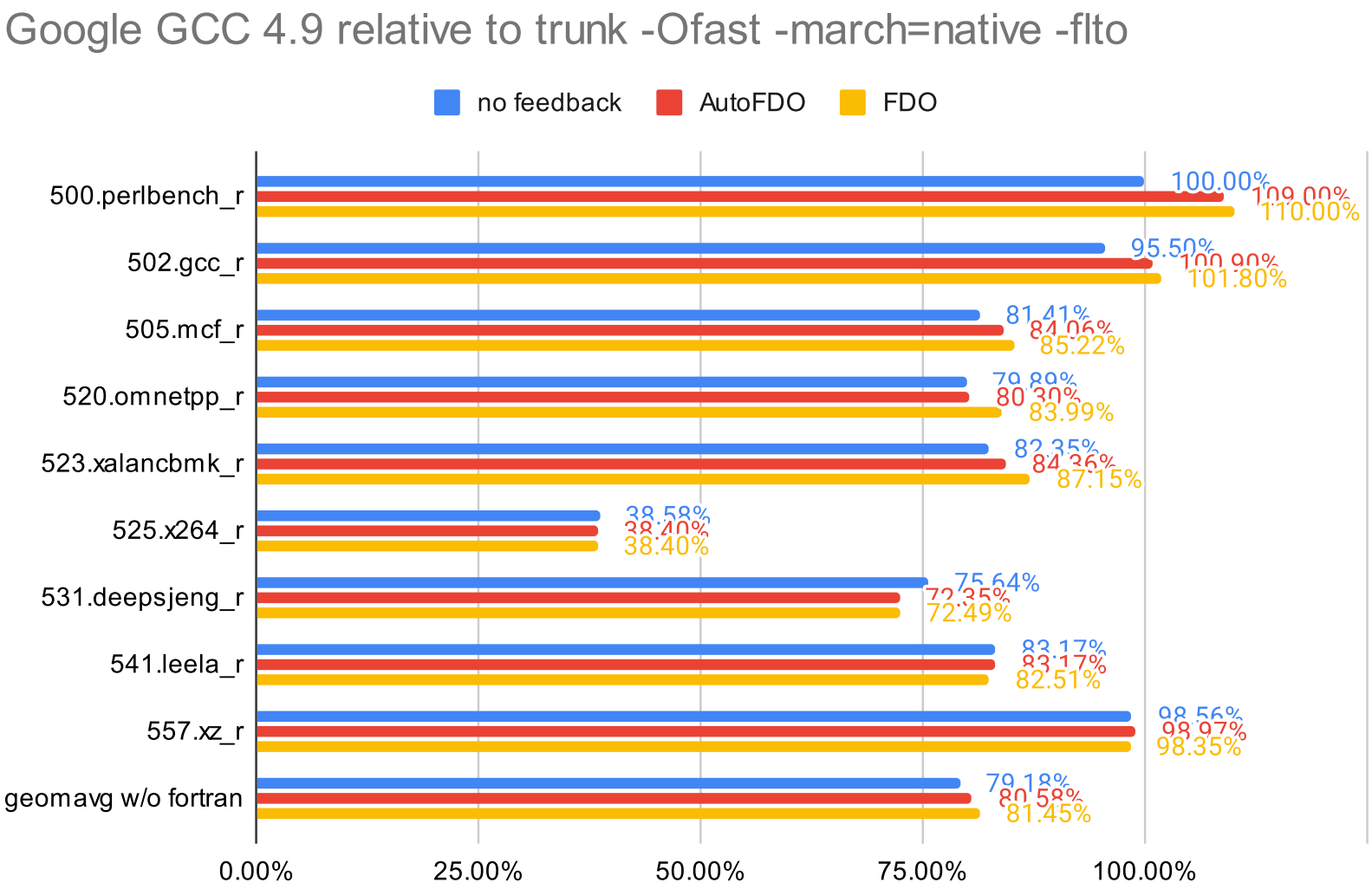
```
OPTIMIZE    = -Ofast  -march=native -flto=auto
fdo_pre0 = rm -rf ${benchmark}.data ${benchmark}.gcov
fdo_run1 = perf record -e ex_ret_brn_tkn:Pu -c 100003 -b  -o ${benchmark}.data -- ${command} || exit 1;\
           create_gcov  --suffix_elision_policy=none --binary=${baseexe} --profile=${benchmark}.data   \
             --gcov=current.gcov -gcov_version=2 || exit 1;                                            \
           if test -e ${benchmark}.gcov ; then                                                         \
             profile_merger current.gcov ${benchmark}.gcov --output_file ${benchmark}.gcov || exit 1 ; \
           else mv current.gcov ${benchmark}.gcov || exit 1 ; fi
PASS1_OPTIMIZE = -g -fno-reorder-blocks-and-partition \
  -fno-ipa-icf -fno-partial-inlining
PASS2_OPTIMIZE = -fauto-profile=${benchmark}.gcov
```

