Online Code Layout Optimizations via OCOLOS

Yuxuan Zhang, *University of Pennsylvania, Philadelphia, PA, 19104, USA*
Tanvir Ahmed Khan, *University of Michigan, Ann Arbor, MI, 48109, USA*
Gilles Pokam, *Intel Corporation, Santa Clara, CA, 95054, USA*
Baris Kasikci, *University of Michigan, Ann Arbor, MI, 48109, USA*
Heiner Litz, *University of California, Santa Cruz, Santa Cruz, CA, 95064, USA*
Joseph Devietti, *University of Pennsylvania, Philadelphia, PA, 19104, USA*

The processor front end has become an increasingly important bottleneck in recent years due to growing application code footprints, particularly in data centers. Profile-guided optimizations performed by compilers represent a promising approach, as they rearrange code to maximize instruction cache locality and branch prediction efficiency along a relatively small number of hot code paths. However, these optimizations require continuous profiling and rebuilding of applications to ensure that the code layout matches the collected profiles. In this article, we propose Online COde Layout OptimizationS (OCOLOS), the first online code layout optimization system for unmodified applications written in unmanaged languages. OCOLOS allows profile-guided optimization to be performed on a running process instead of being performed offline and requiring the application to be relaunched. Our experiments show that OCOLOS can accelerate MySQL by up to 41%.

As the world demands ever more from software, code sizes have increased to keep up. Google, for example, reports annual growth of 20% in the instruction footprint of important internal workloads.[3] This code growth has created bottlenecks in the front end of the processor pipeline, where latency-sensitive structures cannot be easily scaled up—both Intel and AMD today have 32-kB level 1 instruction (L1i) caches, the same as they did a decade ago. Cramming ever more code into a fixed-size L1i leads to a rising number of processor front-end stalls.

To address these front-end stalls, large software companies have turned to profile-guided optimizations[a] (PGOs) from the compiler community that reorganize code within a binary to optimize the utilization of the limited L1i for the common-case control-flow paths. Google's AutoFDO[1] and Propeller,[9] Meta's Binary Optimization and Layout Tool (BOLT),[6,7] and gcc's and clang's built-in PGO passes are popular examples of this approach. While these systems have seen successful deployment at scale, there remain three significant challenges.

First, because PGO is an offline optimization, there is a fundamental lag between when profiling information is collected and when it is used to optimize the code layout. If program inputs shift during this time, previous profiling information is rendered irrelevant or even harmful when it contradicts newer common-case behavior. Maintaining profiles for each input or program phase is prohibitive in terms of storage costs, so profiles are merged together to capture average-case behavior at the cost of input-specific optimization opportunities.

Second, even if we have secured timely profiling information, if the program code itself changes, then it is difficult to map the profiling information onto the

[a]Many optimizations can be driven by profiling information, so the term "PGO" is quite broad. In this article, we use it to refer exclusively to profile-driven *code layout* optimizations.

0272-1732 © 2023 IEEE
Digital Object Identifier 10.1109/MM.2023.3274758
Date of publication 10 May 2023; date of current version 29 June 2023.

new code.[1] Profiling information is captured at the machine code level, and even modest changes to the source code can lead to significant differences in machine code. In large software organizations, code changes can arrive every few minutes for important applications, creating a constant challenge when applying PGO with profiling data collected from version k to the compilation of the latest version k'. Profiling data that cannot be mapped to k' are discarded, leading to missed optimization opportunities.

The third key challenge with offline PGO approaches is that recording, storing, and accessing PGO profiles adds an operational burden to code deployment.

In this article, we propose OCOLOS, a novel system for *online* PGOs in unmanaged languages. OCOLOS performs code layout optimizations at runtime on a running process. By moving PGO from compile time to runtime, we avoid the challenges listed previously. Profile information is always up to date with the current behavior of the program, profiling data always map perfectly onto the code being optimized, and there is no profile management burden since a profile is produced and then immediately consumed. Some managed language runtimes (e.g., Oracle's HotSpot Java Virtual Machine) support online code layout optimizations and achieve similar benefits. We are not aware, however, of any system before OCOLOS that brings the benefits of online PGO to unmanaged code written in languages like C/C++.

