Lightning BOLT: Powerful, Fast, and Scalable Binary Optimization

Maksim Panchenko
Facebook, Inc.
Menlo Park, CA, USA
maks@fb.com

Rafael Auler
Facebook, Inc.
Menlo Park, CA, USA
rafaelauler@fb.com

Laith Sakka
Facebook, Inc.
Seattle, WA, USA
lsakka@fb.com

Guilherme Ottoni
Facebook, Inc.
Menlo Park, CA, USA
ottoni@fb.com

Abstract

Profile-guided binary optimization has proved to be an important technology to achieve peak performance, particularly for large-scale binaries that are typical for data-center applications. By applying the profile data at the same representation where sampling-based profiling is collected, binary optimizers can provide double-digit speedups over binaries compiled with profile-guided optimizations using similarly collected profile data. The main blocker for adoption of binary optimizers in practice is the overhead that they add to the already long and demanding build pipelines used for producing highly optimized binaries, which already include aggressive compiler optimizations guided by profile data and also link-time optimizations. This paper addresses the overheads of binary optimizers in the context of BOLT, a modern and powerful open-source binary optimizer. More specifically, this paper describes Lightning BOLT, which is an improved version of the BOLT binary optimizer that drastically reduces BOLT's processing time and memory requirements, while preserving BOLT's effectiveness in improving the final binary's performance. Using a set of real-world data-center and open-source applications, we show that Lightning BOLT speeds up BOLT's processing by an average of 4.71× and reduces BOLT's memory consumption by 70.5% on average. Furthermore, Lightning BOLT also provides an adjustable mechanism to further reduce BOLT's overheads at the cost of some lost performance for the final binary.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computing methodologies** → **Parallel algorithms**; • **General and reference** → **Performance**.

Keywords: binary optimization, code optimization, parallelization

ACM Reference Format:
Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. 2021. Lightning BOLT: Powerful, Fast, and Scalable Binary Optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC '21), March 2–3, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3446804.3446843

1 Introduction

A binary optimizer is a special flavor of a compiler that transforms an input binary into a more performant output binary. Binary optimization has proved to be a powerful approach to achieve peak performance [12, 15, 22]. Previous work has demonstrated that significant, double-digit speedups can be achieved via binary optimization on top of highly optimized binaries compiled with mature compilers like GCC [10] and Clang [14], even when these compilers are empowered with link-time optimizations (LTO) and profile-guided optimizations (PGO) [22]. In particular, binary optimizers are well positioned to perform code-layout optimizations with much greater accuracy than compilers can achieve with profile-guided optimizations. As Panchenko et al. [22] clearly identified, these opportunities arise from the compilers' intrinsic inaccuracy and limitations of mapping profile data collected at the machine-instruction level back to their intermediate representations, where optimizations are applied.

Unfortunately, the benefits of employing a binary optimizer come with a cost. Even though profile data collection can be made essentially free [24], binary optimization still incurs extra processing overheads in the binary-building

ACM ISBN 978-1-4503-8325-7/21/03.
https://doi.org/10.1145/3446804.3446843

[†]Artifact available at: `https://doi.org/10.5281/zenodo.4451492`

pipeline. These overheads, in terms of build time and memory usage, can be significant to the point that they prevent the use of binary optimizers in real production environments, despite providing significant speedups to widely deployed applications [26, 28].

To address these scalability concerns about binary optimization, this paper studies techniques to reduce the processing overheads in the context of BOLT [22], a modern, production-quality, open-source binary optimizer. Specifically, this paper shows how BOLT's overheads can be greatly reduced by employing two techniques: parallel processing and selective optimizations.

Parallel compilers have been studied and employed before. However, they require significant engineering effort. In general, this effort is only justified in dynamic compilation systems, where the benefits of quickly producing optimized code are bigger due to less time spent executing interpreted or less optimized code [5, 17, 18, 21]. On the static compilation domain, mainstream compilers are still sequential applications due to the engineering challenges in parallelizing them [3, 4]. Instead, build systems leverage process-level parallelism by compiling independent files concurrently [2]. However, binary optimizers, by processing a single linked binary, do not have the luxury of leveraging such parallelism. This paper describes our approach to parallelizing BOLT.

The second technique we employ to reduce the overhead of binary optimization is to selectively apply optimizations. BOLT's original design processed all the functions in the input binary. In this paper, we demonstrate how that overhead can be significantly reduced by restricting BOLT's optimizations to only a portion of the binary. Although straightforward at the high level, this approach requires careful engineering because even the portions of the binary that are not optimized still require patching to account for the optimized portions of the binary. This paper also describes how Lightning BOLT leverages this technique to increase BOLT's robustness and applicability, allowing processing of binaries even when they cannot be correctly disassembled.

By combining parallel processing and selective optimizations, this paper demonstrates how Lightning BOLT reduces BOLT's processing time and memory overheads by 78.8% and 70.5%, respectively, when optimizing real-world data-center and open-source workloads. Overall, this paper makes the following contributions:

1. It describes how parallel processing and selective optimization can be applied on a state-of-the-art binary optimizer to drastically reduce its processing overheads.
2. It provides extensive evaluation demonstrating the impact of the studied techniques when optimizing real large-scale data-center workloads.
3. It points to future directions for further reducing the overheads of binary optimizers.

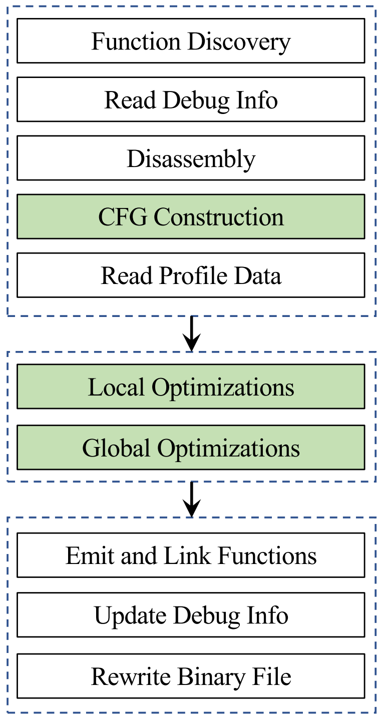

Figure 1. Overview of BOLT pipeline. Highlighted are the steps parallelized in Lightning BOLT.