Daily testing is now done at `https://lnt.opensuse.org/db_default/v4/SPEC/recent_activity`

Benchmarking AutoFDO and AMD EPYC 9755

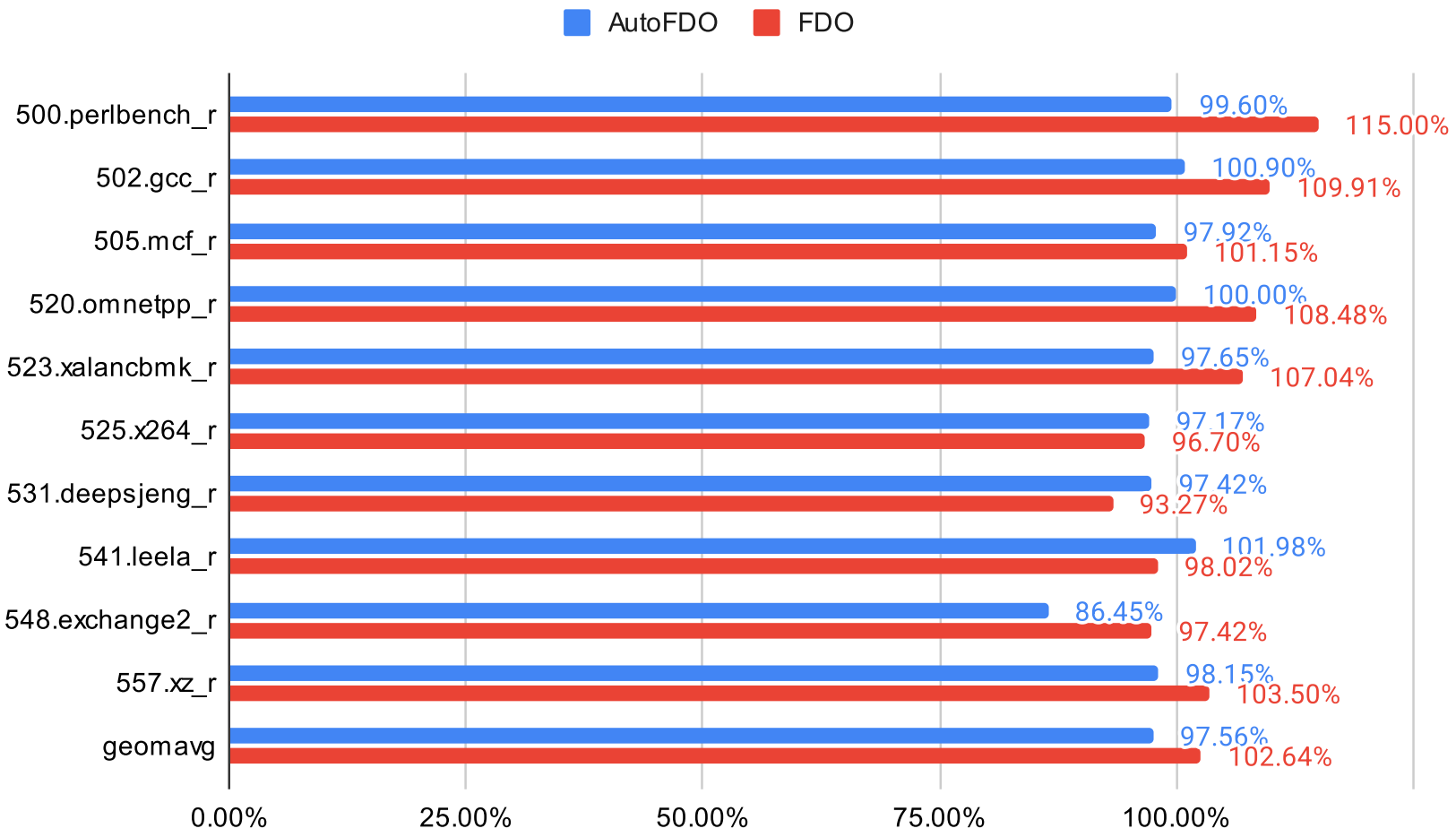


Benchmarking AutoFDO and AMD EPYC 9755



Benchmarking AutoFDO and AMD EPYC 9755

trunk



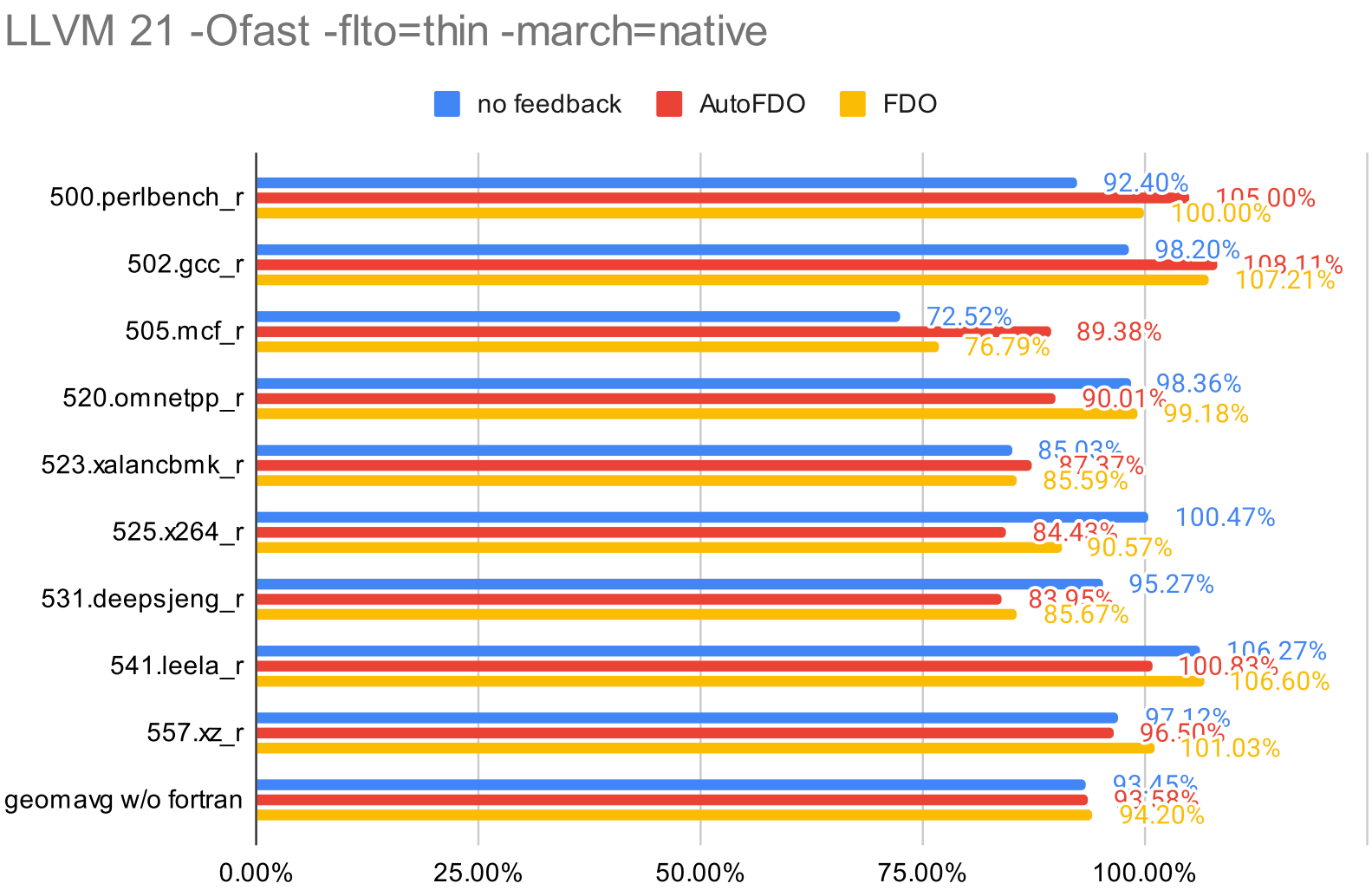
yesterday

Benchmarking AutoFDO and AMD EPYC 9755

trunk

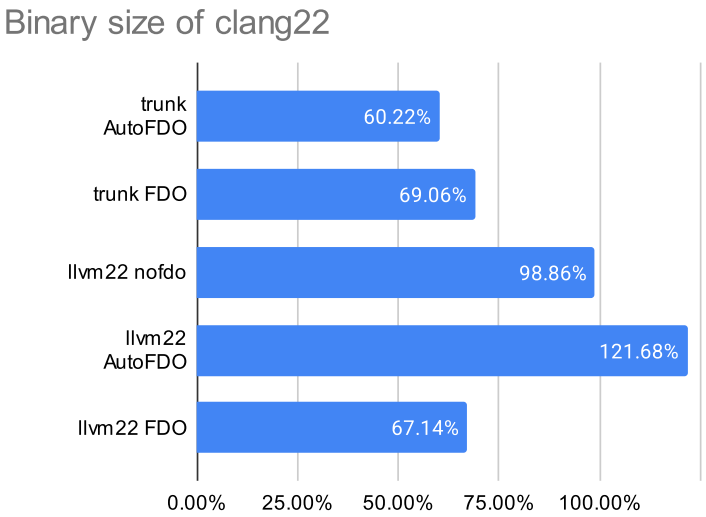
this morning

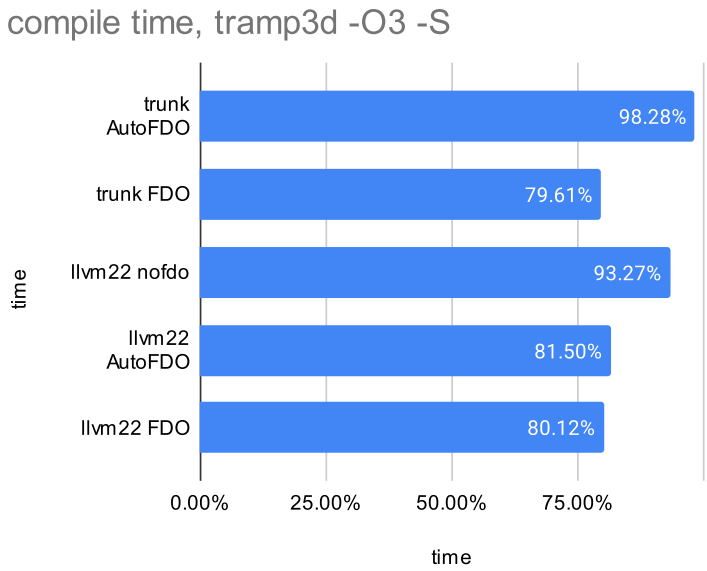
Benchmarking AutoFDO and AMD EPYC 9755



Benchmarking AutoFDO and AMD EPYC 9755

Clang22 built with GCC trunk and LLVM21 with `-O3 -flto`.
ThinLTO was used for LLVM.
Trained by building tramp3d.





Reported copmile time is a compile time of resulting clang22 binary building tramp3d again.

Changes upstreamed to trunk

1. Testsuite support for aarch64
2. Stripping of late suffixes (`isra`, `constprop`, `part`, `cold`)
3. `-Wauto-profile` warning
4. AFDO offline pass (to not lose profile with LTO)
5. Significant changes to profile inference algorithm
6. Better handling of zeros in AutoFDO profiles
7. Scaling of AutoFDO profile to reduce roundoff errors
