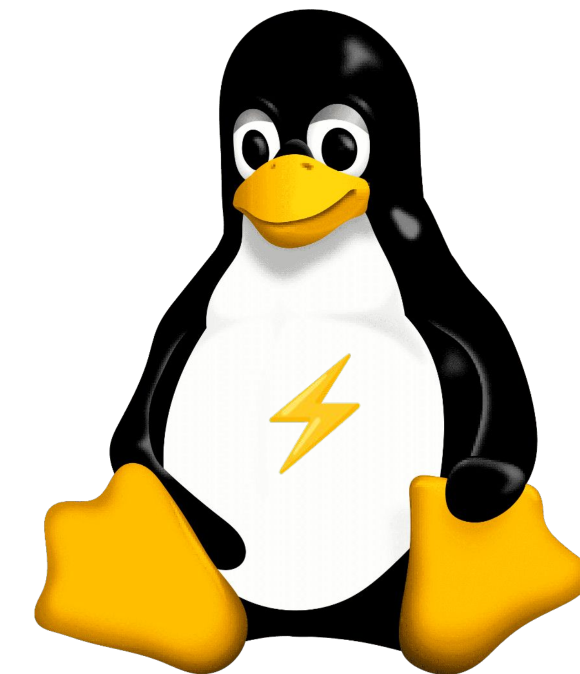
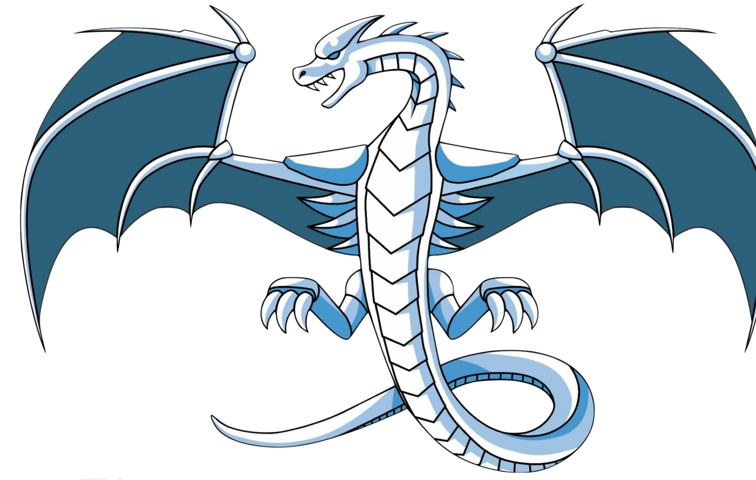
BOLT - Binary Optimizer for Linux Kernel



