Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications

Han Shen*
Google
USA
shenhan@google.com

Krzysztof Pszeniczny*
Google
Switzerland
kpszeniczny@google.com

Rahman Lavaee*
Google
USA
rahmanl@google.com

Snehasish Kumar*
Google
USA
sneaky@google.com

Sriraman Tallam*
Google
USA
tmsriram@google.com

Xinliang David Li*
Google
USA
davidxl@google.com

ABSTRACT

While profile guided optimizations (PGO) and link time optimizations (LTO) have been widely adopted, post link optimizations (PLO) have languished until recently when researchers demonstrated that late injection of profiles can yield significant performance improvements.However, the disassembly-driven, monolithic design of post link optimizers face scaling challenges with large binaries and is at odds with distributed build systems. To reconcile and enable post link optimizations within a distributed build environment, we propose Propeller, a relinking optimizer for warehouse scale workloads. To enable flexible code layout optimizations, we introduce basic block sections, a novel linker abstraction. Propeller uses basic block sections to enable a new approach to PLO without disassembly. Propeller achieves scalability by relinking the binary using precise profiles instead of rewriting the binary. The overhead of relinking is lowered by caching and leveraging distributed compiler actions during code generation.

Propeller has been deployed to production at Google with over tens of millions of cores executing Propeller optimized code at any time. An evaluation of internal warehouse-scale applications show Propeller improves performance by 1.1% to 8% beyond PGO and ThinLTO. Compiler tools such as Clang improve by 7% while MySQL improves by 1%. Compared to the state of the art binary optimizer, Propeller achieves comparable performance while lowering memory overheads by 30%-70% on large benchmarks.

CCS CONCEPTS

• **Software and its engineering → Retargetable compilers**.

KEYWORDS

Profile Guided Optimization, Post-Link Optimization, Binary Optimization, Warehouse-Scale Applications, Distributed Build System, Datacenters

*In alphabetical order of first name.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3575727

ACM Reference Format:
Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. 2023. Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3575727

1 INTRODUCTION

Due to the large computational footprint of warehouse scale applications, any improvement in application performance has an outsized impact [9, 32, 38]. When precise profiles are injected late in compilation [51], performance can be improved even beyond existing technologies such as profile guided optimizations and link time optimizations. However, we find that the monolithic design of binary optimizers is ill-suited for use with distributed build systems. With Propeller we introduce a new paradigm for post link binary optimizations, enabling their use with distributed builds while reducing overheads and improving coverage.

1.1 Why Do We Need a New Paradigm?

The design of disassembly driven binary optimizers is at odds with the characteristics of warehouse scale applications. To maximize performance, applications are built with all optimizations and libraries are linked statically. They also use link time cross-module optimizations and hot code sequences are tuned with hand-written assembly. Unfortunately, the performance of disassemblers degrades with increasing complexity [7]. The most recent iteration of BOLT, Lightning BOLT [52] parallelizes key aspects of the binary optimizer, however, it is still limited to a single machine. Sequential processing is necessary during function discovery, disassembly, global optimizations and finally emit-and-link. These stages account for more than half of the total BOLT runtime. Furthermore since disassembly is an inexact science for CISC architectures, the results may be unreliable. This may lead to failures during optimization or worse, application run time crashes.

Similar scalability problems are encountered with link time optimizations in GCC and LLVM. The traditional, monolithic approach attempts to link all the intermediate language files together and optimize them as a single translation unit. However, this method

consumes a disproportionately large amount of memory and increases runtime. To reduce overheads, LLVM[1] offers a multi-stage LTO implementation called ThinLTO [37] with

(1) Per-module, distributed summary generation.
(2) Fast, serial whole program summary based analyses.
(3) Per-module, distributed function importing and middle-end optimizations.

This allows the overheads to be reduced by taking advantage of the insight that only a small number of functions across modules are relevant in a particular context. Furthermore, this allows distributed build systems to leverage multi-process parallelism for all stages apart from (2). Thus, inspired by the approach adopted to scale LTO to distributed build systems, we propose a new paradigm for binary optimizations with the Propeller framework.

1.2 Our Contributions

To bring further performance improvements to warehouse scale PGO-optimized binaries, we present Propeller, a profile guided relinking optimizer which leverages distributed build systems. Propeller has been deployed to production at Google with over *tens of millions of cores executing Propeller optimized code at any time*. The contributions we make in this work are:

- The design of a profile guided, *relinking* optimizer for warehouse scale workloads, a new paradigm for post link optimizations.
- Demonstrate how post link optimizations can leverage the power of distributed build systems while adhering to running time and memory constraints.
- Basic block sections, a novel linker abstraction which enables flexible and fine grained code layout optimizations.
- An exhaustive evaluation of the impact of profile guided relinking on commercial and open-source workloads.

2 BACKGROUND

Homogeneity in hardware, software and control systems allow a datacenter to be treated as a warehouse scale computer (WSC), a new class of computing infrastructure[10, 38]. WSCs often run a small number of very large internet scale applications belonging to a single organization. Warehouse scale applications require the use of warehouse scale tooling to build, test and deploy. In this section, we introduce distributed build systems, profile guided optimizations, link time optimizations, post link optimization and their use in internet scale applications.

2.1 Distributed Build Systems

As the complexity of software increases, companies such as Google and Microsoft have adopted a *monorepo* (monolithic repository) approach to maintaining their source code [53, 68]. A monorepo approach has the advantages of simplified dependency management, versioning, and atomic changes, thus fostering collaboration. As the codebase grows, dependencies are shared widely across teams. The build action dependency growth rate is quadratic, i.e. increase in submit rate (more engineers) and increase in test pool size (more functionality). To help cope with this increased build/test workload when developing in monorepos, distributed build systems are commonly used. In contrast to local build systems, a distributed build executes actions on network connected, remote machines. It is beneficial when the number of actions required to build a target is orders of magnitude larger than the processing capability of any individual system.

Distributed Build Systems are Essential for Developer Productivity: A 2020 study at Google [67] shared that over 15M actions were executed daily. More than 100K changes are submitted every day by 50K+ developers towards a codebase of over 3 billion lines of code [67]. Over a 5 year period, daily build actions have increased 3×, changes submitted by 2× [30, 67]. The only way to tame this is massive parallelism enabled by a distributed build system. Coupled with aggressive content based caching with a hit rate >90%, most developers spend less than 10s waiting for plain builds to complete. To ensure predictable resource consumption growth all actions are provided with a small (fixed) number of CPU cores and 12 GB of RAM. Microsoft CloudBuild[24], IBM ClearMake[33] are additional examples of distributed build systems used in production. Apart from bespoke solutions, *distcc* and *ccache* are often used to accelerate builds [40, 61] via distributed execution and caching respectively.

2.2 Profile Guided Optimizations (PGO)

Profile guided compiler optimizations are effective in further improving performance of applications when trained with appropriate inputs. PGO in compilers like LLVM and GCC includes profiling of branch weights and indirect call targets among other profile types. The information collected during the training phase is used to augment heuristics driving function inlining, function/block layout, register allocation among other passes.

PGO employs a two stage build where the first step is to build an instrumented binary. This binary is executed using a set of inputs suitable for training, and profile data is collected from this run. The second build step compiles the binary again with the profile collected from the first run. Due to the high overheads incurred by instrumentation, load tests mimicking production scenarios are used to collect profiles. A newer technique, AutoFDO [17], uses hardware performance counters to collect profiles from workloads serving production traffic with low overhead. Both techniques are widely used in the industry and performance improvements of 5% are observed on SPEC2006 benchmarks [17]. However, as compilation progresses through the optimization pipeline, code transformations can cause a mismatch between the profile data and the code being optimized, reducing the applicability and effectiveness of the profile data. This creates opportunities for further performance improvement which can be addressed by post-link optimizers.

2.3 Link Time Optimizations (LTO)

Previously, the phrase "Link Time Optimization" (LTO) has been used to describe disassembly driven, whole program optimizations which are micro-architecture specific [22, 46, 58]. More recently, compilers have redefined this to refer to *"a compilation mode in which an intermediate language (an IL) is written to the object files*

[1]GCC offers scalable LTO [12] too, however, it has not been adapted for use with distributed build systems.

and the optimizer is invoked during the linking stage" [28]. Operating on the intermediate representation rather than relying on disassembly enables reuse of the existing optimization pipeline and avoids the caveats of disassembly driven reconstruction. Both GCC and LLVM offer link time optimization capabilities as standard. The intermediate representation is read into the compiler at link time treating the whole program as a single source level compilation unit. More scalable forms of LTO use summary based analyses to reduce runtime and memory overheads (see §1.1). LLVM based LTO improved performance of SPEC2006 benchmarks on average by 2%-4% [37].

2.4 Post Link Optimizations (PLO)

Post link optimizers (or *static binary optimizers*) operate directly on the executable file [22, 44, 46, 47, 51, 58]. The binary is disassembled and analyses are performed on the reconstructed control flow graph. Optimizations are introduced by rewriting the executable as opposed to dynamic binary optimizers [14, 15, 66] which modify the code in memory at runtime. Some of the optimizations performed are code layout, data compaction, alignment, and peephole optimizations such as conditional branch reversal. PLO may be guided by profile data collected via instrumentation [46, 47, 58] or hardware performance counters [44, 51]. Binary optimizers have advantages such as the ability to operate without source code, enabling tailored optimizations for proprietary libraries. Like LTO, they have a global view of the program enabling more optimizations. However, they are limited by challenges in disassembly such as data embedded in code. Disassembly driven optimizers require heavy weight pre-processing prior to analyses increasing memory overhead. The stability and performance of binary optimizers degrade with increasing complexity of the executable, i.e. higher optimization levels, presence of hand-tuned assembly and the use of static linking [7].

