Adding a BOLT pass

MAKSIM PANCHENKO
AMIR AYUPOV

2023 EuroLLVM Developer’s Meeting

Meta

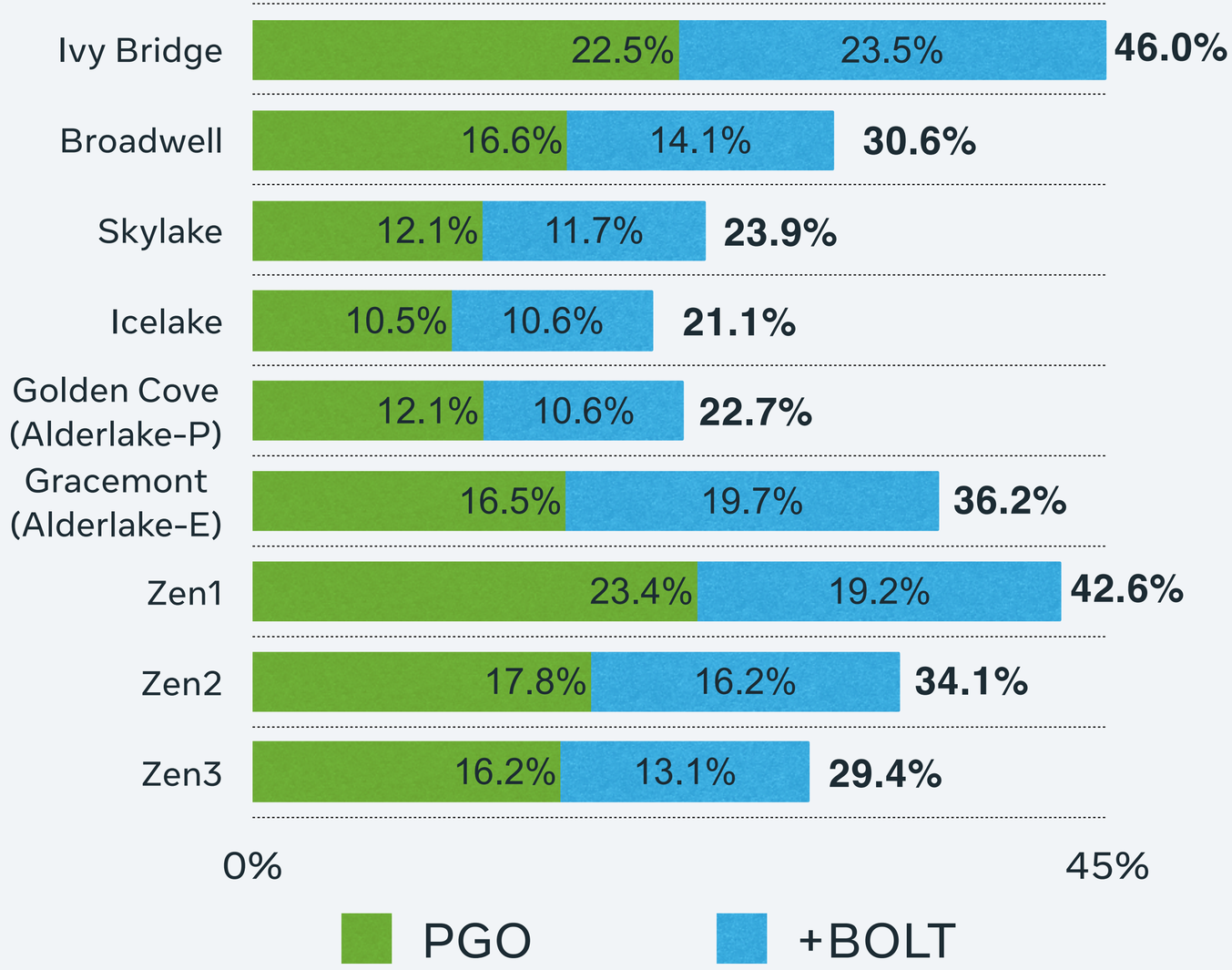
Agenda

1. Intro
2. BOLT pipeline and IR
3. Simple peephole rule
    - Triaging crashes
4. Adding a BinaryPass
    - CFG visualization

What’s BOLT?

