20+ years CFG profile in GCC

Honza Hubička

SuSE ČR s.r.o
Prague

GNU Cauldron 2023, Cambridge

What is CFG profile

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

- Expected branch probabilities
- expected basic block execution counts

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

Profile based optimizations

- Originally an excuse to re-organize GCC backend to use commonized control flow graph module
- Re-organized reg-stack to use CFG in 1998
- Started to work on profile infrastructure in 2000
- School project together with Zdeněk, Josef and Pavel in 2001-2002
- AMD Project in 2002-2003

Profile based optimizations

- Originally an excuse to re-organize GCC backend to use commonized control flow graph module
- Re-organized reg-stack to use CFG in 1998
- Started to work on profile infrastructure in 2000
- School project together with Zdeněk, Josef and Pavel in 2001-2002
- AMD Project in 2002-2003

Profile feedback & unit-at-a-time – p.2/14

GCC in 2000

1. Virtually all optimizations done at RTL form
2. RTL function is a single doubly-linked list of statements (no CFG!)
3. Few optimization passes built and used their own CFG (reg-stack, register allocator, Haifa scheduler, dead code elimination, . . .)
4. Instruction-level notes used to represent information about loops, libcalls, debug info, . . .

Richard Henderson did initial work on generalizing `flow.c` to general CFG infrastructure shared by multiple passes.

Original design

Two forms of profile

1. Edge profiling
   - `-fprofile-generate` and `-fprofile-use`
   - Originally by James Wilson, Cygnus, 1990
   - Ball T, Larus JR. Optimally profiling and tracing programs. ACM Transactions on Programming Languages and Systems (TOPLAS). 1994 Jul 1;16(4):1319-60.
2. Static profile estimation
   - `-fguess-branch-probability`
   - Originally by Jason Eckhart & Stan Cox, Cygnus, 2000
   - Ball T, Larus JR. Branch prediction for free. ACM SIGPLAN Notices. 1993 Jun 1;28(6):300-13.

Original design

~~Two~~ forms of profile

1. Edge profiling
   - `-fprofile-generate` and `-fprofile-use`
   - Originally by James Wilson, Cygnus, 1990
   - Ball T, Larus JR. Optimally profiling and tracing programs. ACM Transactions on Programming Languages and Systems (TOPLAS). 1994 Jul 1;16(4):1319-60.
2. Static profile estimation
   - `-fguess-branch-probability`
   - Originally by Jason Eckhart & Stan Cox, Cygnus, 2000
   - Ball T, Larus JR. Branch prediction for free. ACM SIGPLAN Notices. 1993 Jun 1;28(6):300-13.
3. Value profiling
   - `-fprofile-values`
   - Added by Zdeněk Dvořák in 2003

Original design

~~Two~~ forms of profile

1. Edge profiling
   - `-fprofile-generate` and `-fprofile-use`
   - Originally by James Wilson, Cygnus, 1990
   - Ball T, Larus JR. Optimally profiling and tracing programs. ACM Transactions on Programming Languages and Systems (TOPLAS). 1994 Jul 1;16(4):1319-60.
2. Static profile estimation
   - `-fguess-branch-probability`
   - Originally by Jason Eckhart & Stan Cox, Cygnus, 2000
   - Ball T, Larus JR. Branch prediction for free. ACM SIGPLAN Notices. 1993 Jun 1;28(6):300-13.
3. Value profiling
   - `-fprofile-values`
   - Added by Zdeněk Dvořák in 2003
4. Auto-FDO (based on low overhead profiling)
   - `-fauto-profile`
   - Contributed by Google in 2014, now maintained by Eugene Rozenfeld

Original design

In 2000, as part of work on improving Itanium, branch probability and execution count notes was added to RTL to enable basic block reordering pass.

Original design

In 2000, as part of work on improving Itanium, branch probability and execution count notes was added to RTL to enable basic block reordering pass.

We decided to take this as an excuse to introduce persistent CFG as part of the RTL IL.

Original design

In 2000, as part of work on improving Itanium, branch probability and execution count notes was added to RTL to enable basic block reordering pass.

We decided to take this as an excuse to introduce persistent CFG as part of the RTL IL.

