BOLT: A Practical Binary Optimizer for Data Centers and Beyond

Maksim Panchenko, Rafael Auler, Bill Nell, Guilherme Ottoni
Facebook, Inc.
Menlo Park, CA, USA
{maks,rafaelauler,bnell,ottoni}@fb.com

***Abstract*—Performance optimization for large-scale applications has recently become more important as computation continues to move towards data centers. Data-center applications are generally very large and complex, which makes code layout an important optimization to improve their performance. This has motivated recent investigation of practical techniques to improve code layout at both compile time and link time. Although post-link optimizers had some success in the past, no recent work has explored their benefits in the context of modern data-center applications.**

In this paper, we present BOLT, an open-source post-link optimizer built on top of the LLVM framework. Utilizing sample-based profiling, BOLT boosts the performance of real-world applications even for highly optimized binaries built with both feedback-driven optimizations (FDO) and link-time optimizations (LTO). We demonstrate that post-link performance improvements are complementary to conventional compiler optimizations, even when the latter are done at a whole-program level and in the presence of profile information. We evaluated BOLT on both Facebook data-center workloads and open-source compilers. For data-center applications, BOLT achieves up to 7.0% performance speedups on top of profile-guided function reordering and LTO. For the GCC and Clang compilers, our evaluation shows that BOLT speeds up their binaries by up to 20.4% on top of FDO and LTO, and up to 52.1% if the binaries are built without FDO and LTO.

I. Introduction

Given the large scale of data centers, optimizing their workloads has recently gained a lot of interest. Modern data-center applications tend to be very large and complex programs. Due to their sheer amount of code, optimizing code locality for these applications is key to improving their performance.

The large size and performance bottlenecks of data-center applications make them good targets for *feedback-driven optimizations* (FDO), also called *profile-guided optimizations* (PGO), particularly code layout. At the same time, due to their large sizes, applying FDO to these applications poses scalability challenges. Instrumentation-based profilers incur significant memory and computational performance costs, often making it impractical to gather accurate profiles from a production system. To simplify deployment and increase adoption, it is desirable to have a system that can obtain profile data for FDO from unmodified binaries running in their normal production environments. This is possible through the use of *sample-based profiling*, which enables high-quality profiles to be gathered with minimal operational complexity. This is the approach taken by tools such as Ispike [1], AutoFDO [2], and HFSort [3]. This same principle is used as the basis of the BOLT tool presented in this paper.

Profile data obtained via sampling can be retrofitted to multiple points in the compilation chain. The point in which the profile data is used can vary from compilation time (e.g. AutoFDO [2]), to link time (e.g. LIPO [4] and HFSort [3]), to post-link time (e.g. Ispike [1]). In general, the earlier in the compilation chain the profile information is inserted, the larger the potential for its impact, since more phases and optimizations can benefit from this information. This benefit has motivated recent work on compile-time and link-time FDO techniques. At the same time, post-link optimizations, which in the past were explored by a series of proprietary tools such as Spike [5], Etch [6], FDPR [7], and Ispike [1], have not attracted much attention in recent years. We believe this lack of interest in post-link optimizers is due to folklore and the intuition that this approach is inferior because the profile data is injected very late in the compilation chain.

In this paper, we demonstrate that the intuition described above is incorrect. The important insight that we leverage in this work is that, although injecting profile data earlier in the compilation chain enables its use by more optimizations, injecting this data later enables more accurate use of the information for better code layout. In fact, one of the main challenges with AutoFDO is to map the profile data, collected at the binary level, back to the compiler's intermediate representations [2]. In the original compilation used to produce the binary where the profile data is collected, many optimizations are applied to the code by the compiler and linker before the machine code is emitted. In a post-link optimizer, which operates at the binary level, this problem is much simpler, resulting in more accurate use of the profile data. This accuracy is particularly important for low-level optimizations such as code layout.

We demonstrate the finding described above in the context of a static binary optimizer we built, called BOLT. BOLT is a modern, open-source, retargetable binary optimizer built on top of the LLVM compiler infrastructure [8]. Our experimental evaluation on large real-world applications shows that BOLT can improve performance by up to 20.4% on top of FDO and LTO. Furthermore, our analysis demonstrates that this improvement is mostly due to the superior code layout that

†Artifact available at: `https://doi.org/10.5281/zenodo.2542117`

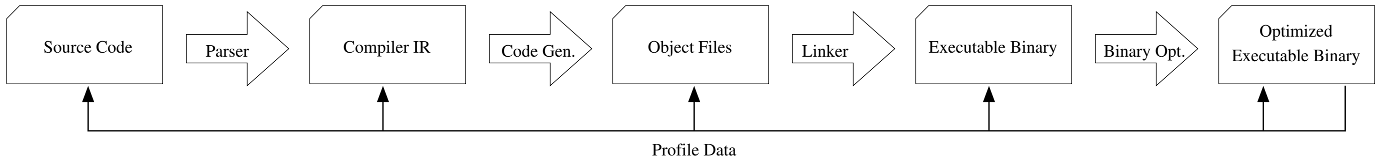

Fig. 1. Example of a compilation pipeline and the various alternatives to retrofit sample-based profile data.

is enabled by the more accurate usage of sample-based profile data at the binary level.

Overall, this paper makes the following contributions:

1) It describes the design of a modern, open-source post-link optimizer built on top of the LLVM infrastructure.[1]
2) It demonstrates empirically that a post-link optimizer is able to better utilize profiling data to improve code layout compared to a compiler-based approach.
3) It shows that neither compile-time, link-time, nor post-link-time FDO supersedes the others but, instead, they are complementary.

This paper is organized as follows. Section II motivates the case for using sample-based profiling and static binary optimization to improve the performance of large-scale applications. Section III then describes the architecture of the BOLT binary optimizer, followed by a description of the optimizations that BOLT implements in Section IV and a discussion about profiling techniques in Section V. An evaluation of BOLT and a comparison with other techniques is presented in Section VI. Finally, Section VII discusses related work, and Section VIII concludes the paper.

II. Motivation

In this section, we motivate the post-link optimization approach used by BOLT.

A. Why sample-based profiling?

Feedback-driven optimizations (FDO) have been proven to help increase the impact of code optimizations in a variety of systems (e.g. [2, 4, 9, 10, 11]). Early developments in this area relied on *instrumentation-based profiling*, which requires a special instrumented build of the application to collect profile data. This approach has two drawbacks. First, it complicates the build process, since it requires a special build for profile collection. Second, instrumentation typically incurs very significant CPU and memory overheads. These overheads generally render instrumented binaries inappropriate for running in real production environments.

In order to increase the adoption of FDO in production environments, recent work has investigated FDO-style techniques based on *sample-based profiling* [2, 3, 12]. Instead of instrumentation, these techniques rely on much cheaper sampling using hardware profile counters available in modern CPUs, such as Intel's Last Branch Records (LBR) [13]. This approach is more attractive not only because it does not require a special build of the application, but also because the profile-collection overheads are negligible. By addressing the two main drawbacks of instrumentation-based FDO techniques, sample-based profiling has increased the adoption of FDO-style techniques in complex, real-world production systems [2, 3]. For these same practical reasons, we opted to use sample-based profiling in this work.

