Optimizing the Linux Kernel with LLVM BOLT

Maksim Panchenko
Agenda

- Why Optimize the Kernel?
- Challenges Applying BOLT
- Progress & Plans
Why Optimize the Linux Kernel?
Why Optimize the Linux Kernel?
Why Optimize the Linux Kernel?

- Often optimized for size
- Heavily hand-tuned for performance
- Plenty of assembly code
- Slow PGO adoption
- Even slower LTO adoption
BOLT Overview

What?
Binary Optimization & Layout Tool

Why?
Accurate Binary Profile

How?
Break Binary and Assemble a Better One

01 Why Optimize the Kernel?
Opportunities for BOLT

- Focusing on X86-64 for now
- 16 MiB `.text`
- High I$ miss ratio
- Exposure to Assembly
- Indirect Call Promotion
  - To offset retpoline penalties
- Specialized Kernel?

<table>
<thead>
<tr>
<th>Instruction Count</th>
<th>Description</th>
<th>Percent Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>24907705</td>
<td>executed forward branches</td>
<td>+4.4%</td>
</tr>
<tr>
<td>1830988</td>
<td>taken forward branches</td>
<td>-77.8%</td>
</tr>
<tr>
<td>6954275</td>
<td>executed backward branches</td>
<td>-13.2%</td>
</tr>
<tr>
<td>3594639</td>
<td>taken backward branches</td>
<td>-21.0%</td>
</tr>
<tr>
<td>1049173</td>
<td>executed unconditional branches</td>
<td>-46.6%</td>
</tr>
<tr>
<td>15478597</td>
<td>all function calls</td>
<td>-5.8%</td>
</tr>
<tr>
<td>0</td>
<td>indirect calls (=)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>PLT calls (=)</td>
<td></td>
</tr>
<tr>
<td>275956063</td>
<td>executed instructions</td>
<td>-0.7%</td>
</tr>
<tr>
<td>74095406</td>
<td>executed load instructions</td>
<td>(=)</td>
</tr>
<tr>
<td>32940929</td>
<td>executed store instructions</td>
<td>(=)</td>
</tr>
<tr>
<td>0</td>
<td>taken jump table branches</td>
<td>(=)</td>
</tr>
<tr>
<td>0</td>
<td>taken unknown indirect branches</td>
<td>(=)</td>
</tr>
<tr>
<td>32911153</td>
<td>total branches</td>
<td>-2.7%</td>
</tr>
<tr>
<td>6474800</td>
<td>taken branches</td>
<td>-56.1%</td>
</tr>
<tr>
<td>26436353</td>
<td>non-taken conditional branches</td>
<td>+38.5%</td>
</tr>
<tr>
<td>5425627</td>
<td>taken conditional branches</td>
<td>-57.5%</td>
</tr>
<tr>
<td>31861880</td>
<td>all conditional branches</td>
<td>-0.0%</td>
</tr>
</tbody>
</table>
Challenges Applying BOLT

- Code Volatility
- Updating ELF
- Testing
- Debugging
Dealing with Code Volatility

• What You See Is *NOT* What You Get
• Simple disassembly does not reflect the state of the code while running the kernel
• Code is patched at boot time
  - SMP (lock prefix) vs single core (NOP)
  - Arch-specific instruction sequences
• Code is patched at run time
  - Ftrace
  - Static Keys
  - Static Calls
Static Keys

.Ltmp1319 (7 instructions, align : 1)
Predecessors: .Ltmp1317
00000123: movq %rbp, %rdx
00000126: movl $0xc0000100, %ecx
0000012b: movl %ebp, %eax
0000012d: shrq $0x20, %rdx
00000131: wrmsr
00000133: nop # Size: 5
00000138: jmp .Ltmp1320
Successors: .Ltmp1320 (mispreds: 0, count: 2)
Static Keys

.Ltmp1319 (7 instructions, align : 1)
  Predecessors: .Ltmp1317
      movq  %rbp, %rdx
      movl  $0xc0000100, %ecx
      movl  %ebp, %eax
      shrq  $0x20, %rdx
      wrmsr
      nop  # Size: 5
      jmp   .Ltmp1320
  Successors: .Ltmp1320 (mispreds: 0, count: 2)
Static Keys