PLO mitigates profile quality loss: As retargetable compilers such as GCC and LLVM matured, their architecture-specific code generation improved. Scalable whole program analysis is now a common feature in both compilers. Finally, profile information was incorporated early in the optimization pipeline. It seemed that the value of binary optimizers for performance had been wholly subsumed by the compiler.

Recently, BOLT[51] demonstrated that injecting hardware profiles with a binary optimizer can significantly improve code layout. BOLT showed that profiles applied to the compile, link or post link are complementary to each other. Post link profiles fix inaccuracies accrued by instrumented profiles as optimizations transform the source. This novel insight has expanded the scope of PLO beyond whole program and architecture specific optimizations. Our experiments show that identifying cold blocks using hardware sample profiles collected from an PGO optimized binary is more effective than directly identifying cold blocks in the PGO profile. Performance improves by an additional 1% when function splitting (see §4.6) is driven with the former on a clang benchmark. BOLT demonstrated the effectiveness of PLO on large Facebook workloads, improving performance by 2-7% [51]. With these encouraging results, binary optimizers have new-found relevance for those seeking peak performance.

3 DESIGN OF A PROFILE GUIDED, RELINKING OPTIMIZER

The design of Propeller is driven by the need for scalable and sound post-link optimization (PLO). The key barrier to scalability and soundness is the disassembly oriented approach of existing PLO toolchains. Disassembly is one of the primary serializing bottlenecks in state of the art PLO tools [52]. Accurate recursive disassembly [7] of complex applications is challenging to distribute across many machines due to the incremental nature of discovery. The design of Propeller is guided by the following insights of the distributed build system used in production at Google.

1. Extensive caching is available: Build environments cache intermediate build artifacts to reduce computation at the expense of cheap, fast storage. Artifacts such as optimized LLVM IR (intermediate representation) and native object files are readily available for reprocessing. Reprocessing a small, targeted fraction of cached artifacts does not add significant overhead.

2. Monolithic, heavy-weight actions are discouraged: Warehouse scale applications depend on thousands of source files. Distributed build actions often operate on a single module at a time. Cross module optimizations are supported via summary based analyses, e.g. ThinLTO. The key non-distributed action with the largest memory footprint is the final native link of the object files.

From these insights we derive the cornerstone of Propeller's design, **relinking**, which obviates the need for disassembly. Relinking makes it possible to distribute the optimization of the individual modules across machines. This keeps the resource usage of relinking actions equivalent to compiler actions and allows Propeller to scale to warehouse scale applications. We note that BOLT [51] has proposed MCPlus serialization [4, 50] as future work to support caching and relinking as an alternative to disassembly. This further reinforces the advantage of a design which eschews the use of disassembly.

Figure 1 illustrates the Propeller end-to-end workflow while Table 1 lists each component and its tool or framework. Each phase is described in order, in the subsections that follow.

Table 1: Propeller Components

Component	Tool / Framework
Optimized IR cache	Distributed build system [67]
BB address map	LLVM Backend [27]
Hardware profiles	Linux perf [39]
Whole Program Analysis	Standalone tool [29]
Local code layout	LLVM Backend [25]
Object file cache	Distributed build system [67]
Global code layout	LLD [64]

3.1 Phase 1 - Compile and Cache Optimized LLVM IR

In Phase 1 of the workflow, individual program modules are compiled to optimized LLVM IR modules. The optimized IR is cached for reuse by the distributed build system used in production. At

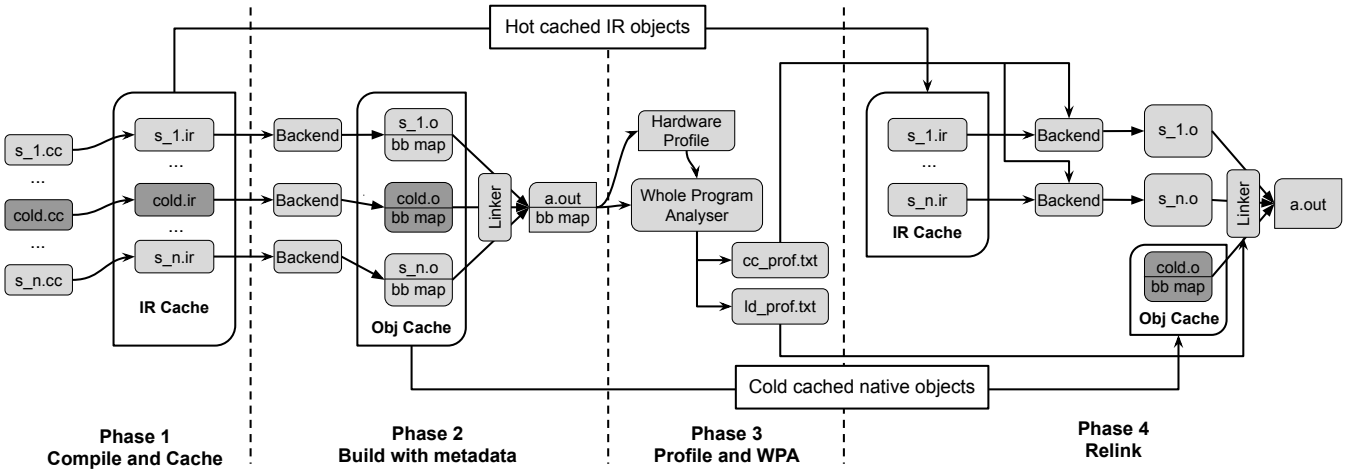


Figure 1: Design of a Profile Guided, Relinking Optimizer

this stage, all optimizations are enabled for each module including profile guided optimizations such as instrumented PGO or AutoFDO [17]. In the later phases, Propeller optimizes and relinks these cached artifacts, avoiding disassembly. The memory and time overheads to relink and build the final Propeller optimized binary is lower than building a vanilla optimized binary. This is due to the reuse of cached build artifacts from the prior phases. The backend actions that optimize and convert the cached IR to native object files are distributed and the memory requirements of relinking are within the thresholds enforced by the distributed build system. For a distributed build environment that includes ThinLTO [37], this step does not introduce any additional storage or compute requirements. However, note that Propeller is complementary to ThinLTO and can be applied independently too.

3.2 Phase 2 - Build Optimized Binary with Profile Mapping Metadata

In Phase 2, after cross module optimizations and codegen operations the compiler backend generates mapping metadata. The metadata enables the association of hardware profiles containing virtual addresses to machine basic blocks. The **Basic Block Address Map** [27], i.e profile to basic block mapping metadata is generated for each function and stored in a separate ELF section (see Table 1). The offset of each machine basic block from the beginning of a function is stored along with an identifier. Additional block characteristics may also be stored such as next fall-through basic block to aid whole program analysis in Phase 3. Application performance is unaffected since this ELF section is not loaded into memory. Experimental results show that the binary size is increased by 7% to 9% for the benchmarks we study in § 5. Cold object files which do not undergo profile driven transformations are reused from the cache in Phase 4.

3.3 Phase 3 - Profile Collection and Whole Program Analysis (WPA)

While the workload is executing representative load, sampled hardware profiles are collected using the Linux perf tool. We collect Last Branch Records (LBR) [39] from Intel machines which contain the source and destination address pairs of the last 32 retired jump instructions. The Basic Block Address Map is used to map the samples to machine IR [5] basic blocks. A *dynamic* control flow graph (DCFG) is created for every function encountered in the LBR samples. The graph is built incrementally, defining edges as samples are processed. Reconstructing the control flow does not require disassembly. Each basic block is annotated with metadata from the Basic Block Address Map (§ 3.2) and profile counts from the sample data. We use the Ext-TSP algorithm proposed by Newell and Pupyrev [49] to approximate the optimal basic block reordering. The outputs of the computation are code layout directives to the compiler and linker. Fig 1 shows the outputs named *cc_prof.txt* and *ld_prof.txt* and their use in Phase 4. The whole program analysis has been released as a standalone tool [29] (see Table 1). The memory consumption of this step is directly proportional to the cumulative size of the DCFGs. The peak heap memory used is at most 2.5GB for the warehouse scale applications, each containing 90MB to 600MB of code (see Fig. 4 and Table 2).

3.4 Phase 4 - Relinking to Yield a Propeller Optimized Binary

The number of cold functions outnumber hot functions, ergo in practice, most object files contain only cold code. For these, no samples were observed during profiling in Phase 2 (see § 3.2). Cold object files (see *cold.o* in Fig. 1) are retrieved from the distributed build system [67] cache and passed to the final **relink** action. For object files containing hot code, we rerun the LLVM backend code generator with the desired block layout specified in *cc_prof.txt* (see Table 1). Propeller updates the basic block layout of ≃10% of object files on average for our workloads. The largest fraction of hot object

files, i.e., those with at least one hot function, was observed in the clang benchmark (33%).

For code layout optimizations evaluated in this work, the backend generates multiple basic block clusters for each hot function. The hot basic blocks form the primary cluster, whereas cold blocks (if any) are placed in a separate cluster. For inter-procedural basic block layout, additional clusters are generated as described in section 4.7. As described in § 4, each cluster is placed in a separate section and assigned a symbol. The primary cluster retains the symbol of the parent function, while the cold cluster gains a suffix *.cold*. Any additional clusters generated for inter-procedural layout (when profitable) are named by appending a numeric identifier to the original symbol. Barring object size overheads, the design allows for an arbitrary number of basic block cluster sections per function. The final **relink** action is invoked with the desired global layout in *ld_prof.txt* (see Fig. 1). The running time of Phase 4 is minimized by reusing the cached LLVM IR from Phase 1, distributed invocation of codegen backends, and reusing the cached object files from Phase 2 (see Fig. 1).

