facebook

Building Binary Optimizer with LLVM

Maksim Panchenko

maks@fb.com

BOLT

Binary Optimization and Layout Tool

- Built in less than 6 months
- x64 Linux ELF
- Runs on large binary (HHVM, non-jitted part)
- Improves I-Cache, ITLB, branch misses
- Deployed to limited production

Overview

- Why a binary optimizer
- Is LLVM the best choice?
- Challenges
- Approaches to implementation
- Results
- Future plans

Why Binary Optimizer

- No need to link sample-based profile data to source code or IR
- Can optimize 3rd-party libraries without source code
- Has “whole-program” view
- Some optimizations could only be done to a binary

Existing Binary Optimizers

and Binary Rewriters

- HP ISpike
- Microsoft Vulcan/BBT
- Sun/Oracle Studio Binary Optimizer
- Intel PIN
- Dynamic binary optimizers
- Many More

Usage Model

Example with HHVM binary running in production

- perf record -b -e .... -a -- sleep 300
- perf2bolt perf.data -o perf.fdata -b hhvm
- llvm-bolt -data=perf.fdata hhvm -o hhvm.bolt

Why LLVM

- Disassembler
- Assembler
- ... sharing the same representation

- ELFs, DWARFs, and ORCs

Implementation Overview

- Code discovery
- Disassembly
- CFG construction
- Optimizations
- Available storage discovery
- Code (and data) emission

Discovery Process

Functions and Objects

- Symbol table
  - need unstripped binary
- `.eh_frame`
  - unwind info includes function boundaries
- No general problem solution
- Don’t need to know everything to optimize
- Relocations from the linker

Disassembly

- Relocation reconstruction for code
- `%rip`-relative addressing on x64
- Relocations for %rip operands
- tblgen fixes required for some instructions

CFG Construction

- x86 binary -> `MCInst` with CFG -> ORC -> x86 binary
- `MCInst` vs `MachineInstruction`
- No higher than `MachineInstruction`
- Conservative approach that works
- Modify code that we 100% understand

Optimizations

- Feedback-directed basic block reordering (modified Pettis-Hansen)
- Sample-based profiling with LBR
  - Can gather profile on a binary running in production
- On top of the linker script that does function placement

Allocating New Code and Data

ELF-specific

- Pretend we are linking for jitting
- Map address spaces for relocation processing
- No prior allocation required
- Tricky to relocate ELF program header table
- Fix section header table

Ready to run?

C++ Exceptions

IA64 “zero-cost”

- `.eh_frame` updated with new CFIs
  - Heavy usage of `RememberState/RestoreState`
- `.eh_frame_hdr` section and `GNU_EH_FRAME` program header
- `.gcc_except_table` with new call site table

