2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)

2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00045

OCOLOS: Online COde Layout OptimizationS

Yuxuan Zhang* Tanvir Ahmed Khan† Gilles Pokam‡ Baris Kasikci† Heiner Litz§ Joseph Devietti*
*University of Pennsylvania †University of Michigan ‡Intel Corporation §University of California, Santa Cruz
*{zyuxuan, devietti}@seas.upenn.edu †{takh, barisk}@umich.edu
‡gilles.a.pokam@intel.com §hlitz@ucsc.edu

***Abstract*—The processor front-end has become an increasingly important bottleneck in recent years due to growing application code footprints, particularly in data centers. First-level instruction caches and branch prediction engines have not been able to keep up with this code growth, leading to more front-end stalls and lower Instructions Per Cycle (IPC). 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. If an application's code is frequently updated, it becomes challenging to map profiling data from a previous version onto the latest version, leading to ignored profiling data and missed optimization opportunities.**

In this paper, we propose 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. By running online, profile data is always relevant to the current execution and always maps perfectly to the running code. OCOLOS demonstrates how to achieve robust online code replacement in complex multithreaded applications like MySQL and MongoDB, without requiring any application changes. Our experiments show that OCOLOS can accelerate MySQL by up to 1.41×, the Verilator hardware simulator by up to 2.20×, and a build of the Clang compiler by up to 1.14×.

I. INTRODUCTION

As the world demands ever more from software, code sizes have increased to keep up. Google, for example, reports annual growth of 30% in the instruction footprint of important internal workloads [6, 45]. This code growth has created bottlenecks in the front-end of the processor pipeline [50], as the sizes of front-end hardware resources close to the processor have been relatively static over time [52]. Figure 1 shows that, despite Moore's Law, the per-core L1 instruction cache (L1i) capacity of server microarchitectures from Intel and AMD has remained effectively constant (literally so in Intel's case) over the past 15 years, because the L1i is so latency-critical [23]. As we attempt to cram ever more code into a fixed-size L1i, strain on the processor front-end is inevitable. The inability to deliver instructions to the processor leads to front-end stalls that tank IPC and end-to-end application performance as even advanced techniques like out-of-order processing cannot hide these stalls [58].

To address front-end stalls, large software companies have turned to Profile-Guided Optimizations[1] (PGO) 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 (we describe these compiler optimizations in detail in Section II). Google's AutoFDO [10] and Propeller [29], Meta's BOLT [76, 77], 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 exist three significant challenges.

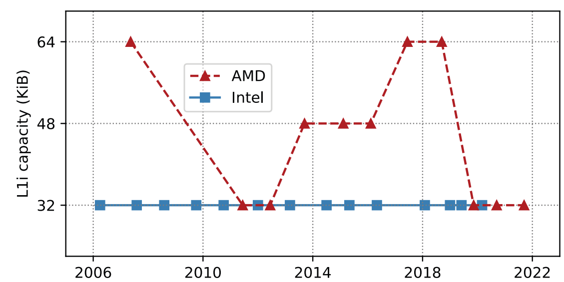

Fig. 1: AMD & Intel per-core L1i capacity over time

First, the results of PGO are only as good as the profiling information that drives PGO's optimization decisions [53, 108, 110]. PGO requires relevant, fresh profiling information to produce high-performance code layouts [72]. However, PGO is an offline optimization, applied either during compilation (AutoFDO, gcc, clang) or to a compiled binary (BOLT and Propeller), creating a fundamental lag between when profiling information is collected and when it is used to optimize the code layout [10]. If program inputs shift during this time, previous profiling information is rendered irrelevant or even harmful when it contradicts newer common-case behavior [69, 105]. 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 [104, 111].

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 new code [10]. By its nature, profiling information is captured at the machine code level, and

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

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00045

even modest changes to the source code can lead to significant differences in machine code [36]. To make things worse, in large software organizations, code changes can arrive every few minutes for important applications [3]. Since we always need to deploy the latest version of an application, there is 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' is discarded, causing us to miss optimization opportunities [22].

