Feedback-Directed Optimization for OCaml

Greta Yorsh

Tools and Compilers Group

Jane Street

Which one is faster?

list.mli

```
val map2_exn : 'a t -> 'b t -> f:('a -> 'b -> 'c) -> 'c t

val map2 : 'a t -> 'b t -> f:('a -> 'b -> 'c) -> 'c t Or_unequal_lengths.t
```

examples

```
map2_exn [1;2;3] [3;2;1] ~f:(fun x y -> x + y)
```

```
[4; 4; 4]
```

```
map2_exn [1;2;3] [] ~f:(fun x y -> x + y)
```

```
raise Invalid_argument
```

```
map2 [1;2;3] [] ~f:(fun x y -> x + y)
```

```
Unequal_lengths
```

```
map2 [1;2;3] [3;2;1;] ~f:(fun x y -> x + y)
```

```
OK [4; 4; 4]
```

```
module Or_unequal_lengths : sig
  type 'a t =
     | Ok of 'a
     | Unequal_lengths
end
```

Which one is faster?

list.mli

```
val map2_exn : 'a t -> 'b t -> f:('a -> 'b -> 'c) -> 'c t

val map2 : 'a t -> 'b t -> f:('a -> 'b -> 'c) -> 'c t Or_unequal_lengths.t
```

list.ml

```
let check_length2_exn name l1 l2 =
  let n1 = length l1 in
  let n2 = length l2 in
  if n1 <> n2 then invalid_argf "length mismatch in %s: %d <> %d" name n1 n2 ()

let check_length2 l1 l2 ~f =
  if length l1 <> length l2 then Or_unequal_lengths.Unequal_lengths else Ok (f l1 l2)

let map2 l1 l2 ~f = check_length2 l1 l2 ~f:(map2_ok ~f)

let map2_exn l1 l2 ~f =
  check_length2_exn "map2_exn" l1 l2;
  map2_ok l1 l2 ~f
```

How much noise?

list.mli

```
val map2_exn : 'a t -> 'b t -> f:('a -> 'b -> 'c) -> 'c t

val map2 : 'a t -> 'b t -> f:('a -> 'b -> 'c) -> 'c t Or_unequal_lengths.t
```

- up to 1%
- up to 2%
- up to 5%
- up to 10%
- more than 10%

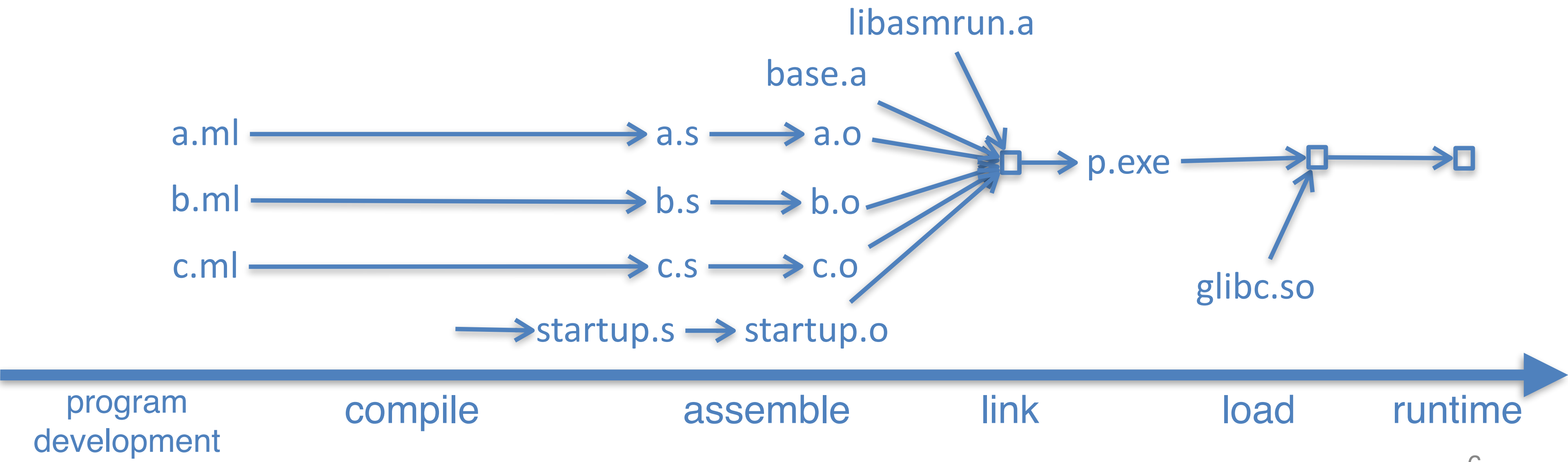
My benchmark says that
map2 is 10% faster than map2_exn
and I have no idea why that might be.
I'm not hitting the exception case.

Performance is sensitive to code layout

- order of functions
- order of basic blocks within a function
- alignment of branch targets
- density of branches
- page alignment

When is code layout determined?

- order of functions
- order of basic blocks within a function
- alignment of branch targets
- density of branches
- page alignment



When is code layout determined?

bench.ml

```
let l1 = .. in
let l2 = .. in
let f x y = x + y
let test_map2 () =
  Base.List.map2 l1 l2 f
let test_map2_exn () =
  Base.List.map2_exn
      l1 l2 f
...
```

base/list.ml

```
let check_length2 l1 l2 ~f =
  if length l1 <> length l2
  then Unequal_lengths
  else Ok (f l1 l2)

let map2 l1 l2 ~f = ...
```

stdlib/list.ml

```
let rev_map2 l1 l2 ~f =
  ...
```

bench.s

```
.section .text
camlBench__f_21:
 ...
camlBench__test_map2_45:
 ...
L149:
 movq (%rsp), %rbx
 cmpq %rbx, %rax
 je   L148
 movl $1, %eax
 addq $40, %rsp
 ret
 .align    2
L148:
 movq 8(%rsp), %rax
 movq 16(%rsp), %rbx
 movq 24(%rsp), %rdi
 call camlStdlib__list__rev_map2_63
 ...
camlBench__map2_exn_73:
 ...
.section .data
camlBench__l1_47: ...
camlBench__l2_57: ...
camlBench__map2_45_closure: ...
```

ELF file

```
header
text section
data section
symbol tables
debug info
...
```

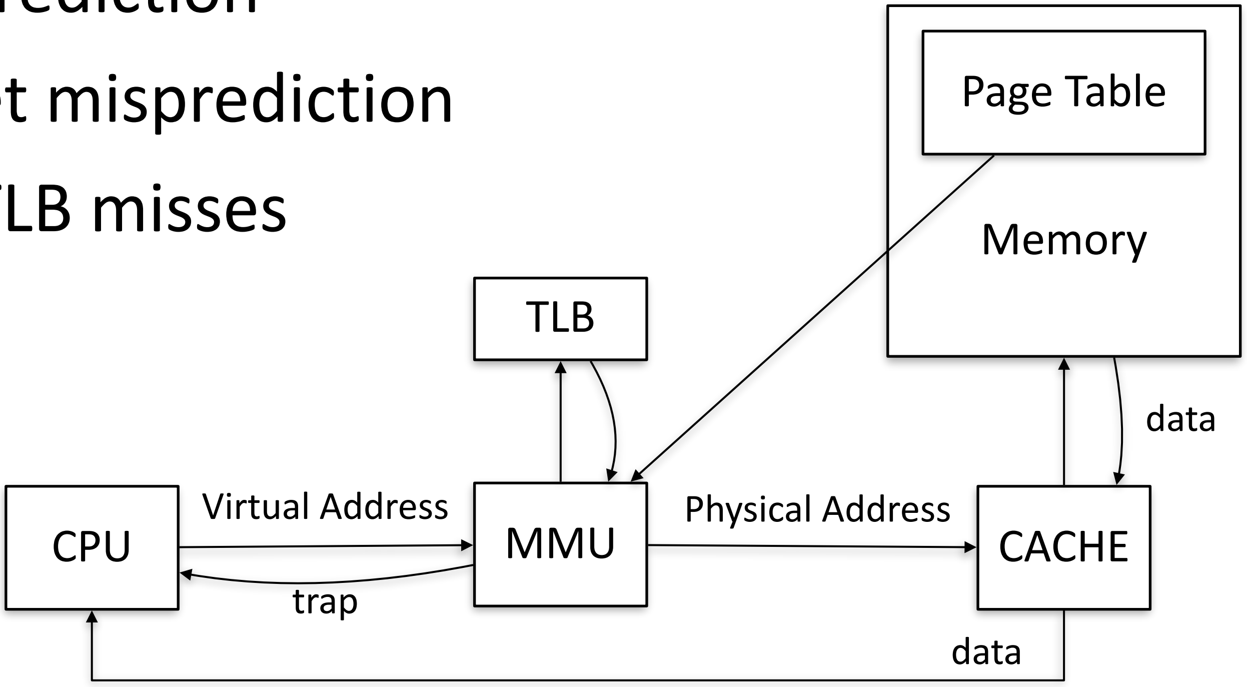
virtual memory

```
operating system
shared libraries

dynamic data

static data

text segment
...
```

program development → compile → assemble → link → load → runtime

Why code layout matters?

- How does the behavior of the underlying system and hardware depend on the layout of code in memory?
  - instruction cache misses and conflicts
  - branch misprediction
  - branch target misprediction
  - instruction TLB misses



Order of functions

nm -n bench_list.exe

```
...
004022d0 T main
004022e6 T _start
00402310 t deregister_tm_clones
00402340 t register_tm_clones
00402380 t __do_global_dtors_aux
..
004050f0 T caml_apply2
..
004052e0 T camlBench_list__map2_65
00405430 T camlBench_list__anon_fn...
00405440 T camlBench_list__init_aux_968
004054a0 T camlBench_list__init_aux_901
00405500 T camlBench_list__entry
...
0046efc0 T camlStdlib__list__aux_2667
0046efd0 T camlStdlib__list__aux_2677
0046efe0 T camlStdlib__list__aux_2699
0046f000 T camlStdlib__list__find_2651
0046f020 T camlStdlib__list__length_aux_179
0046f040 T camlStdlib__list__length_193
0046f060 T camlStdlib__list__cons_203
..
0046f130 T camlStdlib__list__nth_opt_286
0046f160 T camlStdlib__list__nth_aux_296
```

```
0046f1c0 T camlStdlib__list__rev_append_331
0046f210 T camlStdlib__list__rev_345
0046f260 T camlStdlib__list__init_tailrec_aux_355
...
0046f9e0 T camlStdlib__list__rev_map2_637
0046fa30 T camlStdlib__list__rmap2_f_644
0046fad0 T camlStdlib__list__iter2_679
0046fb40 T camlStdlib__list__fold_left2_708
..
004bbcc0 T caml_oldify_local_roots
004bbff0 T caml_darken_all_roots_slice
..
004bd1d0 t sweep_slice
004bd2e0 t clean_slice
004bd500 t mark_slice
..
004c06e0 T caml_alloc_shr_for_minor_gc
..
004da140 t bf_insert_sweep
004da1c0 t bf_split
004da210 t bf_allocate_from_tree
004da390 t bf_init_merge
..
004da640 t ff_allocate
004daa90 t bf_remove
004dab90 t bf_merge_block
004dad80 t bf_allocate
004daf30 T caml_set_allocation_policy
004db060 T caml_fl_reset_and_switch_policy
...
```