To realize the benefits of PGO in the online setting, OCOLOS builds on the BOLT[6,7] offline PGO system, which takes a profile and a compiled binary as inputs and produces a new, optimized binary as the output. OCOLOS, instead, captures profiles during the execution of a *deployed, running* application, uses BOLT to produce an optimized binary, extracts the code from that BOLTed binary, and patches the code in the running process. To avoid corrupting the process, code patching requires careful handling of the myriad code pointers in registers and throughout memory. OCOLOS takes a pragmatic approach that requires no changes to application code, which enables support for complex software like relational databases.

OCOLOS is different from other dynamic binary instrumentation (DBI) frameworks, like Intel Pin,[5] in that OCOLOS 1) focuses on code replacement instead of providing application programming interfaces (APIs) for instrumentation and 2) has a "one-time" cost model where major work is done only during code replacement, and the program runs with native performance once the replacement is complete. Existing DBI frameworks optimize for common code paths with code caches, but the resulting benefits are overshadowed by nontrivial ongoing overheads to intercept control-flow transfers and analyze code on cache fills. Instead, OCOLOS exacts a one-time cost for code replacement, which is readily amortized, along with a small amount of runtime instrumentation on function pointer creation (see the "Continuous Optimization" section).

BACKGROUND

We start with some background on state-of-the-art PGO systems, like BOLT[6,7] and Propeller.[9]

Hardware Performance Profiling

Profile collection is the first step of all PGO workflows. Large-scale deployments generally leverage hardware profiling support, like Intel's last branch record (LBR)[4] facility, which dates back to the Pentium 4 and is widely available. When LBR tracing is enabled, the processor records the program counter (PC) and target of taken branches in a ring buffer. The recording overhead via LBR is negligible, and software can sample the buffer to learn the branching behavior of an application. By aggregating these samples, the approximate frequency of the branch-taken/-not-taken paths through the code can be reconstructed. With these branch frequencies in hand, we can make intelligent decisions about optimizing the code layout.

Basic Block Reordering

Whenever programs contain `if` statements, the compiler must decide how to place the resulting basic blocks into a linear order in memory.[8] The ideal layout places the common-case blocks consecutively, maximizing L1i and instruction translation lookaside buffer locality while reducing pressure on the branch prediction structures.

Consider the example program in Figure 1. Assuming both conditions are typically true, the shaded basic blacks constitute the common-case execution. A naive layout that places the blocks from each `if` statement together results in two taken branches (shown by arrows). The optimal layout, however, avoids any taken branches and results in better performance.

BOLT

BOLT[6,7] is a postlink optimization tool built in the Low-Level Virtual Machine (LLVM) framework, which operates on compiled binaries. Given LBR profiling information and a binary, BOLT decompiles the binary, performs a series of optimizations, and then performs code generation to emit a new BOLTed binary. Of BOLT's optimizations, basic block reordering provides, by far, the biggest

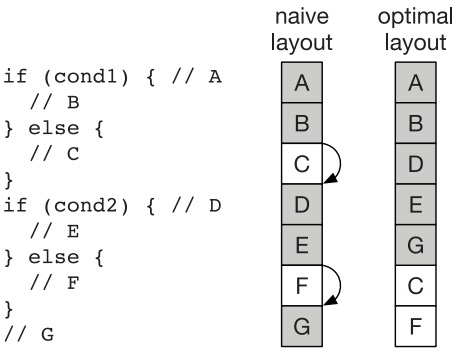

FIGURE 1. Example program that benefits from profile-guided optimization (PGO).

speedup.[6] One helpful feature of BOLTed binaries is that, if a function f grows in size after optimization (e.g., reordering basic blocks may cause branch offsets to grow and require larger instructions), f will be placed in a new text section of the binary. Other functions whose sizes do not grow are optimized in place.

CHALLENGES

A well-known and intuitive challenge with offline profile-based optimizations, like conventional PGO, is ensuring that the gathered profile data are of high quality. In experiments with MySQL, we found that, with the Sysbench read_only input, feeding the profiling data from that same read_only input to BOLT (which is like testing on the training data) results, unsurprisingly, in the biggest speedup compared with using any other profiling data. However, the worst case input resulted in a 21% *slowdown* compared to the best profile, showing that using poor profiling data can exact a high price.