The rest of this paper is organized as follows. Section 2 reviews the architecture of binary optimizers in general and BOLT in particular. After that, Section 3 and Section 4 respectively describe how we have enhanced BOLT with parallel processing and selective optimizations. Section 5 then presents the results of our extensive evaluation demonstrating the impact of this work on large-scale applications. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2 Background

This section gives an overview of the architecture of the BOLT binary optimizer [22]. Although we focus on BOLT here, this general architecture is common to other binary optimizers [7, 12, 15, 25].

Figure 1 illustrates a block diagram of BOLT's binary rewriting pipeline. BOLT starts by identifying all the functions in the binary by leveraging ELF's symbol table. In case the binary was compiled with debug information, BOLT then reads this information. After that, BOLT proceeds to disassemble the functions and to create an in-memory representation of the functions using a control-flow graph (CFG) with basic blocks containing sequences of machine instructions. This is the single internal representation of the binary code in which BOLT's optimizations are applied. Next, BOLT

Table 1. BOLT's optimization pipeline.

Optimization	Description	Local	PGO
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	Replace loads from read-only data section with moves of immediate values	✓	
6. icf	Identical code folding (second pass)		
7. plt	Remove indirection from PLT calls	✓	
8. reorder-bbs	Reorder basic blocks and split hot/cold blocks into separate sections	✓	✓
9. peepholes	Simple peephole optimizations (second pass)	✓	
10. uce	Unreachable code elimination	✓	
11. fixup-branches	Fix basic block terminator instructions to match the CFG and the current layout	✓	
12. reorder-functions	Reorder functions to improve locality		✓
13. sctc	Simplify conditional tail calls	✓	
14. align-functions	Assign alignment values to functions	✓	
15. frame-opts	Remove unnecessary caller-saved register spilling		
16. shrink-wrapping	Moves callee-saved register spills closer to where they are needed if profitable		✓

reads the profile data used to drive its optimizations and annotates it onto the CFG representation. After this, BOLT has all the representation of the code annotated with profile data in memory and can apply its various optimizations. Once the optimizations are finished, BOLT finally emits the optimized version of the code and rewrites the final binary. If debug information was available in the binary, BOLT will also update it to reflect the effect of the optimizations.

We now briefly describe the optimizations that BOLT applies to the input binary to improve its performance. Table 1 lists the optimizations currently implemented in BOLT in the order they are applied. As Panchenko et al. [22] demonstrate, the most powerful of these optimizations are the code layout optimizations, namely basic-block and function reordering. For these optimizations, BOLT implements state-of-the-art algorithms, including ExtTSP [16] for basic-block reordering and C^3 [20] for function reordering. In the third column, Table 1 notes which optimizations are local to each function, i.e. they can be applied to each function independently. As the table shows, most of the optimizations in BOLT are function local. This observation allows for a general approach to parallelize BOLT's optimizations by leveraging parallelism available at the function granularity. Section 3 demonstrates how Lightning BOLT leverages these parallelization opportunities. Finally, the fourth column in Table 1 notes which optimizations are guided by profile information. While BOLT only applies these optimizations for functions with profile data, the other optimizations are applied to all functions in the binary indistinctively. This provides an opportunity to reduce BOLT's processing overheads without degrading performance of the final binary. Section 4 discusses how Lightning BOLT leverages this opportunity.

3 Parallel Processing

Compilation of large projects in parallel is mandatory in modern systems. However, parallelization in compilers is usually achieved at the build-system level, invoking the compiler multiple times in separate processes, as it is easier to track dependencies across artifacts specified in build files. Parallelizing the compiler's internals, on the other hand, is a daunting task that involves updating and understanding legacy code, its dependencies and how it interacts with data, and making sure that all global structures are accessed safely. There is enough parallelism at the build system level to well utilize the system resources in a balanced way, making the parallelization at the compiler level avoidable, with the exception of cases where the user does not break up the project into separate compilation units or if monolithic LTO [13] is used.

When it comes to binary optimizers, parallelization at the same level used by most compilers is not an option since binary optimizers consume a single large binary. Binary optimizers lack the human help done at the source code level to break up the project into separate translation units that can be independently processed, and their scalability is further threatened by the hefty sizes of binaries used in data centers. This warrants the additional work and maintenance overhead required to parallelize binary optimizers with threads.

Lightning BOLT focuses on parallelizing the most expensive steps of a binary optimizer, which are highlighted in Figure 1. Here, each pass is parallelized individually with a synchronization point before the start of the next pass. This design is a good compromise on efficiency and maintainability, since the state of the intermediate representation of the whole program is well defined between two passes, making it easy to debug issues. This is also efficient as long as each pass has enough work to be done to load the thread pool. Not every pass is expensive, so light passes or passes that require heavy synchronization do not run in parallel. Also, since the number of input functions can be quite large for data-center binaries, there is enough work to distribute among threads and obtain meaningful speedups.

One important caveat of running passes in parallel is that they might result in non-deterministic output. Lightning BOLT will never generate non-deterministic output, as it can be disastrous to build systems and infrastructure that rely on build artifacts having the same contents as long as inputs do not change. If the overhead of ensuring determinism offsets the speedup brought by parallelism, we stick with the sequential version of that pass.

3.1 General Changes

This section presents Lightning BOLT's general changes that are used by parallelized passes.

Common interface to run passes in parallel. Several parallelizable loops in BOLT are built around performing the same work on each function. Passes that run local optimizations iterate on a subset of functions of interest and require, most of the time, trivial changes to run in parallel. For this class of optimizations, the interactions with global and shared variables are minimal. Yet, some more involved global passes can be broken up into subtasks that exhibit this pattern.

We wrote an API that captures the intent to run such passes in parallel with the following interface. The user specifies a work function and a predicate function. The work function will concurrently run on each `BinaryFunction` (a function, in BOLT IR's terminology) that satisfies the predicate function. The API implementation manages a set of threads through an LLVM thread pool class that is kept alive throughout BOLT's lifetime and is re-used by this API each time a pass invokes it. Figure 2 shows an example of a pass (reorder basic blocks) implemented using this idea.

```
auto workLambda = [&](BinaryFunction &F) {
  modifyFunctionLayout(F);
  if (F.hasLayoutChanged()) {
    ++ModifiedFuncCount;  // atomic variable
  }
};

auto applyPredicate = [&](const BinaryFunction &F) {
  return shouldOptimize(F);
};

runOnEachFunction(
  Context, SchedulingPolicy::SP_BB_LINEAR,
  workLambda, applyPredicate, "ReorderBasicBlocks");
```

Figure 2. A pass implementation using the interface for parallel execution.