Maksim Panchenko
Meta, Inc

LINUX PLUMBERS CONFERENCE | Vienna, Austria
Sept. 18-20, 2024

Binary Optimization and Layout Tool



Agenda

01 Introduction to BOLT

02 Kernel Performance Numbers

03 Usage

04 Building Binary Optimizer

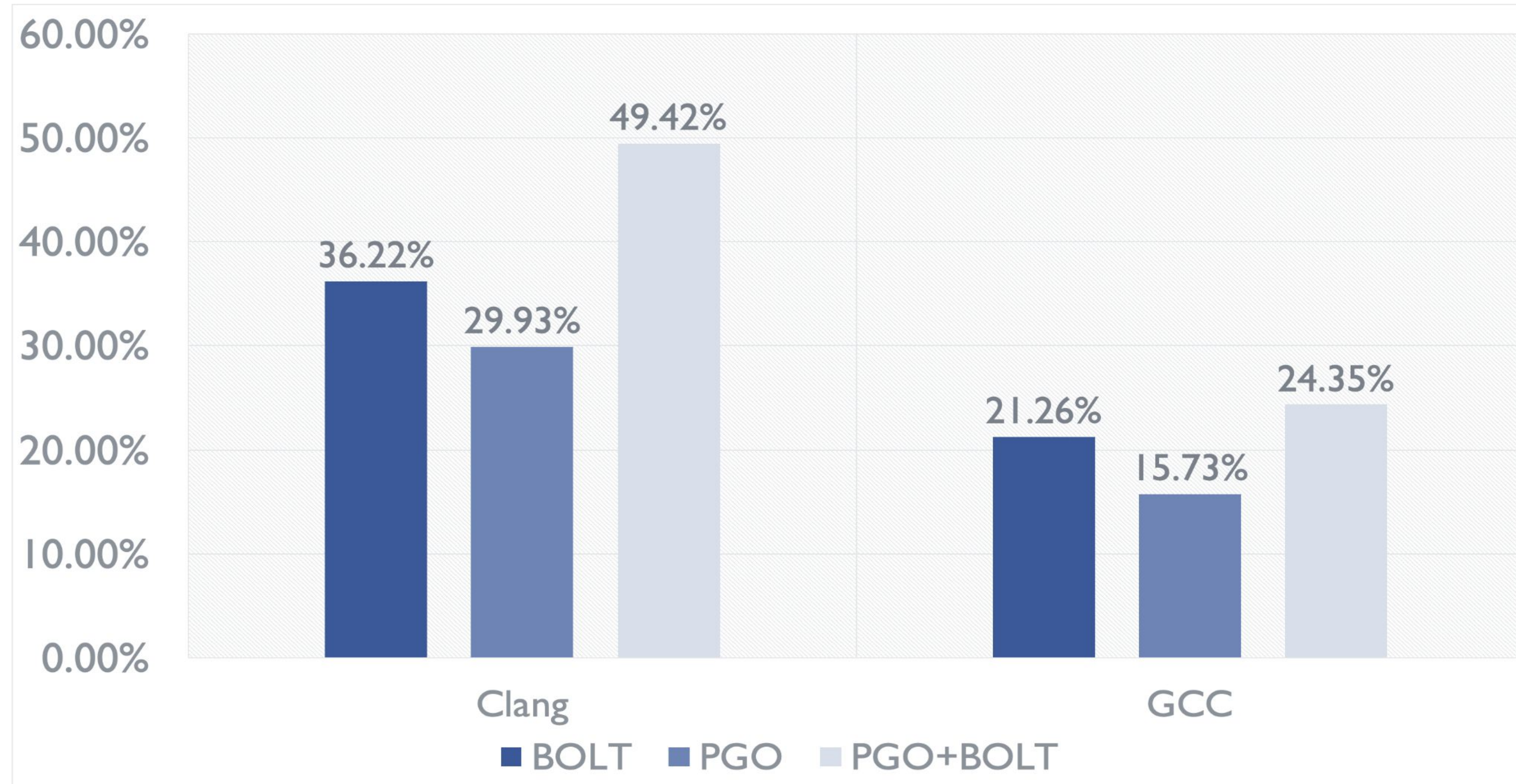
05 Linux Kernel Specifics

06 Advanced (Off)Topics

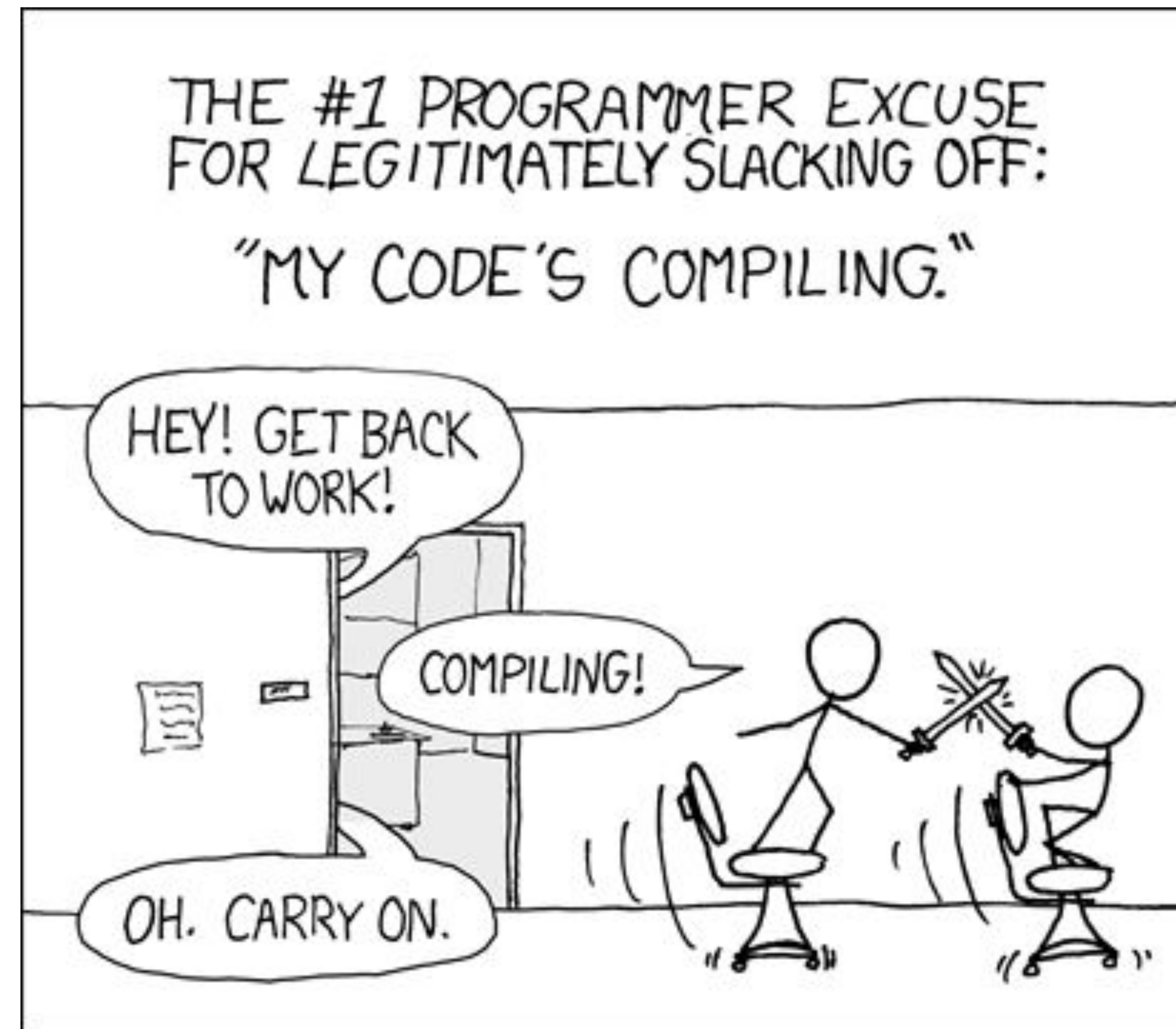
History

- 10 years ago O3+PGO+LTO was the state of the art in the open source world
- 2015: conceived profile-based BOLT to work with any compiler toolchain
- Double digit gains on top of compiler PGO+LTO
  - Speeding up both GCC and Clang
- Optimizing FB/Meta services since 2016
- Open-sourced since 2018
- In LLVM since 2022

History



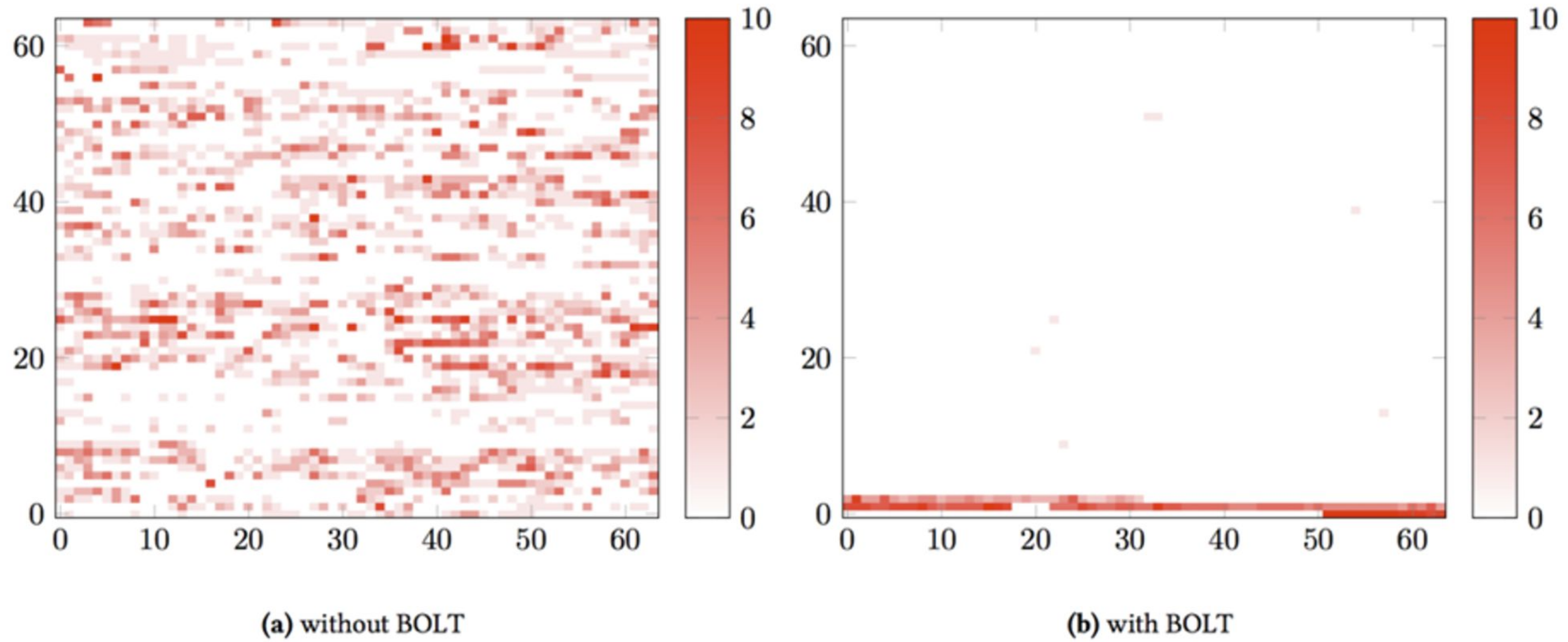
Why faster?