1. Static profile represented as:
   1. probabilities of edges (in range 0...10000) and
   2. frequencies of basic blocks (integers in range 0...10000)
2. Profile feedback (edge profile) represented as:
   1. execution counts of edges (64bit integers)
   2. execution counts of basic blocks (64bit integers)
3. Value profile was read and immediately used for code transformations

Most work was involved in redesigning existing passes to maintain and use CFG.

Static profile estimation (`analyze_brprob.py`)

heuristics	hitrate		perfect hitrate	Coverage
combined	69.74%	/	80.61%	100.0%
first match	77.81%	/	78.31%	69.0%
no prediction	33.65%	/	85.08%	15.6%
DS theory	70.14%	/	86.40%	15.4%
First match:				
loop iterations	67.99%	/	67.99%	39.1%
guessed loop iterations	91.73%	/	92.49%	23.2%
loop exit	85.36%	/	87.83%	5.8%
noreturn call	100.00%	/	100.00%	0.8%
Fortran loop preheader	99.81%	/	99.88%	0.6%
extra loop exit	82.80%	/	88.17%	0.2%
loop iv compare	52.06%	/	52.15%	0.0%
Fortran overflow	100.00%	/	100.00%	0.0%
Fortran fail alloc	100.00%	/	100.00%	0.0%
loop guard with recursion	17.17%	/	93.91%	0.0%
Dempster–Shaffer (DS) theory:				
opcode values nonequal (on trees)	67.63%	/	81.38%	7.2%
call	67.26%	/	92.26%	3.3%
early return (on trees)	54.39%	/	86.51%	3.2%
opcode values positive (on trees)	64.55%	/	90.39%	1.7%
pointer (on trees)	69.59%	/	87.18%	1.6%
continue	66.66%	/	82.85%	1.0%
loop guard	61.88%	/	88.38%	0.6%
guess loop iv compare	97.75%	/	97.79%	0.4%
null return	91.47%	/	93.08%	0.3%
negative return	97.94%	/	99.23%	0.1%
const return	69.39%	/	87.09%	0.0%
loop exit with recursion	72.17%	/	92.33%	0.0%
recursive call	75.19%	/	76.33%	0.0%
Fortran repeated allocation/deallocation	100.00%	/	100.00%	0.0%
Fortran zero-sized array	100.00%	/	100.00%	0.0%

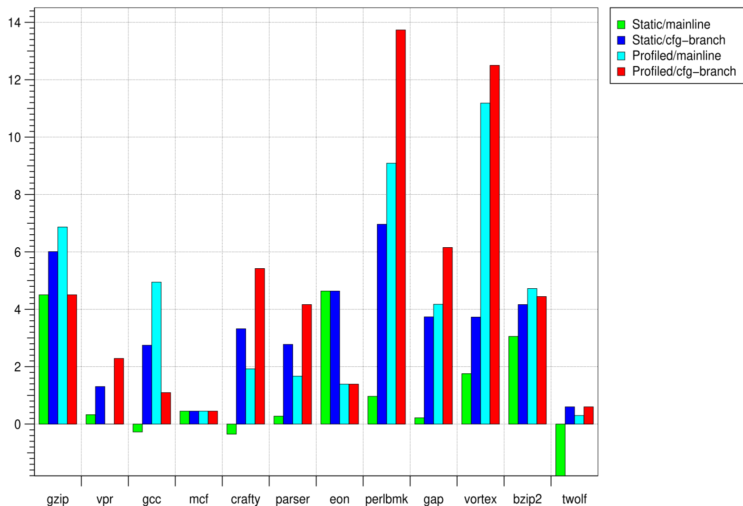
Consumers of the profile information

1. Basic-block reordering pass
2. Code alignment (function/loop/jump target alignment)
3. Register allocation (to spill on cold paths)
4. Loop optimizer (to determine unrolling and peeling factors)
5. Loop array prefetching
6. Tracer (new pass doing tail duplication over common paths)
7. Cold code discovery (optimize cold regions for size)