In order to reduce the scheduling overhead, multiple `BinaryFunction` objects can be batched together to be processed by one thread. This is ideal for the common case where the number of functions is high, but the pass finishes each function quite fast. While this works best in a scenario where the workload of each batch is roughly the same, the processing time to finish them is not always proportional to the number of functions within a block of work as function sizes observe a high variance for typical inputs. We mitigate this with the option *scheduling policy* that is used to model the runtime of the work function when applied to `BinaryFunction` objects, which we rely on to better balance work across threads. The policy estimates the relationship between pass runtime and size of input function in the number of instructions (`SP_INST` prefix) or basic blocks (`SP_BB` prefix). In the example of Figure 2, the runtime of the *reorder basic blocks* pass is linearly proportional to the number of basic blocks in the function, which justifies the use of `SchedulingPolicy::SP_BB_LINEAR` as its policy. The implementation of the API then honors this request by dividing the list of all input functions into blocks that sum up an equal number of basic blocks and sends these to thread workers. Other scheduling policies include `SP_BB_QUADRATIC`, `SP_INST_LINEAR` and `SP_INST_QUADRATIC`. The granularity of the scheduled blocks can be changed using an optional parameter *TasksPerThread*, which can be used to control the total number of tasks created and hence the size of each task.

Allocations. The two most common patterns of accessing shared resources in passes are (1) using LLVM's `MCContext` object to create new labels that manage local references and (2) performing heap allocations using a memory pool shared globally. The former is itself an IR object allocator used by LLVM that we cannot easily change without impacting LLVM core libraries. Accessing it is therefore handled with a shared lock. The latter, on the other hand, is in BOLT's control and we can tune it to account for a thread-friendly allocation. For that, we used two changes: first, jemalloc [9] was chosen as the default memory manager because it showed a better performance than glibc's malloc, making not only the sequential requests faster but also boosting parallelization speedups. Second, we created a pool of allocators to hold pass annotations to BOLT's IR instructions.

Temporary pass-generated information in BOLT is stored as annotations that are attached to the instructions as extra operands. These annotations are allocated using a special bump allocator that is not thread-safe and is heavily used in passes that rely on data flow analysis, such as shrink wrapping. Using locks to manage accesses to this bump allocator adds an unacceptable overhead. To work around this issue and allow lock-free annotation allocation, we rely on a set of multiple annotation allocators that are assigned to each thread. This is supported via a variant of the parallel pass API mentioned earlier, which provides the work function with a bump allocator that is guaranteed not to be used by any other workers at the same time.

3.2 Parallelized Passes

This section presents the passes that were parallelized in Lightning BOLT in the order they appear in the processing pipeline.

CFG construction. After a binary function is disassembled, a control-flow graph construction pass takes place by partitioning instructions into basic blocks and creating edges. During this construction, synchronization on LLVM's `MCContext` is required to allow creation of labels, and the bump allocators are used to store instruction-level information using our distributed allocation scheme.

Local optimizations. Local optimization passes were parallelized with additional usage of locks. The following passes were converted to use the parallel API:

- Aligner pass;
- Reorder basic blocks;
- Eliminate unreachable blocks.

Identical code folding. Identical code folding (ICF) is a global optimization that scans all functions and eliminates identical functions by preserving only one copy for each equivalence class. The pass runs in two stages. First, input functions are clustered together into buckets of functions that share a preliminary level of similarity. Identical functions must satisfy this similarity property, but non-identical functions may appear similar at this stage. The check for functions' similarity is done through a quick linear scan across the function space, looking at contents of each function at a shallow level that, among other information, ignores the destination of functions calls. Clustering is then accomplished by computing a hash of these contents for each function. The runtime of this first stage is dominated by the hashing process, which is proportional to the total size of the binary, while the actual clustering is proportional to the number of functions. This was parallelized in a lock-free fashion by first pre-computing hashes in parallel and then clustering sequentially.

The second stage of ICF runs on each cluster of similar functions and proceeds by performing a more precise comparison to narrow down on *identical* functions, folding them until convergence is reached. This loop runs in parallel where each thread handles separate clusters of similar functions. When a set of functions is found to be identical, the pass needs to update the global state by deleting some functions and updating their callers to reference the elected single copy that will remain. Locks are used to manage this state update.

Frame optimizations and shrink wrapping. These are good examples of a set of expensive global optimizations that were parallelized. They are responsible for removing or changing the position of loads and stores of both callee- and caller-saved registers. The former is accomplished by shrink wrapping, while the latter is done by frame optimizations. Both of these optimizations start with a frame analysis pass that is performed at a whole-program level, building a call graph and gathering information on how the entire program uses registers and stack slots. During this step, an expensive substep is the stack-pointer analysis, which is a dataflow analysis that reconstructs the value of the stack pointer at each point of the program. This dataflow step was rearranged to be precomputed earlier and cached rather than being calculated on-demand, allowing it to run in parallel at the function granularity while its cached results can be used later during the sequential analysis step. At the end of these optimizations, many temporary annotations are cleared in addition to the cached stack-pointer analysis. Unfortunately, removing annotations has a non-trivial runtime cost as well but it is a necessary step to save memory. This step was delayed to the very end, to avoid being interleaved with sequential steps, and then parallelized.

Shrink wrapping in particular performs multiple local dataflow analyses, and each dataflow analysis uses the bump allocator extensively for IR annotation. With the help of independent bump allocators, shrink wrapping was converted to run with function-level parallelism using our regular API described earlier.

3.3 Future Opportunities

There are additional parallelization opportunities in BOLT that have not been explored yet. Table 4 in Section 5 shows the runtime distribution of BOLT doing both a sequential and a threaded run while processing Clang as an input binary. Two expensive and currently sequential phases are *disassembly* and *emit and link*, taking about half of the total processing time. Even though parallelizing these passes could have a significant impact, they also introduce additional challenges in reproducibility and in dealing with LLVM code that is outside the scope of BOLT. Emitting and linking assembly code in-memory using LLVM's `MCStreamer` is expensive, but it is almost entirely handled by LLVM core libraries. The emission and linking phase iterates on each function at a time, emitting symbols, instructions and data while LLVM's in-memory assembly builds a picture of the layout of the final program.

Even though function disassembly could conceptually be done in parallel, there are complications that come from functions that reference code inside other functions, requiring synchronization among threads to agree on established entry points for a function.