Reorder functions to reduce iTLB misses

nm -n bench_list.fdo.exe

```
0000000002000000 T camlStdlib__list__length_aux_179
0000000002000020 T camlStdlib__list__rmap2_f_644
00000000020000c0 T camlStdlib__list__rev_append_331
0000000002000110 T caml_apply2
0000000002000150 t mark_slice
0000000002000940 t sweep_slice
0000000002000a50 t bf_allocate
0000000002000c00 T caml_oldify_one
0000000002000e20 T caml_page_table_lookup
0000000002000ea0 t bf_merge_block
0000000002001090 T camlBench_list__anon_fn...
00000000020010a0 T caml_oldify_mopup
0000000002001260 T caml_process_pending_signals_exn
0000000002001330 T caml_write_fd
00000000020013b0 t bf_split
0000000002001400 T caml_alloc_shr_for_minor_gc
0000000002001510 T caml_darken_all_roots_slice
0000000002001640 T caml_oldify_local_roots
000000000200224e T _start
0000000002002280 t deregister_tm_clones
00000000020022b0 t register_tm_clones
00000000020022f0 t __do_global_dtors_aux
0000000002002310 t frame_dummy
000000000200233d T caml_startup__code_begin
0000000002002340 T caml_program
0000000002002ac0 T caml_curry11
0000000002002b10 T caml_curry11_1_app
...
```

page aligned

functions from ocaml standard library

C functions from ocaml runtime

user-defined ocaml functions

Link-time function reordering

- Compile each function into its own uniquely-named section
- Linker merges all input .text sections into a single output .text section
- Instruct the linker to place hot function sections first

bench.ml

```
let l1 = .. in
let l2 = .. in
let f x y = x + y
let test_map2 () = ...
let test_map2_exn () = ...
```

```
gcc -ffunction-sections runtime/roots_nat.c ...

ocamlopt -function-sections bench.ml ...
```

bench.s

```
.section .text.camlBench__f_21
camlBench__f_21:
 ...
.section .text.camlBench__test_map2_45
camlBench__test_map2_45:
 ...
.section .text.camlBench__test_map2_exn_73
camlBench__map2_exn_73:
 ...
.section .data
camlBench__l1_47: ...
camlBench__l2_57: ...
camlBench__map2_45_closure: ...
```

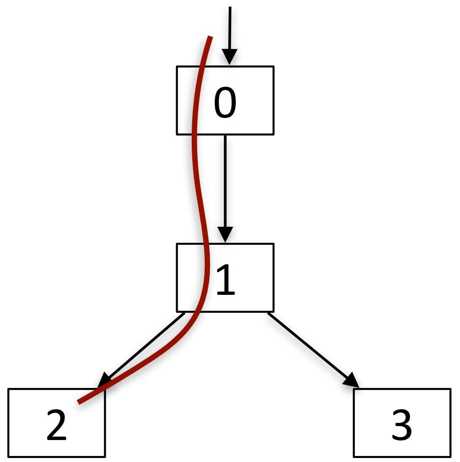
gnu linker script

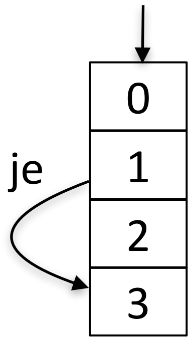
```
.text: {
 . = ALIGN(0x2000000);
 *(.text.camlStdlib__list__length_aux_179)
 *(.text.camlStdlib__list__rmap2_f_644)
 *(.text.camlStdlib__list__rev_append_331)
 *(.text.caml_apply2)
 *(.text.mark_slice)
  ...
  *(.text .text.*)
}
```

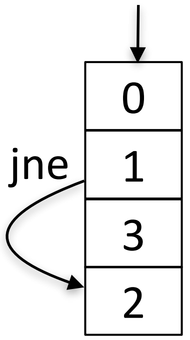
Order of basic blocks

base/list.ml

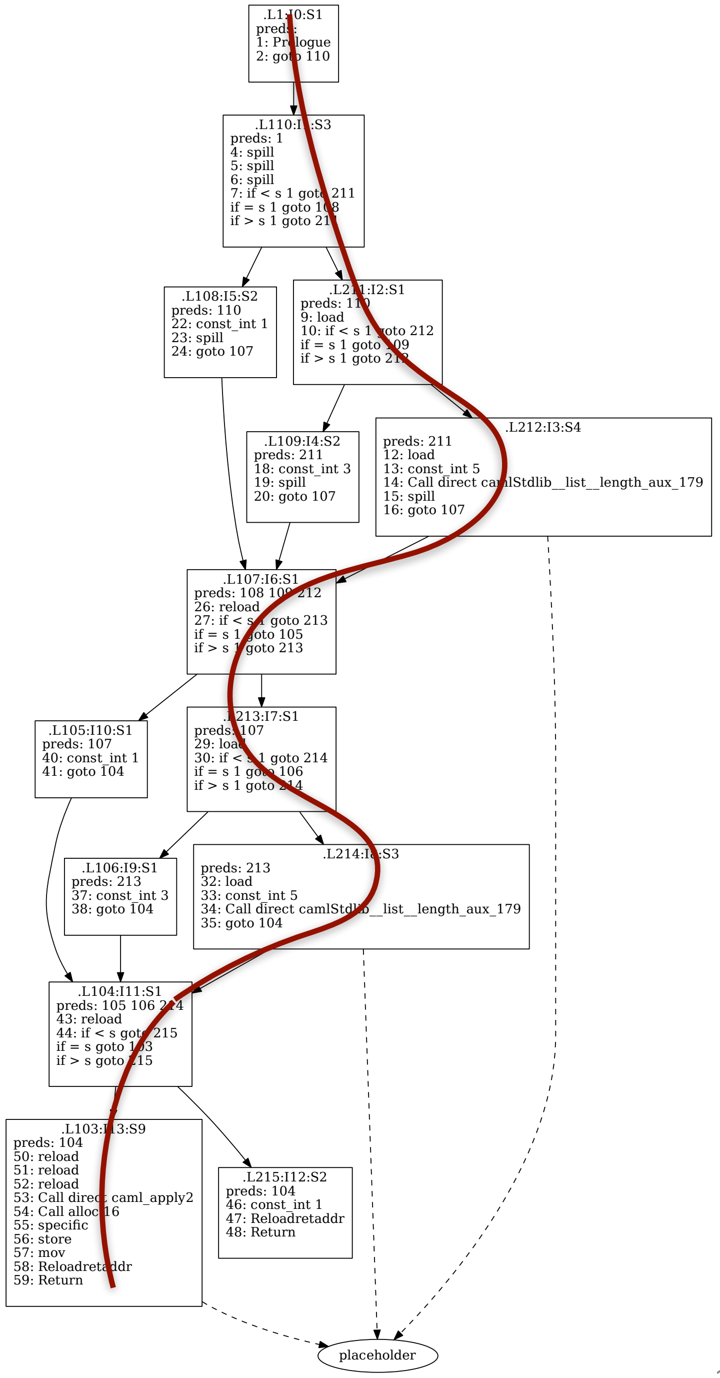
```
0: let check_length2 l1 l2 ~f =
1:   if length l1 <> length l2
2:   then Unequal_lengths
3:   else Ok (f l1 l2)
```







- conditional branches predicted as "not taken"
- hot path falls through is fastest



```
.L1:I0:S1
preds:
1: Prologue
2: goto 110

.L110:I1:S3
preds: 1
4: spill
5: spill
6: spill
7: if < s 1 goto 211
if = s 1 goto 108
if > s 1 goto 211

.L108:I5:S2
preds: 110
22: const_int 1
23: spill
24: goto 107

.L211:I2:S1
preds: 110
9: load
10: if < s 1 goto 212
if = s 1 goto 109
if > s 1 goto 212

.L109:I4:S2
preds: 211
18: const_int 3
19: spill
20: goto 107

.L212:I3:S4
preds: 211
12: load
13: const_int 5
14: Call direct camlStdlib__list__length_aux_179
15: spill
16: goto 107

.L107:I6:S1
preds: 108 109 212
26: reload
27: if < s 1 goto 213
if = s 1 goto 105
if > s 1 goto 213

.L105:I10:S1
preds: 107
40: const_int 1
41: goto 104

.L213:I7:S1
preds: 107
29: load
30: if < s 1 goto 214
if = s 1 goto 106
if > s 1 goto 214

.L106:I9:S1
preds: 213
37: const_int 3
38: goto 104

.L214:I8:S3
preds: 213
32: load
33: const_int 5
34: Call direct camlStdlib__list__length_aux_179
35: goto 104

.L104:I11:S1
preds: 105 106 214
43: reload
44: if < s goto 215
if = s goto 103
if > s goto 215

.L103:I13:S9
preds: 104
50: reload
51: reload
52: reload
53: Call direct caml_apply2
54: Call alloc 16
55: specific
56: store
57: mov
58: Reloadretaddr
59: Return

.L215:I12:S2
preds: 104
46: const_int 1
47: Reloadretaddr
48: Return

placeholder
```

Performance is sensitive to code layout

- Benchmarking: randomize layout

*"**STABILIZER**: Statistically Sound Performance Evaluation"*
Charlie Curtsinger and Emery Berger (ASPLOS'13)
https://github.com/ccurtsinger/stabilizer

UMass Amherst

- Compilation: optimize code layout

What's hot and what's not?