But How?

- Code Layout Optimizations
- L1 I$ is small 32KB/64KB
- Compiler does not have enough information
  - Inlining one of the reasons
- BOLT profiles at the lowest level
  - Knows “final” edge profiles
- Two-stage PGO became de facto for *maximum* performance
  - “Large”/datacenter applications
  - Context-sensitive PGO flavors
  - More PLO tools

But How?



(a) without BOLT

(b) with BOLT

Binary Optimization and Layout Tool

- Post-link Optimizer
- Part of LLVM project
- Complements compiler
- Support popular architectures

Binary Optimization and Layout Tool

- Post-link Optimizer
- Part of LLVM project
- Complements compiler
- Support popular architectures

Runs on ELF binary, e.g. *vmlinux*

Binary Optimization and Layout Tool

- Post-link Optimizer
- Part of LLVM project
- Complements compiler
- Support popular architectures

Runs on ELF binary, e.g. *vmlinux*

Runs on GCC-compiled code too

Binary Optimization and Layout Tool

- Post-link Optimizer
- Part of LLVM project
- Complements compiler
- Support popular architectures

Runs on ELF binary, e.g. *vmlinux*

Runs on GCC-compiled code too

PGO+LTO+BOLT for max performance

Binary Optimization and Layout Tool

- Post-link Optimizer — Runs on ELF binary, e.g. *vmlinux*
- Part of LLVM project — Runs on GCC-compiled code too
- Complements compiler — PGO+LTO+BOLT for max performance
- Support popular architectures
  - x86-64
  - AArch64
  - RISC-V

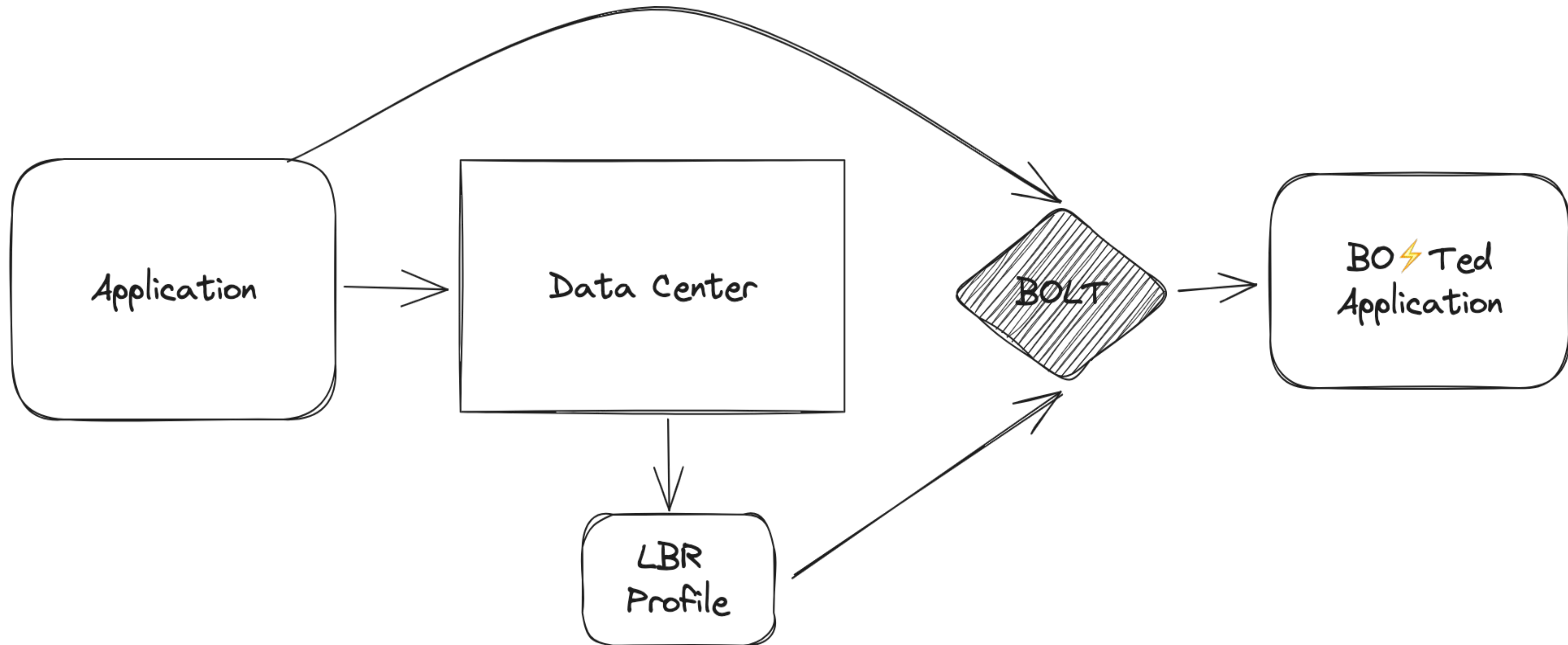
System Performance Improvements

- BOLT applied to the Linux kernel
- On top of PGO:
  - 2% QPS gain on Meta's TAO distributed data store
  - 2.5% gain on RockDB *db_bench fillseq*
  - ~30% reduction in *br_inst_retired.near_taken:k*
- Important considerations:
  - Kernel configuration
  - Time spent in the kernel
  - Micro- vs Macro- benchmark
  - System configuration and bottlenecks
  - Quality of profile

BOLT Usage

- No recompilation required
  - Only if function splitting was enabled. BOLT splits better.
- Link with *-q* *(-Wl,-q)* a.k.a. *--emit-relocs*
- Takes seconds to optimize *vmlinux*

BOLT Usage Example



Profile

- Sampling branch data
  - *perf* with Intel LBR or similar
- Instrumentation when LBR not available
- Traces (PT or ETM)
- Cycles sampling

What does it take to build Binary Optimizer for Linux kernel?