4 Selective Optimizations

This section describes how Lightning BOLT extends BOLT with selective optimizations to both reduce BOLT's processing overheads and improve its robustness and applicability.

As described in Section 2, the original BOLT design applied its suite of optimizations to all the functions in the binary. The only exception to this were function-local profile-guided optimizations (c.f. Table 1), which were only applied to the functions with profile data. This approach used by BOLT of aggressively optimizing the input as much as possible is common to static compilers and optimizers. The reasoning behind this approach is that compilers usually lack the knowledge about what portions of the code will execute more often in practice, thus fully optimizing the entire application ensures that the hot portions of the application are fully optimized.

However, a profile-guided binary optimizer can do better than that by not applying its optimizations to the entire binary, while still maintaining the peak performance of the final binary. Contrary to static compilers, a profile-guided binary optimizer is well positioned to selectively apply optimizations, for two main reasons. First, a profile-guided binary optimizer can leverage its input profile data to decide what portions of the binary are worth optimizing. In other words, the same profile data that it uses to decide *how* to optimize each function can also be used to decide *what* functions are worth optimizing. We note that this approach can also be used by profile-guided static compilers. However, the same difficulty in mapping profile data back to a compiler's intermediate representation that prevent compilers from fully benefiting from profile data, which was observed by Panchenko et al. [22], can also prevent the compiler from accurately determining what functions are worth optimizing. For this reason, static compilers typically aggressively optimize the entire binary, even when profile data is available. The second advantage that a binary optimizer has over a static compiler to leverage selective optimizations is the fact that its input and output languages are the same: both are binary machine code. Because of this, a binary optimizer can avoid the overhead of processing a given function almost entirely if it decides not to process that function. The input machine code provides a natural and reasonably well optimized fallback in such cases.

Lightning BOLT applies the insight described above to greatly reduce BOLT's processing time and memory overheads in two different modes, one that maximizes performance of the final binary and one that trades some performance for further overhead reductions. Lightning BOLT's first mode ensures that the performance of the final binary is the same as the one obtained by BOLT by skipping all optimizations for functions with absolutely no profile data. As long as the profile data is representative of the actual binary usage in the field, this mode results in no performance degradation compared to BOLT. The second Lightning BOLT mode further reduces the processing overheads by also skipping some functions with profile data. In this mode, BOLT sorts the functions in decreasing order of CPU time assigned to them according to its input profile, and only the top functions are optimized. There are two options available to control the amount of functions that are processed, one specifying the maximum absolute number of functions to be processed and another specifying the percentage of functions with profile to be processed.

As mentioned above, by not optimizing a function, most of the overhead of processing it can be avoided, but not all. Note that unoptimized functions may reference optimized functions, which may be moved in memory (e.g. due to function reordering or identical code folding). Because of that, unoptimized functions still need to be loaded in memory and disassembled for minimal processing. This processing is fairly fast compared to full disassembly with CFG reconstruction, as it only consists of scanning the function for references to optimized functions and updating them. The references are updated in-place, which allows reducing processing time and memory by skipping the code emission phase for such functions.

Since the references that BOLT scans and updates are located in the cold part of the code that is rarely or never executed, it is natural to assume that updating them will not affect the application's performance. In practice, we have seen this assumption be broken for applications that contain startup code that initializes pointers to frequently executed callback functions. For example, this can happen when the application's functionality is controlled via a configuration file or command-line option. The initialization code is only run once and thus unlikely to become registered with sample-based profiling. If the reference in the initialization function is not updated, then the runtime will invoke the original function via the original function pointer. Thus, without knowing the specifics of the application, Lightning BOLT by default scans and patches all function references.

Note that function references discovered during the scanning phase are often available to BOLT via relocations saved in the binary by the linker. However, there are cases where the necessary relocations are absent in the object files and thus not exposed to the linker. Examples of these include some program-counter-relative relocations used for call instructions. When a call instruction destination is present in the same object file as the call instruction and its binding rules are strict, the compiler may decide to process the relocation internally and omit it from the object file. In such case, neither the linker nor BOLT may know if the function has a missing relocation and hence BOLT has to rely on its scanning algorithm for all functions in the input binary. If BOLT had the information about the original object file section boundaries, it could use that to entirely skip such sections if they contained no functions with profile data, because references to functions external to these sections *must* be visible to the linker. Unfortunately, the information about input sections is not preserved by the linker.

Although the primary motivation for selective optimizations in Lightning BOLT was to reduce its processing time and memory overheads, this approach also helped to increase Lightning BOLT's robustness and applicability compared to BOLT. BOLT sometimes is unable to optimize a binary, for example, due to data embedded in the middle of the code[1] or the use of new or exotic machine instructions that are not supported by the disassembler yet. In such cases, BOLT fails to disassemble the input binary and exits with an error message pointing to the problematic piece of code. In contrast, Lightning BOLT is able to leverage its ability to skip functions in the binary to successfully optimize the binary even in cases it cannot properly disassemble some functions.

[1]For ARM processors, for which data is commonly embedded in the code, the ABI requires the programs to clearly mark all data-in-code in the symbol table, and BOLT relies on this information to correctly process ARM binaries.

Naturally, Lightning BOLT can skip processing those functions, similar to how it skips functions regarded not worth optimizing. However, a complication arises because these functions still need to be scanned to update their references to optimized functions, but this is not possible if the function cannot be correctly disassembled. Lightning BOLT solves this issue in the following way. For functions that are optimized, Lightning BOLT still preserves their original code and patches entries with an unconditional jump to their new, optimized location. This solution not only guarantees that references in functions that cannot be correctly disassembled and scanned still properly work, but it also ensures that, in case such a reference is used to call the unoptimized version of a function, the execution goes back to the optimized portion of the binary. However, if the body of the function is comparable to cache line size, BOLT can skip patching its entries and keep two copies of the function. The reasoning behind this logic is that, by the time the CPU has reached the original entry point, it has already fetched most of the function code from memory, and the benefit of executing the optimized version will likely be lost completely or become insignificant. When Lightning BOLT decides to keep both copies of the function, skipping the patching process, it has to duplicate the metadata for the function.

5 Evaluation

This section evaluates the impact of Lightning BOLT versus BOLT, breaking down the impact of parallelization and selective optimizations. Furthermore, a study of Lightning BOLT's mechanism to trade performance of the output binary for extra savings is also presented.