OCOLOS requires modification of the code pointers at runtime to perform its optimizations. First, we distinguish between code pointers that refer to the starting address of a function versus those that reference a specific instruction within a function (e.g., the target of a conditional branch). We discuss function starting addresses first. Functions can call each other via direct calls, encoding the callee function's starting address as a PC-relative offset. There may also be indirect calls via function pointers stored in a virtual table (v-table)[b] or programmer-created function pointers stored on the stack or heap, in global variables, or in processor registers.

[b]A virtual function/method table (v-table) is used to implement dynamic dispatch or virtual functions in object-oriented languages. The table itself stores function pointers to the methods of a class.

Code pointers that do not refer to the start of a function are also commonplace. Jump and conditional branch instructions within a function reference code locations via PC-relative offsets. Sometimes, indirect jumps rely on compile-time constants that are used to compute a code pointer at runtime, e.g., in the implementation of some switch statements. Return addresses on the stack are code pointers to functions that are on the call stack. Each thread's PC is a pointer to an instruction in the currently running function. A thread may be blocked doing a system call, in which case its PC is effectively stored in the saved context held by the operating system. The libc setjmp/longjmp API can be used to create programmer-managed code pointers to essentially arbitrary code locations.

Thus, the address space of a typical process contains *a large number* of code pointers. Tracking them so that they can be updated if a piece of code moves is essentially impossible for any serious program. Thus, OCOLOS retains the original code within a process and adds optimized code at a new location, patching up as many code pointers as possible to steer execution toward the optimized code in the common case.

OCOLOS

In Figure 2, we show a high-level overview of the steps OCOLOS performs to optimize the code of a target process at runtime. First, we gather profiling information from the target process via Linux's perf utility to extract LBR samples ❶. Then, we invoke BOLT to build the BOLTed binary ❷; pause the target process via Linux's ptrace API ❸; inject code ❹; update pointers to refer to the injected code ❺; and, finally, resume the process ❻. Note that steps ❶ and ❷, which consume the most time, are done concurrently in the background while the target process continues to run. Though operations like running BOLT are CPU intensive, they compete for cycles with the target process for only a limited time. Steps ❸–❺ are done synchronously while the target process is paused. Steps ❶ and ❷ are largely taken from prior work, so we focus on ❸–❺, which are the core of OCOLOS.

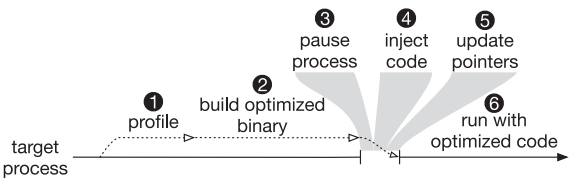

FIGURE 2. Main steps Online Code Layout Optimizations (OCOLOS) takes to optimize a target process.

To better describe key operations within OCOLOS, we first describe the important regions of the address space of the target process, shown in the left part of Figure 3(a). The code from the *original binary* we refer to as C_0, which here consists of three functions: a_0, b_0, and c_0. A v-table contains a pointer to b_0. Finally, each thread's stack is also important, as it contains the return addresses of currently executing functions. In Figure 3(a), c_0 is on the call stack.

OCOLOS takes as input an *optimized binary*, with modified code for functions in C_0 or code for entirely new functions. While OCOLOS's code replacement ultimately requires a short stop-the-world period (see the "Updating Code Pointers" section) to modify code and update code pointers, OCOLOS performs some bookkeeping in advance. In particular, OCOLOS parses the original binary offline to identify the locations of all direct call instructions. OCOLOS patches these calls at runtime, but identifying the call sites a priori abbreviates the stop-the-world period.

Adding Code

As we described in the "Challenges" section, finding and updating all code pointers is fraught with corner cases. This leads to the first principle guiding OCOLOS's design:

Principle 1: Preserve the addresses of C_0 instructions.

Instead of updating the code of a function in place, OCOLOS injects a new version of the code C_1 into the address space while leaving the original code intact [see Figure 3(a)]. OCOLOS then changes a subset of code pointers within C_0 to redirect execution to the C_1 code. The remaining code pointers are not perturbed and continue to point to C_0 code.

Updating Code Pointers

When patching code pointers to make the C_1 code reachable, OCOLOS follows our second design principle:

Principle 2: Run C_1 code in the common case.

OCOLOS executes code from C_0 instead of C_1 *occasionally* to ensure correctness. However, the more frequently OCOLOS executes code from C_0, the more it reduces the potential performance gains C_1 can provide. Therefore, we seek to make C_1 the common case.