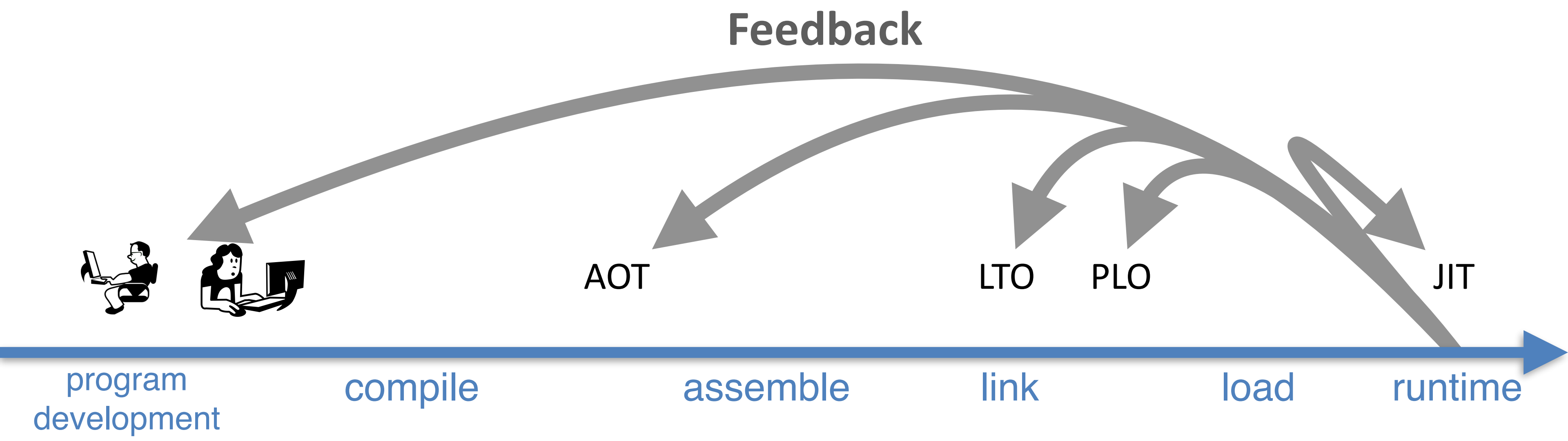
Profile-guided optimization

- Obtain execution **profile**
  - what code paths are **hot**
  - frequency of execution of code
- Use execution profile to **guide** optimization decisions
- Profile **does not affect safety** of transformations

How does the compiler obtain program's execution profile?

- Code patterns
  - exceptions are cold
- Source annotations
  - `@hot @cold`
  - `likely unlikely __builtin_expect`
- Data collected during program execution

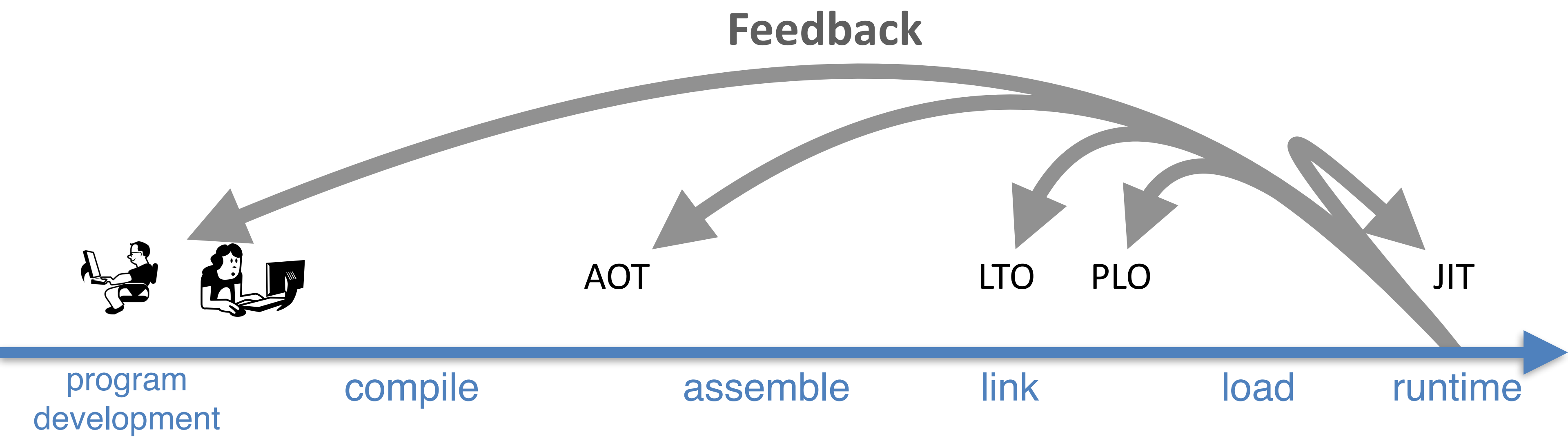
Feedback-directed optimization



- FDO: Feedback-Directed Optimization
- PGO: Profile-Guided Optimization
- LTO: Link-Time Optimization
- PLO: Post-Link Optimization
- AOT: Ahead-of-time compilation
- JIT: Just-int-time compilation

- code layout
- inlining
- register allocation
- indirect branch/call promotion

Feedback-directed optimization



overhead vs accuracy

- How to collect relevant data at runtime?
- How to relate dynamic data back to static representation used for optimizations?

Profile collection methods

- instrumentation
  - instrumented build `gcc -fprofile-generate ... -o p.with_instr.exe`
  - training run `./p.with_instr.exe inputs`
  - optimized build `gcc -fprofile-use ... -o p.exe`

base/list.ml

```
0: let check_length2 l1 l2 ~f =
1:   if length l1 <> length l2
2:   then Unequal_lengths
3:   else Ok (f l1 l2)
```

bench.ml

```
let () = main ();
```

with instrumentation

```
val call_to_check_length2 = ref 0;
val cond_at_l1_true = ref 0;
val cond_at_l1_false = ref 0;
let check_length2 l1 l2 ~f =
  incr call_to_check_length2;
  if length l1 <> length l2
  then (incr cond_at_l1_true; Unequal_lengths)
  else (incr cond_at_l1_false; Ok (f l1 l2))
```

```
let () = main (); write_profile ()
```

Profile collection methods

- instrumentation
  - instrumented build `gcc -fprofile-generate ... -o p.with_instr.exe`
  - training run `./p.with_instr.exe inputs`
  - optimized build `gcc -fprofile-use ... -o p.exe`
- sampling
  - normal build `gcc ... -o p.exe`
  - run with sampling `perf record -e cycles:u -j any,u ./p.exe inputs`
  - decode raw data `create_gcov --profile=perf.data --binary=p.exe`
  - build with profile `gcc ... -fauto-profile -o p.exe`

Sampling

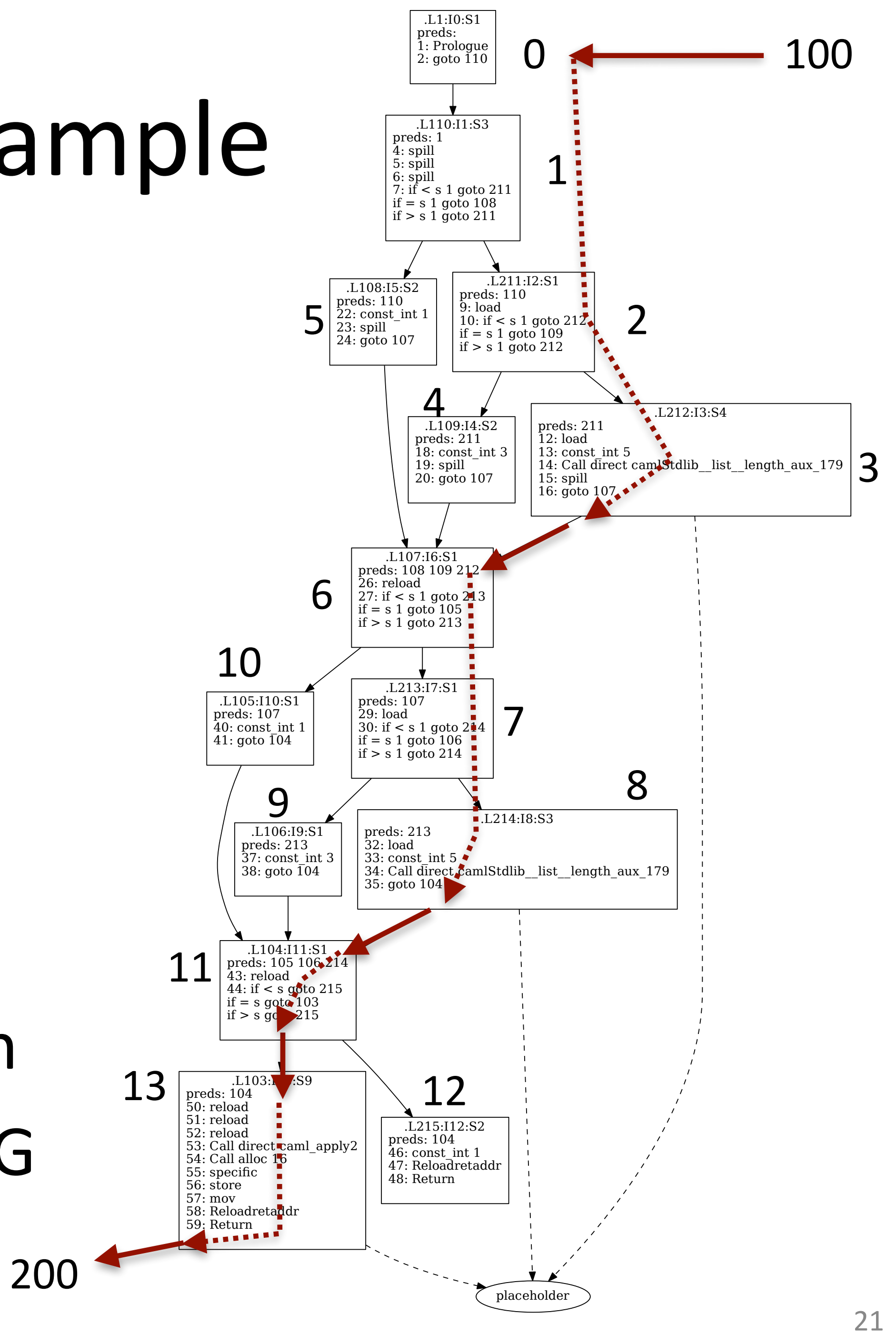
- Rely on hardware support
- Interrupt-based sampling
  - hardware execution counters (cycles, branches, etc)
  - interrupt when counter overflows
- Event-based sampling (PEBS)
  - hardware logs current state when counter overflows
- Sampling with last branch record (LBR)
  - hardware continuously records the last **k** branches **taken**
  - for example, k=32 in Skylake
  - hardware logs LBR when counter overflows