5.1 Workloads and Experimental Setup

We evaluated Lightning BOLT on both open-source workloads and very large binaries deployed at Facebook. As open-source benchmarks, we used GCC 10 bootstrapped (built with itself) and Clang 11 bootstrapped. The profiles for these workloads were collected while using them to build Clang 11. Lightning BOLT has already been upstreamed into the BOLT project, so we built it using a recent BOLT rev (fab6ae4a[2]) with Clang 11 and linked against jemalloc. Our experiments were performed on an Intel Broadwell machine with 56 logical cores and 256 GB of RAM. All experiments that investigate parallelism use 56 threads.

For data-center workloads, we used five binaries that are widely deployed across Facebook's data centers. The first is Proxygen, a load balancer used to route requests coming from outside the data center [23]. The second is Multifeed, used to select what is shown in the Facebook News Feed. The third is AdIndexer, used to select ads. The fourth is TAO, a data caching service [6]. The last one is HHVM, the Hack virtual machine that serves web requests for a large part of Facebook's products [1]. These binaries were selected because they are some of the largest ones built for Facebook's server fleet and better suited to study the scalability of BOLT with respect to the size of the input. The open-source counterparts, GCC and Clang, were selected because they are large enough to suffer from problems of high i-cache misses and branch mispredictions and therefore benefit from BOLT. Table 2 shows the static code size and other statistics for this set of applications.

[2]Available at https://github.com/facebookincubator/BOLT/commit/fab6ae4a.

Table 2. Statistics of input binaries used for evaluation.

Binary	Code	Hot code	Functions	Hot functions
Proxygen	222 MB	9 MB	435,281	12,469
Multifeed	570 MB	34 MB	633,943	31,628
AdIndexer	573 MB	24 MB	673,854	21,049
TAO	163 MB	6 MB	257,728	4,860
HHVM	107 MB	21 MB	194,763	16,574
GCC	18 MB	8 MB	39,965	10,629
Clang	49 MB	16 MB	79,021	16,737

5.2 Processing Time and Memory

The graphs in Figure 3 and Figure 4 present speedups in wall time and reductions in peak memory consumption when running Lightning BOLT with parallelization, with selective optimizations and with both. The BOLT flags used for this evaluation were the following:

```
-reorder-blocks=cache+ -reorder-functions=hfsort
 -split-functions=3 -split-all-cold -icf=1
 -split-eh -frame-opt=hot -no-threads -lite=0
```

For the *Parallelization* data set, we removed `-no-threads`; for *Selective Optimizations*, we removed `-lite=0`; and for *Both*, we removed both options. For example, for Proxygen, Figure 3 shows that, compared to BOLT, Lightning BOLT results in a 1.56× speedup using parallelization, 6.99× using selective optimizations, and a 8.51× speedup using both techniques. Also for Proxygen, Figure 4 shows that there is a small memory overhead of 1.2% in resident set size (RSS) in order to support execution with multiple threads, but a large 81.1% reduction in memory RSS once selective optimizations are turned on. Overall, we observe an average memory reduction of 70.5% and wall-time speedup of 4.71× for Lightning BOLT when compared with BOLT across our set of applications.

Table 2 helps to explain the gains observed from selective optimizations. For example, while Proxygen contains 435,281 functions that would need to be fully disassembled and stored in memory, selective optimizations will only process the 12,469 hot functions. This is a difference of 34.9× in the number of functions processed, which translates to Lightning BOLT being 6.99× faster while using about 1/5 of peak memory. The table also shows the size of all code in the binary and the size of hot code, meaning how many bytes of code are processed by BOLT versus Lightning BOLT.

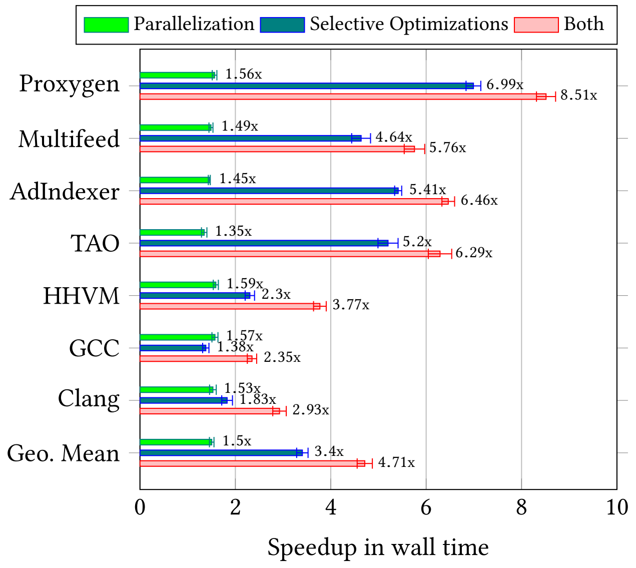

Figure 3. Wall time speedups when separately evaluating the Lightning BOLT techniques of *Parallelization*, *Selective Optimizations*, and *Both*. Baseline is regular BOLT.

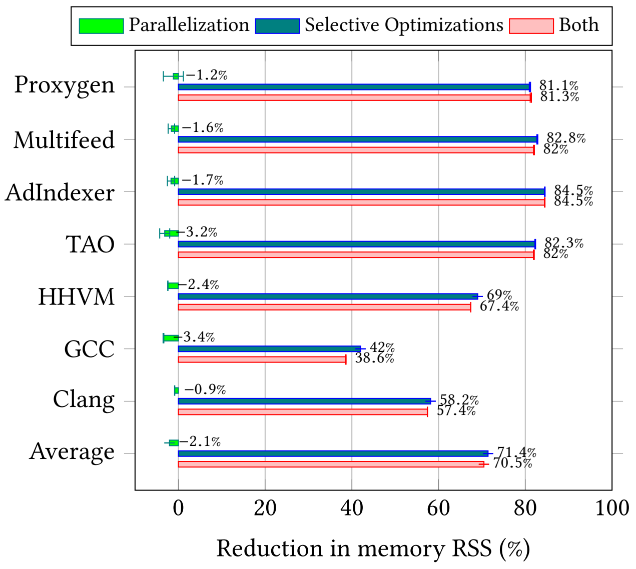

Figure 4. Memory reduction when separately evaluating the Lightning BOLT techniques of *Parallelization*, *Selective Optimizations*, and *Both*. Bars are calculated by 1 minus the ratio of RSS of a specific mode over baseline BOLT.