B. Why a binary optimizer?

Sample-based profile data can be leveraged at various levels in the compilation pipeline. Figure 1 shows a generic compilation pipeline to convert source code into machine code. As illustrated in Figure 1, the profile data may be injected at different program-representation levels, ranging from source code to the compiler's intermediate representations (IR) to the linker and post-link optimizers. In general, the designers of any FDO tool are faced with the following trade-off. On the one hand, injecting profile data earlier in the pipeline allows more optimizations along the pipeline to benefit from this data. On the other hand, the mapping of the profiling data at a given level is more accurate the closer that representation is to the level at which the data was recorded. Since sample-based profile data must be collected at the binary level, a post-link binary optimizer allows the profile data to be used with the highest level of accuracy.

AutoFDO [2] retrofits profile data back into a compiler's intermediate representation (IR). Chen et al. [12] quantified the precision of the profile data that is lost by retrofitting profile data even at a reasonably low-level representation in the GCC compiler. They quantified that the profile data had 84.1% accuracy, which they were able to improve to 92.9% with some techniques described in that work.

The example in Figure 2 illustrates the difficulty in mapping binary-level performance events back to a higher-level representation. In this example, both functions `bar` and `baz` call function `foo`, which gets inlined in both callers. Function `foo` contains a conditional branch for the `if` statement on line `(02)`. On modern processors, it is advantageous to make the most common successor of forward branches like this be the fall through, which can lead to better branch prediction and instruction-cache locality. This means that, when `foo` is inlined into `bar`, block `B1` should be placed before `B2`, but the blocks should be placed in the opposite order when inlined into `baz`. When this program is profiled at the binary level, two branches corresponding to the `if` in line `(02)` will be profiled, one within `bar` and one within `baz`. Assume that functions `bar` and `baz` execute the same number of times

[1]BOLT is available at `https://github.com/facebookincubator/BOLT`.

```
(01)      function foo(int x) {
(02)        if (x > 0) {
(03)          ... // B1
(04)        } else {
(05)          ... // B2
(06)        }
(07)      }

(08)      function bar() {
(09)        foo(... /* > 0 */); // gets inlined
(10)      }

(11)      function baz() {
(12)        foo(... /* < 0 */); // gets inlined
(13)      }
```

Fig. 2. Example showing a challenge in mapping binary-level events back to higher-level code representations.

at runtime. Then, when mapping the branch frequencies back to the source code in Figure 2, one will conclude that the branch at line `(02)` has a 50% chance of branching to both `B1` and `B2`. And, after `foo` is inlined in both `bar` and `baz`, the compiler will not be able to tell which layout is best in each case.

Since our initial motivation for BOLT was to improve large-scale data-center applications, where code layout plays a major role, a post-link binary optimizer was very appealing. Traditional code-layout techniques are highly dependent on accurate branch frequencies [14], and using inaccurate profile data can actually lead to performance degradation [12]. Nevertheless, as we mentioned earlier, feeding profile information at a very low level prevents earlier optimizations in the compilation pipeline from leveraging this information. Therefore, with this approach, any optimization that we want to benefit from the profile data needs to be applied at the binary level. Fortunately, code layout algorithms are relatively simple and easy to apply at the binary level.

C. *Why a* static *binary optimizer?*

The benefits of a binary-level optimizer outlined above can be exploited either statically or dynamically. We opted for a static approach for two reasons. The first one is the simplicity of the approach. The second was the absence of runtime overheads. Even though dynamic binary optimizers have had some success in the past (e.g. Dynamo [15], DynamoRIO [16], StarDBT [17]), these systems incur non-trivial overheads that go against the main goal of improving the overall performance of the target application. In other words, these systems need to perform really well in order to recover their overheads and achieve a net performance win. Unfortunately, since they need to keep their overheads low, these systems often have to implement faster, sub-optimal code optimization passes. This has been a general challenge to the adoption of dynamic binary optimizers, as they are not suited for all applications and can easily degrade performance if not tuned well. The main benefit of a dynamic binary optimizer over a static one is the ability to handle dynamically generated and self-modifying code.

III. Architecture

Large-scale data-center binaries may contain over 100 MB of code from multiple source-code languages, including assembly language. In this section, we discuss the design of the BOLT binary optimizer that we created to operate in this scenario.

A. *Initial Design*

We developed BOLT by incrementally increasing its binary code coverage. At first, BOLT was only able to optimize the code layout of a limited set of functions. With time, code coverage gradually increased by adding support for more complex functions. Even today, BOLT is still able to leave some functions in the binary untouched while processing and optimizing others, conservatively skipping code that violates its current assumptions.

The initial implementation targeted x86-64 Linux ELF binaries and relied exclusively on ELF symbol tables to guide binary content identification. By doing that, BOLT was able to optimize code layout within existing function boundaries. When BOLT was not able to reconstruct the control-flow graph of a given function with full confidence, it would leave the function untouched.

Due to the nature of code layout optimizations, the effective code size may increase for a couple of reasons. First, this may happen due to an increase in the number of branches on cold paths. Second, there is a peculiarity of x86's conditional branch instruction, which occupies 2 bytes if a (signed) offset to a destination fits in 8 bits but otherwise takes 6 bytes for 32-bit offsets. Naturally, moving cold code further away showed a tendency to increase the hot code size. If an optimized function did not fit into the original function's allocated space, BOLT would split the cold code and move it to a newly created ELF segment. Note that such function splitting was involuntary and did not provide any extra benefit beyond allowing code straightening optimizations as BOLT was not filling out the freed space between the split point and the next function.

B. *Relocations Mode*

A second and more ambitious mode was later added to change the position of all functions in the binary. While multiple approaches were considered, the most obvious and straightforward one was to rely on relocations recorded and saved in the executable by the linker. Both BFD and Gold linkers provide such an option (`--emit-relocs`). However, even with this option, there are still some missing pieces of information. An example is the relative offsets for PIC jump tables which are removed by the linker. Other examples are some relocations that are not visible even to the linker, such as cross-function references for local functions within a single compilation unit (they are processed internally by the compiler). Therefore, to detect and fix such references, it is essential to disassemble all the code correctly before trying to rearrange the functions in the binary. Nevertheless, with relocations, the job of gaining complete control over code re-writing became much easier. Handling relocations gives BOLT

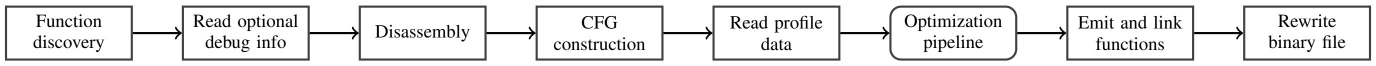

Fig. 3. Diagram showing BOLT's binary rewriting pipeline.

the ability to change the order of functions in the binary and split function bodies to further improve code locality.

Since linkers have access to relocations, it would be possible to use them for similar binary optimizations. However, there are multiple open-source linkers for x86 Linux alone, and selecting one to use for any particular application depends on a number of circumstances that may also change over time. Therefore, in order to facilitate the tool's adoption, we opted for writing an independent post-link optimizer instead of being tied to a specific linker.

C. Rewriting Pipeline

Analyzing an arbitrary binary and locating code and data is not trivial. In fact, the problem of precisely identifying and disassembling machine code is undecidable in general. However, at a very minimum, every executable binary has clear entry point information available for the operating system or a dynamic linker indicating where execution should start. In practice, there is more information than just an entry point available, and BOLT relies on correct ELF symbol table information for code discovery. Since BOLT works with 64-bit Linux binaries, the ABI requires the inclusion of function frame information that contains function boundaries as well. While BOLT could have relied on this information, it is often the case that functions written in assembly omit frame information. Thus, we decided to employ a hybrid approach using both symbol table and frame information when available.