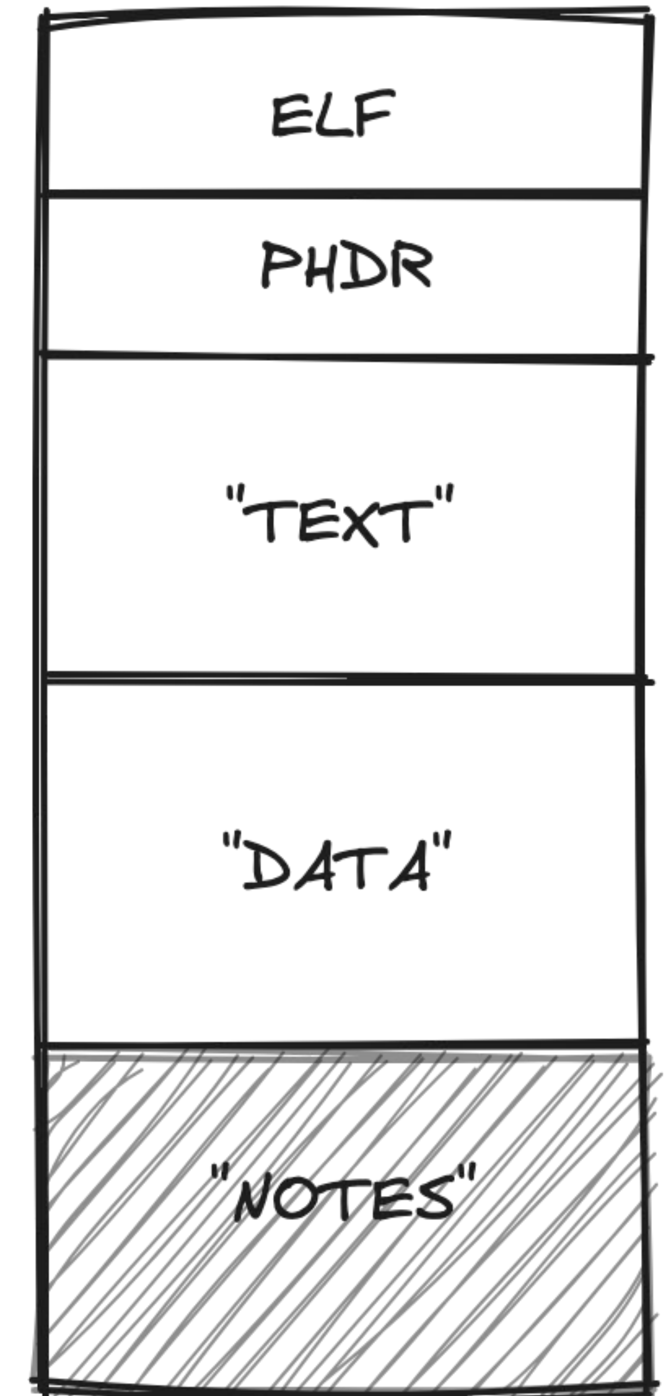
- Assumptions:
  - Well-formed unobfuscated code
  - Compiler-generated code
  - Assembly with good form
  - Similar to *objtool?*
- Unstripped x86-64 ELF

Steps:

- Identify Code and Data
- Identify Functions and Boundaries
- Disassemble Functions
- Build IR
- Attach Profile
- Run Optimizations
- Emit and Write Optimized Code
- Update Metadata

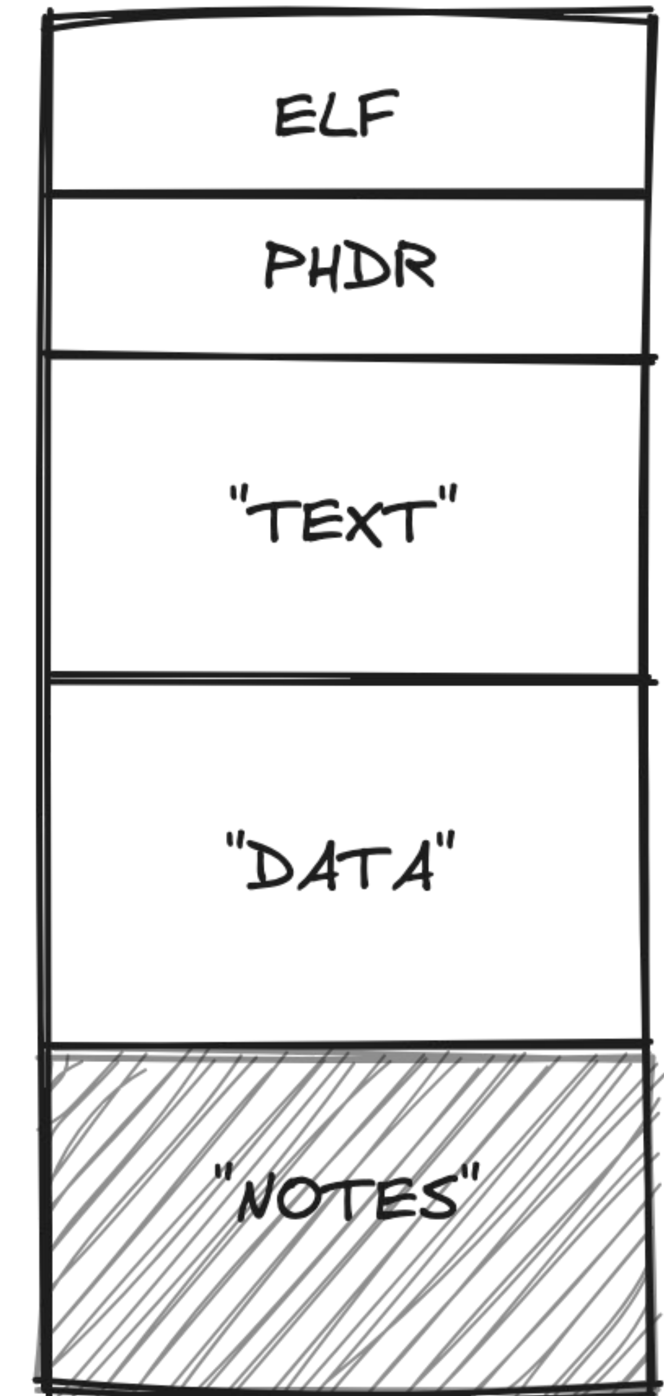
Identify Code and Data

- Segments
- Sections
- ELF flags
  - *SHF_ALLOC | SHF_EXECINSTR*
- Not a problem for unstripped binaries



Identify Functions and Boundaries

- Symbol Table
- Dynamic symbol table
- FDEs in *.eh_frame*
- Not a problem for unstripped binaries