In this phase, the hot LLVM IR objects are retrieved from the cache for code layout optimizations. The cold native objects are reused from Phase 2. The linker arranges the layout of all basic block sections in the final binary based on the symbol ordering specified. Any address map metadata sections in the cold native objects are dropped by the linker.

3.5 Design of Propeller Optimizations

The Propeller workflow illustrated in Figure 1 is designed to take advantage of distributed build systems. This design is inspired by ThinLTO [37] where the thin link step is kept lightweight and uses summaries to guide subsequent compiler optimizations. This model requires that an optimization be split into two parts. The first part is done in a distributed fashion independently on each IR object. The second part is done as a whole-program transformation on the entire binary in the relink phase. With code layout optimizations presented in this work, the former is the intra-function layout, performed independently on each function. The global layout of basic block clusters is done at link time by leveraging the section ordering feature in the linker.

Additionally, optimizations must be designed to keep memory consumption low. This is due to distributed build systems enforcing limits on how much CPU and RAM each action may consume (see § 2.1 for rationale). In this work, the parts that consume the most memory are the whole program analysis in Phase 2 and the final relink action in Phase 4. The memory usage of whole program analyses can exceed the limits if fine-grained information about the program is retained. Therefore, we keep minimal control flow information required to perform code layout (see Phase 2 in Fig. 1); no instructions are disassembled. The final relink step is also memory intensive as it consumes all the object files and writes out the final binary.

Profile guided, post link software prefetch insertion [45] is another optimization that can be implemented in Propeller. The whole-program analysis of cache miss profiles determine prefetch insertion points. A summary-based directive can then drive the distributed code generation actions that modify the objects and insert prefetch instructions.

4 BASIC BLOCK SECTIONS

Propeller code layout optimizations are enabled by a novel linker abstraction. A section is a contiguous range of bytes containing either code, data, debug info, relocations, or metadata that the linker operates on as a single unit. Compilers such as gcc and clang feature function sections, an abstraction which allows the linker to deduplicate code and perform coarse-grained code layout. We introduce *Basic Block Sections*, a new abstraction where *one or more* basic blocks of a single function are placed in a unique text section in the object file. Basic block sections enable fine granularity code layout decisions without disassembly and binary rewriting. While we implemented our approach for the ELF format [2], the idea can be generalized to PE, COFF [1] and Mach-O [3] executable formats. Symbols are generated to refer to each basic block section allowing tools to manipulate the ordering at a finer granularity than previously possible in the executable. The ordering is performed via a *symbol ordering file*, a concept supported by modern linkers [63, 64]. Ordering basic block sections enables optimizations such as inter-procedural block layout, global function layout and function splitting. The following section outlines the challenges to be addressed and discusses the aforementioned use cases.

4.1 Object File Metadata

Each additional section added to an object file increases its size by tens of bytes. The number of basic blocks can be much larger than the number of functions. For instance, the *clang* binary contains $2.1M$ basic blocks from approximately $160K$ functions. Enabling sections for each block can increase object sizes significantly. Furthermore the peak memory usage of the final relink action (see Fig 1) is increased. Propeller creates basic block sections only where necessary by computing block layout in Phase 3 (§ 3.3). This allows the compiler backends in Phase 4 (§ 3.4) to create sections with *clusters* of basic blocks. Mapping multiple blocks from the same function to a single cluster keeps overheads low.

4.2 Branch Relocations and Explicit Fall-Through

To improve performance, one side of a branch terminating a basic block is retained as an implicit fall-through. To allow the linker to reorder basic blocks, the compiler must retain the direct jump instruction to the next basic block. This makes the fall-through explicit while retaining flexibility. Static relocations are added for each branch whose address needs to be updated since resolution of branch targets is deferred to the linker. The compiler uses short branch instructions on some ISAs[8] for branch offsets that can be accommodated in one byte. With basic block sections, since long branch instructions are used, the offset is not determined at compile time. After code layout has been performed, a bespoke linker relaxation pass[62] removes fall-through branches. Additionally it shrinks branch instructions where the offset can be encoded in fewer bytes.

4.3 Debug Information

Preserving debug information across code layout optimizations is necessary to ensure ease of debugging. The potential discontiguous layout of basic blocks requires debug information to be generated per cluster. The DWARF standard allows for the generation of address range based descriptors using the *DW_AT_ranges* tag which can be discontiguous yet refer to the same entity. Additionally, two relocations point to symbols at the start and end of the basic block cluster. The overhead incurred is directly proportional to the number of basic block cluster sections generated. Thus basic block clusters (see § 4.1) are essential to keep final binary overheads low.

4.4 Call Frame Information (CFI)

Call frame information directives allow precise unwinding in the presence of optimizations where the function prologue may not be easily recognizable. Unlike the DWARF debug information, CFI does not account for the non-contiguous range of addresses occupied by a function. The DWARF standard explicitly requires emitting separate CFI Frame Descriptor Entries for each contiguous fragment of a function. The CFI for all callee-saved registers (possibly including the frame pointer) have to be emitted along with redefining the Call Frame Address (CFA), viz. where the current frame starts. Each additional basic block section increases the overhead of the *.eh_frame* section (ELF binaries). Clustering multiple basic blocks from the same function in a single section amortizes the overhead.

4.5 Exception Handling

Exceptions [34] are handled with basic block sections by splitting the call-site table into various ranges corresponding to the sections created. The call-site table explicitly specifies the landing pad start (*@LPStart*), as this is not necessarily the start of the function. The exception landing pads are kept together in a single section, although they may be combined with non-landing pad blocks. Further, the C++ ABI requires that landing pads have non-zero offsets relative to *@LPStart* to avoid ambiguity with the no-landing-pad case. Thus, if the landing pad section begins with a landing pad block, we insert a *nop* at the start of the landing pad section.

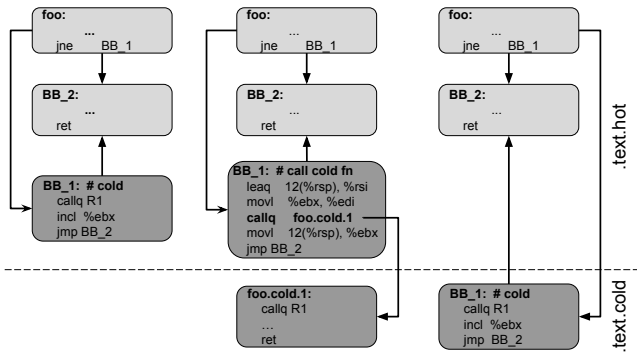


Figure 2: Original function layout (L), Splitting via function call (C), Splitting with basic block sections (R)

4.6 Use Case: Low Overhead Function Splitting

A study of warehouse scale workloads at Google showed that in half of the hottest functions, more the 50% of the code bytes are untouched [9]. Function splitting is an optimization where the cold basic blocks of a function are extracted to a separate region. This improves the locality of iTLB and icaches reducing frontend pipeline stalls. An implementation of this optimization in LLVM uses a heuristic to evaluate potential gain from splitting. A heuristic is necessary since the extraction mechanism uses a function call to jump to the cold blocks, incurring overhead in layout and possibly during runtime (see Fig. 2). With basic block sections, the function body can be discontiguous without code overhead introduced during extraction thus eliminating the need for a heuristic. Our experiments show that this approach is ≃2X [41] more effective than prior work in LLVM. Compared to a baseline optimized with PGO and ThinLTO, we observe up to 40% reduction in iTLB misses and 5% reduction in icache misses. Using an additional round of hardware profiling in the Propeller framework further improves performance by 1% for the clang benchmark.

4.7 Use Case: Inter-Procedural Code Layout

Basic block sections allow us to split and reorder functions across the whole program beyond hot-cold splitting. In particular, a function may be split into multiple sections placed apart in the binary. This improves performance for large, multi-modal functions which exhibit different behaviors within different calling contexts. For instance, consider Figure 3, illustrating the control flow between three functions. Function *foo* branches into either of its two loops. Each loop calls a different function that is not inlined. Placing the loop bodies close to their callees improves spatial locality, thereby reducing the icache and iTLB misses. As shown in Figure 3, an optimal **intra**-function layout keeps both callees close to *foo*, but cannot place them near their callsites. In contrast, an **inter**-function layout can split the layout of the two loops and place both callees near their callsites. This improves the overall spatial locality since the calls from the loops are much hotter than the edges outside the loops.

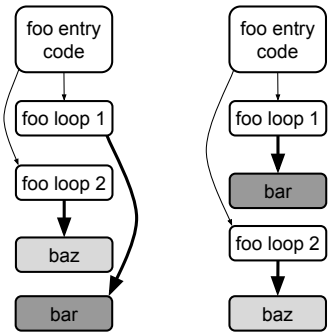


Figure 3: Optimal intra-function (L) and inter-function (R) layout. Hotter edges are shown by thicker lines.

To compute the optimal inter-procedural layout, we apply the Ext-TSP algorithm [49] on the whole program control flow graph (CFG). This graph includes edges for inter-procedure calls. The

unmodified algorithm does not scale with the size of whole program CFGs constructed for warehouse scale applications. We implemented several improvements such as logarithmic time retrieval of the most profitable action. Even with such improvements, generating the optimal inter-function layout takes 3× to 10× longer than intra-function layout.