Figure 3 shows a diagram with BOLT's rewriting steps. Function discovery is the very first step, where functions are created, assigned names, and bound to addresses. Later, functions are disassembled while debug information is retrieved if available.

BOLT uses the LLVM compiler infrastructure [8] to handle disassembly and modification of binary files. There are a couple of reasons why LLVM is well suited for BOLT. First, LLVM has a nice modular design that enables relatively easy development of tools based on its infrastructure. Second, LLVM supports multiple target architectures, which allows for easily retargetable tools. To illustrate this point, a working prototype for the ARM architecture was implemented in less than a month. In addition to the assembler and disassembler, many other components of LLVM proved to be useful while building BOLT. Overall, this decision to use LLVM has worked out well. The LLVM infrastructure has enabled a quick implementation of a robust and easily retargetable binary optimizer.

As Figure 3 shows, the next step in the rewriting pipeline is to build the control-flow graph (CFG) representation for each function. The CFG is constructed using the `MCInst` objects provided by LLVM's Tablegen-generated disassembler. BOLT reconstructs the control-flow information by analyzing all branch instructions encountered during disassembly. Then, in the CFG representation, BOLT runs its optimization pipeline, which is explained in detail in Section IV. For BOLT, we have added a generic annotation mechanism to `MCInst` in order to expand the semantics of `MCInst` to fit BOLT's CFG requirements. For example, annotations are used to describe `invoke` instructions, tail calls, and indirect branches using jump tables, among other things. Annotations are also used to facilitate certain optimizations, e.g. by providing a way of recording data-flow information. The final steps involve emitting functions and using LLVM's dynamic linker mechanism (created for the LLVM JIT systems) to resolve references created by BOLT for code and data. Finally, the binary is rewritten with the new contents while also updating ELF structures to reflect the changes.

D. C++ Exceptions and Debug Information

BOLT is able to recognize DWARF [18] information and update it to reflect the code modifications and relocations performed during the rewriting pass.

Figure 4 shows an example of a CFG dump demonstrating BOLT's internal representation of the binary for the first two basic blocks of a function with C++ exceptions and a throw statement. The function is quite small with only five basic blocks in total, and each basic block is free to be relocated to another position, except for the entry point. Placeholders for DWARF *Call Frame Information* (CFI) instructions are used to annotate positions where the frame state changes (for example, when the stack pointer advances). BOLT rebuilds all CFI for the new binary based on these annotations, so the frame unwinder works properly when an exception is thrown. The `callq` instruction at offset `0x00000010` can throw an exception and has a designated landing pad as indicated by a landing-pad annotation displayed next to it (`handler: .LLP0; action: 1`). The last annotation on the line indicates a source line origin for every machine-level instruction.

IV. Optimizations

BOLT runs passes with either code transformations or analyses, similar to a compiler. BOLT is also equipped with a data-flow analysis framework to feed information to passes that need it. This enables BOLT to check register liveness at a given program point, a technique also used by Ispike [1]. Some passes are architecture-independent while others are not. In

```
Binary Function "_Z11filter_onlyi" after building cfg {
  State       : CFG constructed
  Address     : 0x400ab1
  Size        : 0x2f
  Section     : .text
  LSDA        : 0x401054
  IsSimple    : 1
  IsSplit     : 0
  BB Count    : 5
  CFI Instrs  : 4
  BB Layout   : .LBB07, .LLP0, .LFT8, .Ltmp10, .Ltmp9
  Exec Count  : 104
  Profile Acc : 100.0%
}
.LBB07 (11 instructions, align : 1)
  Entry Point
  Exec Count : 104
  CFI State : 0
    00000000:  pushq  %rbp # exception4.cpp:22
    00000001:  !CFI   $0      ; OpDefCfaOffset -16
    00000001:  !CFI   $1      ; OpOffset Reg6 -16
    00000001:  movq   %rsp, %rbp # exception4.cpp:22
    00000004:  !CFI   $2      ; OpDefCfaRegister Reg6
    00000004:  subq   $0x10, %rsp # exception4.cpp:22
    00000008:  movl   %edi, -0x4(%rbp) # exception4.cpp:22
    0000000b:  movl   -0x4(%rbp), %eax # exception4.cpp:23
    0000000e:  movl   %eax, %edi # exception4.cpp:23
    00000010:  callq  _Z3fooi # handler: .LLP0; action: 1
                              # exception4.cpp:23
    00000015:  jmp    .Ltmp9 # exception4.cpp:24
  Successors: .Ltmp9 (mispreds: 0, count: 100)
  Landing Pads: .LLP0 (count: 4)
  CFI State: 3
.LLP0 (2 instructions, align : 1)
  Landing Pad
  Exec Count : 4
  CFI State : 3
  Throwers: .LBB07
    00000017:  cmpq   $-0x1, %rdx # exception4.cpp:24
    0000001b:  je     .Ltmp10 # exception4.cpp:24
  Successors: .Ltmp10 (mispreds: 0, count: 4),
              .LFT8 (inferred count: 0)
  CFI State: 3
....
```

Fig. 4. Partial CFG dump for a function with C++ exceptions.

this section, we discuss the passes applied to the Intel x86-64 target.

Table I shows each individual BOLT optimization pass in the order they are applied. For example, the first line presents `strip-rep-ret` at the start of the pipeline. Notice that passes 1 and 4 are focused on leveraging precise target architecture information to remove or mutate some instructions. A use case of BOLT for data-center applications is to allow the user to trade any optional choices in the instruction space in favor of I-cache space, such as removing alignment NOPs and AMD-friendly REPZ bytes, or using shorter versions of instructions. Our findings show that, for large applications, it is better to aggressively reduce I-cache occupation, except if the change incurs D-cache overhead since the cache is one of the most constrained resources in the data-center space. This explains BOLT's policy of discarding all NOPs after reading the input binary. Even though compiler-generated alignment NOPs are generally useful, the extra space required by them does not pay off and simply stripping them from the binary provides a small but measurable performance improvement.

BOLT features identical code folding (ICF) to complement the ICF optimization done by the linker. An additional benefit of doing ICF at the binary level is the ability to optimize functions that were compiled without the `-ffunction-sections` flag and functions that contain jump tables. As a result, BOLT is able to fold more identical functions than the linkers. We have measured the reduction of code size for the HHVM binary [19] to be about 3% on top of the linker's ICF pass.

TABLE I
SEQUENCE OF TRANSFORMATIONS APPLIED BY BOLT'S OPTIMIZATION PIPELINE.

Pass Name	Description
1. strip-rep-ret	Strip `repz` from `repz retq` instructions used for legacy AMD processors
2. icf	Identical code folding
3. icp	Indirect call promotion
4. peepholes	Simple peephole optimizations
5. simplify-ro-loads	Fetch constant data in .rodata whose address is known statically and mutate a load into a mov
6. icf	Identical code folding (second run)
7. plt	Remove indirection from PLT calls
8. reorder-bbs	Reorder basic blocks and split hot/cold blocks into separate sections (layout optimization)
9. peepholes	Simple peephole optimizations (second run)
10. uce	Eliminate unreachable basic blocks
11. fixup-branches	Fix basic block terminator instructions to match the CFG and the current layout (redone by reorder-bbs)
12. reorder-functions	Apply HFSort [3] to reorder functions (layout optimization)
13. sctc	Simplify conditional tail calls
14. frame-opts	Removes unnecessary caller-saved register spilling
15. shrink-wrapping	Moves callee-saved register spills closer to where they are needed, if profiling data shows it is better to do so

Passes 3 (indirect call promotion) and 7 (PLT call optimization) leverage call frequency information to mutate a function call into a more performant version. BOLT also has an experimental inlining pass, which is currently disabled by default. BOLT's function inliner is a limited version of what compilers perform at higher levels. We expect that most of the inlining opportunities will be leveraged by the compiler (potentially using FDO). The remaining inlining opportunities for BOLT are typically exposed by more accurate profile data, BOLT's indirect-call promotion (ICP) optimization, cross-module nature, or a combination of these factors.