- Binary Optimization and Layout Tool
- Supports X86 and ARM ELF programs and shared libraries
- Part of LLVM monorepo

Cumulative speedup over bootstrapped build, Building Clang



What does BOLT do?

- **Profile-guided optimizations** at **whole program level**
  - Code layout optimizations
  - Sampling (LBR) or instrumentation profile driven
  - Supporting third-party libraries with no source, and hand-written assembly
- Other uses
  - Whole program transformations or instrumentation (e.g. spectre mitigation)
  - Reverse engineering
  - Profile analysis

Why would you want to add a BOLT pass?

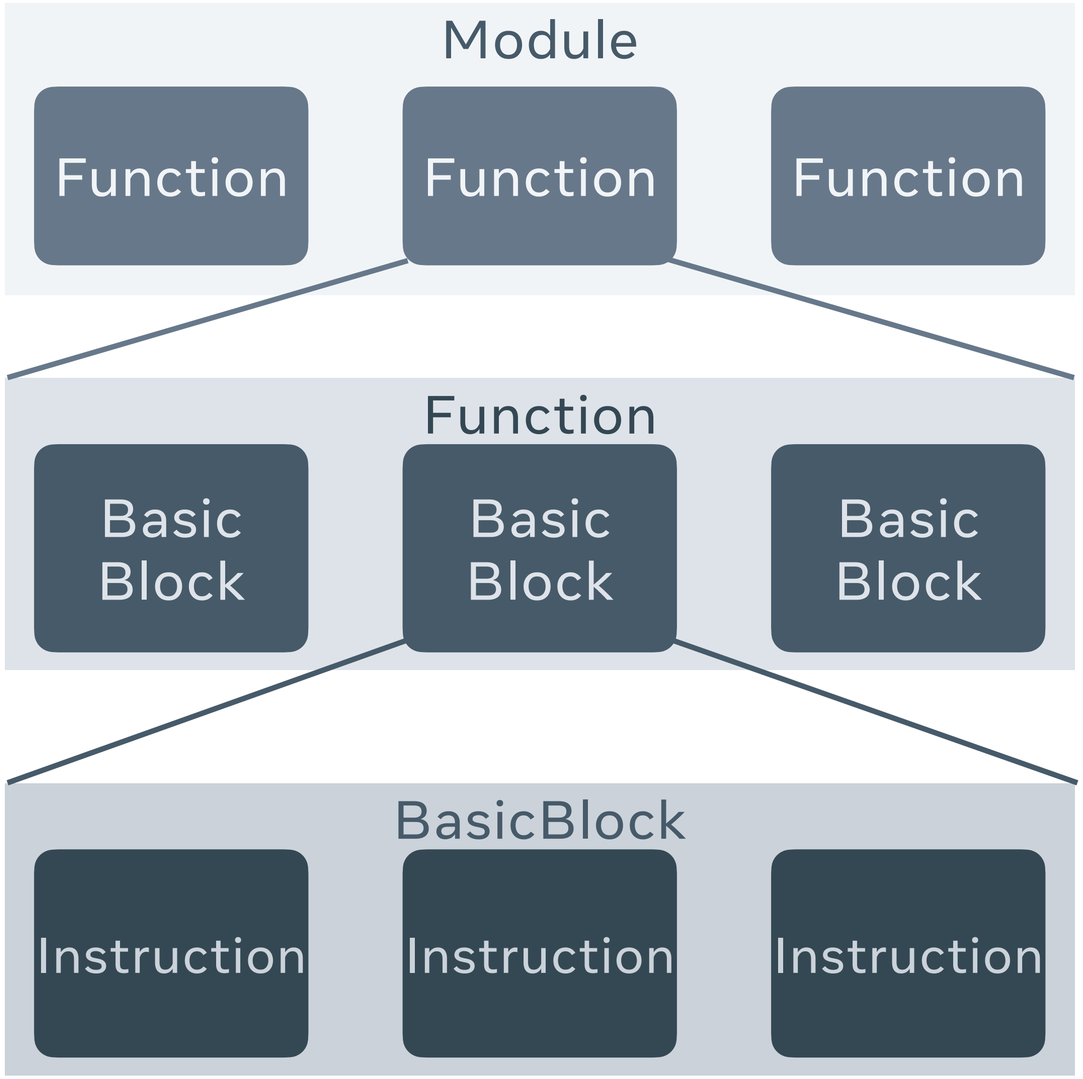
- Exploring **optimization opportunities** leveraging **accurate profiling information**:
  - HW mechanisms: alignment for macro-fusion (Intel/AMD), atomic execution (Intel TSX/ARM TME)
  - OS/HW feedbacks: memory profile (Linux perf), branch mispredictions and latency information (Intel LBR/ARM BRBE)
- Looking for binary patterns leveraging **rich analysis framework**:
  - Metadata: CFI/EH information, DWARF parsing and updates
  - Functions and instructions: call graph, register/stack slot liveness