Since our goal for the current version of OCOLOS is minimizing (but not eliminating) the time spent in C_0, OCOLOS updates as many code pointers to refer to C_1 as it is worthwhile to update. Note, first of all, that hot code gets optimized by BOLT and resides in C_1. Direct calls in C_1 will already refer to C_1 (e.g., c_1 calls b_1) and do not require updating.

Figure 3(a) illustrates changes OCOLOS makes. We update function pointers in v-tables and direct calls in C_0 for functions on the call stack (like c_0). Recall that these C_0 changes preserve instruction addresses, honoring our first design principle. We found that, in practice, updating direct calls in *all* functions (i.e., including those, like a_0, not on the stack) does not improve performance—because functions like a_0 are cold—though it does slow code replacement.

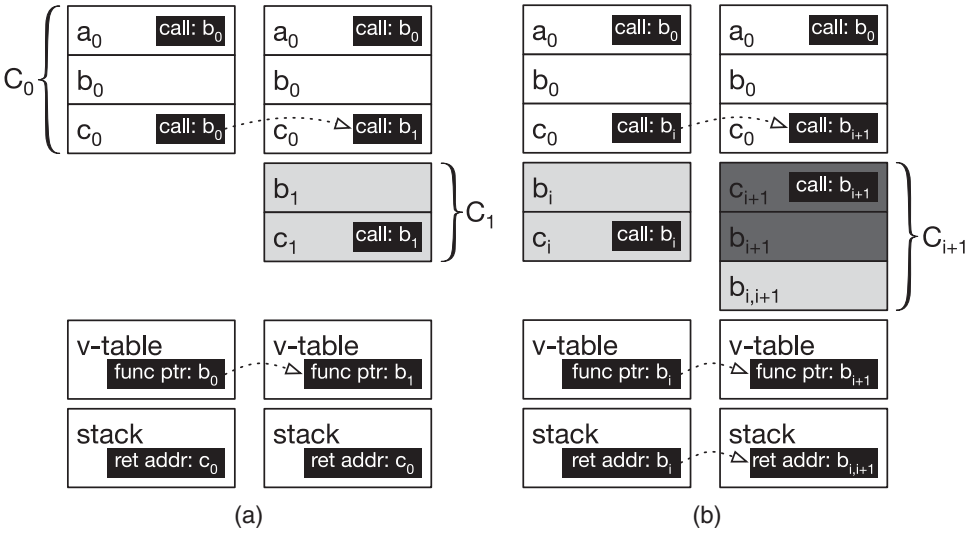

FIGURE 3. (a) Starting state of the address space (left) and state after code replacement (right). (b) Before (left) and after (right) continuous optimization.

We could additionally seek out function pointers in registers and memory, though doing so would require expensive always-on runtime instrumentation to track their propagation throughout the program's execution. This tracking would violate OCOLOS's "fixed costs only" cost model:

Principle 3: Code replacement can incur fixed costs but must avoid all possible recurring costs.

Our experiments show that leaving these remaining function pointers (which our workloads do contain) pointing to C_0 code is fine since C_0 code does not execute for very long before it encounters a direct call or a virtual function call that steers execution back to C_1.

Continuous Optimization

A natural use case for OCOLOS is to perform *continuous optimization*, whereby OCOLOS can replace C_1 with C_2 and C_i with C_{i+1} more generally. These subsequent code versions C_i can be generated by periodic reprofiling of the target process to account for program phases, daily patterns in workload behavior like working versus at-home hours, and so on. OCOLOS can perform continuous optimization largely through the same code replacement algorithm described earlier, though functions on the stack and function pointers require delicate handling, as will be explained.

A key challenge with continuous optimization is the need to *replace* code instead of just adding new code elsewhere in the address space. If we continuously add code versions without removing old versions, the code linearly grows over time, wasting dynamic random-access memory (RAM) and hurting front-end performance. To address this challenge, we introduce a *garbage collection* mechanism for removing dead code. We define dead code as code that can no longer be reached via any code pointers and, hence, is safe to remove.

Instead of waiting for code version C_i to naturally become unreachable, as in conventional garbage collection, we can proactively update code pointers to *enforce* the unreachability of C_i. OCOLOS patches v-tables, direct calls from C_0, return addresses on the stack, and threads' PCs to refer to the incoming C_{i+1} code instead, as described in the "Updating Code Points" section and illustrated in Figure 3(b).