LBR example

- branches taken

from	to
100	0
3	6
8	11
11	13
13	200

- all other blocks fall through
- connect the trace using CFG



```
.L1:I0:S1
preds:
1: Prologue
2: goto 110

.L110:I1:S3
preds: 1
4: spill
5: spill
6: spill
7: if < s 1 goto 211
if = s 1 goto 108
if > s 1 goto 211

.L108:I5:S2
preds: 110
22: const_int 1
23: spill
24: goto 107

.L211:I2:S1
preds: 110
9: load
10: if < s 1 goto 212
if = s 1 goto 109
if > s 1 goto 212

.L109:I4:S2
preds: 211
18: const_int 3
19: spill
20: goto 107

.L212:I3:S4
preds: 211
12: load
13: const_int 5
14: Call direct camlStdlib__list__length_aux_179
15: spill
16: goto 107

.L107:I6:S1
preds: 108 109 212
26: reload
27: if < s 1 goto 213
if = s 1 goto 105
if > s 1 goto 213

.L105:I10:S1
preds: 107
40: const_int 1
41: goto 104

.L213:I7:S1
preds: 107
29: load
30: if < s 1 goto 214
if = s 1 goto 106
if > s 1 goto 214

.L106:I9:S1
preds: 213
37: const_int 3
38: goto 104

.L214:I8:S3
preds: 213
32: load
33: const_int 5
34: Call direct camlStdlib__list__length_aux_179
35: goto 104

.L104:I11:S1
preds: 105 106 214
43: reload
44: if < s goto 215
if = s goto 103
if > s goto 215

.L103:I13:S9
preds: 104
50: reload
51: reload
52: reload
53: Call direct caml_apply2
54: Call alloc 16
55: specific
56: store
57: mov
58: Reloadretaddr
59: Return

.L215:I12:S2
preds: 104
46: const_int 1
47: Reloadretaddr
48: Return
```

Last Branch Record (LBR)

- Cheap way to gather partial context of a sample code address
- Recording does not incur overhead on the execution, only reading
- Overhead low enough to use in production
- More accurate than instruction and branch counters

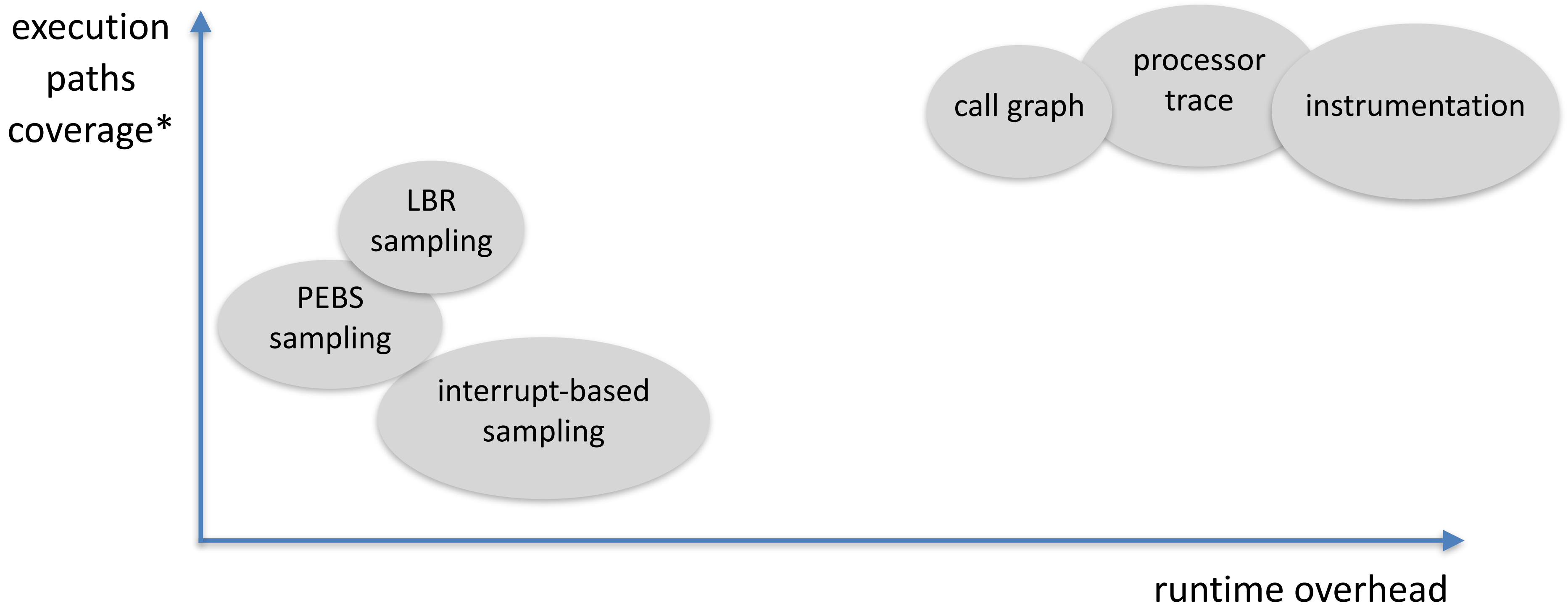
"Taming Hardware Event Samples for Precise and Versatile Feedback Directed Optimizations" Dehao Chen, Neil Vachharajani, Robert Hundt, Xinliang D. Li, Stéphane Eranian, Wenguang Chen, Weimin Zheng (IEEE Transactions on Computers, 2013)

Instrumentation vs Sampling

- complete information
- source level mapping is easy

- profiling run is slow: instrumentation causes overhead and affects optimizations
- instrumentation can alter execution frequencies
- no timing information only frequency
- how to find representative inputs
- complicates workflow

Profile collection



For the purpose of code layout, is LBR as accurate as full call graph or instrumentation, at least in practice?

* profile accuracy depends on where the data is used

Modern tools rely on sampling with LBR

"**AutoFDO**: automatic feedback-directed optimization for warehouse-scale applications"
Dehao Chen, David Xinliang Li, and Tipp Moseley (CGO 2016)
https://github.com/google/autofdo

Google

"**BOLT**: A Practical Binary Optimizer for Data Centers and Beyond"
Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni (CGO 2019)
https://github.com/facebookincubator/BOLT

facebook

"**OCamlFDO:** feedback-directed optimization for OCaml"
work in progress, 2019-present
https://github.com/gretay-js/ocamlfdo