Benchmark

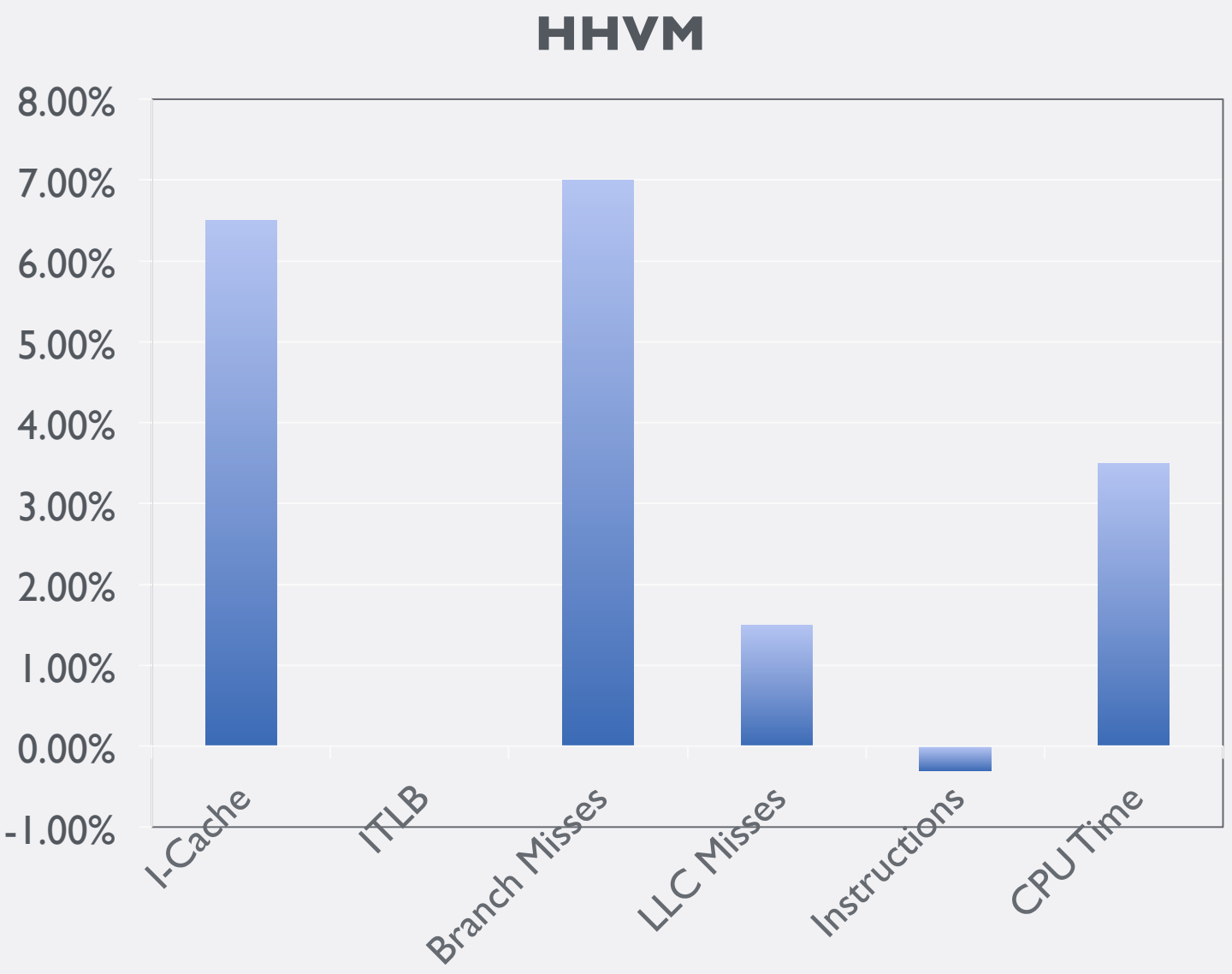
HHVM

- No SpecCPU2006
- PHP JIT
- github.com/facebook/hhvm
- More components linked-in at FB
- >100MB `.text`
- ~4GB with debug info

Benchmark

HHVM

- Hot paths marked with `__builtin_expect()`
- Hottest small functions written in assembly
- Carefully tuned inlining
- Linker script for function placement
- Huge pages for code
- <90% functions optimized by BOLT
- Execution time split between binary and jitted code


HHVM
8.00%
7.00%
6.00%
5.00%
4.00%
3.00%
2.00%
1.00%
0.00%
-1.00%
I-Cache
ITLB
Branch Misses
LLC Misses
Instructions
CPU Time

Updating Debug Information

DWARF

- WIP
- `.debug_info` mostly unchanged
- `DW_AT_ranges` replaces contiguous attributes
- `.debug_line` rewritten and `DW_AT_stmt_list` updated
- `.debug_ranges`, `.debug_aranges` modified
- `.debug_loc` modified
- More work with more optimizations

Limitations

- Well-formed C/C++
- Properly marked assembly functions
- Self-modifying code
- Self-validating code
- Not implemented
  - Multiple-entry functions
  - Switch tables

Future Optimizations

- Inlining
- De-virtualization
- Conditional tail-call
- ABI-breaking optimizations
  - Remove unnecessary spills/reloads after analyzing call chain
- Data reordering

Future Plans

- Linker-style optimizations
  - ICF
  - Unreachable/dead-code (gc-sections)
  - Function re-ordering
- 100% coverage
- Replace linker script and optimizations
- Move entry points
- Integrate into dynamic engine

Compared to AutoFDO/LTO

- No direct comparison
- Mixed results from AutoFDO when it works
- BOLT is faster than running linker with linker script
- The goal is to complement compiler and extract every single bit of performance out of a binary

Example

```
void foo(int c) {
  if (c > 0) {
    A; // macro A
  } else {
    B; // macro B
  }
}
```

```
void bar() {
  ...
  foo(/* > 0*/);
  ...
}
```

```
void baz() {
  ...
  foo(/* <= 0*/);
  ...
}
```

Example

```
void foo(int c) {
  if (c > 0) {
    A; // macro A
  } else {
    B; // macro B
  }
}
```

1000

```
void bar() {
  ...
  foo(/* > 0*/);
  ...
}
```

1000

```
void baz() {
  ...
  foo(/* <= 0*/);
  ...
}
```

Example

```
void foo(int c) {
  if (c > 0) {
    A; // macr
  } else {
    B; // macr
  }
}
```

1000

1000

1000

```
void bar() {
  ...
  foo(/* > 0*/);
  ...
}
```

1000

```
void baz() {
  ...
  foo(/* <= 0*/);
  ...
}
```

Example

```
void foo(int c) {
  if (c > 0) {
    A; // macro
  } else {
    B; // macro
  }
}
```