For example, even though HHVM has an 11.7× reduction in number of functions, the hot functions are larger than the ones that are cold, so the ratio of total code size over hot code size that is processed is 5×. Lightning BOLT can skip processing these cold functions and cold bytes, but it still has important work that cannot be skipped, such as reading the full symbol table and identifying limits of every function in the binary. Lightning BOLT also needs to scan references from cold code to hot code that got optimized and update these references as well. In Proxygen, for example, scanning references can take up to 30% of the total time. This explains why we do not necessarily observe the same reduction of compute resources with respect to the reduction in number of processed functions or bytes, although the scanning of references can be optionally disabled (see Section 5.4 for a comparison of tradeoffs).

Table 3 shows absolute numbers for average wall time and peak memory RSS when using Lightning BOLT. This data confirms that Lightning BOLT is able to process very large binaries, containing up to 500MB of code, such as AdIndexer, faster than the time required to run the compiler (thin)LTO for these builds, which is in the order of tens of minutes and up to one hour, depending on how engineers tune knobs that control optimization passes. Currently, it takes about 30 minutes for Clang's thinLTO to process AdIndexer and about 10 minutes for GCC's WHOPR to process HHVM. For comparison, the time for LLD (a linker) to process Clang is 7s with 470MB of peak RSS, although the linker is doing much less work compared to BOLT.

Table 3. Absolute overheads for Lightning BOLT, with both parallelization and selective optimizations.

Input binary	Avg. wall time	Avg. peak Memory RSS
Proxygen	72.6s	6.7 GB
Multifeed	168.3s	14.0 GB
AdIndexer	161.3s	12.9 GB
TAO	51.0s	4.6 GB
HHVM	54.7s	5.7 GB
GCC	19.5s	2.1 GB
Clang	29.8s	3.4 GB

5.3 Parallelization of Individual Passes

Figure 5 shows the speedups from parallelizing individual passes in Lightning BOLT's pipeline. The distribution of time spent in each pass for a run while optimizing Clang in sequential mode (regular BOLT) is shown in Table 4. From this table, we gather that the most expensive pass in BOLT is emit and link, which is responsible for calling LLVM's in-memory assembler to create an object file and ORC's runtime linker to link and resolve references in this object file in memory. This pass cannot be easily parallelized. The next most expensive pass is building CFGs for all functions of the input, taking 16% of the execution time of BOLT for this input, followed by ICF, taking 14%. Both of these are parallelized and their speedups are reported in Figure 5. For example, for Clang, we observe that parallelization brings a 2.2× speedup for ICF. The table also shows the distribution of time after parallelization. Passes that cannot run in parallel

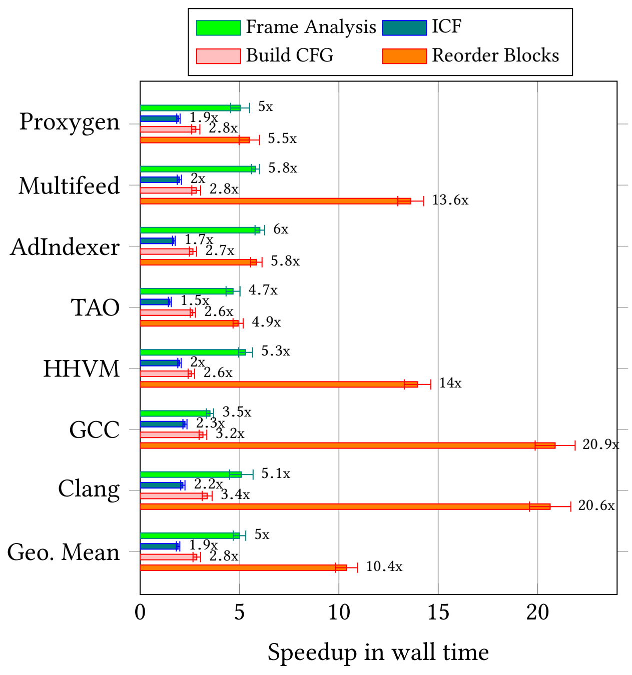

Figure 5. Wall time speedups for selected passes when evaluating parallelization alone.

Table 4. Distribution of time spent in each step of BOLT in a sequential run of 100.6s while optimizing Clang compared with Lightning BOLT with parallelization alone. Passes that did not execute for a significant portion of time are omitted.

Pass	Wall time sequential (% of total)	Wall time threads (% of total)
Discover functions†	3%	5.4%
Disassemble functions†	8%	17%
Build CFG	16%	15.5%
Identical code folding	14%	10.5%
Reorder basic blocks	5%	0.5%
Split functions	5%	2.5%
Reorder functions†	1%	1.4%
Aligner	4%	0.25%
Frame analysis & opts.	12%	9.1%
Shrink wrapping	9%	1.4%
Emit & link†	21%	36.2%

†Pass is not parallelized

take a larger percentage of the total time and become larger bottlenecks.

Reorder basic blocks is both expensive and easy to parallelize, so this pass observes bigger speedups in Figure 5. Passes that run very fast or that frequently need synchronizations observe smaller speedups. Very fast passes cannot offset the overhead of distributing the work across threads. With threads, one important resource that needs to be synchronized in BOLT is access to LLVM's MCContext, which internally runs an allocator to store IR objects. Therefore, passes that perform operations requiring instantiation by MCContext suffer higher contention.

5.4 Trading Output Binary Performance for Reduced Processing Overheads

This section evaluates Lightning BOLT's mechanism to further reduce its processing overheads by processing fewer functions in the binary, and how this affects the performance of the final binaries.