The third key challenge with offline PGO approaches is that recording, storing, and accessing PGO profiles adds an operational burden to code deployment. In particular, for end-user mobile applications, managing profiles can itself be a non-trivial task [69, 90].

In this paper, we propose OCOLOS, a novel system for *online* profile-guided optimizations in unmanaged languages. OCOLOS performs code layout optimizations at run time, on a running process. By moving PGO from compile time to run time, we avoid the challenges listed previously. Profile information is always up-to-date with the current behavior the program is exhibiting. OCOLOS also supports *continuous optimizations* to keep up with input changes over time. In OCOLOS, profiling data always maps perfectly onto the code being optimized since we profile and optimize the currently-running process. There is no burden of profile management, as the profile is produced and then immediately consumed. Some managed language runtimes (*e.g.*, Oracle HotSpot JVM [39] and Meta HHVM [73, 74]) 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 [76, 77] 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 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 [64], DynamoRIO [8, 35], and Valgrind [71] in that OCOLOS 1) focuses on code replacement, instead of providing APIs for instrumentation, and 2) has a "1-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 would be unsuitable for our online PGO use-case because programs running under, say, Pin experience a non-trivial ongoing overhead to intercept control-flow transfers and maintain the code cache. The performance benefits of the improved code layout would, in practice, often be outweighed by these ongoing overheads. Instead, OCOLOS exacts a 1-time cost for code replacement which is readily amortized, along with a small amount of run-time instrumentation on function pointer creation (see Section IV-C).

For some short-running programs, even the 1-time cost of OCOLOS is too high to be effectively amortized at run time. To address this problem, we propose BATCH ACCELERATOR MODE (BAM), a technique that allows batch workloads consisting of a large collection of short-running processes, like large software builds, to benefit from OCOLOS. BAM works by profiling initial executions of a binary, generating an optimized binary with BOLT, and using that optimized binary in subsequent executions. BAM operates transparently to the workload, via `LD_PRELOAD` injection, allowing BAM to accelerate builds of the Clang compiler without any changes to Clang code or build scripts.

While in this paper we focus on using OCOLOS to enable online PGO, we envision OCOLOS also being applicable in a range of other cases such as performance optimization and security. We will open-source OCOLOS to facilitate this exploration.

In summary, this paper makes the following contributions:

- We describe the design of OCOLOS, the first *online* profile-guided code layout optimization tool for unmanaged code.
- We show how to perform online code replacement efficiently in unmodified, large-scale C/C++ programs.
- We evaluate OCOLOS on a series of big-code applications like MySQL and MongoDB, demonstrating speedups of up to 1.41× on MySQL.
- We evaluate a variant of OCOLOS targeted at batch workloads with many short-running processes, demonstrating a 1.14× speedup on a from-scratch Clang build.

II. BACKGROUND

In this section, we provide the necessary background of PGO passes implemented in tools like AutoFDO [10] and BOLT [76, 77], which are now the state of the art at all major cloud companies including Google and Meta.

A. Hardware Performance Profiling

Profile collection is the first step of all PGO workflows. There are two different methods of profile collection: 1) through compiler instrumentation of branch instructions (*e.g.*, Clang and GCC), and 2) through hardware support (*e.g.*, Intel's Last Branch Record [56] and Processor Trace [55]). Due to the high overheads of compiler instrumentation [10], cloud providers generally leverage hardware profiling support [10, 13, 18, 67, 76, 77]. For instance, Intel's LBR [56] facility, which dates back to the Netburst architecture (Pentium 4), is widely available at this point. When LBR tracing is enabled, the processor records the Program Counter (PC) and target of taken branches in a ring buffer[2]. The recording overhead via LBR is negligible [43, 67] and software can then sample this ring buffer to learn the branching behavior of an application. The Linux `perf` utility

[2]The LBR buffer in Skylake and newer cores has 32 entries.

provides simple access to LBR sampling, including the ability to start and stop LBR recording of arbitrary running processes.