Inter-function layout improves the performance of *clang* by 0.8% in comparison to the **intra**-function layout. For clang, the icache and iTLB miss rates reduce by 11% and 13%, respectively. Although promising on some applications, optimal **inter**-function layout requires a more extensive study which we leave for future work. In the evaluation of this work, we focus on **intra**-function layout enabled by basic block sections.

5 EVALUATION

The design of Propeller is focused on scalability, see § 1.1 for motivation. Thus we evaluate Propeller optimizations with an emphasis on meeting the limits imposed by the distributed build system at Google. We contrast Propeller's relinking approach to the most recent iteration of BOLT, Lightning BOLT [52]. When measuring memory usage of BOLT optimizations, we use the options that yield the fastest runtime. For workload performance, we present results where all BOLT optimization passes are enabled. We demonstrate the efficacy of Propeller optimized benchmarks, both in standalone benchmarks and using data gathered from a production environment. To put these numbers into perspective, we present the cost (in time) to release processes which integrate Propeller optimizations. Finally, we discuss the challenges we encountered while evaluating BOLT on warehouse-scale applications.

Benchmarks: We have evaluated Propeller on 4 warehouse-scale applications, 8 benchmarks from *SPEC2017* integer benchmarks and 2 open source workloads, *MySQL* and *Clang*. All the benchmarks are built for x86-ELF on Linux. Table 2 details characteristics of benchmarks relevant to this work. The following metrics provide insight into the complexity of the benchmark - size of the text section (Text), number of functions (Funcs) and the total number of basic blocks (BBs). Optimization run time and memory usage are directly proportional to these characteristics. The fraction of cold objects (% Cold) allow us to reason about the effectiveness of caching. The characteristics of each benchmark in Table 2 is drawn from the baseline binary.

Table 2: Benchmark Characteristics

Benchmark	Text	#Funcs	#BBs	% Cold
Clang	72 MB	160 K	2.1 M	67%
MySQL	26 MB	61 K	1.4 M	93%
Spanner [20]	175 MB	562 K	7.8 M	83%
Search	413 MB	1.7 M	18 M	95%
Bigtable [16]	93 MB	368 K	4.2 M	88%
Superroot [70]	598 MB	2.7 M	30 M	82%
SPEC 2017	34 KB - 4 MB	80-12 K	1-107 K	21%-88%

Methodology. In all evaluations, the baseline binary is built with instrumented PGO profiles and ThinLTO optimizations are enabled. Hardware sample profiles are collected from the PGO optimized binary. To ensure fairness, we use the same set of hardware profiles to drive both Propeller and Lightning BOLT. We have used a recent version of trunk LLVM BOLT[2]. For evaluations where memory usage and optimization run time are reported (§ 5.1, 5.2 and 5.7), BOLT was used with the following options:

```
-reorder-blocks=cache+ -reorder-functions=hfsort
-split-functions=3 -split-all-cold -split-eh
```

These options were recommended in Lightning Bolt [52] to reduce the peak memory usage and runtime of BOLT optimizations. For evaluations where performance was measured with BOLT (§ 5.4), we add `-lite=0` to enable heavy weight optimizations. This ensures that we report the best performance obtained by BOLT optimized binaries.

The *Clang*, *MySQL* and *SPEC2017* benchmarks, were built on a developer workstations with 72 cores and 192GB of RAM. The profile conversion and whole program analysis step (perf2bolt and Phase 3) were performed on the workstation to minimize differences in evaluation. BOLT optimized binaries were built on the workstation, allowing use of concurrent processing detailed in [52]. Memory usage for BOLT optimizations are unconstrained. The warehouse-scale benchmarks and their Propeller optimized versions were built on a distributed build system [67]. Each distributed action is limited to 12GB of RAM (24GB for *Superroot*) and ≃1 core on a remote machine. Peak memory usage for evaluations is obtained by measuring the max resident set size.

5.1 Peak Memory Usage of Phase 3: Profile Conversion and Whole Program Analysis

Propeller consumes less than 3GB across all workloads, within limits imposed by the build system and scales well for warehouse-scale applications. Prior to performing any optimizations the perf data profiles must be consumed to create data structures such as the whole program control flow graph (see § 3.3). For BOLT the analogous step is performed by *perf2bolt* where the binary is disassembled and the profile is converted to a custom format. Figure 4(L) shows the peak memory usage incurred by Propeller (Phase 3) and BOLT profile conversion. The profile conversion step is a single process that is not distributed. It is important that its peak memory usage be within the limits imposed by the distributed build system. The same hardware sample profiles are used for both BOLT and Propeller. The sizes of the profiles range from 100 MB to 700 MB (may be multiple for each benchmark). For the largest binary *Superroot* (see Table 2 for details) Propeller peak memory usage is 2.6GB. For Propeller, the peak memory usage is attributed to the maximum of reading profiles and the in-memory DCFG (see § 3.3). While the former can be reduced simply via chunked reading, the latter requires more effort. Note that BOLT's memory usage is much higher due to function-oriented, linear disassembly. It can potentially be reduced if selective processing used in LightningBOLT [52] is implemented here as well. Figure 4(R) also shows the peak memory usage incurred by Phase 3 for the *SPEC2017* benchmarks. For smaller binaries, like *505.mcf* and *557.xz*, BOLT performs on par

[2]Git revision: 6db71b8f1418170324b49d20f1f7b3f7c5086066

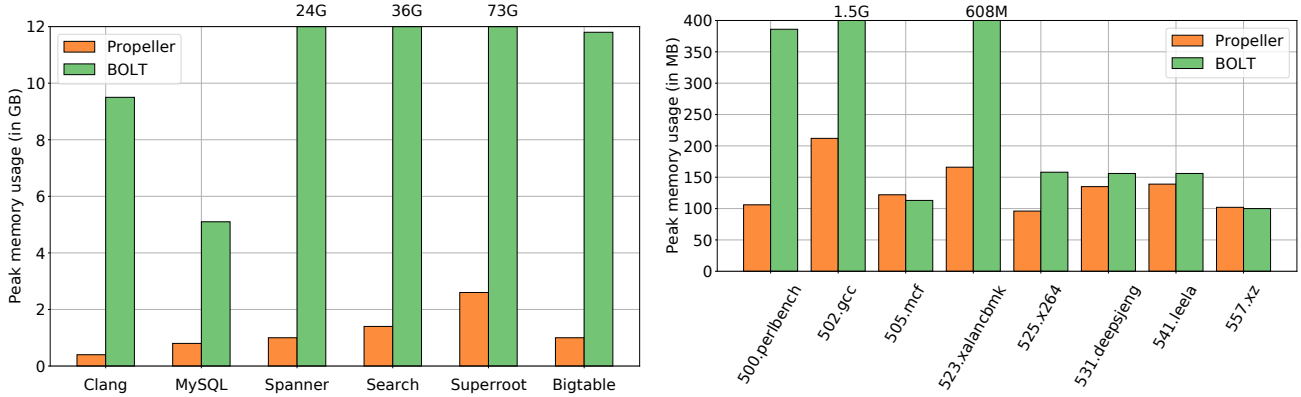


Figure 4: Peak memory usage during profile conversion and whole program analysis for warehouse-scale applications, open-source workloads (L) and SPEC2017 integer benchmarks (R).

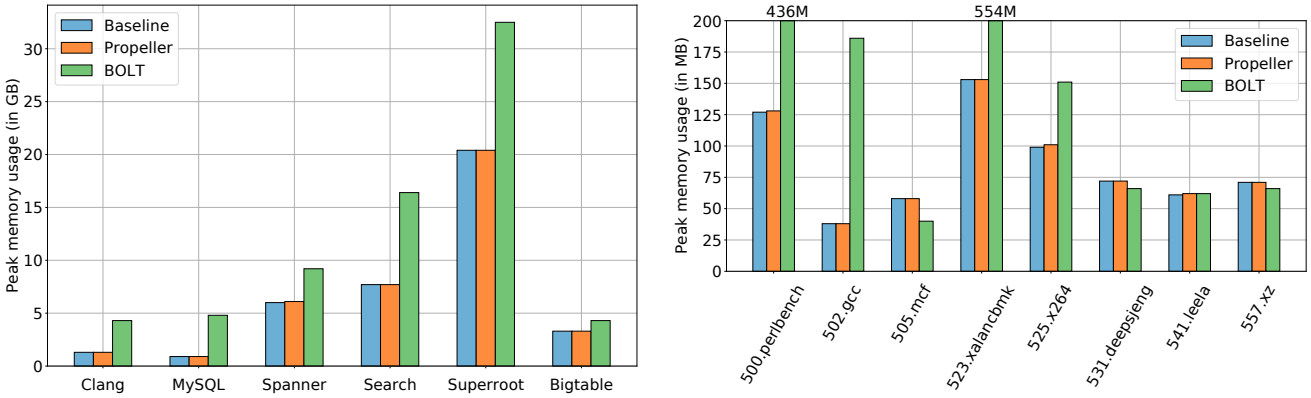


Figure 5: Peak memory usage for Propeller (Phase 4), BOLT optimizations and baseline link action on warehouse-scale applications, open-source workloads (L) and SPEC2017 integer benchmarks (R).

or better than Propeller. For *SPEC2017*, the average profile size is $\sim 30M$. This indicates that BOLT's memory usage during profile conversion does not scale with increasing workload size.