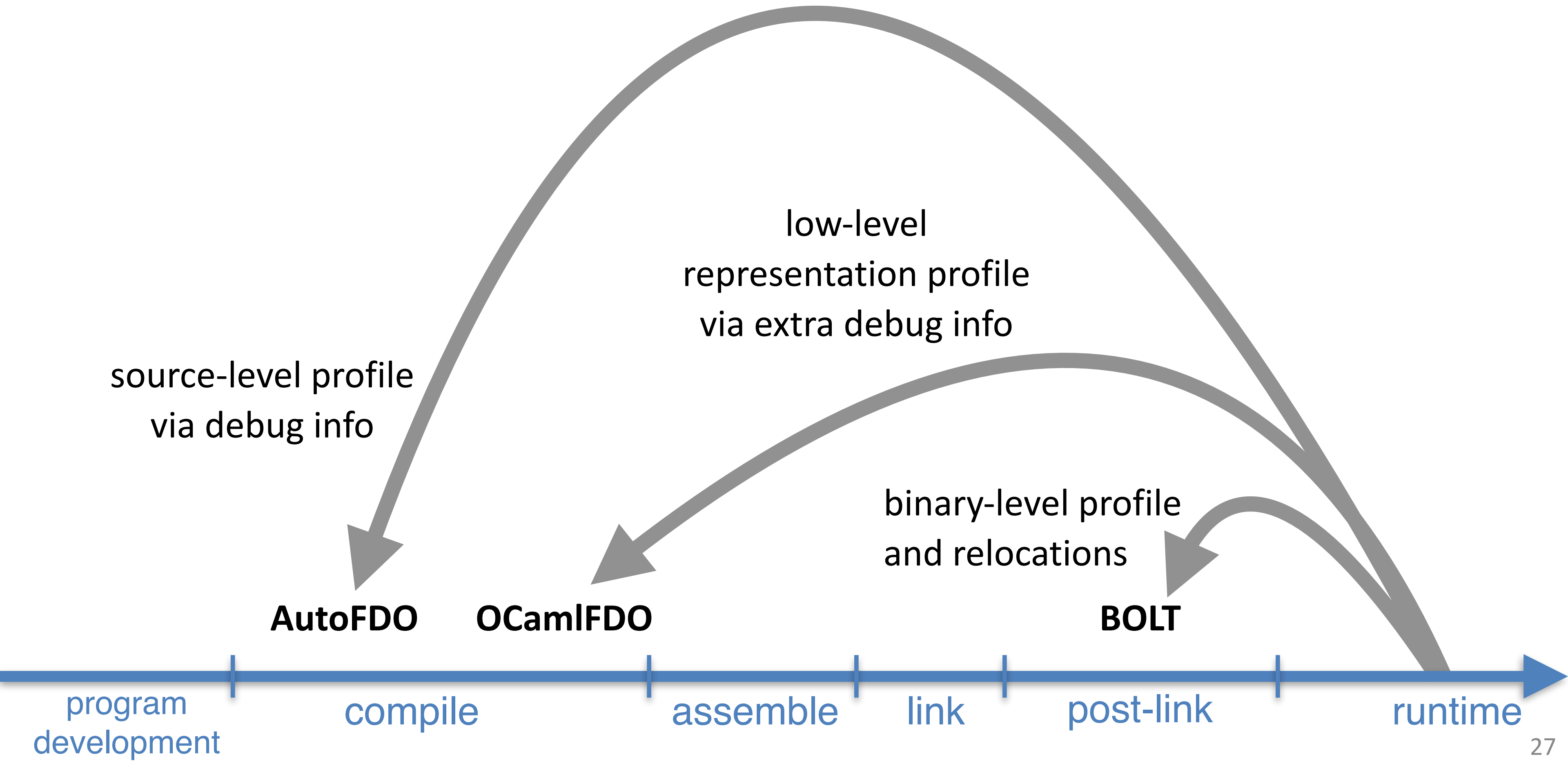
Jane Street

OCamlFDO

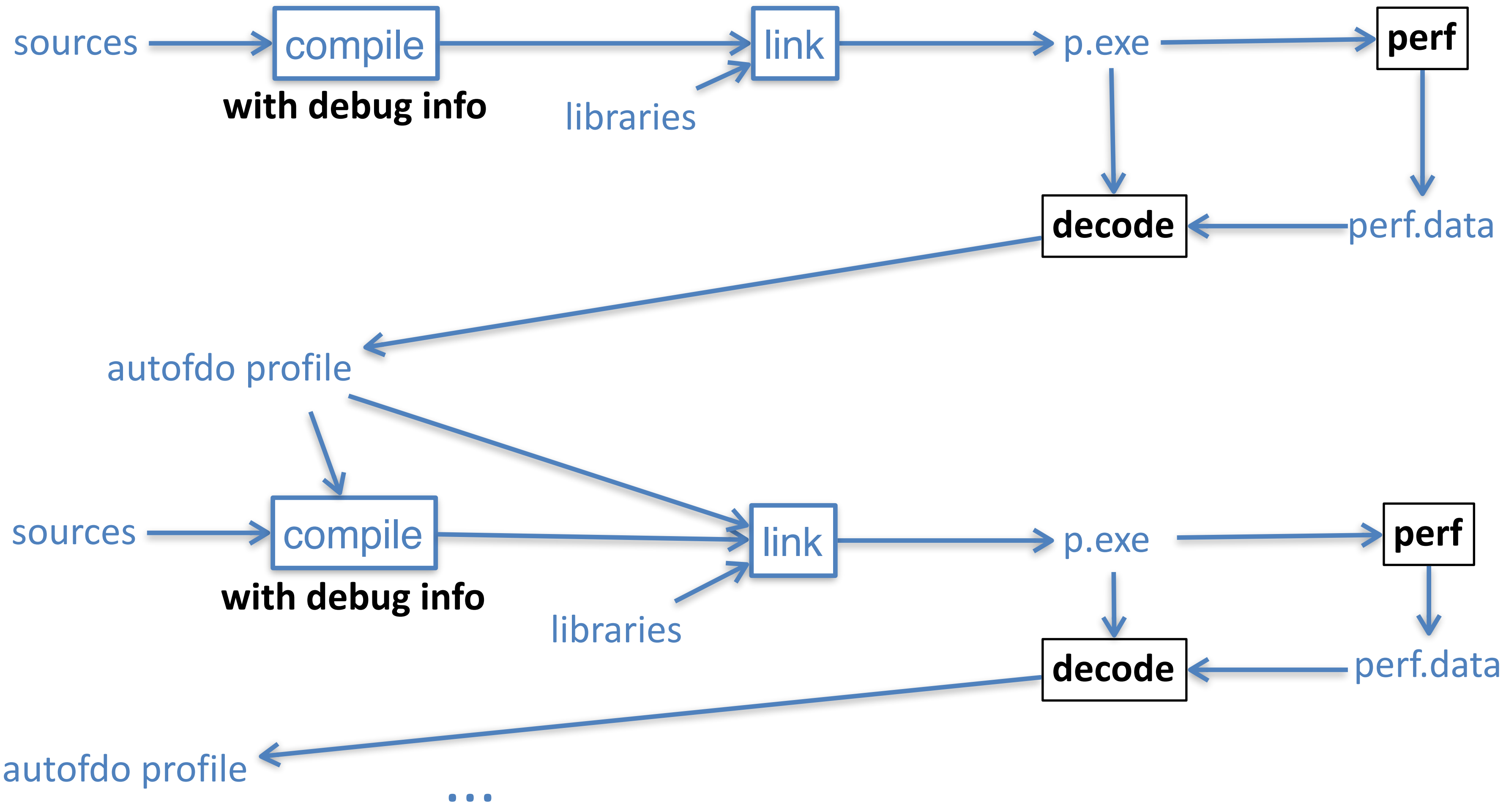
- First code-layout optimizations for ocaml compiler
- First sampling-based profile use in ocaml compiler
- Reduces noise in benchmarking
- Improves performance of large binaries

- New approach to execution profiles
- New internal representation for ocaml compiler backend
- Integration with build systems and workflows

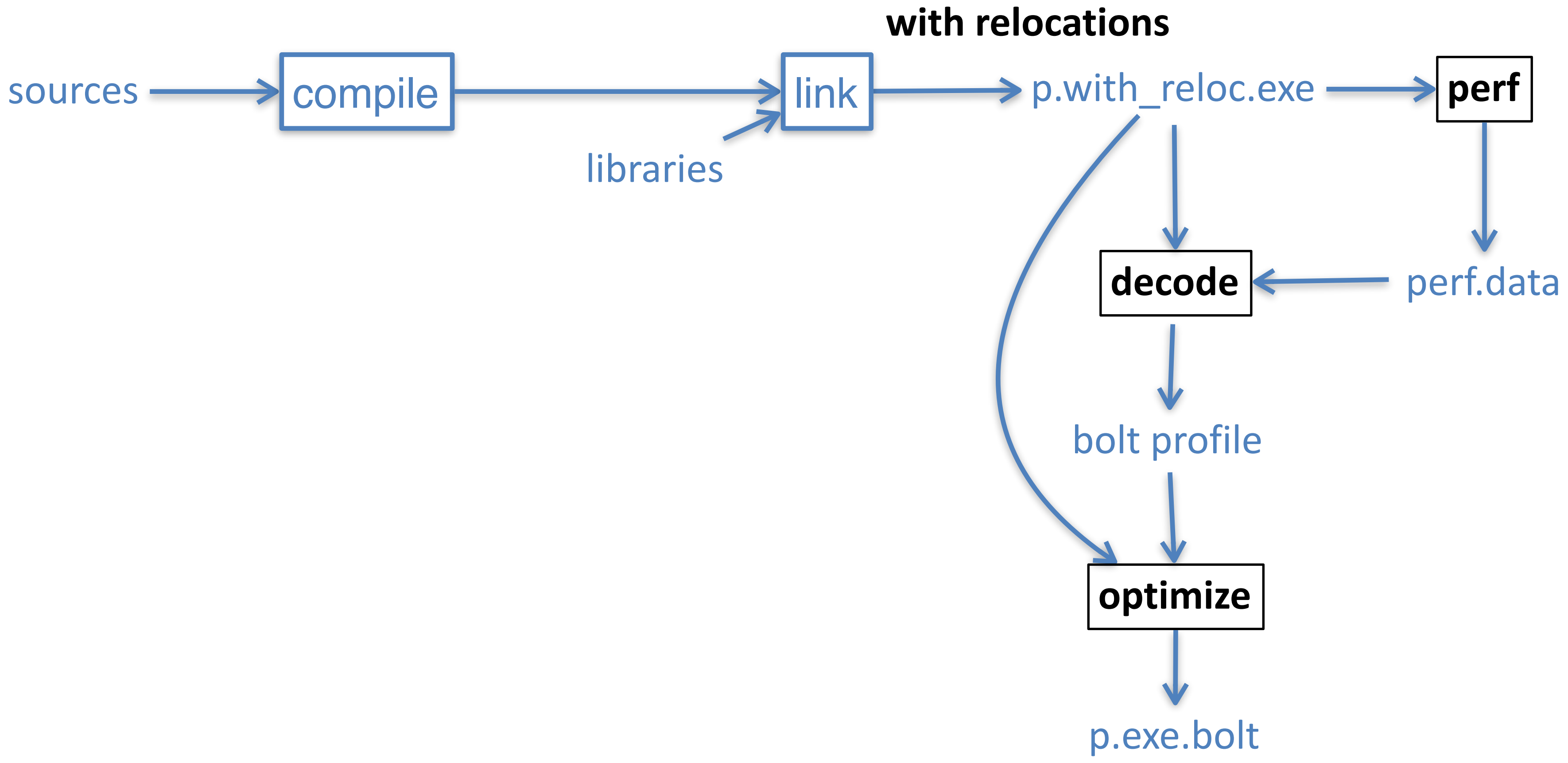
Feedback-directed optimization



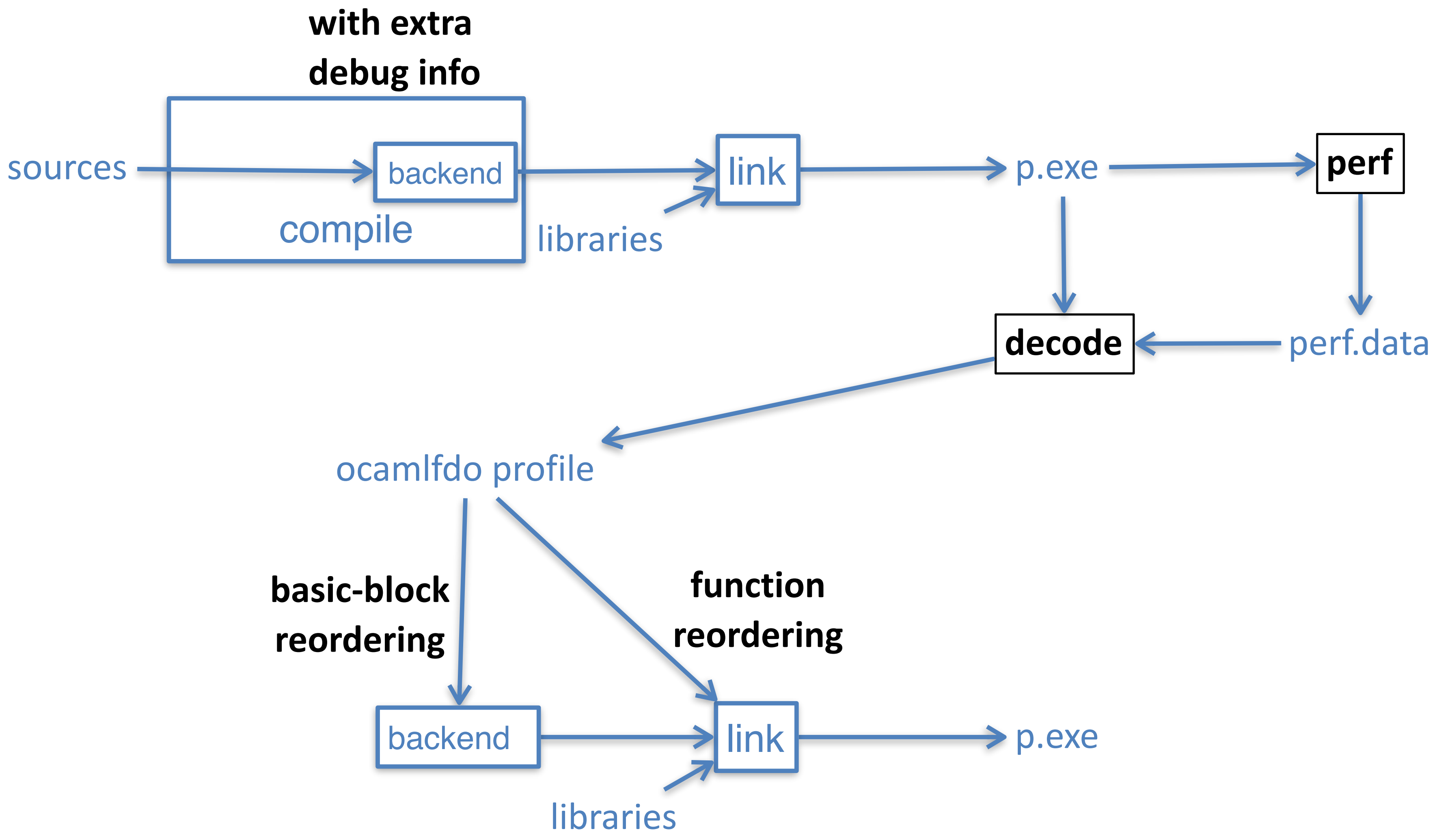
AutoFDO


sources
compile
with debug info
libraries
link
p.exe
perf
decode
perf.data
autofdo profile
sources
compile
with debug info
libraries
link
p.exe
perf
decode
perf.data
autofdo profile
…