Each LBR sample represents a snippet of the program's control flow. By aggregating these snippets, the approximate frequency of 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 as described below.

B. Basic Block Reordering

Basic block reordering is typically the most impactful PGO code transformation [76]. Whenever programs contain if statements, the compiler must decide how to place the resulting basic blocks into a linear order in memory [80]. Without profiling information, the compiler must use some heuristics to decide on a layout based on static code properties [7, 9, 15, 44, 66, 85, 107, 112], often leading to sub-optimal results [69].

The ideal layout places the common-case blocks consecutively, maximizing L1i and instruction Translation Lookaside Buffer (iTLB) locality while reducing pressure on the branch prediction mechanism [72]. In particular, by linearizing the code, the number of taken branches is minimized, reducing the pressure on the Branch Target Buffer (BTB) which only stores information about taken branches [40, 41, 48, 98]. Consider the example program in Figure 2. Assuming both conditions are typically true, shaded basic blacks constitute the common-case execution. A naive layout which 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.

```
if (cond1) { // A
  // B
} else {
  // C
}
if (cond2) { // D
  // E
} else {
  // F
}
// G
```

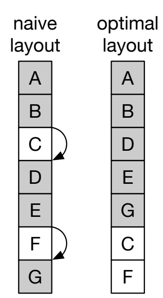

Fig. 2: Example program which benefits from PGO

C. Function Reordering

Function reordering optimizes the linear order of functions within a binary to take advantage of caller-callee relationships. This optimization first uses profiling information to construct a call graph and annotates edges with the frequency of calls.

The classic Pettis-Hansen (PH) algorithm [80] puts functions next to each other if they call or are called by each other frequently. While the PH algorithm uses a greedy approach to place the most frequently-invoked functions adjacent to each other, it makes no distinction between callers and callees.

The C^3 [75] algorithm improves upon Pettis-Hansen by placing callers before callees, which is especially helpful in asymmetric calling relationships where A calls B frequently but B never calls A. This allows C^3 to move the target of a function call closer to the call instruction itself, improving L1i and iTLB locality beyond what PH can provide.

Sometimes PGO passes will incorporate additional optimizations, such as function inlining or peephole optimizations. However, nearly all of the performance benefit of PGO passes comes from basic block reordering and function reordering [76].

D. BOLT: Binary Optimization & Layout Tool

BOLT [76, 77] is a post-link optimization tool built in the LLVM framework, which operates on compiled binaries. The BOLT workflow begins with gathering profiling information. Though BOLT can use a variety of profile formats, LBR samples are preferred. Armed with the profile and the original, non-BOLTed binary, BOLT decompiles the machine code into LLVM's low-level Machine Intermediate Representation (MIR) format, not to be confused with the more commonplace LLVM IR. BOLT performs a series of optimizations, including basic block reordering and function reordering, at the MIR level before performing code generation to emit a new, BOLTed binary.

The layout of a BOLTed binary is unconventional in a few ways. First, cold functions are left in-place in the original .text section of the binary, which is renamed to the bolt.org.text section. These cold functions are subject to small peephole optimizations but their starting addresses do not change and their basic blocks are not reordered because there was insufficient profiling information to justify stronger optimizations. The hot functions, however, are heavily optimized by BOLT (via basic block and function reordering) and are moved to a new .text section at a higher address range. Additionally, BOLT may perform hot-cold code splitting, where the cold basic blocks of a hot function f are not stored contiguously with the hot blocks for f, but are instead exiled to another region of the binary with other cold blocks from other hot functions. Functions that are entirely cold are not worth splitting in this way, and have their code stored contiguously in the bolt.org.text section.

III. Challenges

A well-known and intuitive challenge with offline profiling-based optimizations like conventional PGO is ensuring that the gathered profile data is of high quality [11, 53, 54, 106]. Profiling with one program input and then running on a different input can lead to many sub-optimal optimization decisions [2, 51]. We validate this effect experimentally in Section III-A.