Pass 5, simplification of load instructions, explores a tricky tradeoff by fetching data from statically known values (in read-only sections). In these cases, BOLT may convert loads into immediate-loading instructions, relieving pressure from the D-cache but possibly increasing pressure on the I-cache, since the data is now encoded in the instruction stream. BOLT's policy, in this case, is to abort the promotion if the new instruction encoding is larger than the original load instruction, even if it means avoiding an arguably more computationally expensive load instruction. However, we found that such opportunities are not very frequent in our workloads.

Pass 8, reorder and split hot/cold basic blocks, reorders basic blocks based on profile counts so that the hottest successor will most likely be a fall-through, reducing taken branches and relieving pressure from the branch predictor unit.

Finally, pass 12 reorders the functions via the HFSort technique [3]. This optimization mainly improves I-TLB performance, but it also helps with I-cache to a smaller extent. Combined with pass 8, these are the most effective ones in BOLT because they directly optimize the code layout.

V. Profiling Techniques

This section discusses the pitfalls and caveats of different sample-based profiling techniques when trying to produce accurate profiling data.

A. Techniques

In recent Intel microprocessors, LBR is a list of the last 32 taken branches. LBRs are important for profile-guided optimizations not only because they provide accurate counts for critical edges (which cannot be inferred even with perfect basic block count profiling [20]), but also because they make block-layout algorithms more resilient to bad sampling. We found that the performance impact of BOLT optimizations that rely on profile data collected with LBRs is very consistent even for different sampling events, such as *retired instructions*, *taken branches*, and *cycles*. We also experimented with different levels of *Precise Event-Based Sampling* (PEBS) [13]. In all these cases, for a workload for which BOLT provided a 5.4% speedup, the performance differences between them were within 1%. However, if using profile collected without LBRs with the wrong combination of sampled events and algorithm to reconstruct edge counts, it is possible to observe as much as 5% performance penalty when compared to the best-case scenario with LBRs, meaning BOLT misses nearly all post-link optimization opportunities. We did succeed in tuning non-LBR profile collection techniques to stay under 1% worse than the LBR ones in this example workload, but if LBR is available in the processor, one is better off using it to obtain higher and more robust performance results. We also evaluate this effect for HHVM in Section VI-F.

B. Consequences for Block Layout

Using LBRs, in a hypothetical worst-case biasing scenario where all samples in a function are recorded in the same basic block, BOLT will lay out blocks in the order of the path that leads to this block. It is an incomplete layout that misses the ordering of successor blocks, but it is not an invalid nor a cold path. In contrast, when trying to infer the same edge counts with non-LBR samples, the scenario is that of a single hot basic block with no information about which path was taken to get to it.

In practice, even in LBR mode, many times the collected profile is contradictory by stating that predecessors execute many times more than their single successor, among other violations of flow equations.[2] Previous work [20, 21], which includes techniques implemented in IBM's FDPR [7], report handling the problem of reconstructing edge counts by solving an instance of *minimum cost flow* (MCF [20]), a graph network flow problem. However, these reports predate LBRs. LBRs only store taken branches, so when handling very skewed data such as the cases mentioned above, BOLT satisfies the flow equation by attributing all surplus flow to the non-taken path that is naturally missing from the LBR, similarly to Chen et al. [12]. BOLT also benefits from being applied after the static compiler: to cope with uncertainty, by putting weight on the fall-through path, it trusts the original layout done by the static compiler. Therefore, the program trace needs to show a significant number of taken branches, which contradict the original layout done by the compiler, to convince BOLT to reorder the blocks and change the original fall-through path. Without LBRs, it is not possible to take advantage of this: algorithms start with guesses for both taken and non-taken branches without being sure if the taken branches, those readily available in LBR mode, are real or the result of bad edge-count inference.

[2]I.e., the sum of a block's input flow is equal to the sum of its output flow.

C. Consequences for Function Layout

BOLT uses HFSort [3] to perform function reordering based on a weighted call graph. If LBRs are used, the edge weights of the call graph are directly inferred from the branch records, which may also include function calls and returns. Without LBRs, BOLT is still able to build an incomplete call graph by looking at the direct calls in the binary and creating caller-callee edges with weights corresponding to the number of samples recorded in the blocks containing the corresponding `call` instructions. However, this approach cannot take indirect calls into account. Even with these limitations, we did not observe a performance penalty as severe as using non-LBR mode for basic block reordering (Section VI-F)

VI. Evaluation

This section evaluates BOLT in a variety of scenarios, including Facebook server workloads and the GCC and Clang open-source compilers. A comparison with GCC's and Clang's PGO and LTO is also provided in some scenarios. The evaluation presented in this section was conducted on Linux-based servers featuring Intel microprocessors.

A. Facebook Workloads

The impact of BOLT was measured on five binaries inside Facebook's data centers. The first is HHVM [19], the PHP/Hack virtual machine that powers the web servers at Facebook and many other websites, including Baidu and Wikipedia. The second is TAO [22], a highly distributed, in-memory, data-caching service used to store Facebook's social graph. The third one is Proxygen, which is a cluster load balancer built on top of the open-source library with the same name [23]. Finally, the other two binaries implement a service called Multifeed, which is used to select what is shown in the Facebook News Feed.

In this evaluation, we compared the performance impact of BOLT on top of binaries built using GCC and function reordering via HFSort [3]. The HHVM binary specifically is compiled with LTO to further enhance its performance. Unfortunately, a comparison with FDO and AutoFDO was not possible. The difficulties with FDO were the common ones outlined in Section II-A to deploy instrumented binaries

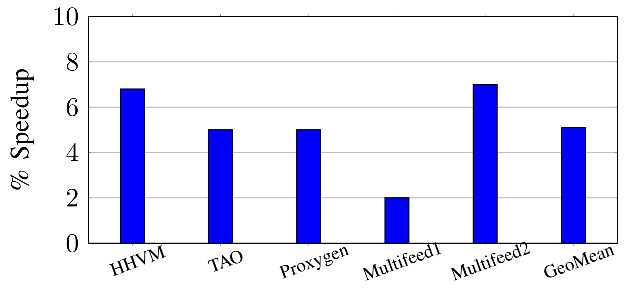

Fig. 5. Performance improvements from BOLT for our set of Facebook data-center workloads.

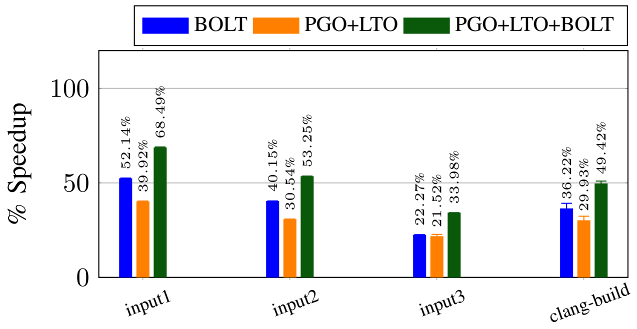

Fig. 6. Performance improvements for Clang.