Return Addresses

Code pointers in return addresses and in threads' PCs may reference C_i, so OCOLOS must update these references to point to C_{i+1}. To update these references, OCOLOS first crawls the stack of each thread via $\mathrm{libunwind}$ to find all return addresses. OCOLOS examines RIP for each thread via ptrace. Collectively, this examination provides OCOLOS with the set of *stack-live* functions that are currently being executed. If any stack-live function is in C_i [such as b_i in Figure 3(b)], OCOLOS must copy its code to C_{i+1}. While there may be an optimized version b_{i+1} in C_{i+1}, it is challenging to update the return address to refer to b_{i+1} because, in general, the optimizations applied to produce b_{i+1} can have a significant impact on the number and order of instructions within a function.

Thus, OCOLOS makes a copy of b_i in C_{i+1}, which we call $b_{i,i+1}$ to distinguish it from the more optimized version b_{i+1}. The $b_{i,i+1}$ version may need to have a different starting address than b_i, so OCOLOS updates PC-relative addressing within $b_{i,i+1}$ to accommodate its new location. OCOLOS must also update the return address to refer to the appropriate instruction within $b_{i,i+1}$, but OCOLOS can treat the original return address into b_i as an offset from b_i's starting address and then use this offset into $b_{i,i+1}$ to compute the new return address.

While copying b_i to $b_{i,i+1}$ is a key part of enabling continuous optimization, it does not improve performance of the currently running call to b_i since the code is the same. However, subsequent calls are likely to reach b_{i+1} instead via other code pointers, like the v-table in Figure 3(b).

Function Pointers

Apart from return addresses, function pointers may also point to C_i. Instead of trying to track down and update these pointers while moving from C_i to C_{i+1}, OCOLOS enforces a simpler invariant that a program cannot create function pointers to C_i code in the first place—rather, function pointers must always refer to C_0. This allows function pointers to propagate freely throughout the program without the risk that they will be broken during code replacement.

OCOLOS enforces this invariant via a simple LLVM compiler pass that instruments function pointer creation sites to map pointers to C_i back to the corresponding C_0 function instead. This instrumentation has low cost: MySQL running the $\mathrm{read\_only}$ input creates just 45 function pointers per millisecond on average.

Current Status

Having avoided function pointers to C_i, OCOLOS is able to update all other references to C_i code to refer to the incoming C_{i+1} code instead. Thus, OCOLOS can safely overwrite C_i code. While BOLT does not directly support the reoptimization of a BOLTed binary, which initially prevented continuous optimization from being

realized, we have recently learned about an alternative workflow with BOLT that does allow for reoptimization.

We are in the process of updating OCOLOS to leverage this. BOLT's strategy for offline reoptimization is to translate profiling information from a BOLTed binary to appear as if it were from the original non-BOLTed binary and then reapply BOLT to the original binary with the translated profile. To facilitate this, BOLT creates a detailed basic-block-level translation table.

One technical hurdle we have already overcome in continuous optimization is translating profiling information gathered from an OCOLOS process like that in Figure 3(a), which is a mix of C_0 and C_1 code, to appear as if it contains only C_0, which is the format that BOLT needs. We have extended BOLT to handle cases such as when b_1 is moved by BOLT but OCOLOS retains b_0 as well, and, thus, both appear in the profile.

EVALUATION

We run our experiments on a two-socket Intel Broadwell Xeon E5-2620v4 server with eight cores and 16 threads per socket (16 cores and 32 threads total) running at 2.1 GHz with 128 GB of RAM. Our benchmarks are MySQL 8.0.28, MongoDB 6.0.0, Memcached 1.6.12, and Verilator 3.904.

Performance

Figure 4 shows the throughput improvement OCOLOS provides across our set of benchmarks. We compare OCOLOS to three baselines. *Original* is the performance of the original binary, compiled with only static optimizations (nothing profile guided). *BOLT oracle input* is the performance offline BOLT provides when profiling and running the same input. Finally, *BOLT average-case input* is the performance offline BOLT achieves when aggregating profiles from all inputs and then running on the input shown on the *y*-axis. We show throughput normalized to *original*.