OCOLOS offers a solution to offline PGO's input sensitivity: since OCOLOS profiles and optimizes a running process, the profile data is always for the current binary and the current inputs. However, performing code replacement at run time introduces other challenges. Chief among them is that changing code can break any explicit or implicit *code pointers* (pointers to other instructions, not data) that referenced the changed code. In Section III-B, we catalog the myriad sources of code pointers in a running process, to better motivate the design of OCOLOS in Section IV which can update or preserve these code pointers as necessary.

A. Input Sensitivity

Figure 3 quantifies the sensitivity of BOLT's performance to the quality of its profile data. The bars show along the x-axis the throughput of a BOLTed MySQL binary running the read_only input from Sysbench [1]. The y-axis lists the Sysbench input used to provide profile data to BOLT. Thus, the bottom bar shows the performance when profiling the insert input, BOLTing the binary, and then running with the read_only input. For reference, the dashed line shows the performance of the original MySQL binary without BOLT optimizations. While BOLT improves performance regardless of the training input used, the worst profile (insert) is 21% slower than the best profile (read_only). Aggregating all profiling inputs (the bar labeled *all*) experiences some destructive interference between profiles and is about 8% slower than the best profile. Because OCOLOS (shown with the solid line) runs online instead of ahead of time, it always profiles the current input, and achieves high performance comparable to the best profile.

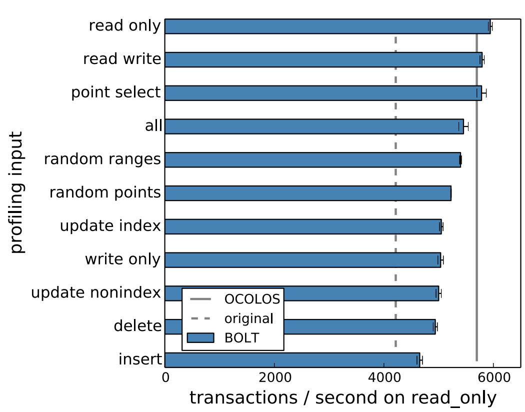

Fig. 3: Performance achieved when running MySQL with the Sysbench read_only input, using BOLT to produce a binary from the given profiling input or, with the *all* bar, from profiles of all inputs combined.

B. Challenges of Changing Code Pointers

OCOLOS requires modification of code pointers at run time to perform its optimizations. To better understand the challenges of changing these code pointers, we first discuss the different flavors of code pointers that arise in a running process. The conventional compilation flow of offline PGO deals only with static code, so many of the challenges we discuss below are unique to OCOLOS's run-time approach.

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 v-table[3], or programmer-created function pointers stored on the stack, heap, or in global variables. As with other pointers in C/C++, a function pointer might be cast to an integer, obfuscated via arithmetic, then restored to the original value, cast back to a pointer again, and dereferenced.

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 run time, *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. The value of PC in each thread (the rip register in x86) is a pointer to an instruction in the currently-running function in each thread. 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. libc's setjmp/longjmp API can be used to create programmer-managed code pointers to essentially arbitrary code locations.

As is clear from the discussion above, the address space of a typical process contains *a large number* of code pointers. Keeping track of all of them so that they can be updated if a piece of code moves is essentially impossible for any serious program. Sometimes a code pointer of interest resides in kernel space where it is inaccessible to user code. In a managed language, the runtime can indeed track all code pointers and update or invalidate them as needed. However, OCOLOS targets complex unmanaged code so we have to be able to live with code pointers that are outside our control. Even small code changes can silently break such code pointers.

In the next section, we discuss how OCOLOS overcomes these challenges by retaining the original code within a process, adding optimized code at a new location, and patching up as many code pointers as possible to steer execution towards the optimized code in the common case.