02 BOLT pipeline and IR

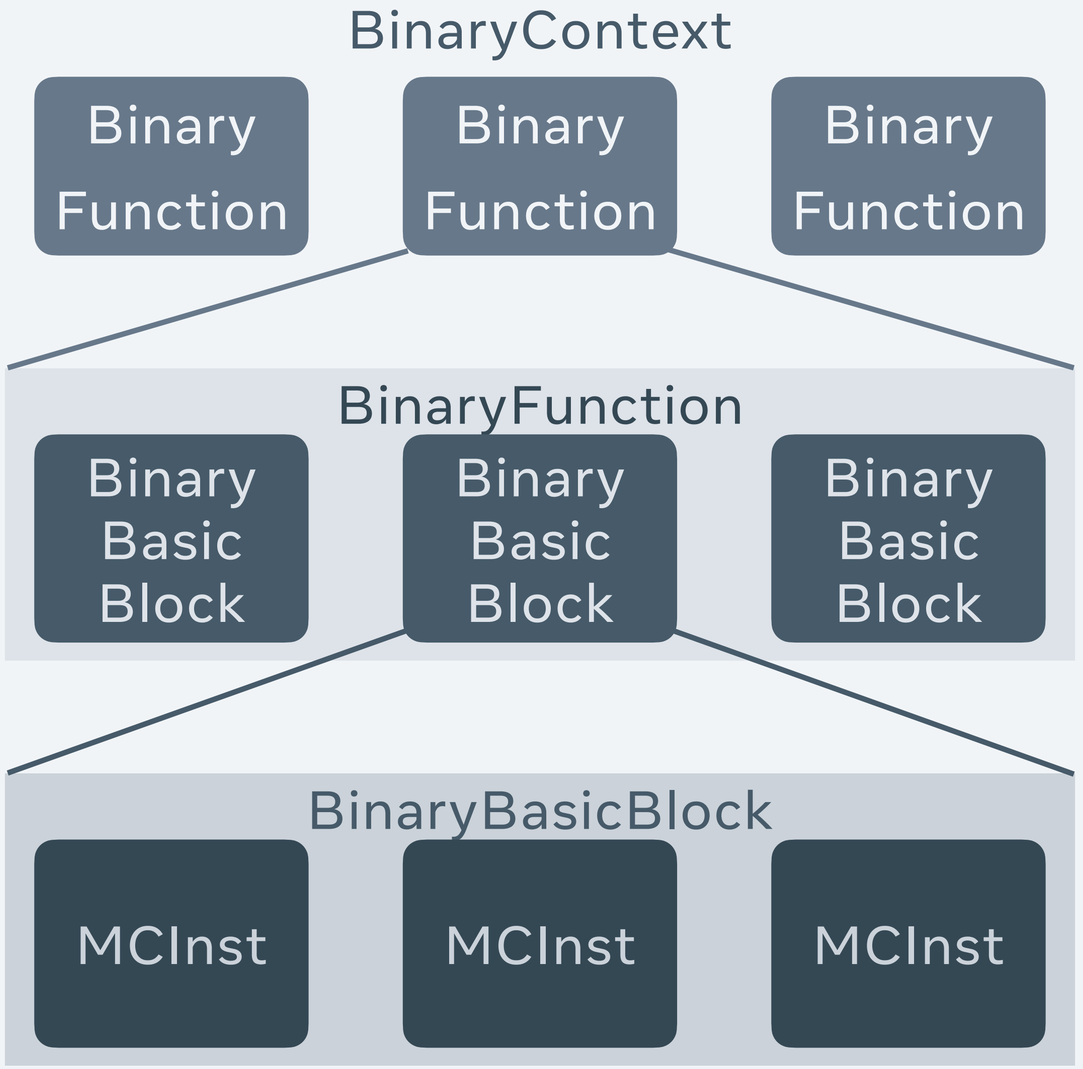
BOLT processing pipeline

- Discover
- Disassemble
- Build CFG
- Read Profile
- **Optimize**
- Emit
- Rewrite

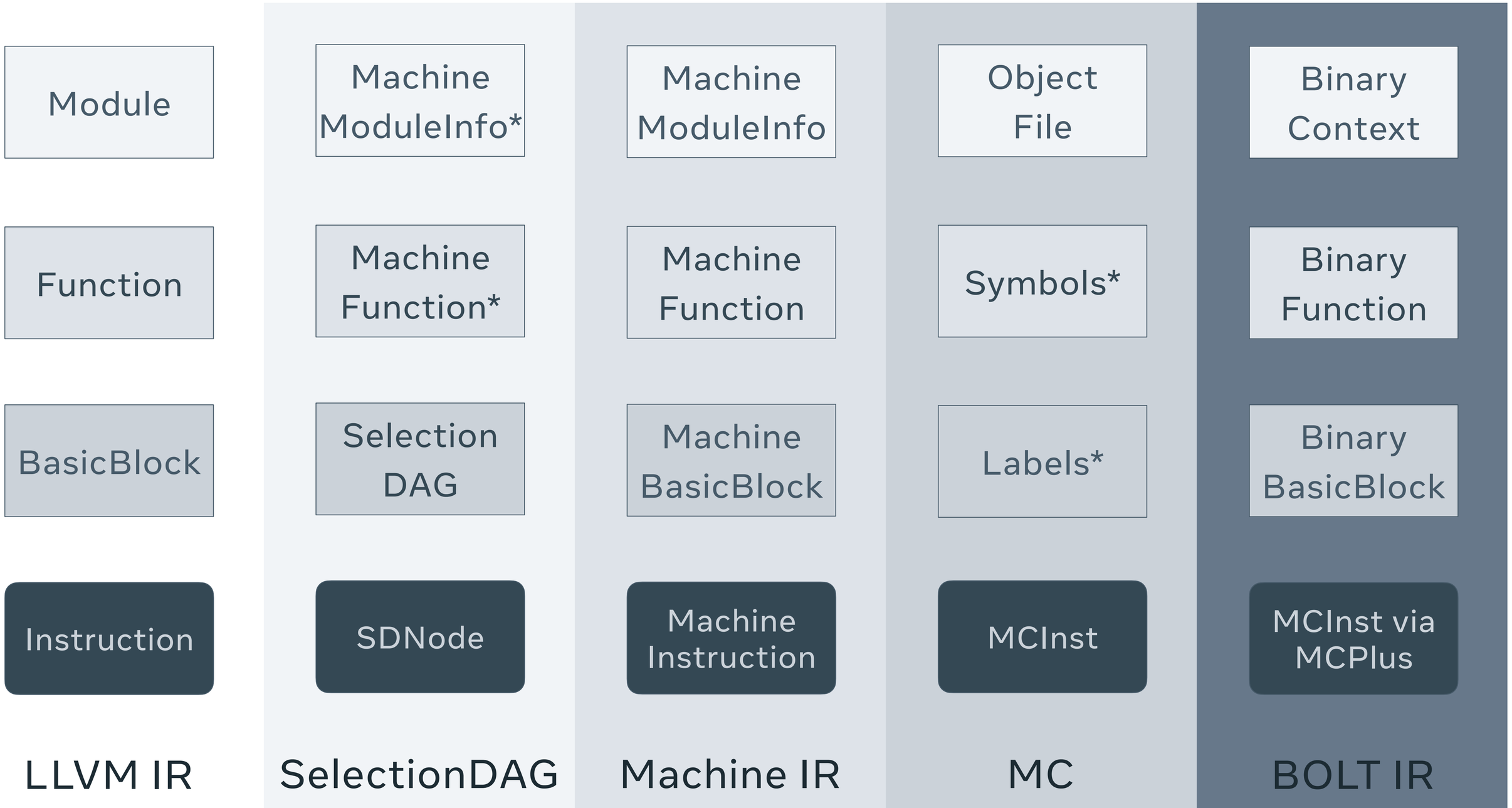
1. Find functions and data in code
2. Identify instructions inside functions
3. Analyze instructions and build CFG
4. Read profile and attribute to CFG edges
5. **Execute local and global optimization passes**
6. Generate code and process relocations
7. Write code to file and update ELF