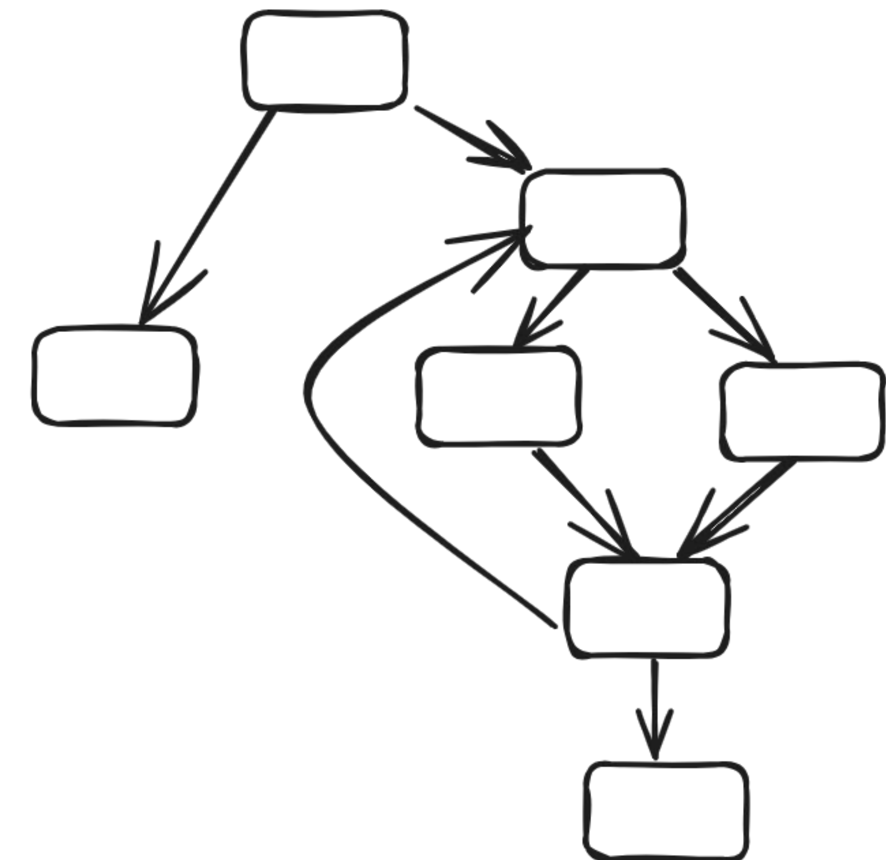
Disassemble

- Symbolic disassembler
- How to distinguish constant from an address?
- x86-64 rip-relative addressing
- Linker relocations
- Function pointers detection in code

```
movq $0x80000, 0xfaceb0 -> movq $VAR, 0xfaceb0
```

Build IR

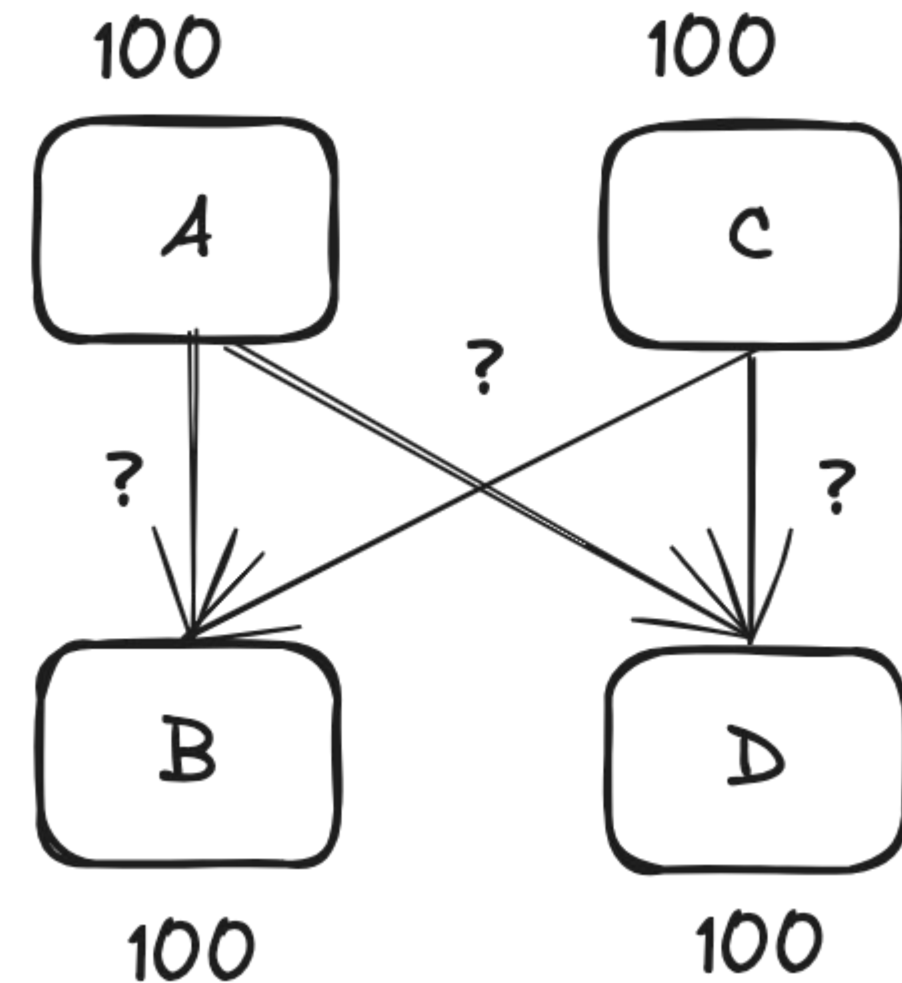
- MCPlus
- LLVM MC-level instructions with annotations
- E.g. ORC annotations per instruction
  - Indicates frame/stack modification
- With CFG (basic blocks with edges)
- Leverage LLVM for low-level target analysis
- Challenge:
  - Indirect jumps



```
pushq   %rbx # ORC: {sp: 8, bp: 0, info: 0x5}
movq    %rdi, %rbx # ORC: {sp: 16, bp: 0, info: 0x5}
```

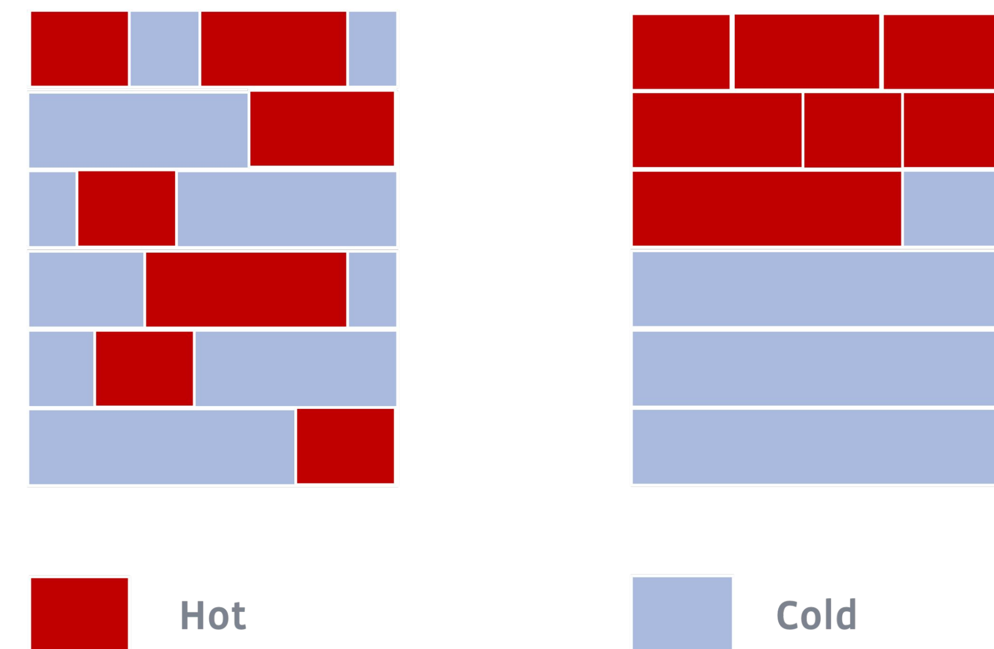
Attach Profile

- Profile on edges for optimal layout decisions
- A → B could be anywhere in [0, 100]
  - Optimal path for layout unknown
- Without edge info, some info can be recovered