.Ltmp1319 (7 instructions, align : 1)
Predecessors: .Ltmp1317

00000123:  movq   %rbp, %rdx
00000126:  movl   $0xc0000100, %ecx
0000012b:  movl   %ebp, %eax
0000012d:  shrq  $0x20, %rdx
00000131:  wrmsr
00000133:  jmp   .Ltmp46490  # STATICKEY
00000138:  jmp   .Ltmp1320

Successors: ?
Static Keys

- Eliminates global variable condition check
- More than removing a branch
- Assembler macro: \texttt{STATIC\_JUMP\_IF\_\{TRUE\|FALSE\} target, key}
- Metadata placed in \texttt{.rodata}
- \texttt{__jump_table} "section" is marked by [\texttt{__start\_jump\_table}, \texttt{__stop\_jump\_table}]
- Entry Contents:
  - PCREL32 JumpAddress
  - PCREL32 TargetAddress
  - PCREL64 KeyAddress

```cpp
/* Documentation/staging/static-keys.rst */
if (static_branch_unlikely(&key))
  printk("I am the true branch\n");
```
Static Keys Support

.Ltmp1319 (7 instructions, align : 1)
  Predecessors: .Ltmp1317
  00000123: movq  %rbp, %rdx
  00000126: movl  $0xc0000100, %ecx
  0000012b: movl  %ebp, %eax
  0000012d: shrq  $0x20, %rdx
  00000131: wrmsr
  00000133: nop OR jmp  .Ltmp46490
  00000138: jmp  .Ltmp1320

Successors: ?

Challenges Applying BOLT
Static Keys Support

.Ltmp1319 (7 instructions, align : 1)
Predecessors: .Ltmp1317

00000123:   movq   %rbp, %rdx
00000126:   movl   $0xc0000100, %ecx
0000012b:   movl   %ebp, %eax
0000012d:   shrq   $0x20, %rdx
00000131:   wrmsr
00000133:   jCC   .Ltmp46490
00000138:   jmp   .Ltmp1320

Successors: .Ltmp46490 (mispreds: 0, count: 42), .Ltmp1320 (mispr: 0, count: 2)
Static Keys Support

.Ltmp1319 (7 instructions, align : 1)
  Predecessors: .Ltmp1317
  00000123:    movq  %rbp, %rdx
  00000126:    movl  $0xc00000100, %ecx
  0000012b:    movl  %ebp, %eax
  0000012d:    shrq  $0x20, %rdx
  00000131:    wrmsr
  00000133:    jit  .Ltmp46490  # STATICKEY: 0  # Size: 5
  00000138:    jmp  .Ltmp1320

Successors: .Ltmp46490 (mispreds: 0, count: 42), .Ltmp1320 (mispr: 0, count: 2)
Static Keys Support Implementation

- Disassemble Functions
- Read Static Keys "Jump Table"
- Convert JMP/NOP into JCC with a custom CC (it)
- Build CFG
  ...
- Convert JIT into JMP/NOP
  - Preserve the original 5-byte size
- Emit function code
- Update "Jump Table" with new Jump and Target addresses
- Update Key Address low bit if the condition was reversed
Static Keys Opportunities

- ~20% of hot functions have static key jumps
- Always 5-byte NOP/JMP
  - 2-byte might work in many cases
  - BOLT can detect short vs long JMP codegen
- Invalid static prediction
  - Reverse the key condition

queued_spin_lock_slowpath:
.LBB0815 (2 instructions, align : 1)
Exec Count : 13836
  00000000: callq __fentry__
  00000005: jit .Ltmp46692 # STATICKEY: 0 # Size: 5
Successors: .Ltmp46692 (miss: 1, count: 13836), .LFT3516 (m: 0, count: 0)
Static Calls

- Way to overcome overheads of indirect calls
  - Critical for kernels hardened with retpolines
- CALL to `__SCT_tp_func_*` in disassembly
  - Replaced with new target or NOP
- LBR confusion
  - Might affect function ordering

```c
/*
 * The static call site table needs to be created by external tooling (objtool
 * or a compiler plugin).
 */
struct static_call_site {
  s32 addr;
  s32 key;
};
```
Alternative Instructions