Figure 4 shows that OCOLOS uniformly improves performance over the original binary by up to 1.41× on MySQL `read_only`, 1.29× on MongoDB `read_update`, 1.05× on Memcached, and 2.20× on Verilator. The results for *BOLT oracle input* represent an upper bound for OCOLOS's performance since BOLT has access to the oracle profiling data and ensures that all code pointers refer to optimized code, not just a judicious subset of them as with OCOLOS (see the "Updating Code Pointers" section). However, on average, OCOLOS is close to the BOLT oracle's performance, with a slowdown of just 4.6 points. Compared to offline BOLT with an average-case profile, OCOLOS is 8.9 points faster on average, as different inputs tend to exhibit contradictory control-flow biases that cancel each other out.

MySQL Case Study

To better understand the performance impact of OCOLOS's code replacement mechanism, we performed an experiment with MySQL with Sysbench's oltp_read_only input reporting the client's transaction throughput every second, shown in Figure 5. Region 1 is a warm-up period, after which `perf` profiling begins collecting LBR samples for 1 s (region 2). In region 3, `perf2bolt` processes the LBR samples, and then BOLT generates the optimized binary. This is a CPU-intensive phase, causing a reduction in throughput after the 30-s mark. In region 4, OCOLOS performs code replacement, which

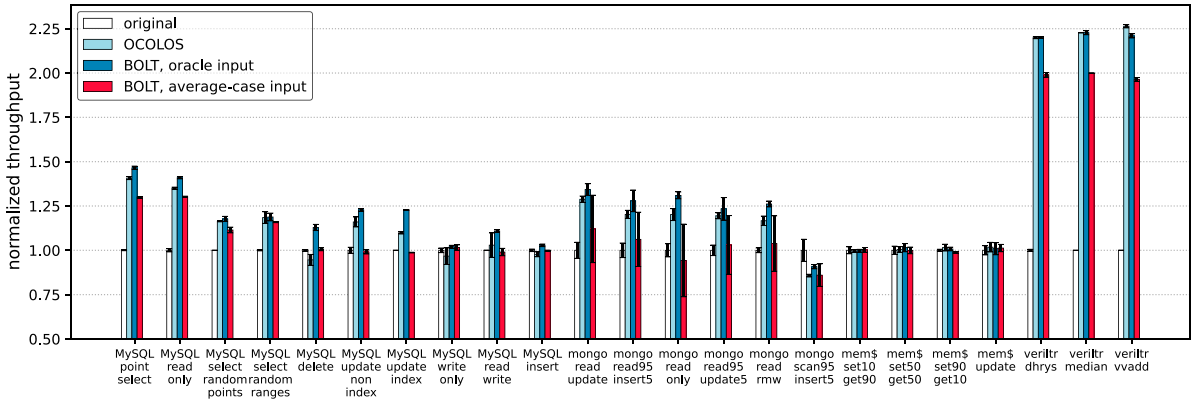

FIGURE 4. Performance of OCOLOS (light blue bars) compared to Meta's Binary Optimization and Layout Tool (BOLT) using an oracle profile of the input being run (dark blue bars), and BOLT using an average-case profiling input aggregated from all inputs (red bars). All bars are normalized to the original non-PGO binaries (white bars).

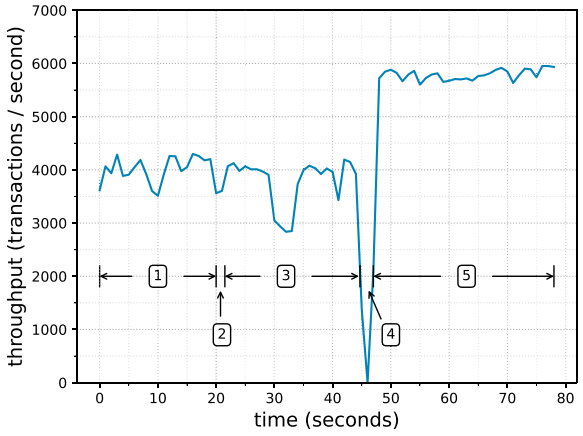

FIGURE 5. Throughput of MySQL read_only before, during, and after code replacement.

entails a stop-the-world phase of 1.9 s. After that, in region 5, MySQL's parallel execution resumes with the optimized code in place, boosting performance by 1.41× compared to region 1.