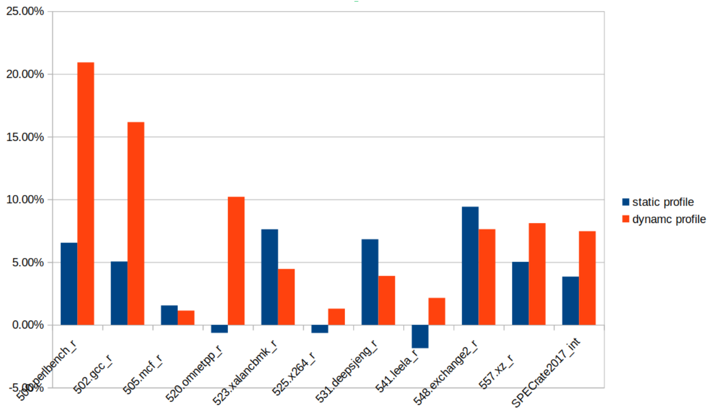
Consumers of the profile information

1. Basic-block reordering pass
2. Code alignment (function/loop/jump target alignment)
3. Register allocation (to spill on cold paths)
4. Loop optimizer (to determine unrolling and peeling factors)
5. Loop array prefetching
6. Tracer (new pass doing tail duplication over common paths)
7. Cold code discovery (optimize cold regions for size)
8. Interprocedural optimizations: Inliner, function cloning
9. Function partitioning
10. Function splitting
11. Higher-level loop optimizations
12. Profile is useful in handling various side cases (code sinking, PRE, . . .)

Results in 2002



Results in 2023



Persistent loop information

GCC was also updated to keep persistent information about the loop structure   We keep following info

1. Upper bound on number of iterations
2. Likely upper bound on number of iterations
3. Expected number of iterations
4. Is loop known to be finite?
5. Force/enable vectorization flag
6. Intended unrolling factor
7. ...

Loop structures are linked with header basic blocks:

1. Code duplication code should mind updating loop structure
2. Some passes move loop headers and should also update the links
   (otherwise loop is lost & rediscovered and all annotations are forgotten)

2017 CFG profile revamp

C++ data-types replacing frequencies, probabilities and counts:

1. `profile_probability`, 32 bits stored in CFG edges:
    1. 29 bits of fixed point probability (value in 0...1)
    2. 3 bits quality information:

2017 CFG profile revamp

C++ data-types replacing frequencies, probabilities and counts:

1. `profile_probability`, 32 bits stored in CFG edges:
   1. 29 bits of fixed point probability (value in 0...1)
   2. 3 bits quality information:
      1. `UNINITIALIZED_PROFILE`
      2. `GUESSED`
      3. `AFDO`
      4. `ADJUSTED` (value used to be known precisely but we duplicated code and it may not be representative anymore)
      5. `PRECISE`

2017 CFG profile revamp

C++ data-types replacing frequencies, probabilities and counts:

1. `profile_probability`, 32 bits stored in CFG edges:
   1. 29 bits of fixed point probability (value in 0...1)
   2. 3 bits quality information:
      1. `UNINITIALIZED_PROFILE`
      2. `GUESSED`
      3. `AFDO`
      4. `ADJUSTED` (value used to be known precisely but we duplicated code and it may not be representative anymore)
      5. `PRECISE`
2. `profile_count`, 64 bits stored in CFG basic blocks and callgraph:
   1. 61 bits fixed point execution count
   2. 3 bits quality information with few extra options:
      1. `GESSED_LOCAL` (value is known only within single function and is relative to the entry block count)
      2. `GESSED_GLOBAL0` (function was never executed in train run, but we have local estimate)

2017 CFG profile revamp

1. `profile_probability` supports:
   1. Predefined values:
      1. `never` (0 precise), `always` (1 precise),
      2. `even` (0.5 guessed),
      3. `likely` (0.8 guessed),, `unlikely` (0.2 guessed)
      4. `very_likely` (0.998 guessed), `very_unlikely` (0.002)
      5. `uninitialized`
      6. ...

2017 CFG profile revamp

1. `profile_probability` supports:
   1. Predefined values:
      1. `never` (0 precise), `always` (1 precise),
      2. `even` (0.5 guessed),
      3. `likely` (0.8 guessed),, `unlikely` (0.2 guessed)
      4. `very_likely` (0.998 guessed), `very_unlikely` (0.002)
      5. `uninitialized`
      6. ...
   2. Basic operations $+, -, *, /$, pow, sqrt with obvious meaning. All capping and propagating quality info.