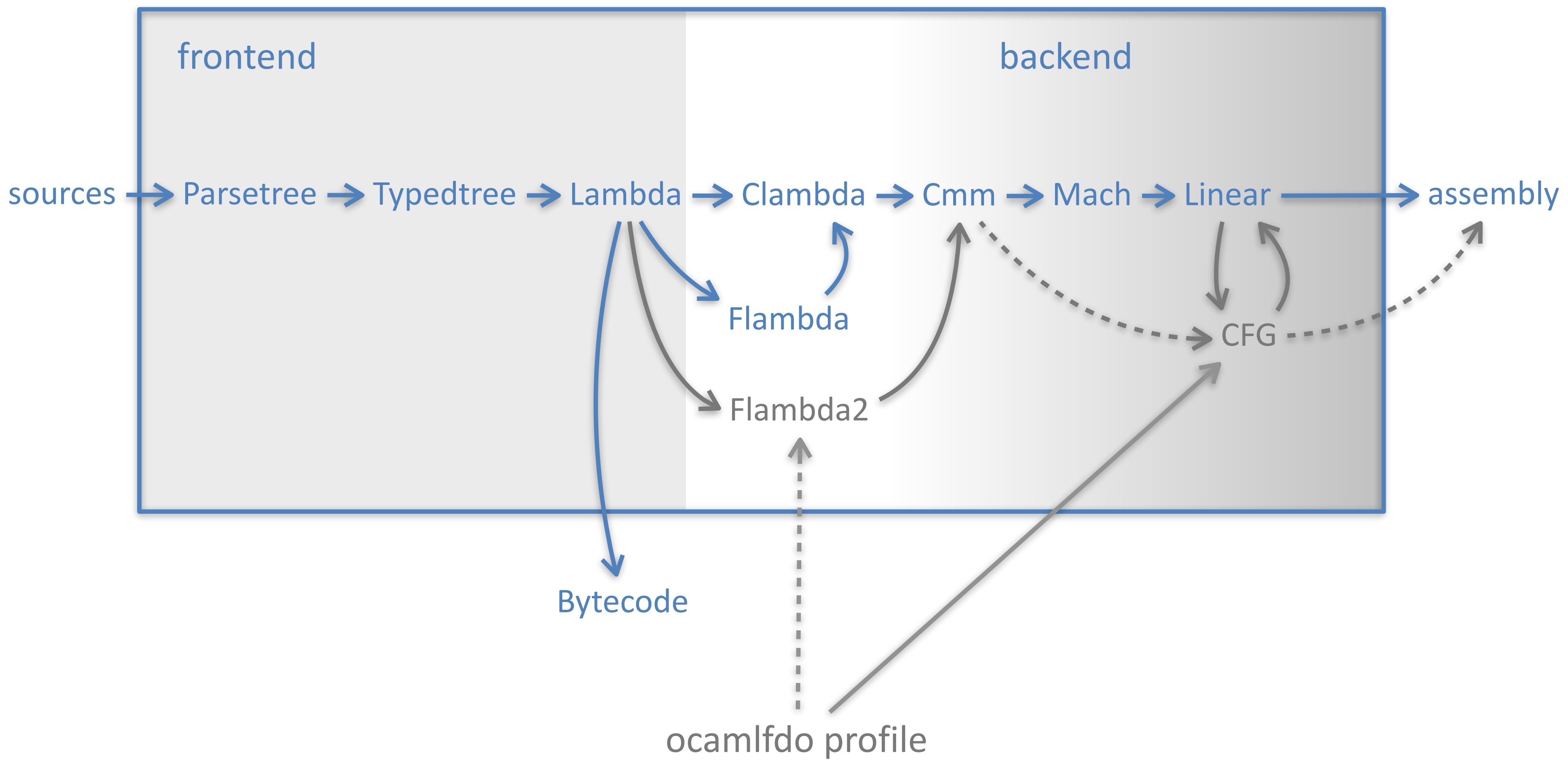
BOLT


with relocations
sources
compile
link
libraries
p.with_reloc.exe
perf
decode
perf.data
bolt profile
optimize
p.exe.bolt

OCamlFDO

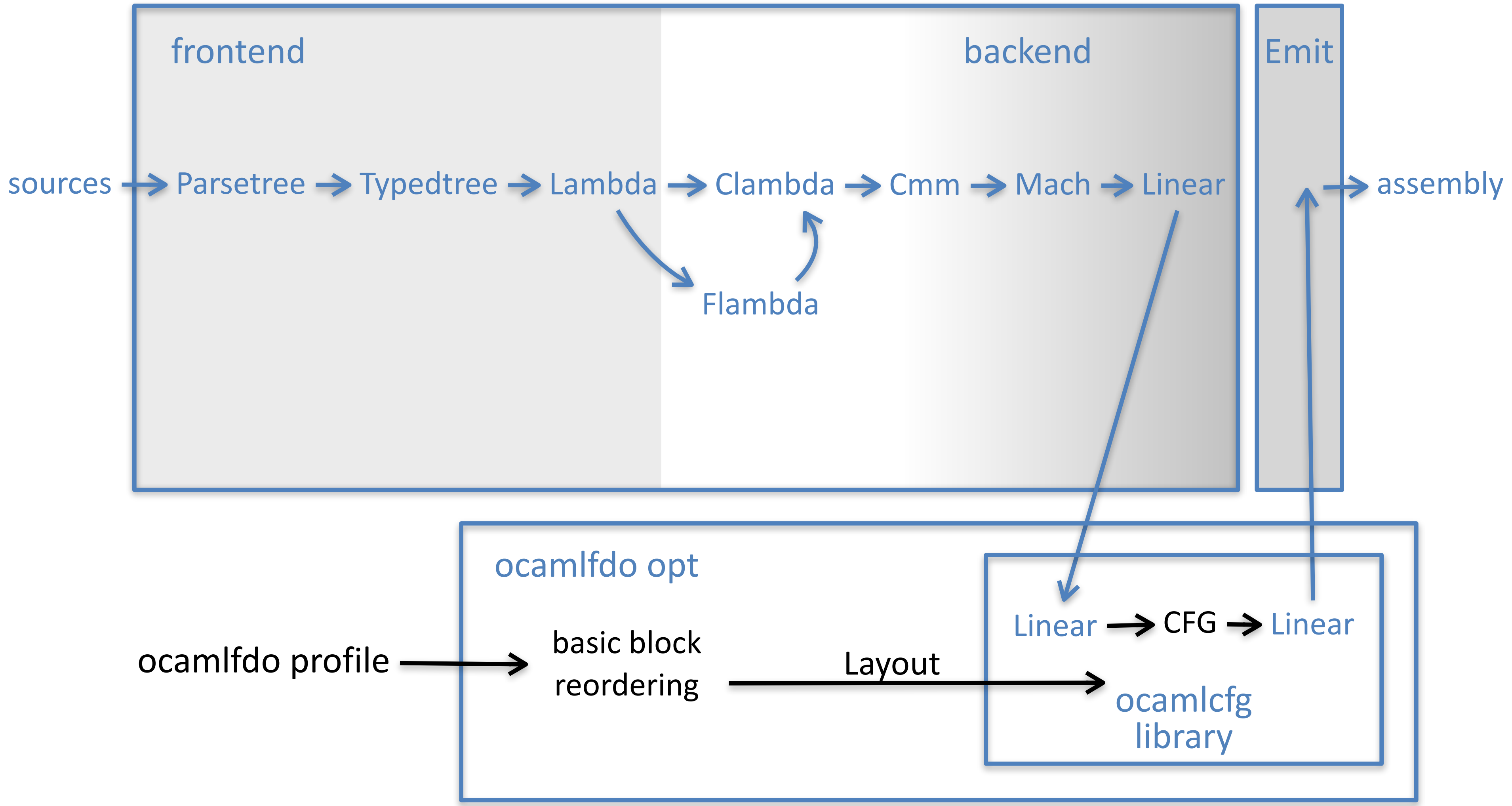

with extra debug info
sources
backend
compile
libraries
link
p.exe
perf
decode
perf.data
ocamlfdo profile
basic-block reordering
function reordering
backend
link
libraries
p.exe

OCaml compiler with FDO



OCaml compiler with FDO

current implementation



Representation: CFG + Layout

- Layout is a sequence of labels of CFG nodes
- Any permutation that preserves entry is a legal layout
- Implemented in **ocamlcfg library**
  - depends only on ocaml compiler internals
  - from Linear to CFG involves reconstruction of exception flow
  - transformations: dead code elimination, jump threading
  - use Layout to transform CFG back to Linear representation
  - interface: CFG is read-only
- Testing
  - Linear --> CFG + Layout --> Linear is identity
  - random reorder basic blocks
  - large code base

Basic-block reordering

- implemented in "ocamlfdo opt" command
- read Linear from file
- use ocamlcfg library to transform Linear to CFG
- if profile exists
    - read profile and check if stale
    - annotate CFG with execution counters from the profile including partial traces from LBR
    - choose layout of basic blocks using clustering heuristic
    - modifies the Layout, not the structure of the CFG
- if profile does not exist
    - add md5 at compilation unit or function level for stale profile detection
    - add extra debug info
- transforms back to Linear and write to file

Debug info

- Map binary addresses to source locations
- Used by many tools
  - gdb ./p.exe
    - next
    - backtrace
  - perf annotate

- **not** OCaml exceptions and backtraces

Debug info

- source location
  - file name and line number
  - discriminator: distinguish multiple execution paths on the same source line
- inlined stack: source locations of inlined calls

base/list.ml

```
0: let check_length2 l1 l2 ~f =
1:    if length l1 <> length l2 then Unequal_lengths else Ok (f l1 l2)
```

Debug info

base/list.s

```
camlBase__list__check_length2_606:
      .file 1       "base/list.ml"
      .loc  1       7       18
      .cfi_startproc
      subq  $40, %rsp
      .cfi_adjust_cfa_offset 40
.L108:
      movq  %rax, 8(%rsp)
      movq  %rbx, 16(%rsp)
      movq  %rdi, 24(%rsp)
      cmpq  $1, %rbx
      je    .L107
      .file 2       "stdlib/list.ml"
      .loc  2       23      4
      movq  8(%rbx), %rbx
      movl  $3, %eax
      .loc  2       23      12
      call  camlStdlib__list__length_aux_83@PLT
.L100:
      movq  %rax, (%rsp)
      jmp   .L106
      .align      4
.L107:
      movl  $1, %eax
      movq  %rax, (%rsp)
.L106:
      movq  8(%rsp), %rax
      cmpq  $1, %rax
      je    .L105
      .loc  2       23      4
      movq  8(%rax), %rbx
      movl  $3, %eax
      .loc  2       23      12
      call  camlStdlib__list__length_aux_83@PLT
```

ELF file

header
text section
data section
symbol tables
debug info
...