5.2 Peak Memory Usage of Phase 4: Code Layout Optimizations and Relink

Propeller code layout optimizations do not increase peak memory consumption. Figures 5(L) and 5(R) compare the peak memory usage incurred by Propeller (Phase 4), BOLT optimizations and the baseline link action. For Propeller, we profile the **relink** action in Phase 4 and for BOLT, we profile the *llvm-bolt* tool. For BOLT, all optimizations are performed in this step while for Propeller, only global optimizations are performed. For Propeller, local optimizations are performed as distributed codegen actions. This comparison demonstrates that introducing BOLT as a monolithic PLO step would shift the peak memory bottleneck from the link action to BOLT. Linker memory usage is somewhat well defined (≃2X size of inputs [21]) whereas the peak memory usage due to disassembly may not be. BOLT optimizations on SPEC benchmarks *505.mcf*, *531.deepsjeng* and *557.xz* consume less memory than the baseline link action. For BOLT even with selective processing, peak memory usage can be as large as 5× the baseline link action (see Figure 4 *MySQL*). For warehouse-scale applications, binary sizes grow with time (see § 2.1) which increases the linker memory usage. Propeller's design minimizes object file size increase via basic block clusters (see § 4.1) thus limiting increased peak memory usage.

5.3 Impact on Binary Size

Propeller's relinking based workflow avoids unnecessary binary size increases, on average less than 1% over baseline. Figure 6 shows binary sizes when building Propeller, BOLT, and baseline binaries. Both optimization frameworks require building with metadata for profiling (see § 3.2). The overall size for each is normalized to the size of the baseline binary. Each bar is subdivided into *.text*, *.eh_frame*, *llvm_bb_addr_map* and *.rela* sections. The remaining *other* component includes information such as read-only data. "Base" refers to the section wise size breakdown for the baseline binary. *"PM"* and *"PO"* corresponds to the binary built with Propeller metadata and optimizations respectively. *"BM"* and *"BO"* are similar but for the BOLT binary.

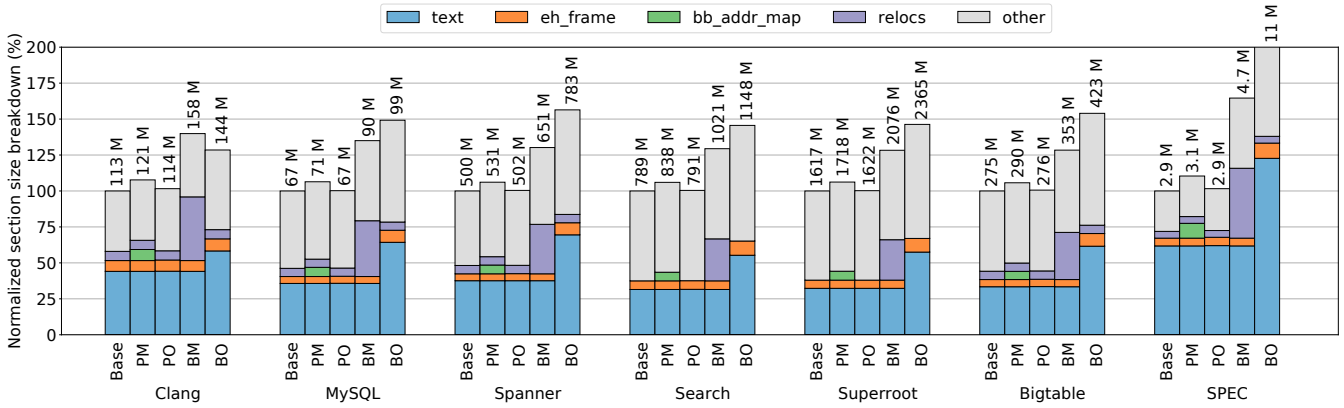


Figure 6: Normalized break down of section sizes for PGO and ThinLTO (Base), Propeller Metadata (PM), Propeller Optimized (PO), BOLT Metadata (BM) and BOLT Optimized (BO) binaries.

Metadata binaries: For Propeller, the metadata binary contains basic block mapping information, see § 3.2 for details. For BOLT, the metadata binary contains static relocations necessary to ease dissasembly and binary rewriting. The executable performance is unaffected since neither section is loaded into memory. However, the requirement for static relocations may be prohibitive if a binary needs to be built with debug information. Measured on a debug build of *Clang*, the *.rela* section (required by BOLT) can be up to 43% of the overall binary size (1.7G). Overall, Propeller and BOLT metadata binaries are 7-9% and 20-60% larger than the baseline respectively.

Optimized binaries: Propeller optimized binaries are on average 1% larger than the baseline binary for warehouse-scale applications, and for open-source and SPEC benchmarks. The increase can be attributed to additional symbol names for cold function parts (see §3.4) and *eh_frame* information (see § 4.4). For BOLT, the binary size increases are significant, 30% on average for *Clang* and *MySQL*, and 150% on average for large *SPEC2017* benchmarks. BOLT retains the original .text section since it only selectively optimizes some functions. For warehouse-scale applications, the BOLT optimized binary is 45% larger than the baseline. Furthermore, the new optimized .text is aligned to a 2M page boundary to take advantage of hugepages[3]. While this can be disabled via an option, by default it significantly increases sizes for binaries in *SPEC2017* (see Fig. 6).

5.4 Performance Evaluation of Code Layout Optimizations

Propeller code layout optimizations improve warehouse-scale applications up to 7%, comparable to BOLT on open-source workloads. We report the performance improvement of Propeller and BOLT in Table 3 compared to the baseline optimized with PGO and ThinLTO. For BOLT, we include results which use the `lite=0` option to enable all optimizations. On *Clang* the BOLT optimized binary and the Propeller optimized binary were ≃ 7% faster than the baseline. On *MySQL*, we measured the mean transaction latency for five, sysbench driven benchmarks. The performance improvement for Propeller and BOLT was 1% and 0.8% over the baseline respectively.

[3]https://github.com/facebookincubator/BOLT/issues/138

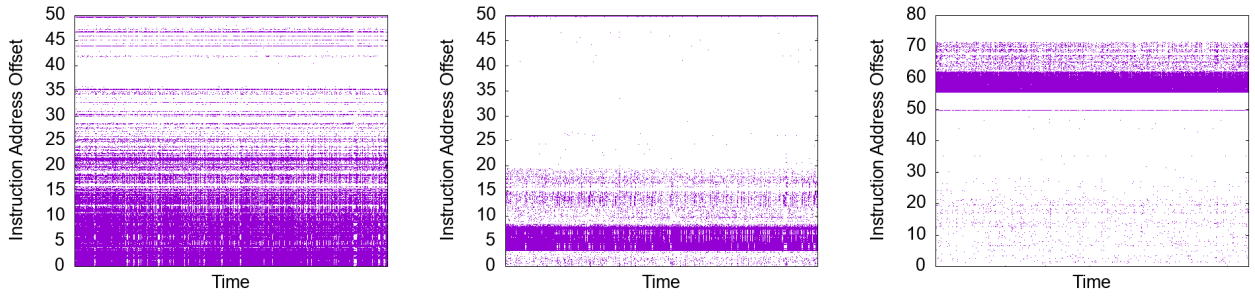
Table 3: Performance improvements of Propeller and BOLT optimized binaries over PGO and ThinLTO.

Benchmark	Metric	% Improvements	
		Propeller	BOLT
Clang	Walltime	7.3 %	7.3 %
MySQL	Latency	1 %	0.8 %
Spanner	Latency	7 %	*Crash*
Search	QPS	3 %	4 %
Superroot	QPS	1.1 %	*Crash*
Bigtable	QPS	3 %	*Crash*

For warehouse-scale applications, the Propeller optimized binary improves by 1%–7% over the baseline. Unfortunately, BOLT optimized binaries failed at startup for all but one of the warehouse-scale applications. For *Search*, the BOLT optimized binary is 4% faster than the baseline whereas the Propeller optimized binary is 3% faster. In addition to code layout, BOLT performs many disassembly driven optimizations (see Table 1 in [52]). Whereas the performance improvements for Propeller are solely due to improved code layout.

We present instruction access heat maps for the baseline, Propeller and BOLT optimized *Clang* binary in Figure 7. Each figure represents the binary address space. Accesses to each offset are marked over time for the *Clang* benchmark. The BOLT heat map (Figure 7(c)) shows a band at a higher offset since the optimized BOLT text section is placed in a new segment. Tightly laid out bands demonstrate the efficacy of code layout optimizations by BOLT and Propeller. This results in reduced iTLB, icache misses and improves performance by ≃7%.

Impact of Code Layout Optimizations on SPEC2017 Integer Benchmarks. We report the performance numbers from BOLT and Propeller on all the *SPEC2017* integer benchmarks except *520.omnetpp*, which fails to build with clang. BOLT improves *500.perlbench* by



(a) Baseline with PGO + ThinLTO **(b) PGO + ThinLTO + Propeller** **(c) PGO + ThinLTO + BOLT**

Figure 7: Whole binary, instruction access heat maps for the Clang benchmark. Y-axis instruction address offset labels truncated for clarity.

0.4% and Propeller improves *541.leela* by 1%. BOLT regresses 5 benchmarks by 0.8% − 6.3% (2.4% on average). Propeller regresses 5 benchmarks too, 4 of them the same as BOLT, by 0.8% − 3.9% (2% on average).