We believe there is scope to significantly reduce region 4 latency by replacing some inefficient scripts with compiled code and parallelizing the code replacement routines, which are all currently serial.

IMPACT

Despite PGO being a long-standing component of optimizing compilers like gcc and clang, barriers to adopting PGO in practice remain high. Deployment at hyperscale in systems like Meta's BOLT[7] and Google's AutoFDO[1] and Propeller[9] has reignited interest in PGO research but not fundamentally improved usability. Offline profiling is still required, and the binary must be rebuilt or rewritten based on the profile. Matching a profile to a binary is a fragile process, and even small code changes can cause a profile to map poorly. PGO, thus, remains a tool used only by those who care deeply about performance and are willing to deal with the complexity of a PGO-enabled build and production environment.

Democratizing PGO

OCOLOS's primary long-term impact will be to democratize the use of PGO and provide its performance benefits to a wide range of users automatically and by default. When PGO can be deployed at runtime via OCOLOS without any need for developers to adjust their code, their build system, or their production environment, taking advantage of PGO can become the default option instead of an expensive detour to higher performance. In the following sections, we explain in more detail how OCOLOS can bring this about.

Simpler Deployment

By profiling and optimizing the currently running process, OCOLOS ensures that profile information can be produced and consumed on the local machine. No persistent storage or management is required. This keeps operational complexity low, avoiding dependencies on storage services, which must themselves be provisioned for PGO to function. Adopting a technology like OCOLOS can, thus, actually reduce overall system complexity compared to a conventional offline PGO system like BOLT or Propeller.

Another important consequence of OCOLOS's simple deployment is that many more software projects can adopt PGO successfully. Smaller teams, or projects that are important but not under active development, struggle to justify the human cost of using offline PGO since there are ongoing costs to recording, storing, and retiring profiles and deploying the optimized binaries that are produced. OCOLOS provides a one-time cost for adoption: installing the requisite packages and then launching the workload under OCOLOS. Everything after that is handled automatically. While offline PGO, with its marginally superior performance when the input is known in advance, may remain in use for very popular workloads that can justify the complexity, OCOLOS can target a long tail of workloads and provide significant aggregate performance gains across a wide user base.

Continuous Optimization

By enabling online PGO, OCOLOS paves a path for continuous optimization. Prior work[2] has shown that applying PGO on top of a binary already optimized by PGO can provide significant additional performance benefits. However, due to the offline nature of existing PGO, such benefits are still outside the scope of modern data center applications. OCOLOS is a natural framework within which we can unlock the compounding benefits of repeated PGO.

Reducing the Data Center Tax

Due to the extremely diverse nature of data center applications, there is no small set of "hotspots" to optimize with traditional hardware acceleration mechanisms.[3] Instead, these applications share common building blocks (the "data center tax") in the form of popular shared libraries. Unfortunately, existing PGO cannot optimize these libraries due to their variance across different applications.[1] As OCOLOS moves PGO

from offline to online, OCOLOS brings these "data center tax" components within the reach of PGO, allowing the tax to be reduced in an application-specific way.

Beyond PGO

OCOLOS is a generic framework for updating the code of a running process at a one-time cost. OCOLOS's ability to steer most (but not necessarily all) execution toward the updated code is well suited to specialization for vector extensions or accelerators that happen to be available at runtime. Logging or other program instrumentation could be selectively added to a process to facilitate debugging in production; afterward, the instrumentation can be completely removed to restore native performance. Our code is open source[c] to facilitate exploring these and other use cases.

CONCLUSION

We have described the design and implementation of OCOLOS, the first online PGO system for unmanaged code. OCOLOS provides the performance benefits of a classic offline PGO compilation flow but applied to a running process. By operating at runtime, OCOLOS always profiles the most up-to-date behavior of the program and avoids problems with mapping the profile to a target binary that can frustrate offline PGO.

ACKNOWLEDGMENTS

This work was supported by gifts from Intel Labs, joint NSF/Intel Grants 2010810 and 2011168, NSF Grant 1942754, a Rackham Predoctoral Fellowship, and the Applications Driving Architectures Research Center, a Joint University Microelectronics Program Center cosponsored by the Semiconductor Research Corporation and the Defense Advanced Research Projects Agency. We thank Maksim Panchenko and Guilherme Ottoni from Meta for helpful discussions about BOLT.