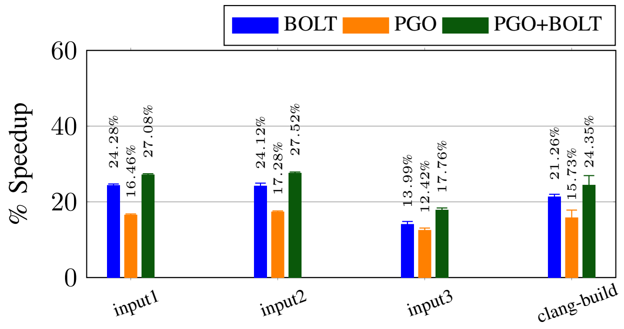

Fig. 7. Performance improvements for GCC. LTO was not used due to build errors.

in these applications' normal production environments. And we found that AutoFDO support in the two versions of GCC available in our environment (version 5.4.1 and 8) is not stable and caused either internal compiler errors or runtime errors related to C++ exceptions. Nevertheless, a direct comparison between BOLT and FDO was possible for other applications, and the results are presented in Section VI-B.

Figure 5 shows the performance results for applying BOLT on top of HFSort for our set of Facebook data-center workloads (and, in case of HHVM, also on top of LTO). In all cases, BOLT's application resulted in a speedup, with an average of 5.1% and a maximum of 7.0% for Multifeed2. Note that HHVM, despite containing a large amount of dynamically compiled code that is not optimized by BOLT, still gets a 6.8% speedup through BOLT. That is because HHVM spends more time in statically compiled code than in the dynamically generated code. Among these applications, HHVM has the largest total code size, which makes it very front-end bound and thus more amenable to the code layout optimizations that BOLT implements. To better understand the performance benefits of BOLT, Section VI-C shows the impact of BOLT on various micro-architectural metrics and also a breakdown of the benefit of different BOLT optimizations.

B. Clang and GCC Compilers

BOLT should be able to improve the performance of any front-end bound application, not just data-center workloads. To demonstrate this, we ran BOLT on two open-source compilers: Clang and GCC.

1) Clang Setup: For our Clang evaluation, we used the `release_70` branch of LLVM, Clang, and compiler-rt open-source repositories [24]. We built a bootstrapped release version of the compiler first. This stage1 compiler provided a baseline for our evaluation. We then built an instrumented version of Clang,[3] and then used the instrumented compiler to build Clang again with default options. The collected profile data was used to do another build of Clang with LTO enabled.[4] This is referred to as `PGO+LTO` in our chart.

Each of the two compilers was profiled with our training input, a full build of GCC. We used the Linux perf utility with options `record -e cycles:u -j any,u`. The profile from perf was converted using `perf2bolt` utility into YAML format (`-w` option). Then the profile was used to optimize the compiler binary using BOLT with the following options:

```
-b profile.yaml -reorder-blocks=cache+
-reorder-functions=hfsort+ -split-functions=3
-split-all-cold -split-eh -dyno-stats -icf=1 -use-gnu-stack
```

The four compilers were then used to build Clang, and the overall build time was measured for benchmarking purposes. For all builds above, we used `ninja` instead of GNU make, and for all benchmarks, we ran them with `-j40 clang` option. We chose to build only the Clang binary (as opposed to the full build) to minimize the effect of link time on our evaluation.

We have also selected three Clang/LLVM source files ranging from small to large sizes and preprocessed those files such that they could be compiled without looking up header dependencies. The three source files we used are:

- input1: tools/clang/lib/CodeGen/CGVTT.cpp
- input2: lib/ExecutionEngine/Orc/OrcCBindings.cpp
- input3: lib/Target/X86/X86ISelLowering.cpp

Each of the files was then compiled with `-std=c++11 -O2` options multiple times, and the results were recorded for benchmarking purposes. Tests were run on a dual-node 20-core (40-thread with hyperthreading) IvyBridge (Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz) system with 32 GB of RAM.

[3] `-DLLVM_BUILD_INSTRUMENTED=ON`

[4] `-DLLVM_ENABLE_LTO=Full -DLLVM_PROFDATA_FILE=clang.profdata`

TABLE II
STATISTICS REPORTED BY BOLT WHEN APPLIED TO CLANG'S BASELINE AND PGO+LTO BINARIES.

Metric	Over Baseline	Over PGO+LTO
executed forward branches	-1.6%	-1.0%
taken forward branches	-83.9%	-61.1%
executed backward branches	+9.6%	+6.0%
taken backward branches	-9.2%	-21.8%
executed unconditional branches	-66.6%	-36.3%
executed instructions	-1.2%	-0.7%
total branches	-7.3%	-2.2%
taken branches	-69.8%	-44.3%
non-taken conditional branches	+60.0%	+13.7%
taken conditional branches	-70.6%	-46.6%

2) GCC Setup: For our GCC evaluation, we used version 8.2.0. First, GCC was built using the default build process. The result of this bootstrap build was our baseline. Second, we built a PGO version using the following configuration:

```
--enable-linker-build-id --enable-bootstrap
--enable-languages=c,c++ --with-gnu-as --with-gnu-ld
--disable-multilib
```

Afterward, `make profiledbootstrap` was used to generate our PGO version of GCC.

Since BOLT is incompatible with GCC function splitting, we had to repeat the builds passing `BOOT_CFLAGS='-O2 -g -fno-reorder-blocks-and-partition'` to the make command. The resulting compiler, ready to be BOLTed, was used to build GCC again (our training input), this time without the bootstrap. The profile was then recorded and converted using `perf2bolt` to YAML format, and the `cc1plus` binary was optimized using BOLT with the same options used for Clang and later copied over to GCC's installation directory.

All four different types of GCC compilers, two without BOLT and two with BOLT, were later used to build the Clang compiler using the default configuration.

3) Experimental Results: Figures 6 and 7 show the experimental results for Clang and GCC, respectively. We observed a significant improvement on both compilers by using BOLT. On top of Clang with LTO and PGO, BOLT provided a 15.0% speedup when doing a full build of Clang. On top of GCC with PGO, BOLT provided a 7.45% speedup when doing a full build of Clang.

Table II shows some statistics reported by BOLT as it optimizes the Clang binaries for the baseline and with PGO+LTO applied. These statistics are based on the input profile data. Even when applied on top of PGO+LTO, BOLT has a very significant impact in many of these metrics, particularly the ones that affect code locality. For example, we see that BOLT reduces the number of taken branches by 44.3% over PGO+LTO (69.8% over the baseline), which significantly improves I-cache locality.

C. Breakdown of Performance Improvements

This section evaluates the impact of various BOLT optimizations on two workloads: HHVM, representing data-center workloads, and Clang, representing open-source compilers.