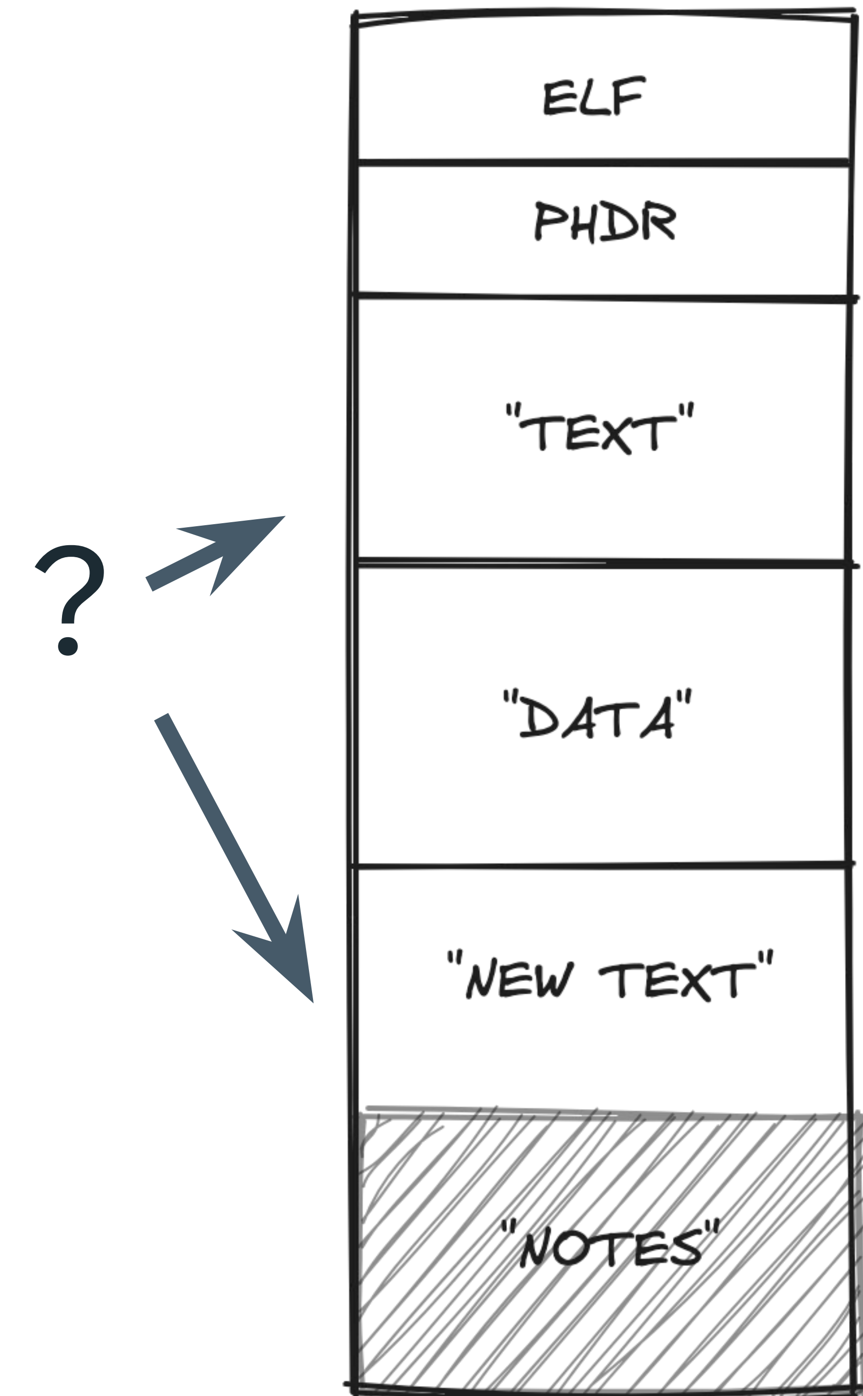
Run Optimizations

- Focus on Code Layout
- Basic-Block Reordering (ext-TSP)
- Function Splitting
  - Hot/Cold is the basic
  - CDSplit
- Fragment/Function Reordering
  - CDSort
- x86-specific branch optimization on hot path