REFERENCES

1. D. Chen, D. X. Li, and T. Moseley, "AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications," in *Proc. Int. Symp. Code Gener. Optim.*, ACM, 2016, pp. 12–23, doi: 10.1145/2854038.2854044.
2. W. He, J. Mestre, S. Pupyrev, L. Wang, and H. Yu, "Profile inference revisited," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–24, Jan. 2022, doi: 10.1145/3498714.
3. S. Kanev et al., "Profiling a warehouse-scale computer," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 158–169, doi: 10.1145/2749469.2750392.
4. A. Kleen. *An Introduction to Last Branch Records*. (2016). LWN.net. [Online]. Available: https://lwn.net/Articles/680985/
5. C.-K. Luk et al., "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 190–200, doi: 10.1145/1064978.1065034.
6. M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "BOLT: A practical binary optimizer for data centers and beyond," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, Washington, DC, USA: IEEE Press, 2019, pp. 2–14, doi: 10.1109/CGO.2019.8661201.
7. M. Panchenko, R. Auler, L. Sakka, and G. Ottoni, "Lightning BOLT: Powerful, fast, and scalable binary optimization," in *Proc. 30th ACM SIGPLAN Int. Conf. Compiler Construction*, 2021, pp. 119–130, doi: 10.1145/3446804.3446843.
8. K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 1990, pp. 16–27, doi: 10.1145/93542.93550.
9. H. Shen, K. Pszeniczny, R. Lavaee, S. Kumar, S. Tallam, and X. D. Li, "Propeller: A profile guided, relinking optimizer for warehouse-scale applications," in *Proc. 28th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, pp. 617–631, vol. 2, 2023, doi: 10.1145/3575693.3575727.

[c]https://github.com/upenn-acg/ocolos-public

YUXUAN ZHANG is working toward her Ph.D. degree with the Computer and Information Science Department, University of Pennsylvania, Philadelphia, PA, 19104, USA. Her research interests include investigating and mitigating the processor front-end bottleneck at runtime. Zhang received her master of science degree in electrical engineering from the University of Michigan, Ann Arbor. She is a student member of the Association for Computing Machinery (ACM). Contact her at zyuxuan@seas.upenn.edu.

TANVIR AHMED KHAN is a final-year Ph.D. candidate and Rackham Predoctoral Fellow at the University of Michigan, Ann Arbor, MI, 48109, USA. His research interests include bringing together techniques from computer architecture, compilers, and operating systems to enable efficient data center processing. He is a student member of ACM. Contact him at takh@umich.edu.

GILLES POKAM is a principal engineer at Intel, Santa Clara, CA, 95054, USA, where he works on designing CPU core front-end microarchitecture mechanisms for next-generation server

processors to improve the performance and architecture efficiency of data-center-scale applications. His research interests include microarchitecture and its interactions with system software and security. Gilles received his Ph.D. degree in computer science from Institut National de Recherche en Sciences et Technologies du Numérique (INRIA) and the University of Rennes, France. He is a Member of IEEE and a member of ACM. Contact him at gilles.a.pokam@intel.com.

BARIS KASIKCI is an assistant professor of computer science and engineering at the University of Michigan, Ann Arbor, MI, 48109, USA. His research interests include building efficient and trustworthy computer systems using a combination of techniques from operating systems, programming languages, and computer architecture. Kasikci received his Ph.D. degree in computer science from École Polytechnique Fédérale de Lausanne (EPFL). He is a Member of IEEE and a member of ACM. Contact him at barisk@umich.edu.

HEINER LITZ is an assistant professor of computer science and engineering at the University of California, Santa Cruz, Santa Cruz, CA, 95064, USA. His research interests include designing novel computer architectures and power-efficient infrastructure for data center systems. Litz received his Ph.D. degree in computer science from the University of Mannheim. He is a Member of IEEE and a member of ACM. Contact him at hlitz@ucsc.edu.

JOSEPH DEVIETTI is an associate professor in the Computer and Information Science Department, University of Pennsylvania, Philadelphia, PA, 19104, USA. His research interests include making use of hardware performance counters to inform performance optimizations statically and at runtime. Devietti received his Ph.D. degree in computer science and engineering from the University of Washington. He is a Member of IEEE and a member of ACM. Contact him at devietti@cis.upenn.edu.