2017 CFG profile revamp

1. `profile_probability` supports:
   1. Predefined values:
      1. `never` (0 precise), `always` (1 precise),
      2. `even` (0.5 guessed),
      3. `likely` (0.8 guessed),, `unlikely` (0.2 guessed)
      4. `very_likely` (0.998 guessed), `very_unlikely` (0.002)
      5. `uninitialized`
      6. ...
   2. Basic operations $+, -, *, /$, pow, sqrt with obvious meaning. All capping and propagating quality info.
   3. Comparisons $<, >, =, <=, >=$ are three-way and returns false if unknown.

2017 CFG profile revamp

1. `profile_probability` supports:
   1. Predefined values:
      1. `never` (0 precise), `always` (1 precise),
      2. `even` (0.5 guessed),
      3. `likely` (0.8 guessed),, `unlikely` (0.2 guessed)
      4. `very_likely` (0.998 guessed), `very_unlikely` (0.002)
      5. `uninitialized`
      6. ...
   2. Basic operations $+, -, *, /$, pow, sqrt with obvious meaning. All capping and propagating quality info.
   3. Comparisons $<, >, =, <=, >=$ are three-way and returns false if unknown.
   4. Conversion to `sreal`; conversion to original `REG_BR_PROB_BASE` fixpoint is deprecated

2017 CFG profile revamp

1. `profile_probability` supports:
   1. Predefined values:
      1. `never` (0 precise), `always` (1 precise),
      2. `even` (0.5 guessed),
      3. `likely` (0.8 guessed),, `unlikely` (0.2 guessed)
      4. `very_likely` (0.998 guessed), `very_unlikely` (0.002)
      5. `uninitialized`
      6. ...
   2. Basic operations $+, -, *, /$, pow, sqrt with obvious meaning. All capping and propagating quality info.
   3. Comparisons $<, >, =, <=, >=$ are three-way and returns false if unknown.
   4. Conversion to `sreal`; conversion to original `REG_BR_PROB_BASE` fixpoint is deprecated
   5. Probability can be applied to `profile_count`
   6. Probability can be scaled by fractions of two `profile_count`s or `gcov_type`s

2017 CFG profile revamp

1. `profile_probability` supports:
    1. Predefined values:
        1. `never` (0 precise), `always` (1 precise),
        2. `even` (0.5 guessed),
        3. `likely` (0.8 guessed),, `unlikely` (0.2 guessed)
        4. `very_likely` (0.998 guessed), `very_unlikely` (0.002)
        5. `uninitialized`
        6. ...
    2. Basic operations $+, -, *, /$, pow, sqrt with obvious meaning. All capping and propagating quality info.
    3. Comparisons $<, >, =, <=, >=$ are three-way and returns false if unknown.
    4. Conversion to `sreal`; conversion to original `REG_BR_PROB_BASE` fixpoint is deprecated
    5. Probability can be applied to `profile_count`
    6. Probability can be scaled by fractions of two `profile_count`s or `gcov_type`s
    7. `reliable_p` predicate

2017 CFG profile revamp

1. `profile_probability` supports:
   1. Predefined values:
      1. `never` (0 precise), `always` (1 precise),
      2. `even` (0.5 guessed),
      3. `likely` (0.8 guessed),, `unlikely` (0.2 guessed)
      4. `very_likely` (0.998 guessed), `very_unlikely` (0.002)
      5. `uninitialized`
      6. ...
   2. Basic operations $+, -, *, /$, pow, sqrt with obvious meaning. All capping and propagating quality info.
   3. Comparisons $<, >, =, <=, >=$ are three-way and returns false if unknown.
   4. Conversion to `sreal`; conversion to original `REG_BR_PROB_BASE` fixpoint is deprecated
   5. Probability can be applied to `profile_count`
   6. Probability can be scaled by fractions of two `profile_count`s or `gcov_type`s
   7. `reliable_p` predicate
   8. Dumping, debug output and LTO streaming
   9. ...