IV. OCOLOS

In Figure 4a, we show a high-level overview of the steps OCOLOS performs to optimize the code of a target process at run time. First, we gather profiling information from the target process ❶, then build the BOLTed binary ❷, pause the target process ❸, inject code ❹, update pointers to refer to the injected code ❺, and finally resume the process ❻. In this section, we assume the presence of the BOLTed binary and focus on the key components of OCOLOS's online code replacement mechanism: Steps ❸-❺. Later, in Section V, we discuss Steps ❶ and ❷, which are conceptually simpler as they leverage existing tools like Linux's perf utility for performance profiling and BOLT. Note that Steps ❶ and ❷, which consume the most time, are done concurrently in the background while

[3]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.

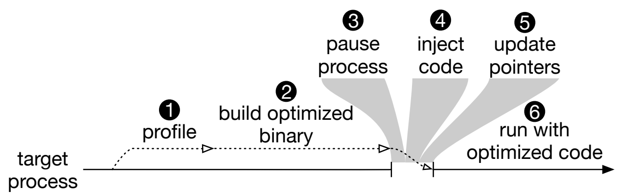

Figure 4a: Main steps OCOLOS takes to optimize a target process

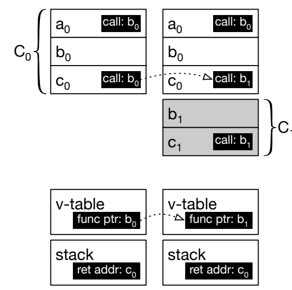

Figure 4b: Starting state of the address space (left) and state after code replacement (right)

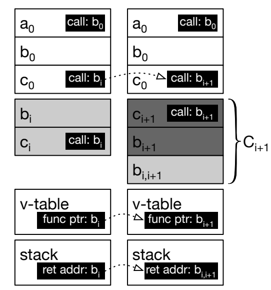

Figure 4c: Before (left) and after (right) continuous optimization

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.

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 4b. The code from the *original binary* we refer to as C_0, which consists of 3 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 return addresses of currently-executing functions. In Figure 4b, 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 (Section IV-B) 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 run time, but identifying the call sites in advance significantly shortens the stop-the-world period. OCOLOS leverages the Linux ptrace API, which allows one process (often a debugger like gdb) to control and inspect another process. OCOLOS uses ptrace to stop the target process and to inspect and adjust its register state.

A. Adding Code

As we describe in Section III-B, finding and updating all code pointers is fraught with corner cases. This leads to the first principle guiding OCOLOS's design:

Design Principle #1: *preserve addresses of C_0 instructions*

To enable significant performance gains by optimizing both the function-level and basic block layout, while preserving correctness, we design the following technique. 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 4b). OCOLOS then changes a subset of code pointers within C_0 to redirect execution to the C_1 code. Remaining code pointers are not perturbed and continue to point to C_0 code. This approach can also handle thorny cases like setjmp/longjmp where a target instruction (not function) address has been saved on the heap or stack at run time.

B. Updating Code Pointers

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

Design 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. Other OCOLOS use-cases such as profiling are likely to also be amenable to this trade-off. For instance, if we need to count function invocations then we can instrument only the C_1 code, ignoring the rare invocations of the old C_0 version of a function. For security or debugging use-cases, however, it may be necessary to redirect all invocations of C_0 functions to their C_1 counterparts instead, *e.g.*, via trampoline instructions [16] at the start of C_0 functions and at call sites within them.

Since our goal for the current version of OCOLOS is minimizing (but not eliminating) 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 4b 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.

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

Design 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 very long before it encounters a direct call or a virtual function call which steers execution back to C_1.

C. 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 re-profiling 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 above, though functions on the stack and function pointers require delicate handling as explained below.

The key challenge in 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 DRAM 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 be removed.

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 Section IV-B and illustrated in Figure 4c.

1) 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 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 4c), 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}. $b_{i,i+1}$ 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 4c.