- map binary address to source location

Debug info

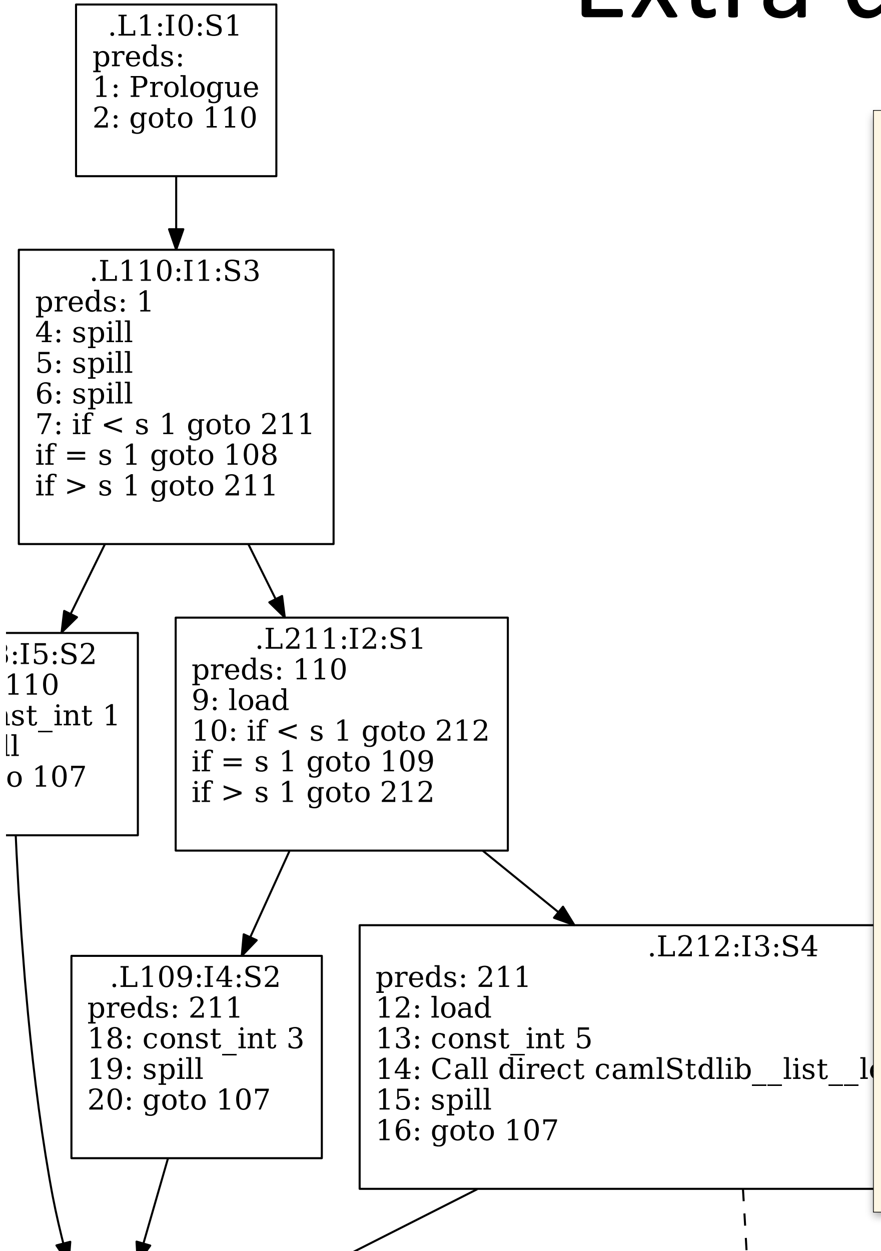
base/list.s

```
camlBase__list__check_length2_606:
        .file 1      "base/list.ml"
        .loc  1      7      18
        .cfi_startproc
        subq  $40, %rsp
        .cfi_adjust_cfa_offset 40
.L108:
        movq  %rax, 8(%rsp)
        movq  %rbx, 16(%rsp)
        movq  %rdi, 24(%rsp)
        cmpq  $1, %rbx
        je    .L107
        .file 2      "stdlib/list.ml"
        .loc  2      23     4
        movq  8(%rbx), %rbx
        movl  $3, %eax
        .loc  2      23     12
        call  camlStdlib__list__length_aux_83@PLT
.L100:
        movq  %rax, (%rsp)
        jmp   .L106
        .align       4
.L107:
        movl  $1, %eax
        movq  %rax, (%rsp)
.L106:
        movq  8(%rsp), %rax
        cmpq  $1, %rax
        je    .L105
        .loc  2      23     4
        movq  8(%rax), %rbx
        movl  $3, %eax
        .loc  2      23     12
        call  camlStdlib__list__length_aux_83@PLT
```

list.s with extra debug info

```
camlBase__list__check_length2_24:
        .file 1      "base/list.ml"
        .loc  1      7      18
        .cfi_startproc
        .file 2      "camlBench__check_length2_24.cmir-linear"
        .loc  2      1
        subq  $40, %rsp
        .cfi_adjust_cfa_offset 40
        .loc  2      4
        movq  %rax, 8(%rsp)
        .loc  2      5
        movq  %rbx, 16(%rsp)
        .loc  2      6
        movq  %rdi, 24(%rsp)
        .loc  2      7
        cmpq  $1, %rbx
        je    .L108
        .loc  2      9
        movq  8(%rbx), %rax
        .loc  2      10
        cmpq  $1, %rax
        je    .L109
        .loc  2      12
        movq  8(%rax), %rbx
        .loc  2      13
        movl  $5, %eax
        .loc  2      14
        call  camlStdlib__list__length_aux_179@PLT
.L100:
        .loc  2      15
        movq  %rax, (%rsp)
        .loc  2      16
        jmp   .L107
        .align       4
```

Extra debug info



bench.s with extra debug info

```
camlBase__list__check_length2_24:
        .file 1       "base/list.ml"
        .loc  1       7       18
        .cfi_startproc
        .file 2       "camlBench__check_length2_24.cmir-linear"
        .loc  2       1
        subq  $40, %rsp
        .cfi_adjust_cfa_offset 40
        .loc  2       4
        movq  %rax, 8(%rsp)
        .loc  2       5
        movq  %rbx, 16(%rsp)
        .loc  2       6
        movq  %rdi, 24(%rsp)
        .loc  2       7
        cmpq  $1, %rbx
        je    .L108
        .loc  2       9
        movq  8(%rbx), %rax
        .loc  2       10
        cmpq  $1, %rax
        je    .L109
        .loc  2       12
        movq  8(%rax), %rbx
        .loc  2       13
        movl  $5, %eax
        .loc  2       14
        call  camlStdlib__list__length_aux_179@PLT
.L100:
        .loc  2       15
        movq  %rax, (%rsp)
        .loc  2       16
        jmp   .L107
```

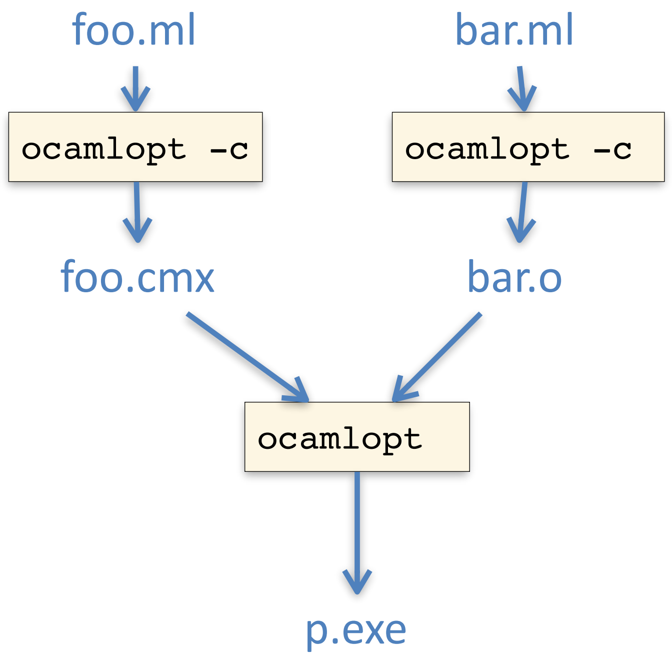
Extra debug info

- Location in the intermediate representation
- Each function has a separate ".file"
- Currently **instead** of source location
- Incomplete toolchain support for debug info
- Do not use binary with extra debug info in prod

Decode perf.data

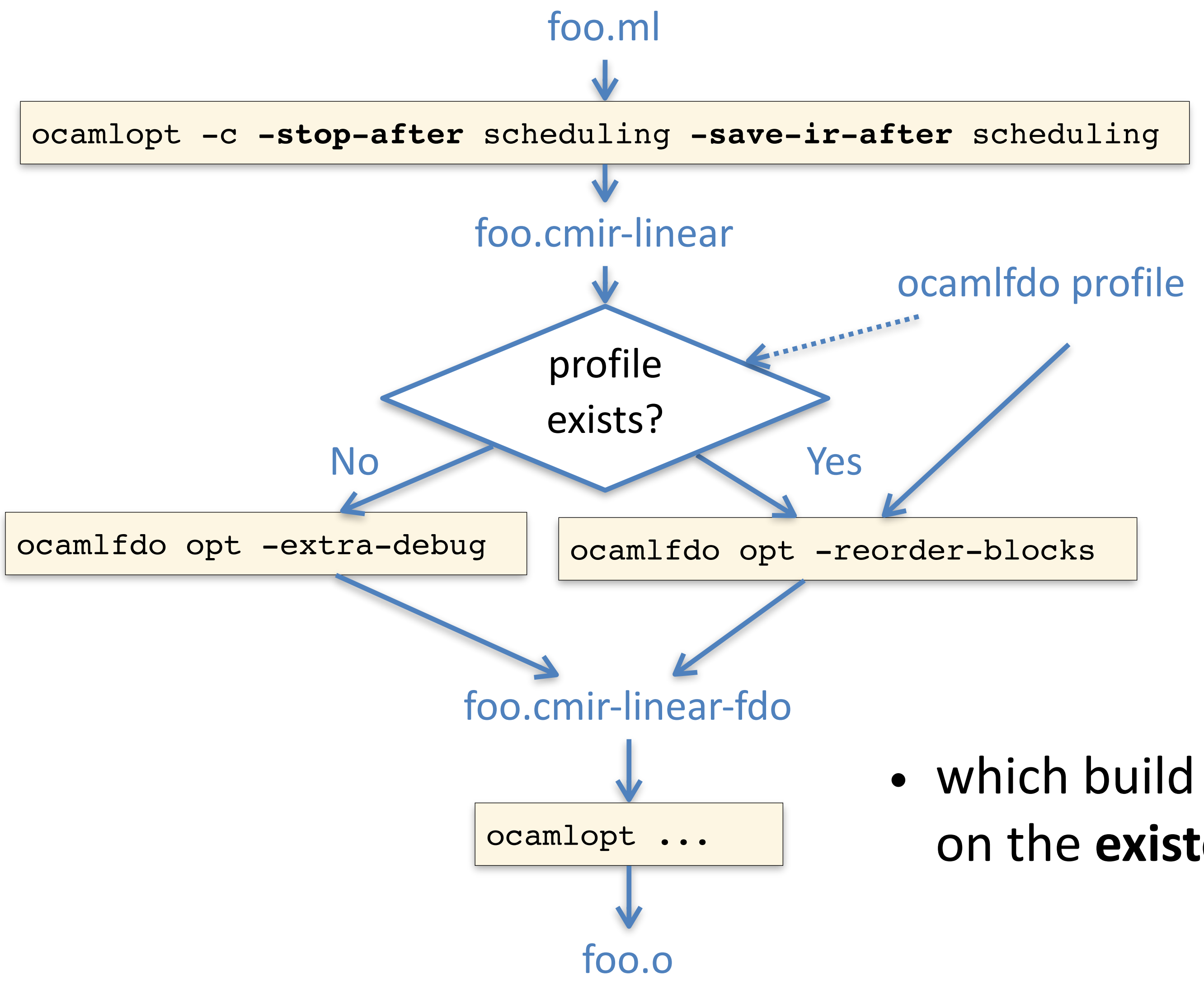
- Implementation
  - parse output of "perf script" to obtain and aggregate raw samples
  - use owee library to read debug info
  - use extra debug info to map dynamic data to Linear representation
  - merge multiple profiles
  - read md5 of compilation units/functions from the binary and save to profile for stale profiles detection
- Testing
  - use BOLT adapted for OCaml native code
    - handle missing frametable relocation
  - compare optimized layout produced by BOLT and OCamlFDO
  - convert BOLT profile to OCamlFDO profile and compare counters