For this evaluation, we used the Clang-7 compiler as a benchmark. The version of Clang used was bootstrapped in default Release build mode. The resulting binary contained 37.53 MB of code. To measure the performance of the binary, we used a pre-processed C++ input file consisting of over 268,000 lines of code and 7.34 MB in size. The file was compiled with Clang-7 compiler using `-std=c++11 -O2 -c` options, and the runtime was measured using Linux's `time` utility recording the elapsed real time. We collected Clang-7 profile when compiling the file, and used it and the following options to optimize the compiler with BOLT:

```
-reorder-blocks=cache+ -reorder-functions=hfsort
 -split-functions=1 -split-all-cold.
```

The runtime and memory of the BOLT command was again measured with the `time` command recording elapsed real time and maximum memory RSS. The new BOLTed Clang-7 binary compiles the same input about 41% faster compared to the bootstrapped Clang-7. We then added option `-lite-threshold-pct=<N>` to the set of BOLT options. This option controls how many functions with profile data are processed. The higher the threshold, the fewer functions are processed and optimized. We gradually increased the threshold, measuring Lightning BOLT's processing time and memory and running Clang-7 benchmark for every threshold setting. The results are plotted in Figure 6. The baseline is Lightning BOLT running without any threshold, i.e. processing all functions with profile data. As the threshold increases, the relative number of functions that are processed is reduced, which is reflected on the X-axis of the graph. Lightning BOLT's runtime and memory decrease and reach just below 50% of the baseline when 95% of the functions are left unprocessed, or only 5% of functions are processed. The graph in Figure 7 shows the speedup of the resulting Clang-7 binary. The more functions get dropped from processing, the smaller the speedup is, reaching just 15% when only the top 5% of functions are optimized. The curve shows a well-defined knee around 65%. To further reduce processing time and memory, we also introduced the `-no-scan` Lightning BOLT option that disables the pass responsible for updating function references in unprocessed code. As a result, all calls from unoptimized to optimized code incur an extra jump overhead. The performance of Clang-7 binary optimized in *no-scan* mode is only about 1% worse compared to standard Lightning BOLT, i.e. when there is no optimization

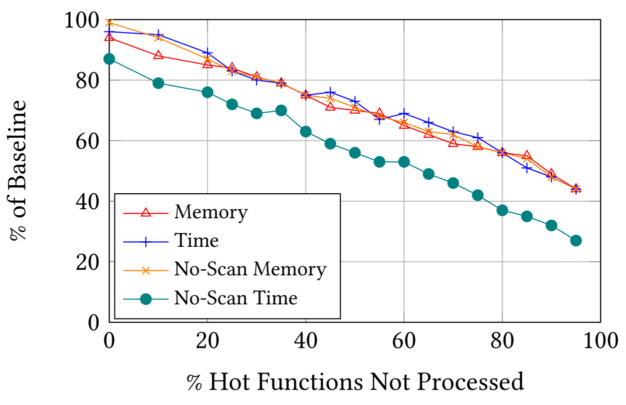

Figure 6. Improvements in Lightning BOLT runtime and memory by gradually reducing the percentage of processed functions. Colder functions are dropped first. *No-Scan* metrics measure disabling Lightning BOLT's patching of unprocessed functions to refer to the new location of optimized functions.

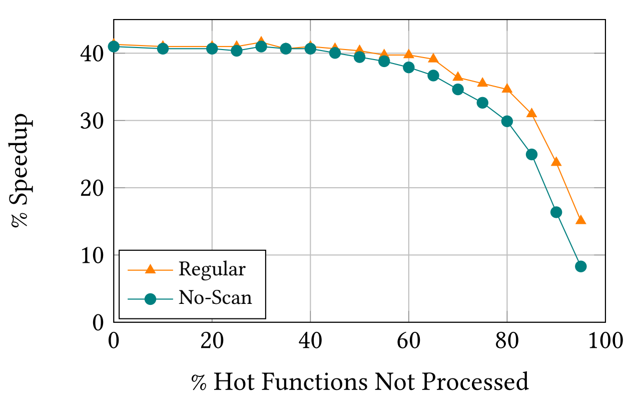

Figure 7. Input binary (Clang) speedups over a version not optimized by BOLT as we gradually reduce Lightning BOLT coverage as a percentage of functions dropped from processing. Colder functions are ignored first. *No-Scan* metrics measure Lightning BOLT disabling patching of colder functions to refer to the new location of hot and optimized functions.

threshold. As the percentage of processed functions drops, the performance of *no-scan* binaries decreases noticeably faster compared to the *scan* binaries. This trend is expected as more and more calls to optimized code are executed from unoptimized but profiled functions.

This experiment demonstrates how Lightning BOLT may run almost twice faster while still producing a binary with nearly identical performance. Trading off performance for processing speed can be important in memory-constrained environments that may prevent executing Lightning BOLT in the default mode.

6 Related Work

This section describes previous work that more closely relates to ours.

Previous work on static binary optimizers have focused on the performance of the resulting binary, but not in the performance of the binary optimizer themselves [7, 12, 15, 22, 25]. In this paper, we aimed at improving the performance of binary optimization. Our work was motivated by large-scale, data-center applications, which tend to be very large binaries and challenging to build and optimize [13, 20]. Although we have conducted our study in the context of BOLT [22], we believe that our approach, based on parallelism and selective optimizations, can also be used to improve the performance of other binary optimizers.

Earlier research work on static compilers has leveraged parallelism to speedup the compilation process, including Gross et al. [11] and Wortman et al. [29]. Despite that, mainstream compilers like GCC [10] and LLVM [14] are still single-threaded, relying purely on process-level parallelism arising from the concurrent compilation of independent source files [2]. Within the GCC project, the Parallel GCC project aimed at speeding up compilation by leveraging multiple threads [3]. This effort has resulted in modest speedups (9% using 8 cores), and it attests to the difficulties in parallelizing an application as complex as a production-quality compiler. Bernardino et al. [4] discusses in more detail some of the difficulties faced in the Parallel GCC project.

In the context of dynamic compilation, previous work have relied on parallel compilation to reduce their overhead [5, 18, 21]. In dynamic-compilation systems, the compilation overhead is extremely important because, until optimized code is produced, the execution proceeds much slower by running either unoptimized code or even an interpreter [18, 21]. In these systems, compilation typically leverages multiple independent compilation units (typically methods) to exploit parallelism, akin to the approach used in our work.

In addition to parallel compilation, our work also leverages selective compilation to reduce compilation overheads. Selective optimization is a common practice in dynamic compilers because of the importance of compilation overheads at runtime. For example, dynamic compilers typically have multiple compilation tiers, so that the code is recompiled in more optimized manners the more it executes [8, 18, 19, 21]. However, selective optimization is uncommon in static compilers, where quality of the optimized code is regarded much more important than the compilation overhead. The insight of our work is that, for a static binary optimizer, selective optimizations can effectively reduce compilation overheads without degrading the quality of the resulting binary. This advantage of binary optimizers over source-code compilers comes from the fact that a baseline version of the code is readily available, so the binary optimizer can just fall back to that. Special patching of the code that is not optimized is still performed (as discussed in Section 4), but the overhead of patching is much smaller than fully processing those functions through the main optimization pipeline.