Performance counters show that Propeller and BOLT, on average, reduced the number of taken branches and icache misses by 10% and 20%, respectively, with the exception of *505.mcf*. This is expected as better code layout increases the number of fall-through code paths and improves icache utilization. *505.mcf*, regresses with both BOLT and Propeller by 1% and 4%, respectively. For this benchmark, the total number of branches and icache misses increased by 2% and 4% respectively. Top-down analysis [71] shows that iTLB is not a major bottleneck for all benchmarks except *502.gcc* and *500.perlbench*. Propeller and BOLT do not change the performance of these two programs.

The top-down analysis shows that DSB [6] misses on average account for 25% of stalled frontend cycles in baseline binaries. Both Propeller and BOLT, on average, increase the DSB misses by 12% and 60%, respectively. The DSB is a cache for fetched instructions and is sensitive to code alignment. Tuning Propeller's optimizations for smaller programs such as SPEC including effective DSB utilization remains future work.

5.5 Analysis of Code Layout Optimizations Using Hardware Performance Counters

We report the hardware performance counter data for two benchmarks, *Clang* and *Search*, in Figure 8. Table 4 lists the events, the labels used in Figure 8 along with a short description. Performance counters which include speculative events are indicated in the description in Table 4. Both workloads were profiled on the Intel Skylake platform. Hugepages (2M) for the .text section were enabled on the Search benchmark only. Workload characteristics are presented in Table 2.

I-cache: Compared to a PGO and ThinLTO baseline, Propeller and BOLT significantly improve code misses at the L1 and L2 for *Search* (up to 30%) and *Clang* (up to 40%). A key observation is that both raw misses (I1) and critical misses (I2) are improved. BOLT optimizations yield an additional 1.8% i-cache miss rate improvement over Propeller on Search. However on *Clang*, Propeller outperforms BOLT on i-cache metrics by 1.4%.

iTLB: Raw iTLB misses (T1) for *Clang* are reduced by 23% and 21% by Propeller and BOLT respectively. On *Clang*, critical iTLB misses (T2) reduced by 28% and 18% for Propeller and BOLT respectively. This is due to the reduced code footprint compared to the baseline as shown in the instruction heat map (see Figure 7).

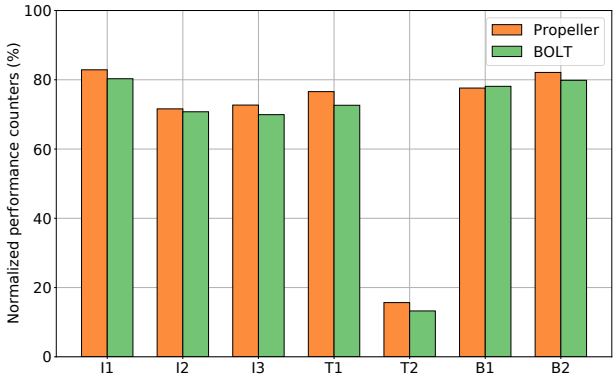
On *Search*, raw iTLB misses (T1) were reduced by 27% and 23% by BOLT and Propeller respectively. iTLB misses which caused a stall (T2) reduced by up to ≃ 85% due to improved code layout by BOLT and Propeller. As opposed to *Clang*, hugepages are enabled for the Search benchmark. While the total hot code for the *Search* workload is ≃20M i.e 5% of 417M (see Table 2), Propeller and BOLT optimizations split out cold basic blocks from hot functions. Compared to the baseline, this further reduces the hot text section by 40 − 50% bringing it within reach of the 8, 2M entries in the Skylake iTLB [23].

Branches: An important characteristic of code layout performed by BOLT and Propeller is an increase in fall-through code paths. Not taken branches do not occupy training resources in the branch predictor unit. Thus increasing fall-through code paths improve branch predictor efficacy. We find that BOLT and Propeller decrease branch resteers (B1) by ≃ 22% on *Search* and 24 − 30% on *Clang* respectively. This effect can be attributed to the decrease in not taken branches (B2) by 18 − 20% on *Search* and 15 − 20% on *Clang*.

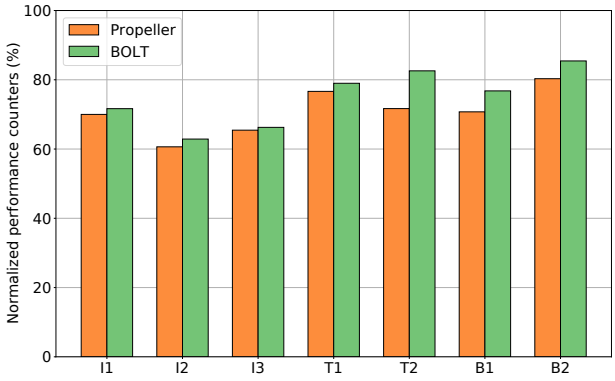
In conclusion, Propeller compares favourably to BOLT with respect to code layout optimizations for large workloads. However, BOLT further improves performance by 1% on Search (see Table 3) likely via additional optimizations, thus leaving room for improvement for Propeller in the future.

5.6 End-to-End Run Time Cost of Propeller Optimizations on Warehouse-Scale Workloads

This section discusses the context within which Propeller optimization running time is measured. Due to the high complexity of warehouse-scale applications, the build and release processes of these applications are also significantly complex. For example, profile guided optimizations require that the application is exercised with representative inputs. This constraint implies that profiling the application has to be performed in a synthetic, yet realistic environment. Table 5 presents the increase in build time for PGO and



(a) Search (lower is better)



(b) Clang (lower is better)

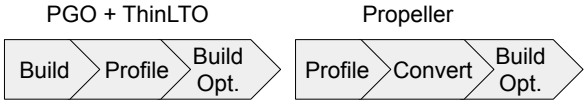
Figure 8: Performance counter data for Search (L) and Clang (R) for BOLT and Propeller optimized binaries. Legend detailed in Table 4.

.

Table 4: Performance counter events (details in Intel Perfmon manual [36]).

Category	Label	Event Name	Description
i-cache events	I1	frontend_retired.l1i_miss	L1 i-cache misses which caused a stall (non-speculative).
	I2	l2_rqsts.code_rd_miss	L2 cache instruction misses.
	I3	icache_16b.ifdata_miss (event=0x80,umask=0x2)	L1 i-cache misses.
iTLB events	T1	icache_64b.iftag_miss	iTLB misses.
	T2	frontend_retired.itlb_miss	iTLB misses which caused a stall (non-speculative).
Branch events	B1	baclears.any	Front-end resteers due to branches not tracked by the predictor.
	B2	br_inst_retired.near_taken	Taken branch instructions (non-speculative).

Table 5: Build phases and the time taken (in minutes) for warehouse-scale applications.



	Phase time taken (in minutes)					
	PGO (Phases 1&2)			Prop. (Phases 3&4)		
	Instr.	Profile	Opt.	Profile	Convert	Opt.
Spanner	7	48	17	45	3	9
Search	10	8	10	8	2	16
Superroot	23	37	36	18	3	15
Bigtable	9	30	13	43	18	10

Propeller over an optimized (-O3) build. For applications of such complexity, the relatively mundane parts dwarf the actual running time of the optimizations themselves. For example, in *Superroot* the profile conversion (4m) and propeller optimized build time (15m) is less than 15% of the overall time. We find that deploying Propeller optimizations extend build-release time by 78% (on average) for warehouse-scale applications even though the Propeller optimizations run time is a small fraction, ≃18% on average, of the whole.

Due to the outsized resource footprint of warehouse-scale applications [9] small improvements yield high aggregate savings. Thus it is imperative to extract as much performance as possible even when the up front cost is high and the yields diminish with each subsequent phase. In the next section, we compare the running time of Propeller Phase 4 with that of BOLT.

5.7 Optimization Run Time (Phase 4)

Propeller reuses cached objects. Thus, relink run time is less than baseline link run time. In this section, we measure the time taken to relink or rewrite the final optimized binary by Propeller and BOLT, respectively. For Propeller, this presents the total time in Phase 4 (see § 3.4). For BOLT, it is the time taken to run *llvm-bolt* [52] which optimizes and rewrites the final binary. Figure 9 compares the time taken for Propeller codegen and relink, BOLT and the baseline link action. Each benchmark shows 3 bars. "Base" corresponds to the time taken to build the baseline optimized binary, equivalent to Phase 2. The bar is divided to present the time taken in the "Backends" (optimized IR to object file generation) and "Linking" (the final link). "Prop." corresponds to the time taken to build the

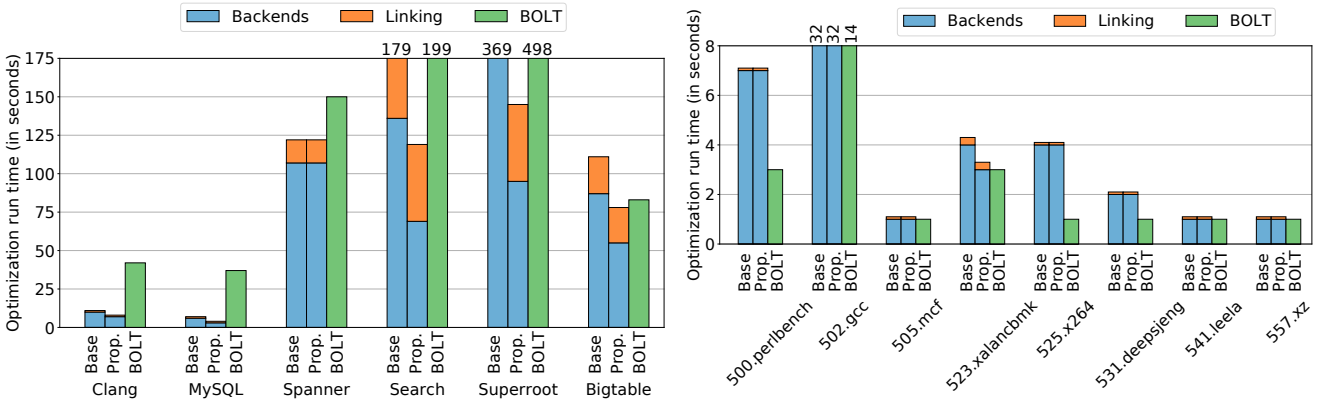


Figure 9: Optimization run time of warehouse-scale applications, open-source workloads (L) and SPEC2017 benchmarks (R).