8. Infrasructure to compare AutoFDO and FDO data
9. Discriminator support rewrite

Approx 70 patches overall.
Regular testing using LNT

In progress

GCC:

1. Hiearchical discriminator support (discriminator,copy-id,multiplicity)

In progress

GCC:

1. Hiearchical discriminator support (discriminator,copy-id,multiplicity)
2. Make dwarf2out to save linkage name of inline functions

In progress

GCC:

1. Hiearchical discriminator support (discriminator,copy-id,multiplicity)
2. Make dwarf2out to save linkage name of inline functions
3. Handle ipa-split clones correctly

In progress

GCC:

1. Hiearchical discriminator support (discriminator,copy-id,multiplicity)
2. Make dwarf2out to save linkage name of inline functions
3. Handle ipa-split clones correctly
4. Improvements to autorpofiledbootstrap

AutoFDO tool:

1. Handle multiple locations per single statement
2. Switch to 64bit format to save hiearchical discriminators
3. Save file names of translation units to distinguish static functions of the same name

What needs to be done

1. Get performance of AutoFDO generated code closer to FDO

What needs to be done

1. Get performance of AutoFDO generated code closer to FDO
2. Redesign datastructures used to hold auto-fdo profile; speed up loading

What needs to be done

1. Get performance of AutoFDO generated code closer to FDO
2. Redesign datastructures used to hold auto-fdo profile; speed up loading
3. Can we extend dwarf to handle multiple call stacks per single address?

What needs to be done

1. Get performance of AutoFDO generated code closer to FDO
2. Redesign datastructures used to hold auto-fdo profile; speed up loading
3. Can we extend dwarf to handle multiple call stacks per single address?
4. Extend gcov-tool to handle merging of sample profiles

What needs to be done

1. Get performance of AutoFDO generated code closer to FDO
2. Redesign datastructures used to hold auto-fdo profile; speed up loading
3. Can we extend dwarf to handle multiple call stacks per single address?
4. Extend gcov-tool to handle merging of sample profiles
5. Rewrite `create_gcov`
   Support streaming profiles from perf; improve scalability and stability of the tool