Motivated by BOLT [22], another project called Propeller was recently started [28]. Tallam [26] reported significant performance improvements from BOLT for Google's large-scale binaries. However, they also reported significant processing time and memory overheads while applying BOLT to their large-scale binaries. Propeller's goal is to bring BOLT's optimizations to Google by leveraging their distributed build system and link-time optimization framework [13]. In order to do basic-block reordering at link time, Propeller requires new *basic-block sections*, which require recompilation of the application with special flags and linker changes, and can also bloat the binary [26]. We believe that these are more invasive changes and make Propeller less applicable than BOLT, because they are tied to a specific compiler (LLVM). Still, Propeller proposes an interesting alternative design to reduce BOLT's processing overheads. Unfortunately, we have not been able to properly optimize any interesting application using Propeller. This appears to be because Propeller is still under development and not ready for general use yet [27]. Although it will be interesting to perform direct comparisons between Propeller and Lightning BOLT at some point, our work demonstrated that Lightning BOLT significantly reduces BOLT's overheads.

7 Conclusion

This paper described Lightning BOLT, an enhanced version of the open-source, production-quality BOLT binary optimizer. Lightning BOLT drastically reduces BOLT's processing time and memory overheads, while keeping the same performance benefits provided by BOLT. To achieve this, Lightning BOLT extends the BOLT binary optimizer with parallel processing and selective optimizations. Our evaluation, using a collection of data-center and open-source applications, demonstrates that Lightning BOLT provides an average 4.71× speedup over BOLT, while also reducing memory consumption by an average of 70.5%. Besides that, Lightning BOLT also provides a controlled mechanism to trade some performance of the optimized binary for further reductions in processing overhead. Our evaluation showed that this mechanism may further double the performance of Lightning BOLT with minimal impact on the performance of the resulting binary. Finally, in addition to the drastic reductions in processing overheads, Lightning BOLT also increases BOLT's robustness and applicability, by enabling the optimization of binaries even when they contain code that cannot be correctly disassembled.

Acknowledgments

We would like to thank Amir Ayupov, Alexander Shaposhnikov and Xun Li for their work on improving BOLT and fixing bugs. We also would like to thank Gabriel Poesia, Theodoros Kasampalis, Angélica Moreira and Tanvir Ahmed Khan for their internships at Facebook working with BOLT. Finally, we would like to thank Jean-Denis Muys and the anonymous reviewers for their helpful suggestions on improving this paper.

References

[1] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. 2014. The Hiphop Virtual Machine. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 777–790.

[2] Erik H. Baalbergen. 1988. Design and implementation of parallel make. *Computing Systems* 1, 2 (1988), 135–158.

[3] Giuliano Belinassi. 2019. The Parallel GCC. Web site: https://gcc.gnu.org/wiki/ParallelGcc.

[4] Matheus T. Bernardino, Giuliano Belinassi, Paulo Meirelles, Eduardo M. Guerra, and Alfredo Goldman. 2020. Improving Parallelism in Git and GCC: Strategies, Difficulties, and Lessons Learned. *To appear in IEEE Software* 0, 0 (2020), 0–0.

[5] Igor Böhm, Tobias J.K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel Topham. 2011. Generalized Just-in-Time Trace Compilation Using a Parallel Task Farm in a Dynamic Binary Translator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 74–85.

[6] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the USENIX Conference on Annual Technical Conference*. 49–60.

[7] Robert Cohn, D. Goodwin, and P. G. Lowney. 1997. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal* 9, 4 (1997), 3–20.

[8] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. 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*. 15–24.

[9] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference*.

[10] GCC Team. 2017. GNU Compiler Collection. Web site: http://gcc.gnu.org.

[11] Thomas Gross, Angelika Zobel, and Markus Zolg. 1989. Parallel Compilation for a Parallel Machine. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 91–100.

[12] Ealan A Henis, Gadi Haber, Moshe Klausner, and Alex Warshavsky. 1999. Feedback based post-link optimization for large subsystems. In *Proceedings of the 2^{nd} Workshop on Feedback Directed Optimization*. 13–20.

[13] Teresa Johnson, Mehdi Amini, and Xinliang David Li. 2017. ThinLTO: scalable and incremental LTO. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 111–121.

[14] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings*

of the International Symposium on Code Generation and Optimization. 75–86.

[15] C-K Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. 2004. Ispike: a post-link optimizer for the Intel Itanium architecture. In *Proceedings of the International Symposium on Code Generation and Optimization.* IEEE, 15–26.

[16] A. Newell and S. Pupyrev. 2020. Improved Basic Block Reordering. *IEEE Trans. Comput.* 69, 12 (2020), 1784–1794.

[17] Scott Oaks. 2014. *Java Performance: The Definitive Guide: Getting the Most Out of Your Code.* O'Reilly Media, Inc.

[18] Guilherme Ottoni. 2018. 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 (PLDI).* 151–165.

[19] Guilherme Ottoni, Thomas Hartin, Christopher Weaver, Jason Brandt, Belliappa Kuttanna, and Hong Wang. 2011. Harmonia: a Transparent, Efficient, and Harmonious Dynamic Binary Translator Targeting x86. In *Proceedings of the ACM International Conference on Computing Frontiers.* 26:1–26:10.

[20] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing Function Placement for Large-scale Data-center Applications. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization.* 233–244.

[21] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java™ HotSpot Server Compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology Symposium.*

[22] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 2–14.

[23] Proxygen Team. 2017. Proxygen: Facebook's C++ HTTP Libraries. Web site: https://github.com/facebook/proxygen.

[24] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* 30, 4 (2010), 65–79.

[25] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. 1997. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, Vol. 1997. 1–8.

[26] Sriraman Tallam. 2019. [RFC] Propeller: A frame work for Post Link Optimizations. Web site: https://lists.llvm.org/pipermail/llvm-dev/2019-September/135393.html.

[27] Sriraman Tallam. 2020. Re: [llvm-dev] [RFC] BOLT: A Framework for Binary Analysis, Transformation, and Optimization. Web site: https://lists.llvm.org/pipermail/llvm-dev/2020-October/146185.html.

[28] Propeller Team. 2020. Propeller: Profile Guided Optimizing Large Scale LLVM-based Relinker. Web site: https://github.com/google/llvm-propeller.

[29] David B. Wortman and Michael D. Junkin. 1992. A Concurrent Compiler for Modula-2+. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI).* Association for Computing Machinery, 68–81.