2017 CFG profile revamp

1. `profile_count` supports:
   1. Predefined values:
      1. `zero` (0 precise),
      2. `adjusted_zero` (0 adjusted),
      3. `guessed_zero` (0 guessed),
      4. `uninitialized`

2017 CFG profile revamp

1. `profile_count` supports:
   1. Predefined values:
      1. `zero` (0 precise),
      2. `adjusted_zero` (0 adjusted),
      3. `guessed_zero` (0 guessed),
      4. `uninitialized`
   2. `ipa_p` predicate if the value is meaningful at inter-procedural level and `ipa` conversion function.

2017 CFG profile revamp

1. `profile_count` supports:
   1. Predefined values:
      1. `zero` (0 precise),
      2. `adjusted_zero` (0 adjusted),
      3. `guessed_zero` (0 guessed),
      4. `uninitialized`
   2. `ipa_p` predicate if the value is meaningful at inter-procedural level and `ipa` conversion function.
   3. Basic operations $+$, $-$. All capping.
   4. $*$, $/$ by integer.

2017 CFG profile revamp

1. `profile_count` supports:
    1. Predefined values:
        1. `zero` (0 precise),
        2. `adjusted_zero` (0 adjusted),
        3. `guessed_zero` (0 guessed),
        4. `uninitialized`
    2. `ipa_p` predicate if the value is meaningful at inter-procedural level and `ipa` conversion function.
    3. Basic operations $+, -$. All capping.
    4. $*, /$ by integer.
    5. Comparisons $<, >, =, <=, >=$ are three-way and returns false if unknown.

2017 CFG profile revamp

1. `profile_count` supports:
   1. Predefined values:
      1. `zero` (0 precise),
      2. `adjusted_zero` (0 adjusted),
      3. `guessed_zero` (0 guessed),
      4. `uninitialized`
   2. `ipa_p` predicate if the value is meaningful at inter-procedural level and `ipa` conversion function.
   3. Basic operations $+, -$. All capping.
   4. $*, /$ by integer.
   5. Comparisons $<, >, =, <=, >=$ are three-way and returns false if unknown.
   6. `apply_probability` to multiply by a probability
   7. `apply_scale` to scale by a given fraction
   8. `probability_in` to determine of probability of one count in another

2017 CFG profile revamp

1. `profile_count` supports:
   1. Predefined values:
      1. `zero` (0 precise),
      2. `adjusted_zero` (0 adjusted),
      3. `guessed_zero` (0 guessed),
      4. `uninitialized`
   2. `ipa_p` predicate if the value is meaningful at inter-procedural level and `ipa` conversion function.
   3. Basic operations $+, -$. All capping.
   4. $*, /$ by integer.
   5. Comparisons $<, >, =, <=, >=$ are three-way and returns false if unknown.
   6. `apply_probability` to multiply by a probability
   7. `apply_scale` to scale by a given fraction
   8. `probability_in` to determine of probability of one count in another
   9. Dumping, debug output and LTO streaming
   10. . . .

2017 CFG profile revamp

Value histograms are attached to statements using on-side hash similar way as we do with EH regions.

1. First execution time profiling
   (for code reordering)
2. Indirect call profiling
   (represented in callgraph to aid inlining)
3. Division/modulo by constant or power of 2 transformation
4. String operation buffer size profiling

Profile maintenance

1. Profile info is estimated or read in early and needs to be maintained across the whole optimization pipeline
2. Low-level API (edge redirection, BB creation, . . .) has no info needed to determine profile
3. Sometimes profile becomes incoherent as a result of optimizations:

```
int foo (int a)
{
  if (a) // 0.5 probability before inlining
    bar ();
}
main()
{
  foo (0); // probability 0 after inline
  foo (1); // probability 1 after inline
}
```

Profile maintenance

1. Profile info is estimated or read in early and needs to be maintained across the whole optimization pipeline
2. Low-level API (edge redirection, BB creation, ...) has no info needed to determine profile
3. Sometimes profile becomes incoherent as a result of optimizations:

```
int foo (int a)
{
  if (a) // 0.5 probability before inlining
    bar ();
}
main()
{
  foo (0); // probability 0 after inline
  foo (1); // probability 1 after inline
}
```

Every pass is responsible to cleanup its own mess!