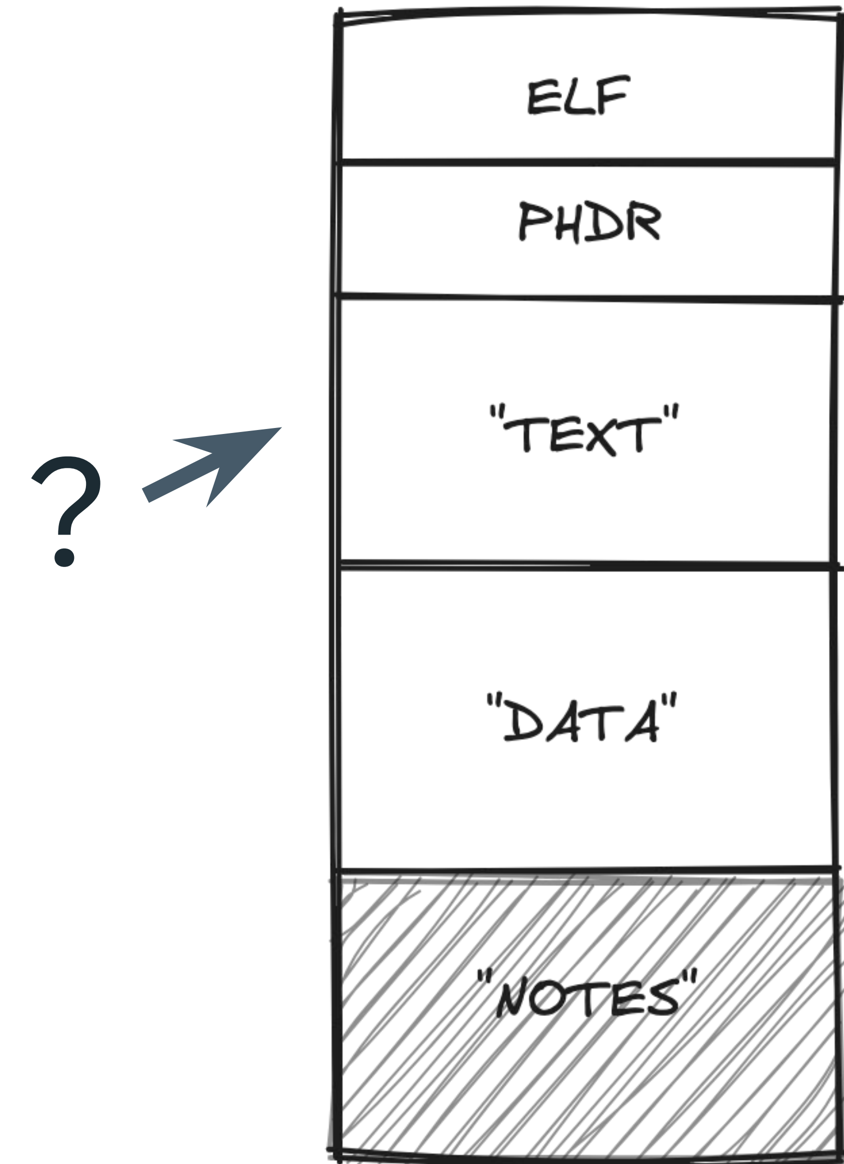
Emit and Write Optimized Code

- No need to re-write data
  - Update code references
  - Jump Tables
- Where place new code?
  - Original function bodies
  - New segment
  - Reuse *.text* but erase old code first



Emit and Write Optimized Code

- Update functions in-place
  - Safe
  - Works without relocations too
  - Piggy-backs on compiler/linker function order
  - Prevents over-specialization of code layout
  - Brings most of performance benefits
  - Con: suboptimal coverage



Update Metadata

- Critical
  - ELF and PHDR tables
  - *.eh_frame* + *.eh_frame_hdr*
  - C++ exception table
    - Existing ranges are interrupted
- Less critical (but still important)
  - Symbol table
  - DWARF
  - Compiler pseudo probes
  - etc.

Linux Kernel Metadata

- Sets kernel apart from user space
- Code modifications at boot time and runtime are common
- What You See Is *NOT* What You Get
  - w/ “*objdump -d*”
- Challenges for Binary Optimization

ORC

- Oops Rewind Capability
  - Similar to DWARF CFI but faster
- Optimized lookup tables
- Every instruction IP is virtually mapped to an entry
  - SPOffset
  - BPOffset
  - SPReg, BPReg, Type, Signal
- *.orc_unwind* + *.orc_unwind_ip* sections
  - *.orc_lookup* - populated at runtime
- Every instruction annotated with ORC entry
- Updated IPs for the new code layout
  - Stack is unchanged

```
pushq   %rbx # ORC: {sp: 8, bp: 0, info: 0x5}
movq    %rdi, %rbx # ORC: {sp: 16, bp: 0, info: 0x5}
```

SMP Locks

- *nop* or *lock* byte - decided at boot time
- Annotate instructions with “SMPLock”
- Update *.smp_locks* with new instruction addresses

```
lock # SMPLock: 1
andb     $-0x21, (%rbx)
```

Static Calls

- Call instructions are updated at runtime
- Tracked by a table with entries:
  - s32 address
  - s32 key
- Annotate instructions with “StaticCall”
- Update addresses in the table
- Table sorted at runtime in *static_call_sort_entries()*
  - Updates are in-place

```
jmp     __SCT__x86_pmu_read # TAILCALL  # StaticCall: 11 # Count: 955
```

Static Keys

- *NOP* or *JMP* toggled at runtime
- Optimized for LIKELY case of static key, true or false
  - NOP “most” of the time
- The static keys table stores:
  - *NOP/JMP* location
  - *JMP* target
  - Key info
- Lower bit of Key indicates if the key is likely true
- Newer kernels optimize for *jmp* size (!)
- *jump_label_update()* updates all code entries for a key (batch mode)
- *text_poke_bp_batch()* overrides the first byte with *int3*

Static Keys

- BOLT recognizes code locations and adjusts CFG
- Special “conditional” branch *jit* (CC “it”)

```
.LBB098 (1 instructions, align : 1)
Exec Count : 6616710
    jit      .Ltmp1597 # ID: 519 # Size: 2 # Likely: 1 # InitValue: 0
BB Successors: .Ltmp1597 (count: 0), .LFT9 (count: 6553941)
```

- BOLT optimizes for the final *JMP* size
- Can always output 5-byte *JMP* for compatibility
- Update table with new addresses

Alternative Instructions: *.alt_instructions*

- Different instructions sequences depending on CPU features
- Multiple alternatives possible
- Alternatives can have their own ORC entries
  - Same ORC table is shared putting restrictions on instruction boundaries (developers can use NOPs)
- Can include control flow instructions
- BOLT annotates disassembly with alternatives
- Hard to properly optimize unless CFG is fixed

```
xsave64 (%rdi) # AltInst: "xsaveopt64 (%rdi)" # AltInst2: "xsavec64 (%rdi)" # AltInst3: "xsaves64 (%rdi)"
```

Paravirtual Instruction Table: *.parainstructions*

- Replaced by alternative instructions in new kernels
- Tracked by a table with entries:
  - u8* instr
  - u8 type
  - u8 len
- Annotate instructions with “ParaSite”
- Skip optimization

Bug Table: *__bug_table*

- Used for kernel debugging
  - WARN()/WARN_ON() & BUG()/BUG_ON()
- *struct bug_entry*
  - Pointer to *ud2* instruction corresponding to a bug
    - Always PC-relative on x86-64
  - Source location and flags
- Update *ud2* location
- *find_bug()* uses linear search

ud2 # BugEntry: 181

Exception Table: __*ex_table*

- Instructions that access user-space memory can cause page faults
- The table references the memory instruction and the code where the execution will resume.
- The table may be expected to be sorted
- Currently BOLT skips functions with exceptions and fixups

```
xsave64 (%rdi) # ExceptionEntry: 459 # Fixup: __ENTRY_save_fpregs_to_fpstate@0xffffffff810390a6
```

PCI Fixup Table: *.pci_fixup*

- Lists code handlers for errors associated with a given PCI
- BOLT verifies that handlers are at the start of a function
- No update needed

Kernel Metadata Summary

- All metadata can be dumped by BOLT
- BOLT Disassembly is annotated with Kernel metadata
- Few changes between 5.19 and 6.8
  - E.g. *struct alt_instr*
  - BOLT automatically detects the correct data structure form
- Undetected Metadata?
  - Relocations pointing at function with displacement
  - Reject optimization

Not Covered in this Talk

- Profiling BOLTed Binaries
- Continuous Profiling
  - New source – new binary
  - Binary Profile Inference
- Re-optimizing
  - Save original BOLT input
- Lightning BOLT
  - Parallel optimizations
- Other BOLT optimizations
- Security Applications