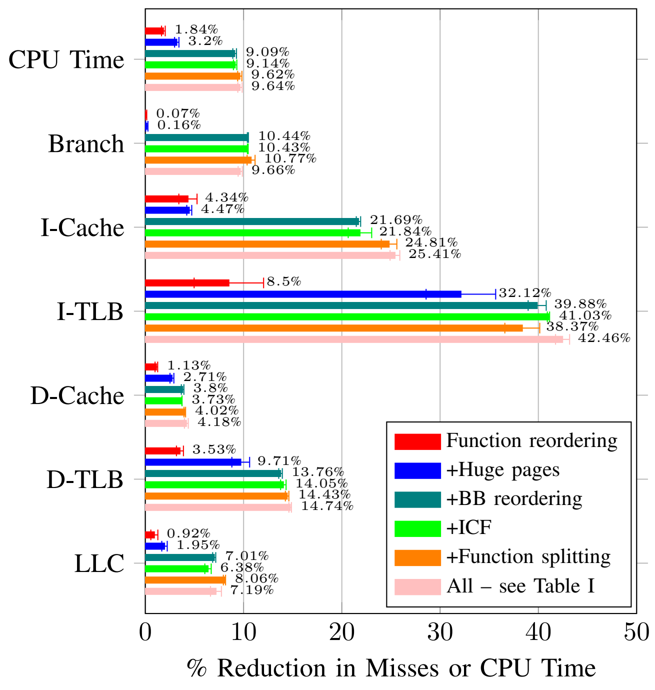

Fig. 8. Breakdown of improvements on different metrics for HHVM (higher is better). *ICF* refers to *identical code folding*. Optimizations are always added on top of the previous bar, i.e., *+BB reordering* has *function reordering*, *huge pages* and *basic block reordering* all turned on.

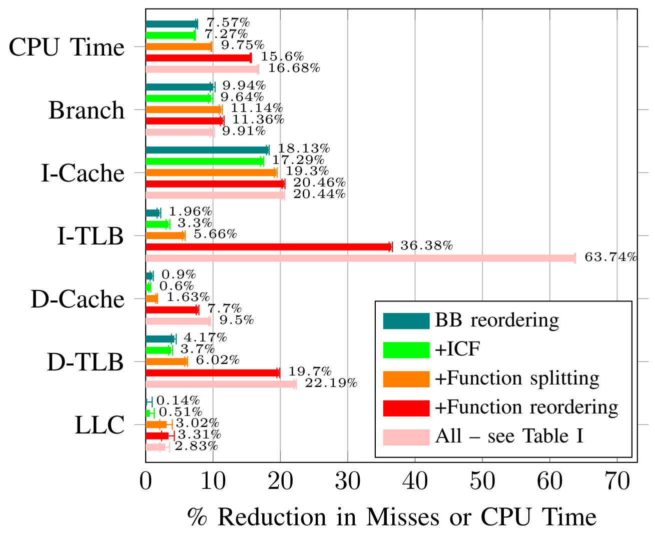

Fig. 9. Breakdown of improvements on different metrics for Clang (higher is better). `ICF` refers to *identical code folding*. Optimizations are always added on top of the previous bar, i.e., *+ICF* includes *BB reordering* and *ICF* both turned on.

For the baseline, this evaluation compares against an HHVM binary built with LTO and a Clang binary built with PGO and LTO.

Figure 8 shows the impact of various BOLT optimizations on key architectural metrics for HHVM. For this evaluation, HHVM was optimized with six different levels of BOLT optimizations, and performance results are all compared against the baseline. The first optimization level is *Function reordering*, while the next, *+Huge pages*, shows the impact of using huge pages *in addition to* the previous scenario with function reordering. We successively add optimizations to show the impact of techniques on top of each other and how improvements compound. The last bar, *All*, includes all optimizations described in Table I and is equivalent to turning on shrink wrapping, indirect call promotion, PLT call optimization and read-only data load simplification. Other passes in Table I that were not explicitly mentioned here were all turned on by default, such as fixup branches, which is mandatory.

Figure 9 shows the breakdown of the impact of BOLT optimizations for Clang. Contrary to HHVM, we do not use huge pages for Clang. The impact of huge pages was evaluated for HHVM because it is an optimization used by HHVM in combination with function reordering to further reduce I-TLB misses [3]. However, Clang does not support this optimization by default. In Figures 8 and 9, the error bars represent the standard deviation and are much smaller for Clang. HHVM experiments exhibit higher noise because of the complexity of the data-center environment in which they run.

Overall, the results in Figures 8 and 9 show that, for both HHVM and Clang, the most impactful optimizations are basic-block and function reordering. For Clang, we also observe a large reduction in I-TLB misses when turning on all optimizations. This reduction is a consequence of using PLT call optimization, which is only enabled in *All*.

D. Analysis of Suboptimal Compiler Code Layout

Using BOLT's `-report-bad-layout` option, we inspected Clang's binary built with PGO+LTO to identify frequently executed functions that contain cold basic blocks interleaved with hot ones. Combined with options `-print-debug-info` and `-update-debug-sections`, this allowed us to trace the source of such blocks. Using this methodology, we analyzed occurrences of suboptimal code layout among the hottest functions in Clang. Our analysis revealed that the majority of such cases originated from function inlining as motivated in the example in Figure 2. Figure 10 illustrates one of these functions at the binary level. This function contains three basic blocks, each one corresponding to source code from a different source file. In Figure 10, the blocks are annotated with their profile counts (`Exec Count`). The source code corresponding to block `.LFT680413` is not cold, except when inlined in this particular call site. By operating at the binary level and being guided by the profile data, BOLT can easily identify these inefficiencies and improve the code layout.

E. Heat Maps

Figure 11 shows heat maps of the instruction address space for Clang compiling itself. Figure 11a illustrates addresses fetched through I-cache for the baseline binary, while Figure 11b shows Clang optimized with PGO+LTO, and Figure 11c shows the result of applying BOLT on top of PGO+LTO.

```
Function:
clang::Redeclarable<clang::TagDecl>::DeclLink::getNext(...)
const
  Exec Count  : 1723213

.Ltmp1100284 (4 instructions, align : 1)
  Exec Count : 1635334
  Predecessors: .Ltmp1100286, .LBB087908
   0000001d:  movq  %r12, %rbx # PointerIntPair.h:152:40
   00000020:  andq  $-0x8, %rbx # PointerIntPair.h:152:40
   00000024:  testb $0x4, %r12b # PointerUnion.h:143:9
   00000028:  je  .Ltmp1100279 # ExternalASTSource.h:462:19
  Successors: .Ltmp1100279 (mispreds: 2036, count: 1635334),
              .LFT680413 (mispreds: 0, count: 0)

.LFT680413 (2 instructions, align : 1)
  Exec Count : 0
  Predecessors: .Ltmp1100284
   0000002a:  testq %rbx, %rbx # ExternalASTSource.h:462:19
   0000002d:  jne .Ltmp1100280 # ExternalASTSource.h:462:19
  Successors: .Ltmp1100280 (mispreds: 0, count: 0),
              .Ltmp1100279 (mispreds: 0, count: 0)

.Ltmp1100279 (9 instructions, align : 1)
  Exec Count : 1769771
  Predecessors: .Ltmp1100284, .LFT680414, .Ltmp1100282,
                .LFT680413
   0000002f:  movq  %rbx, %rax # Redeclarable.h:140:5
   00000032:  addq  $0x28, %rsp # Redeclarable.h:140:5
   00000036:  popq  %rbx # Redeclarable.h:140:5
   00000037:  popq  %r12 # Redeclarable.h:140:5
   00000039:  popq  %r13 # Redeclarable.h:140:5
   0000003b:  popq  %r14 # Redeclarable.h:140:5
   0000003d:  popq  %r15 # Redeclarable.h:140:5
   0000003f:  popq  %rbp # Redeclarable.h:140:5
   00000040:  retq # Redeclarable.h:140:5
```

Fig. 10. Real example of poor code layout produced by the Clang compiler (compiling itself) even with PGO. Block `.LFT680413` is cold (`Exec Count: 0`), but it is placed between two hot blocks connected by a forward taken branch.