LLVM IR



BOLT IR


Module
Function
BasicBlock
Instruction
LLVM IR
Machine ModuleInfo*
Machine Function*
Selection DAG
SDNode
SelectionDAG
Machine ModuleInfo
Machine Function
Machine BasicBlock
Machine Instruction
Machine IR
Object File
Symbols*
Labels*
MCInst
MC
Binary Context
Binary Function
Binary BasicBlock
MCInst via MCPlus
BOLT IR

MCPlus(Plus): extensible abstraction layer

MCPlus is BOLT's abstraction layer for

- target-specific info beyond MCRegInfo/MCInstrAnalysis
- target-independent info beyond opcode/operands
- analyses and manipulations

`MCPlusBuilder` class (`BC->MIB`)

Simple checks: `isNoop`, `isIndirectBranch`

Annotations: `addAnnotation`

Analysis: `analyzePLTEntry`, `evaluateX86MemoryOperand`

Instruction creation: `createTailCall`

Complex transformations: `indirectCallPromotion`

MCPlus examples

- Raising semantical level:
  - tail call, invoke, jump table
- Attaching analysis information to instructions:
  - Liveness, ReachingDefs

```
# MC Jump -> MCPlus Tail Call
  00000002:   ja func # TAILCALL  # CTCTakenCount: 4

# MC Call -> MCPlus Invoke using EH annotations
  00000043:   callq _Z11filteri # handler: .LLP2;
                                action: 1;
                                GNU_args_size = 0

# MC Indirect jump -> MCPlus Jump Table
  00000014:   jmpq  *%rax # JUMPTABLE @0x290
                          # JTIndexReg: 0

# Analysis information
  0040117a:   !PHI  %r8 # ID: 3
              # DF: %r8[23.], %r8[0.] -> %r8[23.]
```

03 Simple peephole rule

The best way to zero a register on X86?