Propeller-optimized binary in Phase 4 (§ 3.4). Here only the object files that need to be modified are re-generated and then passed to the final relink. Finally, "BOLT" shows the time needed to run *llvm-bolt*.

The benchmarks *Clang*, *MySQL*, and *SPEC* were built on a workstation. For these benchmarks, BOLT is faster than Propeller on average by 200% (max 400%). With Propeller, the time taken to generate the backends on a workstation dominates the total time.

For warehouse-scale applications, Propeller reuses the cold object files from the cache - 83% to 95% of total object files. For the remaining object files, the codegen backends are rerun concurrently to generate optimal layout for hot functions. We find that Propeller codegen time and subsequent link (critical path for distributed build actions) is on average 35% lower than the baseline codegen and link time. In the best case, Propeller relink time is 61% lower than the baseline link time (95% of objects are cold). BOLT is hindered by the need for disassembly, taking a large amount of time to process. Propeller is on average, 62% faster than BOLT in this step. On *SPEC2017*, BOLT optimization run time is faster than the baseline and Propeller in all cases. In the next section, we enumerate the challenges we encountered when evaluating BOLT on warehouse-scale applications.

5.8 BOLT Evaluation Challenges

We found and reported multiple issues in BOLT, such as an out-of-bounds memory access when processing exception handling frames in very large binaries [54]. BOLT does not natively support restartable sequences and will rewrite the code in abort sequences pointed to by the *__rseq_cs_ptr_array* API [13, 19]. Similarly, programs containing cryptographic modules subject to FIPS-140-2 [56] are required to perform startup integrity checks verifying that the code of the cryptographic module hasn't been tampered with, which needs special treatment in binary rewriting tools. Furthermore, binaries optimized using BOLT cannot currently be stripped [55] which makes them impractical for systems which store debug information on separate servers. The high memory requirements of BOLT make it infeasible to run it on a distributed build environment as opposed to a powerful developer workstation. However, the latter is not a trusted build environment, which makes it unsuitable for software supply chain verifiability and insider risk reduction programs. Finally, we were unable to evaluate BOLT on 3 of 4 warehouse-scale applications presented in this work due to failures on binary startup. Thus our experience with BOLT reinforces the pitfalls of disassembly driven post link optimizations.

6 RELATED WORK

After BOLT [51, 52], CodeStitcher [42] is the closest work to Propeller and is an inter-procedural basic block layout optimizer that uses sampled hardware profiles from the binary. CodeStitcher is implemented as part of LLVM's LTO [26] framework, where the inter-procedural analysis determines the order of basic blocks. Since CodeStitcher is part of full LTO, it inherits the scalability issues associated with it. CodeStitcher needs to be re-architected to work with ThinLTO and some of the design decisions made with Propeller could be used in that case. CodeMason [69] is another binary optimizer which is similar to BOLT [51] and uses disassembly and binary rewriting to further optimize binaries using post-link optimizations. Particularly, function reordering, alignment and PLT optimizations are applied. In [57], code layout optimizations are implemented on Spike [18] to improve the performance of transaction processing workloads. Spike [18] is similar to BOLT in that it is a binary optimizer and does not relink like Propeller.

MAO [31] is a microarchitectural optimizer that works at the assembly level and has a number of peep-hole and pattern matching optimizations to avoid some of the microarchitectural bottle-necks. MAO cannot do whole program optimizations since it optimizes one assembly module at a time. The optimizations done in MAO could also be done in Propeller, and as whole program optimizations. Other link time and post link optimzers and binary rewriting and patching frameworks like ALTO [46], PLTO [58], ATOM [60], Diablo [65], DynInst [15], FDPR [47], Spike [18], Ispike [44], and TMS320C32x [35] have been developed to perform post link optimizations that are hard to do at compile time. They are either not profile guided or are not built for scalabilty. Propeller differs in

that it does not directly rewrite the binary and these optimizations could also be implemented in Propeller.

Dynamic binary optimizers like DynamoRio [14], PIN [43], Valgrind [48], and QEMU [11] are frameworks that can perform a class of dynamic binary optimizations that are beyond the reach of static optimizers like Propeller. However, dynamic binary optimizers have non-trivial performance overheads from the managed execution engine and have found potential applications in program tracing and program behavior analysis.

7 CONCLUSION

The focus of our work is to bring post link optimizations (PLO) to production environments for warehouse scale computing. Scalability and soundness are the tenets which drive the design of Propeller, a profile guided relinking optimizer for PLO. Propeller enables fine grained PLO using basic block sections, a novel linker abstraction. Propeller achieves scalability and soundness by avoiding disassembly, reusing cached build actions wherever possible, relying on simple whole program analysis, performing local optimizations concurrently, and minimizing the global optimization cost in the final relink phase. Our experiments show that Propeller-optimized warehouse scale applications improve performance while keeping build actions inexpensive. We have enabled Propeller in production with tens of millions of cores running Propeller optimized binaries. The framework is available to the community as part of the LLVM project.

ACKNOWLEDGEMENTS

Our thanks to Teresa Johnson, Tipp Moseley, Parthasarathy Ranganathan, Rajiv Gupta, Svilen Kanev and other anonymous reviewers for their feedback on the manuscript. We would also like to thank prior interns, Manasij Mukherjee, Fatih Bakir and Brett Chalabian for their contributions to the project.

DATA-AVAILABILITY STATEMENT

The scripts that support the findings of this study are openly available in Zenodo at https://doi.org/10.5281/zenodo.7222794, reference number [59].

A ARTIFACT APPENDIX

In this section, we present the means to replicate the results on a standalone machine. We show Propeller can achieve equitable performance to Lightning BOLT with lower peak memory consumption. A key aspect of Propeller enabled optimizations is the integration with a distributed build system which provides caching. To demonstrate the effect of caching we provide scripting which emulates the effect on a single machine. The latest instructions to use Propeller are publicly available at https://github.com/google/llvm-propeller.

A.1 Artifact Check-List (Meta-information)

- **Algorithm:** Profile-guided relinking using basic block sections
- **Program:** Target workload is a clang bootstrap build
- **Git hash of clang**: 6db71b8f1418170324b49d20f1f7b3f7c5086066
- **Compilation:** CMake with clang-16
- **Transformations:** Post link code layout transformations
- **Binary:** 64-bit x86 Linux ELF
- **Run-time environment:** Bare metal, requires access to hardware performance counters
- **Hardware:** Intel Skylake or newer, hardware profiling requires Intel LBR.
- **Metrics:** Runtime, peak memory usage
- **Output:** Post link optimized binary, instruction access heatmaps.
- **Experiments:**
  (1) Performance of binary built with FDO vs Propeller vs BOLT
  (2) Impact of caching on Propeller build latency
- **How much disk space required (approximately)?:** 30GB
- **How much time is needed to prepare the workflow (approximately)?:** 30 minutes
- **How much time is needed to complete experiments (approximately)?:** 4 hours (most of it is waiting on the script to finish)
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache 2.0 License with LLVM exceptions
- **Workflow framework used?:** No
- **Archived:** **https://doi.org/10.5281/zenodo.7222794**

A.2 Description

In the following sections we describe the hardware and software dependencies. The software dependencies include the scripts we provide in the cloud machine to run and execute the experiments.

A.2.1 Hardware Dependencies. The minimum requirements in order to build and run the experiments are:

```
Intel Haswell or newer (preferably Skylake)
16 GB RAM
```

A.2.2 Software Dependencies. The experiments use a bootstrap version of clang. The dependencies of LLVM when built with CMake are documented here: .

```
# System
clang-16, lld
cmake
ninja
linux perf
git

# Built from source by the script
create_llvm_prof
perf2bolt
llvm-bolt
```

A.2.3 How to Access. The latest version of instructions to run Propeller are available at: https://github.com/google/llvm-propeller. The scripts presented in the artifact evaluation are archived at https://doi.org/10.5281/zenodo.7222794.

REFERENCES

[1] 1996. *COFF.* https://wiki.osdev.org/COFF (accessed 2020).
[2] 2003. *ELF - format of Executable and Linking Format (ELF) files.* http://man7.org/linux/man-pages/man5/elf.5.html (accessed Aug 20 2019).
[3] 2009. *OS X ABI Mach-O File Format Reference.* https://developer.apple.com/
[4] 2010. *LLVM MC Project.* http://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html (accessed Aug 15 2019).
[5] 2019. *Machine IR (MIR) Format Reference Manual.* https://llvm.org/docs/MIRLangRef.html (accessed Aug 20 2019).
[6] 2019. *MITE Micro-ops to IDQ.* https://software.intel.com/en-us/forums/intel-performance-bottleneck-analyzer/topic/308522 (accessed Aug 20 2019).

[7] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 583–600.

[8] ARM. 2021. Branch and Call Sequences Explained. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/branch-and-call-sequences-explained [Online; accessed 6-August-2021].

[9] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*. 462–473.

[10] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture* 13, 3 (2018), i–189.