This heat map is built as a matrix of addresses. Each line has 64 blocks and the complete graph has 64 lines. Clang binaries used for this study have 55 MB of text size, which is fully represented in the heat map. Each block represents 14,290 bytes, and the heat map shows how many times, on average, each byte of a block is fetched as indicated by profiling data. For example, the line at $Y = 0$ from $X = 0$ to $X = 63$ plots how the code is being accessed in the first 914,560 bytes of the address space. The average number of times a byte is fetched is transformed by a logarithmic function to help visualize the data, so we can easily identify even code that is executed just a few times. Completely white areas show cold code areas that were never sampled during profiling, while strong red highlights the most frequently accessed areas of instruction memory.

Figure 11c demonstrates how BOLT packs together hot code to use about 6.7 MB of space instead of the original range spanning 55 MB.

It shows how function splitting and reordering are important for moving cold basic blocks out of the hot area, and BOLT uses these techniques on the vast majority of functions in the Clang binary. The result is a tight packing of frequently executed code as shown in $Y = 1$ of Figure 11c, which greatly benefits I-cache and I-TLB.

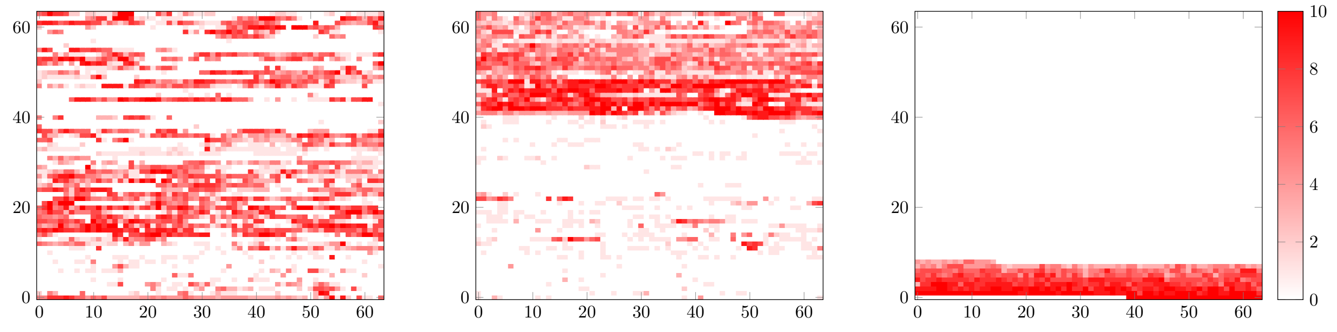

Fig. 11. Heat maps for instruction memory accesses of Clang binaries. Heat is in a log scale.

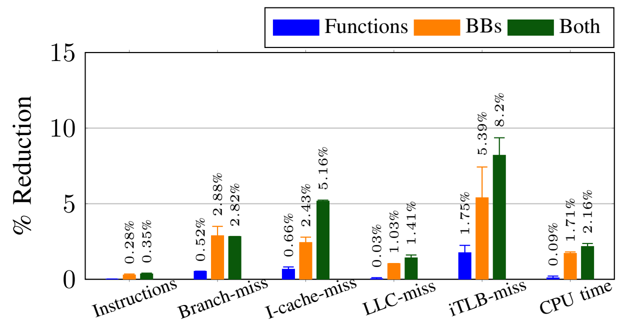

Fig. 12. Improvements on different metrics for HHVM by using LBRs (higher is better).

F. *Importance of LBR*

Not all CPUs support a hardware mechanism to collect a trace of the last branches, such as LBRs on Intel CPUs. This section compares the impact of using LBRs for BOLT profile versus relying on plain samples with no such traces.

Figure 12 summarizes our evaluation of different metrics for HHVM, in three different scenarios: reordering functions using HFSort, reordering basic blocks and applying other optimizations, and with both (all optimizations on). For example, the first data set shows that the overall reduction on instructions executed is 0.35% by having more accurate profiling enabled by LBRs. As Figure 8 shows, the total CPU time reduction by using BOLT on HHVM is 6.4%. Figure 12 shows us that using LBRs is responsible for about 2% of these improvements. Furthermore, the impact is more significant for basic block layout optimizations than it is for function reordering. The reason is that basic-block reordering requires more fine-grained profiling, at the basic-block level, which is harder to obtain without LBRs.

VII. Related Work

Binary or post-link optimizers have been extensively explored in the past. There are two different categories for binary optimization in general: static and dynamic, operating before program execution or during program execution. Post-link optimizers such as BOLT are static binary optimizers. Large platforms for prototyping and testing dynamic binary optimizations are DynamoRIO [16] for the same host or QEMU [25] for emulation. Even though it is challenging to overcome the overhead of the virtual machine with wins due to the optimizations themselves, these tools can be useful in performing dynamic binary instrumentation to analyze program execution, such as Pin [26] does, or debugging, which is the main goal of Valgrind [27].

Static binary optimizers are typically focused on low-level program optimizations, preferably using information about the precise host that will run the program. MAO [28] is an example where microarchitectural information is used to rewrite programs, although it rewrites source-level assembly and not the binary itself. Naturally, static optimizers tend to be architecture-specific. Ispike [1] is a post-link optimizer developed at Intel to optimize for the quirks of the Itanium architecture. Ispike also utilizes block layout techniques similar to BOLT, which are variations of Pettis and Hansen [14]. However, despite supporting architecture-specific passes, BOLT was built on top of LLVM [8] to enable it to be easily ported to other architectures. Ottoni and Maher [3] present an enhanced function-reordering technique based on a dynamic call graph. BOLT implements the same algorithm in one of its passes.

Profile information is most commonly used to augment the compiler to optimize code based on run-time information, such as done by AutoFDO [2]. Even though there is some expected overlap between the gains obtained by AutoFDO and BOLT, since both tools perform code layout, in this paper, we show that the gains with FDO in general (not just AutoFDO) and BOLT can be complementary and both tools can be used together to obtain maximum performance.

VIII. Conclusion

The complexity of data-center applications often results in large binaries that tend to exhibit poor CPU performance due to significant pressure on multiple important hardware structures, including caches, TLBs, and branch predictors. To tackle the challenge of improving the performance of such applications, we created a post-link optimizer, called BOLT, which is built on top of the LLVM infrastructure. The main goal of BOLT is to reorganize the applications' code to reduce the pressure that they impose on those important hardware structures. BOLT achieves this goal with a series of optimizations, with a particular focus on code layout. A key insight of this paper is that a post-link optimizer is in a privileged position to perform these optimizations based on profiling, even beyond what a compiler can achieve.

We tested our assumptions on Facebook data-center applications and obtained improvements ranging from 2.0% to 7.0%. Unlike profile-guided static compilers, BOLT does not need to retrofit profiling data back to source code, making the profile more accurate. Nevertheless, a post-link optimizer has fewer optimizations than a compiler. We show that the strengths of both strategies combine instead of purely overlapping, indicating that using both approaches leads to the highest efficiency for large, front-end bound applications. To show this, we measure the performance improvements on two open-source compilers, GCC and Clang, featuring large code bases dependent on the instruction cache performance. Overall, BOLT achieves 15.0% performance improvement for Clang on top of LTO and FDO.

Appendix

A. Abstract

The open-source workload evaluated on this paper is Clang 7 and GCC 8.2. Our goal is to demonstrate that building Clang with all compile-time and link-time optimizations, including LTO and PGO, still leaves opportunities for a post-link optimizer such as BOLT to do a better job at basic block placement and function reordering, improving workload performance.