- `movl $0x0, %eax    # [0xb8,0x00,0x00,0x00,0x00]`
- `andl $0x0, %eax    # [0x83,0xe0,0x00]`
- `subl %eax, %eax    # [0x29,0xc0]`
- `xorl %eax, %eax    # [0x31,0xc0]`

Leveraging existing passes

shortenInstructions pass calls **MCPlusBuilder::shortenInstruction** – fits the bill.

```
bool shortenInstruction(MCInst &Inst,
                        const MCSubtargetInfo &STI) const override {
...
  Inst.setOpcode(NewOpcode);

  // Replace `mov[lq] $0x0, %[er]ax` with `xor[lq] %[er]ax, %[er]ax`
  switch (NewOpcode) {
  default:
    break;

  case X86::MOV64ri:
  case X86::MOV64ri32:
  case X86::MOV32ri:
    auto OpNum = MCPlus::getNumPrimeOperands(Inst) - 1;
    if (Inst.getOperand(OpNum).isImm() && !Inst.getOperand(OpNum).getImm()) {
      if (NewOpcode == X86::MOV32ri)
        NewOpcode = X86::XOR32rr;
      else
        NewOpcode = X86::XOR64rr;
      MCOperand Op = Inst.getOperand(0);
      Inst.setOpcode(NewOpcode);
      Inst.clear();
      Inst.addOperand(Op);
      Inst.addOperand(Op);
      Inst.addOperand(Op);
    }
    break;
  }

  if (NewOpcode == OldOpcode)
    return false;

  Inst.setOpcode(NewOpcode);
  return true;
}
```

Bughunter script

Bisecting to a function which causes a crash.

Pass the resulting function as

`--funcs=funcname`

to reproduce the issue.

bolt/utils/bughunter.sh

Invocation:

```
BOLT=/build/llvm-bolt \
BOLT_OPTIONS="-v=1" \
INPUT_BINARY=/path/to/binary \
# COMMAND_LINE="--version" or
# OFFLINE=1 \
bolt/utils/bughunter.sh
```

Output:

Text file containing the culprit function.

--print-all

Producing text dumps for all processed functions after each pass.

--print-disasm
--print-cfg
--print-{pass}

--print-only=funcname

```
llvm-bolt /path/to/binary \
  --funcs=funcname --print-all
```

```
Before:
cmpb    $0x0, 0x8(%rax)
movl    $0x0, %edx
movq    -0x8(%rbp), %r13
movq    (%rdi), %rax
cmovel  %edx, %ebx
```

```
After:
cmpb    $0x0, 0x8(%rax)
xorl    %edx, %edx
movq    -0x8(%rbp), %r13
movq    (%rdi), %rax
cmovel  %edx, %ebx
```

AsmDump

Producing an annotated assembly which can be turned into a BOLT test.

--asm-dump[=dir]

```
llvm-bolt /path/to/binary \
  --funcs=funcname --asm-dump=dump_dir

# bolt/test/X86/zero-idiom.s
  .globl _start
  .type _start, %function
_start:
  .cfi_startproc
  cmpb  $0x0, 0x8(%rax)
  movl  $0x0, %edx
  movq  -0x8(%rbp), %r13
  movq  (%rdi), %rax
  cmovel  %edx, %ebx
  .cfi_endproc
.size _start, .-_start
```

04 Adding a pass

Logistics

Inherit from **BinaryFunctionPass**

Add to **bolt/lib/Passes/BinaryPasses.cpp** or **ZeroIdiom.cpp**

```
/* bolt/include/bolt/Passes/ZeroIdiom.h */
class ZeroIdiom : public BinaryFunctionPass {
public:
  explicit ZeroIdiom(const cl::opt<bool> &PrintPass)
      : BinaryFunctionPass(PrintPass) {}

  const char *getName() const override {
    return "zero-idiom";
  }

  void runOnFunctions(BinaryContext &) override;
};

/* bolt/lib/Passes/ZeroIdiom.cpp */
ZeroIdiom::runOnFunctions(BinaryContext &BC) {
```