Profile maintenance

1. Profile info is estimated or read in early and needs to be maintained across the whole optimization pipeline
2. Low-level API (edge redirection, BB creation, ...) has no info needed to determine profile
3. Sometimes profile becomes incoherent as a result of optimizations:

```
int foo (int a)
{
  if (a) // 0.5 probability before inlining
    bar ();
}
main()
{
  foo (0); // probability 0 after inline
  foo (1); // probability 1 after inline
}
```

Every pass is responsible to cleanup its own mess!

2023 is a year of profile fixes for me

Our LNT tester tracks profile quality building tramp3d

`https://lnt.opensuse.org/db_default/v4/CPP/latest_runs_report` counts section

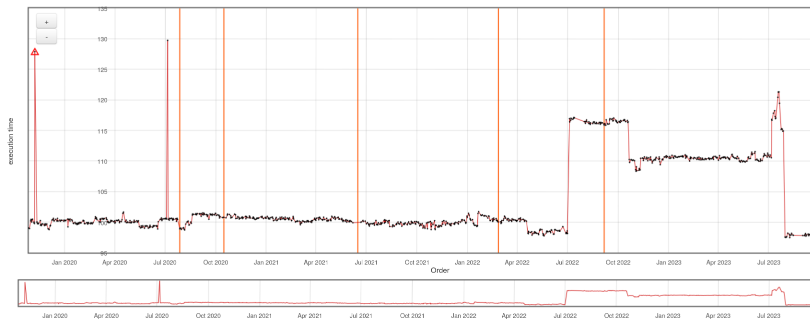
Progress so far

vectorizer	𝍸 𝍸 ||
loop header copying	𝍸 |
loop splitting	𝍸
jump threading	𝍸
loop peeling	|||
loop versioning	||
loop unrolling	||
branch prediction	||
profile datatructures	||
DCE	|
reassoc	|
sreal	|
loop invariant motion	|
loop distribution	|
Patches in various stage of baking	𝍸 ||||

Progress so far

- vectorizer 卌 卌 ||
- loop header copying 卌 |
- loop splitting 卌
- jump threading 卌
- loop peeling |||
- loop versioning ||
- loop unrolling ||
- branch prediction || No RTL yet
- profile datatructures ||
- DCE |
- reassoc |
- sreal |
- loop invariant motion |
- loop distribution |
- Patches in various stage of baking 卌 ||||

Hmmer with `-Ofast -flto -march=native`



Please mind the profile!

1. Use `blocks-details` dump flags to see information about profile mismatches.
2. If you spot some after your transformation try to see if it is carried in or needs to be fixed
3. `-fprofile-report` can be used to get overall data about profile quality
4. Add tests checking profile updates searching for

   `Invalid sum`
5. Try to think of side cases:
   1. What happens when profile is missing
   2. What happens when profile is inconsistent on the input (try to minimize chance of spreading the error further)
   3. ...
6. Transformations affecting CFG needs to update profile (even if you just update conditional to constant 0 or 1 and let cfgcleanup do the work)
7. Transformations affecting loop headers or iteration counts needs to update loop structure