1000

1000

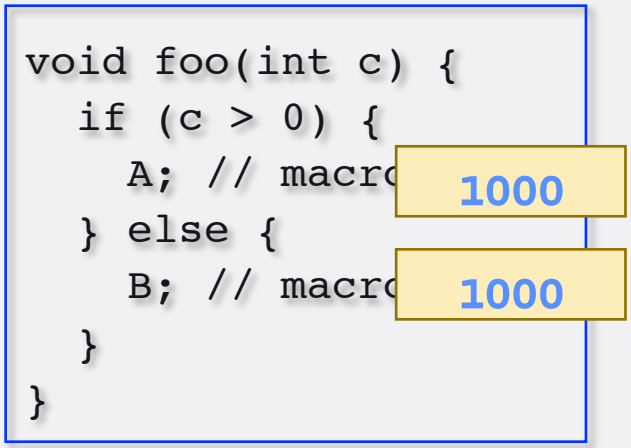
1000

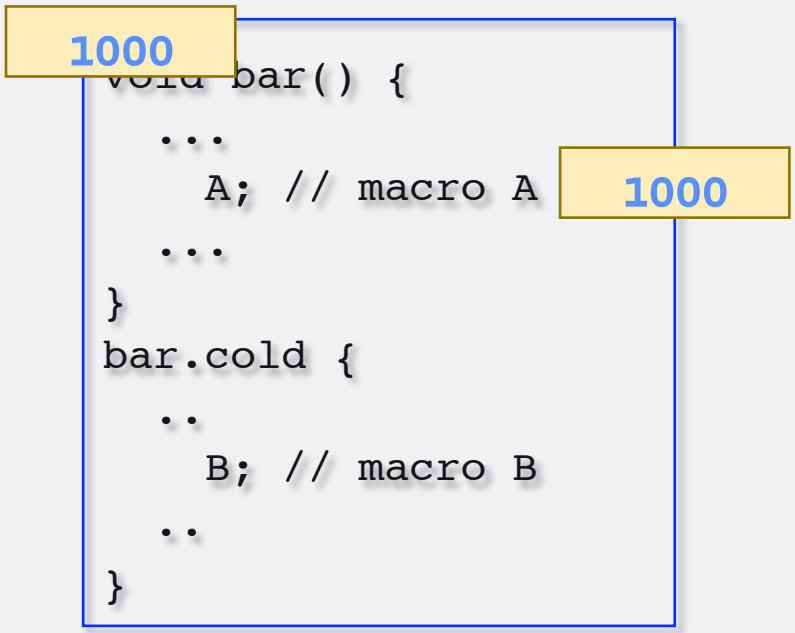
```
void bar() {
  ...
  ..
    A; // macro A
  ..
    B; // macro B
  ..
  ...
}
```

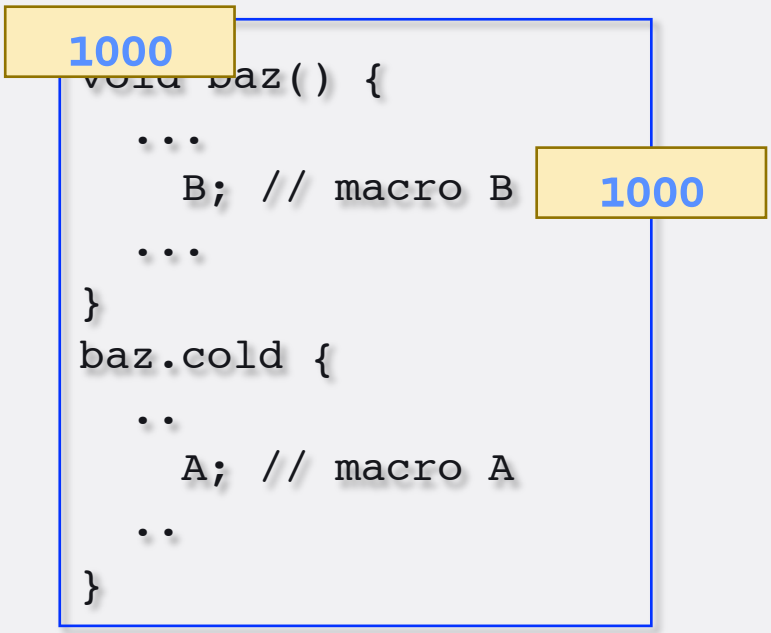
1000

```
void baz() {
  ...
  ..
    A; // macro A
  ..
    B; // macro B
  ..
  ...
}
```

Example


void foo(int c) {
if (c > 0) {
A; // macro
} else {
B; // macro
}
}
1000
1000


1000
void bar() {
...
A; // macro A
...
}
bar.cold {
..
B; // macro B
..
}
1000


1000
void baz() {
...
B; // macro B
...
}
baz.cold {
..
A; // macro A
..
}
1000

Thank You!

- LLVM community

- Rafael Auler - Facebook intern
- Gabriel Poesia - Facebook intern

facebook