Logistics

Append invocation to **BinaryPassManager.cpp**

```
/* bolt/lib/Rewrite/BinaryPassManager.cpp */
static cl::opt<bool> PrintZeroIdiom(
  "print-zero-idiom",
  cl::desc("print functions after zero idiom pass”),
  cl::cat(BoltOptCategory));

void BinaryFunctionPassManager::runAllPasses(
  BinaryContext &BC) {

  Manager.registerPass(
    std::make_unique<ZeroIdiom>(PrintZeroIdiom));

}
```

Extra analyses

Make use of `DataflowManager` providing `LivenessAnalysis`

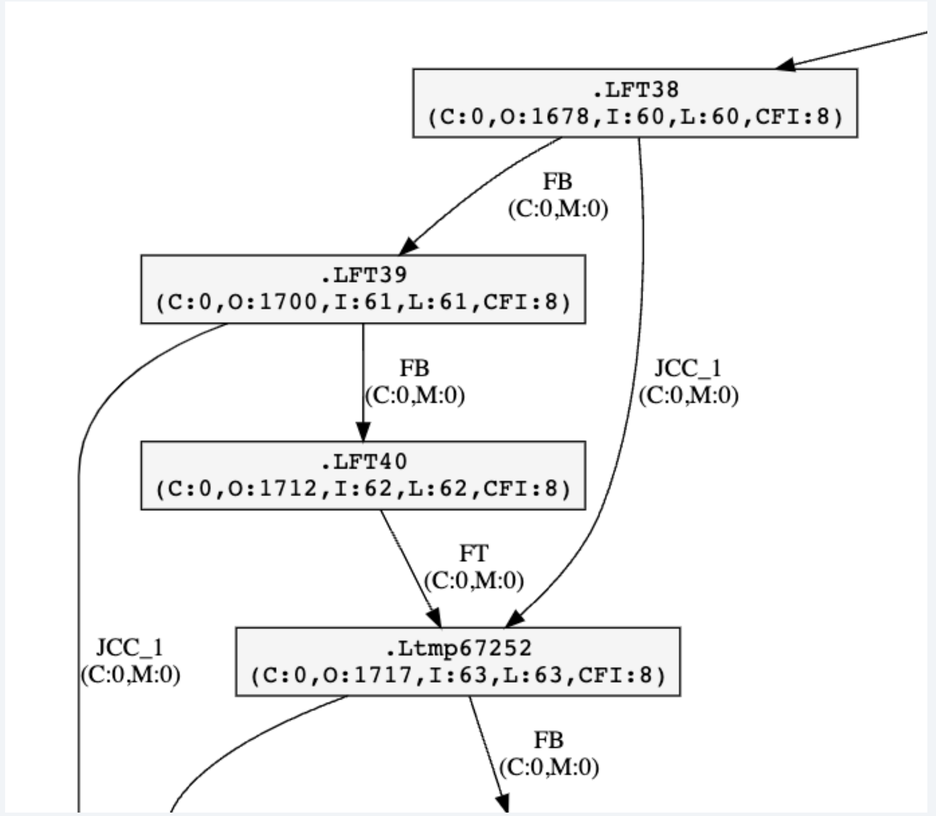
```
/* bolt/lib/Passes/ZeroIdiom.cpp */
void ZeroIdiom::runOnFunction(BinaryFunction &BF,
                              DataflowInfoManager &Info)
{
  BinaryContext &BC = BF.getBinaryContext();
  LivenessAnalysis &LA = Info.getLivenessAnalysis();

  for (BinaryBasicBlock &BB : BF) {
    for (MCInst &Inst : BB) {
      if (LA.isAlive(&Inst, BC.MIB->getFlagsReg()))
        continue;
      BC.MIB->replaceZeroIdiom(Inst);
    }
  }
}
```