[11] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[12] P. Briggs, Doug Evans, B. Grant, R. Hundt, W. Maddox, D. Novillo, Seongbae Park, D. Sehr, Ian Taylor, and Ollie. [n. d.]. WHOPR-Fast and Scalable Whole Program Optimizations in GCC Initial Draft 12-Dec-2007.

[13] Derek Bruening. 2017. Restartable Sequences. https://dynamorio.org/page_rseq.html (accessed 2022).

[14] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 265–275.

[15] Bryan Buck and Jeffrey K Hollingsworth. 2000. An API for runtime code patching. *The International Journal of High Performance Computing Applications* 14, 4 (2000), 317–329.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 205–218.

[17] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*. 12–23. https://doi.org/10.1145/2854038.2854044

[18] Robert S Cohn, David W Goodwin, P Geoffrey Lowney, and N Rubin. 1997. Optimizing alpha executables on windows nt with spike. *Digital Technical Journal* 9 (1997), 3–20.

[19] Jonathan Corbet. 2015. Restartable Sequences. https://lwn.net/Articles/650333/ (accessed 2022).

[20] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages. https://doi.org/10.1145/2491245

[21] Cary Coutant. 2013. *DWARF Extensions for Separate Debug Information Files a.k.a. "Fission" project.* https://gcc.gnu.org/wiki/DebugFission

[22] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, Dominique Chanet, and Koen De Bosschere. 2004. Link-time optimization of ARM binaries. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. 211–220.

[23] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro* 37, 2 (2017), 52–62.

[24] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's Distributed and Caching Build Service. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/2889160.2889222

[25] The LLVM Foundation. 2002. *The LLVM Compiler Infrastructure.* http://llvm.org (accessed Aug 20 2019).

[26] The LLVM Foundation. 2002. *LLVM Link Time Optimization: Design and Implementation.* https://llvm.org/docs/LinkTimeOptimization.html (accessed Aug 20 2019).

[27] The LLVM Foundation. 2020. *SHT_LLVM_BB_ADDR_MAP Section (basic block address map).* https://llvm.org/docs/Extensions.html#sht-llvm-bb-addr-map-section-basic-block-address-map

[28] Taras Glek and Jan Hubicka. 2010. Optimizing real world applications with GCC link time optimization. *arXiv preprint arXiv:1010.2196* (2010).

[29] Google Propeller. 2021. llvm-propeller. https://github.com/google/llvm-propeller (accessed 2021).

[30] Aysylu Greenberg. 2016. Building a Distributed Build System at Google Scale. https://gotocon.com/dl/goto-chicago-2016/slides/AysyluGreenberg_BuildingADistributedBuildSystemAtGoogleScale.pdf

[31] Robert Hundt, Easwaran Raman, Martin Thuresson, and Neil Vachharajani. 2011. MAO–An extensible micro-architectural optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 1–10.

[32] Andrew Hamilton Hunter, Chris Kennelly, Darryl Gove, Parthasarathy Ranganathan, Paul Jack Turner, and Tipp James Moseley. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*.

[33] IBM. 2021. Using remote build clearmake command. https://www.ibm.com/docs/en/rational-clearcase/9.0.0?topic=feature-using-remote-build-clearmake-command [Online; accessed 6-August-2021].

[34] LLVM Compiler Infrastructure. 2003. *Exception Handling in LLVM.* https://llvm.org/docs/ExceptionHandling.html

[35] Texas Instruments. 2015. *TMS320C28x Optimizing C/C++ Compiler.* http://downloads.ti.com/docs/esd/SPRU514I/Content/SPRU514I_HTML/post_link_optimizer.html (accessed Aug 20 2019).

[36] Intel. 2017. Intel Xeon Processor Scalable Family based on Skylake microarchitecture. https://perfmon-events.intel.com/skylake_server.html (accessed 2022).

[37] Teresa Johnson, Mehdi Amini, and David Xinliang Li. 2017. ThinLTO: scalable and incremental LTO. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. 111–121. http://dl.acm.org/citation.cfm?id=3049845

[38] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*. 158–169. https://doi.org/10.1145/2749469.2750392

[39] Andi Kleen. 2016. *An introduction to last branch records.* https://lwn.net/Articles/680985/ (accessed Aug 20 2019).

[40] Konrad Kleine. 2019. 2 tips to make your C projects compile 3 times faster. https://developers.redhat.com/blog/2019/05/15/2-tips-to-make-your-c-projects-compile-3-times-faster

[41] Kumar, Snehasish. 2021. [RFC] Machine Function Splitter. https://groups.google.com/g/llvm-dev/c/RUegaMg-iqc/m/wFAVxa6fCgAJ [Online; accessed 6-August-2021].

[42] Rahman Lavaee, John Criswell, and Chen Ding. 2019. Codestitcher: inter-procedural basic block layout optimization. In *Proceedings of the 28th International Conference on Compiler Construction, CC 2019, Washington, DC, USA, February 16-17, 2019*. 65–75. https://doi.org/10.1145/3302516.3307358

[43] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[44] Chi-Keung 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: feedback-directed and runtime optimization*. IEEE Computer Society, 15.

[45] Chi-Keung Luk, Robert Muth, Harish Patil, Richard Weiss, P Geoffrey Lowney, and Robert Cohn. 2002. Profile-guided post-link stride prefetching. In *Proceedings of the 16th international conference on Supercomputing*. 167–178.

[46] Robert Muth, Saumya K Debray, Scott Watterson, and Koen De Bosschere. 2001. alto: a link-time optimizer for the Compaq Alpha. *Software: Practice and Experience* 31, 1 (2001), 67–101.

[47] Itai Nahshon and David Bernstein. 1996. FDPR: A Post-pass Object-code Optimization Tool. In *International Conference on Compiler Construction*.

[48] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.

[49] Andy Newell and Sergey Pupyrev. 2018. Improved Basic Block Reordering. *CoRR* abs/1809.04676 (2018). arXiv:1809.04676 http://arxiv.org/abs/1809.04676

[50] Maksim Panchenko. 2022. BOLT Open Projects. https://discourse.llvm.org/t/bolt-open-projects/61857 (accessed 2022).

[51] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*. 2–14. https://doi.org/10.1109/CGO.2019.8661201

[52] 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* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/3446804.3446843

[53] Rachel Potvin and Josh Levenberg. 2016. Why Google stores billions of lines of code in a single repository. *Commun. ACM* 59, 7 (2016), 78–87.

[54] Krzysztof Pszeniczny. 2022. llvm-bolt registers .eh_frames which may refer to unmapped sections. https://github.com/llvm/llvm-project/issues/56726 (accessed 2022).
[55] Krzysztof Pszeniczny. 2022. Stripping BOLTed binaries may result in misaligned PT_LOADs. https://github.com/llvm/llvm-project/issues/56738 (accessed 2022).
[56] NIST FIPS PUB. 2001. 140-2: Security requirements for cryptographic modules. *Information Technology Laboratory, National Institute of Standards and Technology* (2001).
[57] Alex Ramirez, Luiz André Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P Geoffrey Lowney, and Mateo Valero. 2001. Code layout optimizations for transaction processing workloads. In *ACM SIGARCH Computer Architecture News*, Vol. 29. ACM, 155–164.
[58] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. 2001. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*. Citeseer.
[59] Han Shen, Rahman Lavaee, Krzysztof Pszeniczny, Snehasish Kumar, Sriraman Tallam, and Xinliang (David) Li. 2022. *Artifacts for "Propeller: A Profile Guided, Relinking Optimizer for Warehouse Scale Applications"*. https://doi.org/10.5281/zenodo.7222794
[60] Amitabh Srivastava and Alan Eustace. 2004. ATOM: A system for building customized program analysis tools. *ACM SIGPLAN Notices* 39, 4 (2004), 528–539.
[61] James Swift. 2017. Crazy Fast Builds Using distcc. https://pspdfkit.com/blog/2017/crazy-fast-builds-using-distcc/
[62] Sriraman Tallam. 2020. *LLD Support for Basic Block Sections*. https://reviews.llvm.org/rG94317878d826 (accessed June 29, 2022).
[63] Ian Lance Taylor. 2008. A New ELF Linker. In *Proceedings of the GCC Developers' Summit*. http://ols.fedoraproject.org/GCC/Reprints-2008/taylor-reprint.pdf
[64] Rui Ueyama. 2017. *LLD - The LLVM Linker*. https://lld.llvm.org/lld
[65] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. 2005. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.* IEEE, 7–12.
[66] Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar R Nair, Mauricio Breternitz, Zhiwei Ying, and Youfeng Wu. 2007. Stardbt: An efficient multi-platform dynamic binary translation system. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*. Springer, 4–15.
[67] Kaiyuan Wang, Greg Tener, Vijay Gullapalli, Xin Huang, Ahmed Gad, and Daniel Rall. 2020. Scalable build service system with smart scheduling service. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 452–462.
[68] Wikipedia contributors. 2021. Monorepo — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Monorepo&oldid=1024603377 [Online; accessed 6-August-2021].
[69] David Williams-King and Junfeng Yang. 2019. CodeMason: Binary-Level Profile-Guided Optimization *(FEAST'19)*. Association for Computing Machinery, New York, NY, USA, 47–53. https://doi.org/10.1145/3338502.3359763
[70] Wired. 2011. Artificial intelligence: it's nothing like we expected - Internet Search. https://www.wired.co.uk/article/artificial-intelligence (accessed 2022).
[71] Ahmad Yasin. 2014. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 35–44.

Received 2022-07-07; accepted 2022-09-22