-fprofile-report on cc1plus; late scalar cleanup

pass	profile mismathces	size	time
86i inline	+44649708141	+33.1%	-10.3%
103t ccp	+48243471353	-0.5%	-0.6%
108t cunrolli	-5321807717	-0.0%	-0.9%
111t forwprop	+6468	-0.1%	-0.1%
114t fre	+24027489258	-1.2%	-2.2%
116t threadfull	+384565874522	-0.1%	-2.2%
117t vrp	+33600542417	-4.7%	-0.5%
118t dse	-1256513433	-0.3%	-0.8%
119t dce	-3122885968	-0.1%	-0.4%
122t cselim	-4219221947	-0.0%	+0.1%
123t copyprop	+1366809955	-0.0%	-0.0%
124t ifcombine	-377126355	+0.1%	+0.6%
125t mergephi	-4203847197		
126t phiopt	+3328253040	-0.2%	-0.1%
127t tailr	+436974	-0.0%	-0.0%
128t ch	+1910701333	+0.3%	-0.1%
131t sra		-0.3%	-1.4%
132t thread	+19416347646	+0.1%	-0.2%
133t dom	+108076654929	+0.6%	-1.4%
134t copyprop	+7207906125	-0.1%	-0.2%
135t isolate-paths	+1308251212	+0.1%	-0.0%
136t reassoc	+3910185448	+0.1%	+0.4%
137t dce	+6480998024	-0.6%	-0.8%
138t forwprop	+3833217719	-0.2%	-0.2%
139t phiopt	-700396863	-0.0%	-0.0%
140t ccp	+249246775	-0.0%	-0.0%
144t lim		+0.0%	-0.3%
146t pre	+6023194805	-1.1%	-1.2%
147t sink	+289483116	-0.2%	-0.8%
151t dse		-0.1%	-0.2%
152t dce	-367616368	-0.0%	-0.0%
156t unswitch	+9830153927	+0.1%	-0.1%

-fprofile-reporton cc1plus; loop opt and cleanup

pass	profile mismathces	size	time
157t lsplit	+14490079645	+0.1%	-0.1%
161t cddce	+669837664745	-0.0%	-0.0%
162t ivcanon	-27256511	-0.0%	+0.2%
163t ldist	-281039842	+0.0%	-0.1%
165t copyprop	+27035589	-0.0%	-0.0%
173t ch_vect	-1046470417	+0.0%	+0.0%
174t ifcvt	+29259121621	+0.3%	+1.1%
175t vect	-29025607210	-0.2%	-0.9%
176t dce	-17302853	-0.0%	-0.1%
178t cunroll	-15270158714	+2.9%	-0.3%
179t fre	+4869608409	-0.1%	-0.3%
185t loopdone	+184441655	-0.0%	-0.0%
187t slp		-0.3%	-0.0%
189t veclower2	-1076208793	-0.0%	-0.0%
190t switchlower	+210516697487	+0.1%	+0.8%
193t reassoc	+21247652009	+0.1%	+0.3%
196t tracer	-336562640930	+2.1%	+0.2%
197t fre	+14786972198	-0.1%	-0.2%
198t thread	+54685466506	+0.2%	-0.3%
199t dom	-77643836674	+0.2%	-0.5%
201t threadfull	+3263568385	+0.2%	-0.1%
202t vrp	+10265768145	-0.2%	-0.1%
203t ccp	+1452217296	-0.0%	-0.1%
205t dse	+296953356	-0.7%	-0.6%
206t dce	-740575118	-0.0%	-0.0%
207t forwprop	+57913221	-0.1%	-0.1%
208t sink	-6059911247	-0.1%	-0.1%
209t phiopt	-640623	-0.0%	-0.0%
210t fab	-26448502	-0.0%	-0.0%
212t store-merging		-0.1%	-0.0%
213t cddce	+266716939779	-0.0%	-0.0%
214t tailc	-18449860	-0.0%	-0.0%

-fprofile-reporton cc1plus; RTL

pass	profile mismathces	size	time
267r cse1	+421885069	-0.1%	+0.0%
268r fwprop1		-5.6%	-3.4%
269r cprop	+8353476384	-0.3%	-0.7%
270r rtl pre		+0.6%	+0.6%
272r cprop	+99659357	-0.6%	-0.8%
274r cse_local	+2592078	-0.1%	-0.3%
275r ce1	-181502659	-0.1%	+0.1%
279r loop2_invariant		+0.1%	+0.1%
280r loop2_unroll	+292954091	+0.0%	-0.0%
282r loop2_done	-14842665470	-0.0%	+0.0%
285r cprop	+4083486645	-0.1%	-0.1%
287r cse2	+209418484	-0.2%	-0.1%
289r fwprop2	+14820737	-0.0%	-0.0%
293r combine	+1095431281	-0.4%	-0.7%
296r ce2	-93648750	-0.0%	+0.0%
297r jump_after_combine	+8056118858	-0.0%	-0.0%
298r bbpart	-162391652	+0.9%	-0.0%
309r ira		+1.6%	+1.2%
310r reload	+332096918	-4.5%	-3.5%
317r pro_and_epilogue	+132365843216	+5.6%	+14.6%
320r jump2	+159341511698	-2.2%	+0.4%
324r ce3	-462446573	-0.0%	+0.0%
328r bbro	-169410373304	-0.5%	-1.8%