dot format

```
llvm-bolt
   --dump-dot-all
```

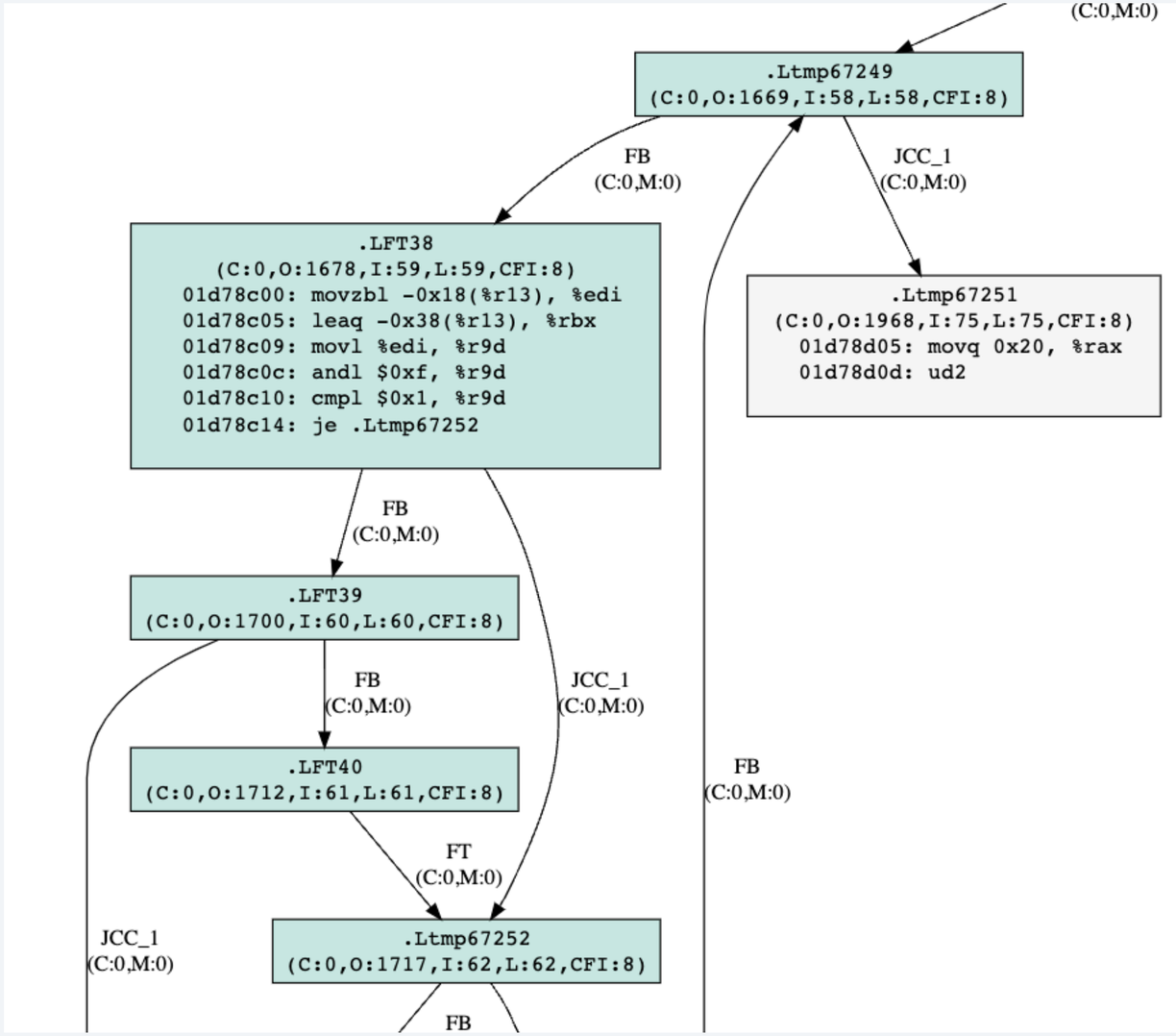
Outputs

```
funcname-00_build-cfg.dot
```



Interactive HTML

```
llvm-bolt
  --dump-dot-all
  --print-loops --dot-tooltip-code

bolt/utils/dot2html/dot2html.py
  main-25_zero-idiom.dot{,.html}
```




Meta