Build system integration



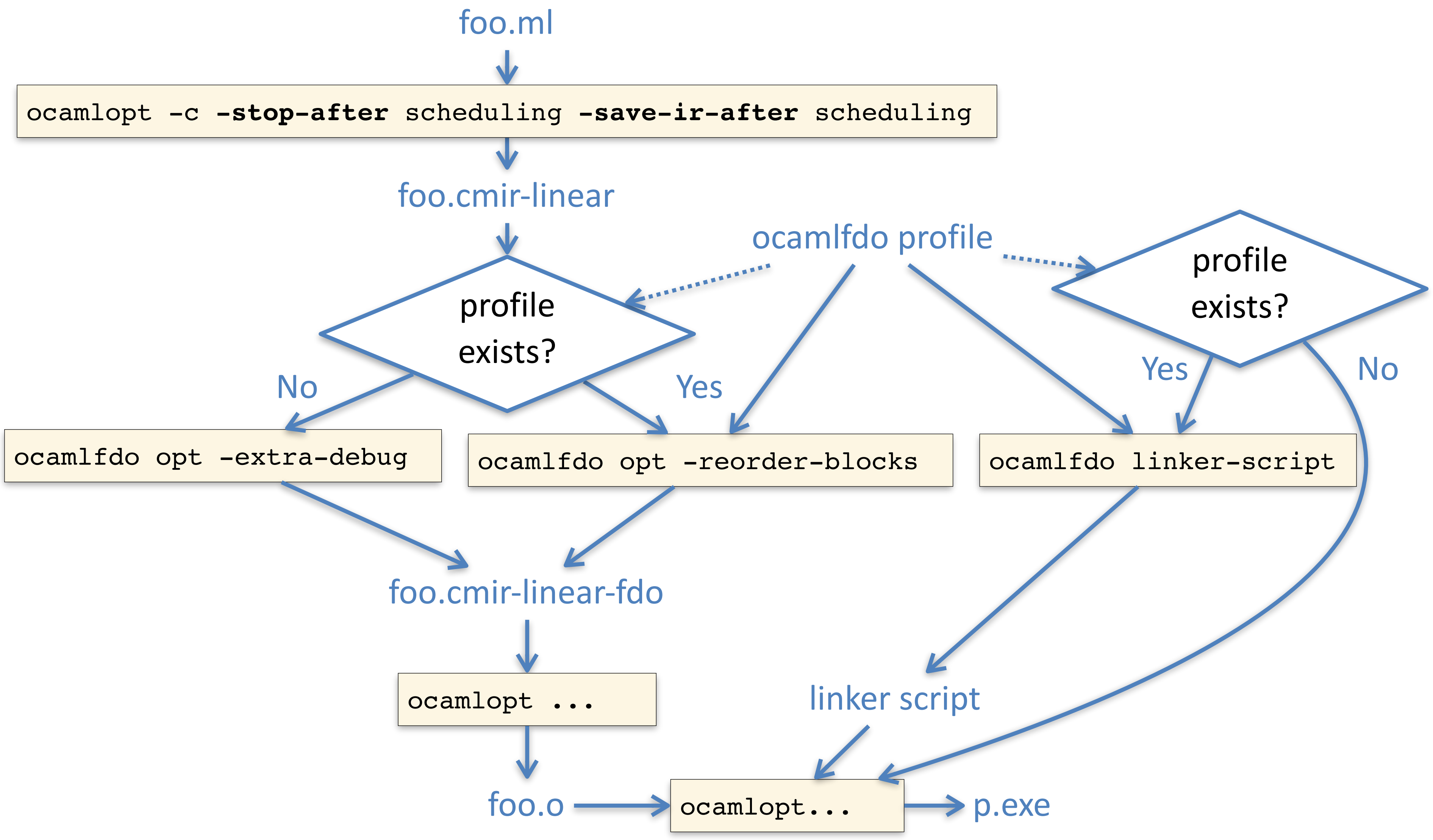
- build rules and their dependencies are statically determined
- build rule fires when the **content** of a file it depends on changes

Build system integration

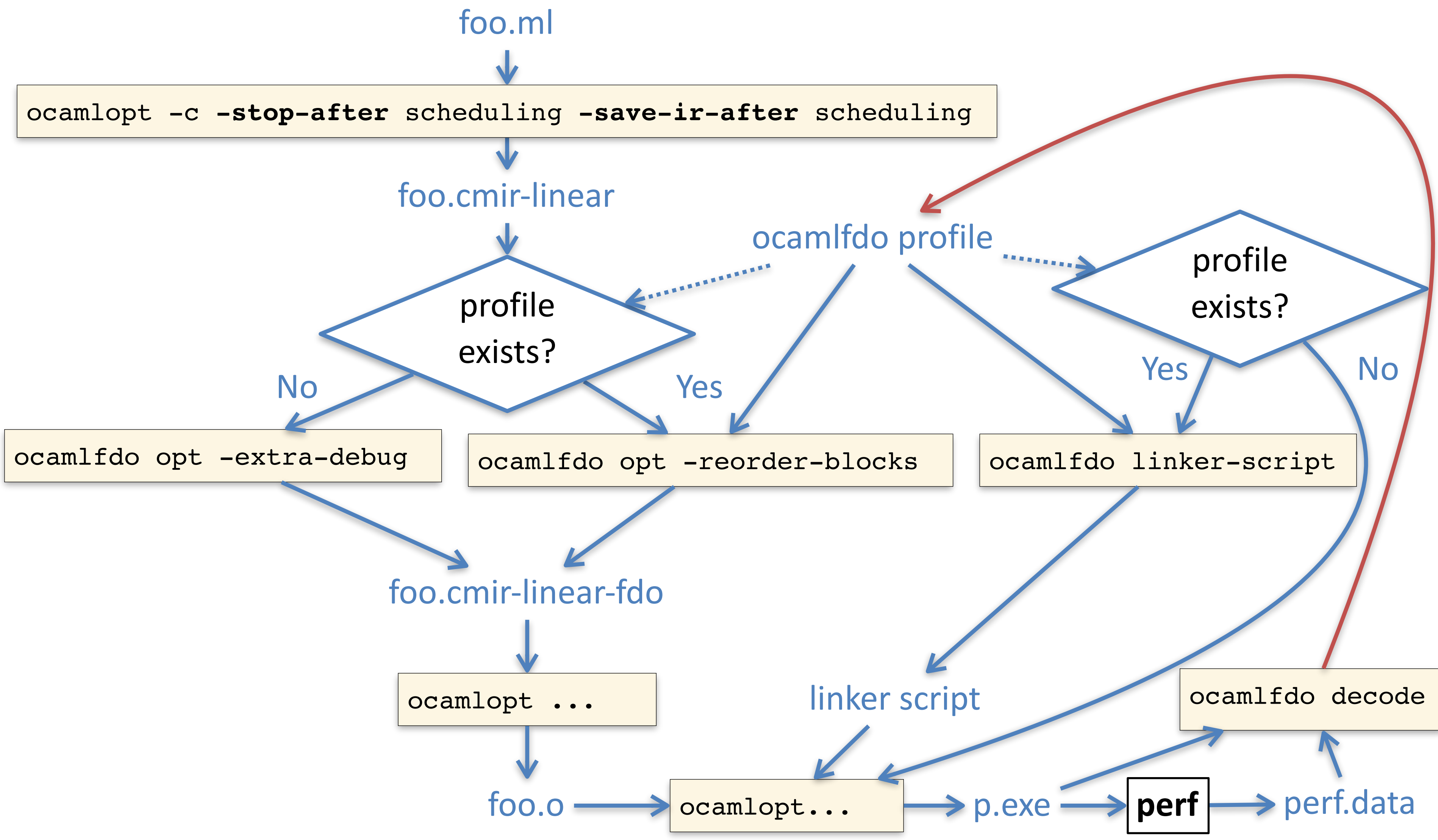

foo.ml
ocamlopt -c -stop-after scheduling -save-ir-after scheduling
foo.cmir-linear
ocamlfdo profile
profile exists?
No
Yes
ocamlfdo opt -extra-debug
ocamlfdo opt -reorder-blocks
foo.cmir-linear-fdo
ocamlopt ...
foo.o

- which build rule to use depends on the **existence** of a file

Build system integration



Further automation?



Practical considerations

- Production quality: safety and scale
- Fit into existing workflows: development and deployment
- Build systems integration: dune and jenga
- Build times and artifact sizes
- Profile storage
- Reuse profile when code changes
  - action on mismatch or missing profile
  - ppx rewriters and source code generators
  - embedded location information [%here]
  - linker-script refers to missing function sections
- Upstreaming compiler changes

Profiles: low-level vs source-level

- map dynamic data directly to static representation where it is used
- approach transferrable to other compilers
- debug info is just an implementation detail
- more accurate for code layout and other low-level and machine-specific optimizations
- faster rebuild when profile changes
- more resilient to source formatting changes
- sensitive to compiler changes
- increase size of executable
- do not increase physical memory use

Further work for OCamlFDO

- Use profile in other compiler transformations
  - inlining
  - spill code and register allocation
- Improve "debug info" encoding
  - reduce size overhead
  - co-exist with source-level debug info
  - formalize "lifting" via control flow dependencies
- Further automation of workflow
- Profile focus and normalization