Please use the profile in your passes

1. Use `optimize_*_for_speed` and `optimize_*_for_size`. (* is one of `function`, `bb`, `insn`, `loop`, `loop_nest`)

Please use the profile in your passes

1. Use `optimize_*_for_speed` and `optimize_*_for_size`.
   (* is one of `function`, `bb`, `insn`, `loop`, `loop_nest`)
2. `optimize_*_for_size` is now two-state predicate. Returned value is `optimize_size_level` enum:
   - 0: `OPTIMIZE_SIZE_NO`
     optimize for speed, this may be hot part of program.
   - 1: `OPTIMIZE_SIZE_BALANCED`
     this is likely not hot part of program but evidence is low; avoid bloat but do not do extreme tradeoffs
   - 2: `OPTIMIZE_SIZE_MAX`
     do everything possible to reduce code size

   (A lot of target specific work is needed here)

Please use the profile in your passes

1. Use `optimize_*_for_speed` and `optimize_*_for_size`.
   (* is one of `function`, `bb`, `insn`, `loop`, `loop_nest`)
2. `optimize_*_for_size` is now two-state predicate.
   Returned value is `optimize_size_level` enum:
   - 0: `OPTIMIZE_SIZE_NO`
     optimize for speed, this may be hot part of program.
   - 1: `OPTIMIZE_SIZE_BALANCED`
     this is likely not hot part of program but evidence is low; avoid bloat but do not do extreme tradeoffs
   - 2: `OPTIMIZE_SIZE_MAX`
     do everything possible to reduce code size

   (A lot of target specific work is needed here)
3. Use persistent loop info in loop transformations
   Most loop transforms are miss for loops iterating 0 or very few times

Please use the profile in your passes

1. Use `optimize_*_for_speed` and `optimize_*_for_size`.
   (* is one of `function`, `bb`, `insn`, `loop`, `loop_nest`)
2. `optimize_*_for_size` is now two-state predicate.
   Returned value is `optimize_size_level` enum:
   - 0: `OPTIMIZE_SIZE_NO`
     optimize for speed, this may be hot part of program.
   - 1: `OPTIMIZE_SIZE_BALANCED`
     this is likely not hot part of program but evidence is low; avoid bloat but do not do extreme tradeoffs
   - 2: `OPTIMIZE_SIZE_MAX`
     do everything possible to reduce code size

   (A lot of target specific work is needed here)
3. Use persistent loop info in loop transformations
   Most loop transforms are miss for loops iterating 0 or very few times
4. Use probabilities and counts to guide decisions about code paths

Please use the profile in your passes

1. Use `optimize_*_for_speed` and `optimize_*_for_size`.
   (* is one of `function`, `bb`, `insn`, `loop`, `loop_nest`)
2. `optimize_*_for_size` is now two-state predicate.
   Returned value is `optimize_size_level` enum:
   - 0: `OPTIMIZE_SIZE_NO`
     optimize for speed, this may be hot part of program.
   - 1: `OPTIMIZE_SIZE_BALANCED`
     this is likely not hot part of program but evidence is low; avoid bloat but do not do extreme tradeoffs
   - 2: `OPTIMIZE_SIZE_MAX`
     do everything possible to reduce code size

   (A lot of target specific work is needed here)
3. Use persistent loop info in loop transformations
   Most loop transforms are miss for loops iterating 0 or very few times
4. Use probabilities and counts to guide decisions about code paths

Future plans

1. Fix remaining bugs
2. More test coverage (302 testcases in trunk compared to 76 in GCC 13)
3. Track more lnt testcases
4. Integrate histogram profiling code (see Ondra's talk!)
5. Make vectorizer to use histogram profiles
6. Set up auto-FDO performance testing
7. Enable partitioning for more targets
8. Profile feedback at LTO linktime (no need to recompile)
9. ...

Thank you!