BOLT Run

```
BOLT-INFO: Target architecture: x86_64
BOLT-INFO: BOLT version: 149ea1e0524505349bbe195fb29cc0e3679b6401
BOLT-INFO: Linux kernel binary detected
BOLT-INFO: first alloc address is 0x0
BOLT-INFO: static input executable detected
BOLT-INFO: enabling lite mode
BOLT-INFO: pre-processing profile using YAML profile reader
BOLT-WARNING: function copy_user_enhanced_fast_string has an object detected in a padding region at address 0xffffffff818716d0
BOLT-WARNING: function __put_user_8 has an object detected in a padding region at address 0xffffffff81874b33
BOLT-INFO: parsed 13312 SMP lock entries
BOLT-INFO: parsed 470 static call entries
BOLT-INFO: parsed 1305 exception table entries
BOLT-INFO: parsed 0 paravirtual patch sites
BOLT-INFO: parsed 10684 bug table entries
BOLT-INFO: setting --alt-inst-has-padlen=0
BOLT-INFO: setting --alt-inst-feature-size=2
BOLT-INFO: parsed 1200 alternative instruction entries
BOLT-INFO: parsed 455063 ORC entries
BOLT-INFO: parsed 865 PCI fixup entries
BOLT-INFO: parsed 10716 static keys jump entries
BOLT-INFO: 5657 out of 55728 functions in the binary (10.2%) have non-empty execution profile
BOLT-WARNING: internal call detected in function __switch_to_asm
BOLT-WARNING: internal call detected in function __switch_to_asm
```

BOLT Run

```
BOLT-INFO: basic block reordering modified layout of 3459 functions (61.15% of profiled, 6.21% of total)
BOLT-INFO: 33 Functions were reordered by LoopInversionPass
BOLT-INFO: program-wide dynostats after all optimizations before SCTC and FOP:
...

           246560887 : executed forward branches (+5.3%)
            15740399 : taken forward branches (-79.3%)
            29878037 : executed backward branches (-29.5%)
            10845129 : taken backward branches (-53.8%)
...
           281608729 : total branches (-3.9%)
            31755333 : taken branches (-72.7%)
           249853396 : non-taken conditional branches (+41.2%)
            26585528 : taken conditional branches (-73.3%)
           276438924 : all conditional branches (-0.0%)

BOLT-INFO: the input contains 717 short and 694 long static keys jumps in optimized functions
BOLT-INFO: written 722 short and 689 long static keys jumps in optimized functions
BOLT-INFO: patched build-id (flipped last bit)
BOLT-WARNING: Linux kernel support is experimental
BOLT: 4885 out of 55728 functions were overwritten.
BOLT-INFO: rewritten functions cover 74.72% of the execution count of simple functions of this binary
```

BOLT Disassembly

__traceiter_sched_process_free()

```
.Ltmp4523 (7 instructions, align : 1)
  Exec Count : 1525
  Predecessors: .Ltmp4523, .LFT3292
    00000017:   movq    (%rbx), %rax # ORC: {sp: 24, bp: 0, info: 0x5}
    0000001a:   movq    0x8(%rbx), %rdi # ORC: {sp: 24, bp: 0, info: 0x5}
    0000001e:   movq    %r14, %rsi # ORC: {sp: 24, bp: 0, info: 0x5}
    00000021:   callq   *%rax # CallProfile: 1768 (942 misses) :
        { __bpf_trace_sched_process_template/1: 1768 (942 misses) } # ORC: {sp: 24, bp: 0, info: 0x5}
    00000023:   cmpq    $0x0, 0x18(%rbx) # ORC: {sp: 24, bp: 0, info: 0x5}
    00000028:   leaq    0x18(%rbx), %rbx # ORC: {sp: 24, bp: 0, info: 0x5}
    0000002c:   jne     .Ltmp4523 # ORC: {sp: 24, bp: 0, info: 0x5}
  Successors: .Ltmp4523 (mispreds: 450, count: 629), .Ltmp4522 (mispreds: 0, count: 1307)
```

BOLT Disassembly

enqueue_task()

```
.Ltmp4589 (1 instructions, align : 1)
  Exec Count : 23112
  Predecessors: .LFT2737, .Ltmp4590
    0000004a:   jit      .Ltmp4592 # ID: 1275 # Size: 2 # Likely: 1 # InitValue: 0 # ORC: {sp: 32, bp: 0, info: 0x5}
  Successors: .Ltmp4592 (mispreds: 3, count: 23112), .Ltmp4588 (mispreds: 0, count: 0)

.Ltmp4588 (9 instructions, align : 1)
  Exec Count : 27838
  Predecessors: .Ltmp4587, .Ltmp4595, .Ltmp4589
    0000004c:   movq    0x378(%r14), %rax # ORC: {sp: 32, bp: 0, info: 0x5}
    00000053:   movq    (%rax), %rax # ORC: {sp: 32, bp: 0, info: 0x5}
    00000056:   movq    %r15, %rdi # ORC: {sp: 32, bp: 0, info: 0x5}
    00000059:   movq    %r14, %rsi # ORC: {sp: 32, bp: 0, info: 0x5}
    0000005c:   movl    %ebx, %edx # ORC: {sp: 32, bp: 0, info: 0x5}
    0000005e:   popq    %rbx # ORC: {sp: 32, bp: 0, info: 0x5}
    0000005f:   popq    %r14 # ORC: {sp: 24, bp: 0, info: 0x5}
    00000061:   popq    %r15 # ORC: {sp: 16, bp: 0, info: 0x5}
    00000063:   jmpl    *%rax # TAILCALL  # ORC: {sp: 8, bp: 0, info: 0x5} # CallProfile: 27540 (12402 misses) :
        { enqueue_task_fair/1: 27458 (12320 misses) },
        { enqueue_task_rt/1: 1 (1 misses) },
        { enqueue_task_stop/1: 81 (81 misses) }
```

BOLT Disassembly

read_tsc()

```
.LBB0101 (7 instructions, align : 1)
  Entry Point
    00000000:   rdtsc # AltInst: "lfence; rdtsc" # AltInst2: "rdtscp" # ORC: {sp: 8, bp: 0, info: 0x5}
    00000002:   nop # Size: 1 # AltInst: "lfence; rdtsc" # AltInst2: "rdtscp" # NOP: 1 # ORC: {sp: 8, bp: 0, info: 0x5}
    00000003:   nop # Size: 1 # AltInst: "lfence; rdtsc" # AltInst2: "rdtscp" # NOP: 1 # ORC: {sp: 8, bp: 0, info: 0x5}
    00000004:   nop # Size: 1 # AltInst: "lfence; rdtsc" # AltInst2: "rdtscp" # NOP: 1 # ORC: {sp: 8, bp: 0, info: 0x5}
    00000005:   shlq    $0x20, %rdx # ORC: {sp: 8, bp: 0, info: 0x5}
    00000009:   orq     %rdx, %rax # ORC: {sp: 8, bp: 0, info: 0x5}
    0000000c:   retq # ORC: {sp: 8, bp: 0, info: 0x5}
```

Thanks!

github.com/llvm/llvm-project/bolt

Meta