B. Artifact Check-List (Meta-Information)

- **Algorithm:** post-link optimizer
- **Program:** Clang 7 and GCC 8.2 (will be downloaded via script)
- **Compilation:** benchmarks will be bootstrapped (Clang 7 is built with Clang 7, GCC 8.2 is built with GCC 8.2, release configuration)
- **Binary:** will be compiled for the target platform
- **Run-time environment:** CentOS Linux version 7.5 or compatible
- **Hardware:** Intel processor with LBR support, 64 GB RAM
- **Execution:** automated via Makefile (make)
- **Metrics:** execution time: milli-second
- **Output:** profile data for pgo, profile data for BOLT, optimized binaries, text file comparing execution times of these binaries when exercising a large workload. Since the optimized binaries are compilers themselves, the workload is to build a large C++ project (Clang).
- **Experiments:** clone our scripts repository, run make under `clang` subfolder to measure the impact of BOLT on Clang, run make under `gcc` subfolder to measure the impact of BOLT on GCC. Check results.txt.
- **How much disk space required (approximately)?** 120 GB
- **How much time is needed to prepare workflow (approximately)?** a few minutes to install dependencies using package manager
- **How much time is needed to complete experiments (approximately)?** 6 hours until all rules are built. This includes downloading all sources and building GCC and Clang several times, in different versions, and then rebuilding Clang 3 times per toolchain setup to measure the speed of the compilers under evaluation. This time can be longer depending on your core count.
- **Publicly available?** Yes
- **Code/data licenses (if publicly available):** BOLT, LLVM and Clang are licensed under University of Illinois Open Source License. GCC is licensed under GPLv3.

C. Description

1) How Delivered: Clone `https://github.com/facebookincubator/BOLT` and follow instructions in `paper/reproduce-bolt-cgo19/README.md`.

2) Hardware Dependencies: We require an Intel processor with LBR support for profile collection. Any implementation of the Sandy Bridge microarchitecture (2011) or later should suffice. Our artifact does require a fair amount of RAM (64 GB) due to the use of full LTO. Experiments can be completed with less RAM if parallelism is reduced during the LTO linking step, but the build will take longer. BOLT itself runs only one instance per time and should not require more than 16 GB of RAM for this workload.

3) Software Dependencies: Besides a relatively modern C++ compiler toolchain such as one based on `GNU g++ 4.8.0`, these experiments expect you to have `git`, `cmake`, `ninja-build` and `flex` available on your system.

D. Installation

```
git clone https://github.com/facebookincubator/BOLT bolt
cd bolt/paper/reproduce-bolt-cgo19
# Read README.md for dependencies, the following should
# solve them on a CentOS 7.5 system.
sudo yum install git cmake ninja flex
# Choose either clang or gcc to evaluate
cd clang
make
cat results.txt
```

E. Experiment Workflow

The experiment workflow is described in both Makefiles, one for Clang and another for GCC evaluation. They are comprised of 15 steps, all the way from downloading our benchmark (Clang and GCC), downloading BOLT itself, until measurement with Linux perf and reporting results.

F. Evaluation and Expected Result

Code reordering as performed by a post-link optimizer can have a very different impact depending on the sizes of the hardware structures of the processor frontend. The results on Figures 6 and 7 (last column, measuring the whole build) should be reproduced with these scripts if running on an ivybridge system. On newer processors with wider TLBs and caches, the performance impact can be less expressive, but still present.

G. Experiment Customization

The experiment workflow is described in both Makefiles and they were written to be easily changed: to download a different compiler version, test a different compiler configuration or test different BOLT flags.

H. Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/sysml2019.html
- http://cTuning.org/ae/submission-20180713.html
- http://cTuning.org/ae/reviewing-20180713.html
- https://www.acm.org/publications/policies/artifact-review-badging

Acknowledgments

We would like to thank Gabriel Poesia and Theodoros Kasampalis for their work on BOLT during their internships at Facebook. We also thank Sergey Pupyrev and Andy Newell for their work on improving the basic block layout algorithms used by BOLT. Finally, we thank Joseph Devietti and the anonymous reviewers for their feedback on earlier drafts of this paper.

References

[1] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney, "Ispike: a post-link optimizer for the Intel Itanium architecture," in *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE, 2004, pp. 15–26.

[2] D. Chen, D. X. Li, and T. Moseley, "AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2016, pp. 12–23.

[3] G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," in *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE, 2017, pp. 233–244.

[4] X. D. Li, R. Ashok, and R. Hundt, "Lightweight feedback-directed cross-module optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2010, pp. 53–61.

[5] R. Cohn, D. Goodwin, and P. G. Lowney, "Optimizing Alpha executables on Windows NT with Spike," *Digital Technical Journal*, vol. 9, no. 4, pp. 3–20, 1997.

[6] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen, "Instrumentation and optimization of Win32/Intel executables using Etch," in *Proceedings of the USENIX Windows NT Workshop*, vol. 1997, 1997, pp. 1–8.

[7] E. A. Henis, G. Haber, M. Klausner, and A. Warshavsky, "Feedback based post-link optimization for large subsystems," in *Proceedings of the 2^{nd} Workshop on Feedback Directed Optimization*, 1999, pp. 13–20.

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[9] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2003, pp. 15–24.

[10] U. Hölzle and D. Ungar, "Optimizing dynamically-dispatched calls with run-time type feedback," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1994, pp. 326–336.

[11] G. Ottoni, "HHVM JIT: A profile-guided, region-based compiler for PHP and Hack," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 151–165.

[12] D. Chen, N. Vachharajani, R. Hundt, X. Li, S. Eranian, W. Chen, and W. Zheng, "Taming hardware event samples for precise and versatile feedback directed optimizations," *IEEE Transactions on Computers*, vol. 62, no. 2, pp. 376–389, 2013.

[13] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, May 2011, no. 325384-039US.

[14] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*. ACM, 1990, pp. 16–27.

[15] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2000, pp. 1–12.

[16] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE, 2003, pp. 265–275.

[17] C. Wang, S. Hu, H.-s. Kim, S. R. Nair, M. Breternitz, Z. Ying, and Y. Wu, "StarDBT: an efficient multi-platform dynamic binary translation system," in *Proceedings of the Asia-Pacific Conference on Advances in Computer Systems Architecture*. Springer, 2007, pp. 4–15.

[18] DWARF Debugging Standards Committee, *DWARF Debugging Information Format version 5*, Feb 2017.

[19] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi, "The Hiphop Virtual Machine," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 777–790.

[20] R. Levin, "Complementing incomplete edge profile by applying minimum cost circulation algorithms," Master's thesis, University of Haifa, August 2007.

[21] D. Novillo, "SamplePGO: The power of profile guided optimizations without the usability burden," in *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*. IEEE Press, 2014, pp. 22–28.

[22] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "Tao: Facebook's distributed data store for the social graph," in *Proceedings of the USENIX*

Conference on Annual Technical Conference, 2013, pp. 49–60.

[23] Proxygen Team, “Proxygen: Facebook’s C++ HTTP libraries,” Web site: https://github.com/facebook/proxygen, 2017.

[24] LLVM Community, “The LLVM open-source code repositories,” Web site: http://llvm.org/releases, 2018.

[25] F. Bellard, “QEMU, a fast and portable dynamic translator,” in *USENIX Annual Technical Conference*, 2005.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2005, pp. 190–200.

[27] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2007, pp. 89–100.

[28] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani, “MAO – an extensible micro-architectural optimizer,” in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2011, pp. 1–10.