- **ALTERNATIVE oldinstr, newinstr, feature**
- Padding required
- NOPs are optimized by `optimize_nops()`
- Jumps and calls are recognized and fixed
  - By manually checking the first byte/opcode

```assembly
# Emit oldinstr to current section
L1:
  oldinstr
  < pad with nops if newinstr is larger>

# Emit newinstr to .altinstr_replacement
L2:
  newinstr

# Emit description to .altinstructions
.long L1 - .
.long L2 - .
.word \feature
.byte \old_len # with padding
.byte \new_len
.byte \pad_len
```
Alternative Instructions

- `ALTERNATIVE_2 oldinstr, newinstr1, feature1, newinstr2, feature2`
  - Generates 2 entries for the same `oldinstr`
  - E.g. one instructions for Intel another for AMD
Alternative Instructions Support

- Optimize conservatively
  - Preserve padding in main code
  - Update alternative instruction targets in `.altinstr_replacement`
- Ignore LBR discrepancies in “alternative regions”
- Advanced: hint BOLT what features are enabled to optimize alternative sequences
- Not as common as e.g. static keys
More Code Variants

- `.parainstructions`
- `__fentry__`
  - LBR profile discrepancy
- `.smp_locks`
  - Locations of SMP lock prefixes
- New/Undiscovered Sections
  - `vmlinux` linked with relocations
  - Ignore functions with unknown references in the middle of code
More Sections to Update

- ORC
  - .orc_unwind_ip, .orc_unwind, .orc_lookup
- __ex_table
- __bug_table
- .pci_fixup
- __ksymtab{__gpl}
Updating ELF

- Incremental Rewriting
- No HFSort / Function Reordering
- In-place mode with relocations
  - Or work w/o relocations with caveats
- No new PHDR / Segment
- No function splitting
Debugging

- Easier to debug in a VM
  - Turnaround time few seconds vs several minutes on HW
  - Some issues only come up on HW
    - Usage of modules/drivers
- Kernel panic
  - Normally easier to debug due to included stack trace
- Kernel stuck at boot time
- Bisecting in BOLT
  - Limit optimizations to a set of functions
  - Works well with VM turnaround time
Kernel Panic

```plaintext
[ 21.406279] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
[ 21.406292] random: fast init done
[ 21.464829] Call Trace:
[ 21.469720] dump_stack+0x4/0x7c
[ 21.476339] panic+0xfb/0x2cb
[ 21.482267] mount_block_root+0x2aa/0x332
[ 21.490276] ? rdinit_setup+0x2c/0x2c
[ 21.497589] prepare_namespace+0x135/0x164
[ 21.505770] kernel_init_freedable+0x21f/0x22c
[ 21.514472] ? rest_init+0xb4/0xb4
[ 21.521268] kernel_init+0xa/0x10c
[ 21.528062] ret_from_fork+0x22/0x30
[ 21.600307] ---[ end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0) ]---
```
Stuck Kernel

- Broken `vmlinux` had the exact loadable contents as "good" `vmlinux`.
- `.vvar` section header was truncated, which was followed by `.data..percpu` section that starts at address 0 in the middle of data segment.
Progress

• Milestone 1: booting VM
• Milestone 2: booting 5.12 on HW
  • *vmlinux/bzImage* is not enough
  • *initramfs* contains kernel modules necessary to mount the real rootfs
  • Kernel modules should be compatible with *vmlinux*
  • (Re-)build the kernel package
  • Regular and “hardened” flavors
• Current (May ‘23): start prod testing & performance measurement
Plans

- Find a good open-source benchmark/suite
  - Stress L1/L2
  - Microbenchmark might not work
  - Server-type or DB workloads seem to fit the bill
    - Rocksdb/LevelDB/Apache/Nginx/MySQL/PostgreSQL (*)
  - Achieve decent SNR
- Gradually increase code coverage
- Full binary rewrite with 100% coverage

(*) One Profile Fits All: Profile-Guided Linux Kernel Optimizations for Data Center Applications, M. Ugur, et al., ACM SIGOPS Operating System Review, 2022