2) Function pointers: Apart from return addresses, function pointers may also point to C_i. At any time during execution, programs can create function pointers that may exist on the stack, heap, or in registers and point to a function in 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 with a callback function: void* wrapFuncPtrCreation(void*)

This function takes as its argument the function pointer being created (which may reference C_i code), and returns the value that the program will actually use - a pointer to the corresponding C_0 function instead. OCOLOS maintains a map from C_i to C_0 addresses to enable this translation. If OCOLOS has not yet replaced any code, or the function pointer being created does not reference C_i (*e.g.*, it references library code), wrapFuncPtrCreation simply acts as the identity function.

Once a function pointer is created, it can freely propagate through registers and memory without any instrumentation - intervention is required only on function pointer creation. This instrumentation has a negligible cost: MySQL running the read_only input creates just 45 function pointers per millisecond on average. While we have not found the need to implement it for our workloads, calls to setjmp could be similarly redirected to C_0.

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.

Due to technical limitations in the current version of BOLT, BOLT assumes the presence of a single .text code section and refuses to run on a BOLTed binary. Unfortunately, this prevents us from evaluating continuous optimization because our profiling data will refer to C_i code, and we need BOLT to run optimizations on C_i to produce C_{i+1}. We plan to add this feature to BOLT in the future.

D. Limitations

OCOLOS currently does not support jump tables, as they rely on compile-time constants to compute the jump target, and hence OCOLOS does not update these constants during code replacement yet. Thus, OCOLOS currently requires that a binary be compiled with the -fno-jump-tables flag. The binaries for BOLT and the non-PGO baseline, however, can include jump tables. This jump table restriction is not fundamental to OCOLOS's approach. With a little extra support from BOLT to identify these constants within the optimized binary, OCOLOS can extract and update them as part of code replacement.

OCOLOS requires a pause time during code replacement, during which the target process cannot respond to incoming requests or do other useful work. This may hurt application performance, especially in terms of tail latency. There is scope to reduce the latency of OCOLOS's code replacement: it requires a few MiB of scattered writes throughout the address space, all of which are currently done sequentially. If OCOLOS, say, updated v-tables in parallel with patching direct calls that should reduce the end-to-end replacement time further.

An additional approach to preserving tail latency during code replacement is to leverage techniques proposed for mitigating the effects of garbage collection pauses in distributed systems [82, 103]. If the system includes a load-balancing tier, as many modern web services do, then the load balancer can be made aware of application pauses (like major garbage collections, or OCOLOS code replacement) and can route traffic to other nodes temporarily. Because code optimizations are explicitly triggered by the operator, pause times are well known and can be scheduled accordingly.

OCOLOS requires that the functionality and ABI of C_1 is unchanged with respect to C_0, so that a function f_0 in C_0 has equivalent application semantics to a function f_1 in C_1. f_1 can, however, vary in non-semantic ways, such as having extra instrumentation or different performance.

Global variables cannot change location in OCOLOS, since C_0 code often hard-codes a global variable's original location via RIP-relative addressing. C_1 code thus needs to reference those same global variables.

V. IMPLEMENTATION

In this section, we discuss some of OCOLOS's implementation issues, including OCOLOS's methodology to profile running processes, steps to run BOLT, and mechanism to transform code. Finally, we describe OCOLOS's BAM mode for accelerating batch workloads such as software builds.

Profiling. As Figure 4a shows, OCOLOS's first step is to profile the target process, to determine whether it suffers from sufficient front-end stalls to merit OCOLOS's optimizations. OCOLOS uses the standard Linux perf utility to record hardware performance counters for this purpose. perf can attach to an already-running process, allowing OCOLOS to be deployed on a new process or an existing one.

OCOLOS adopts a 2-stage approach for profiling. The first stage follows the methodology proposed in DMon [49], which itself is built on Intel's TopDown microarchitectural bottleneck analysis [109]. Note that in many data centers, systems such as GWP [89] already continuously profile all applications in the fleet. We have not integrated this analysis into OCOLOS yet since it is not the primary focus of our work, but we perform measurements to validate the feasibility of this approach in Section VI-C4.

If this first-stage exploration reveals significant time spent in the processor front-end, we continue with the second profiling stage. Here we use perf to record the hot control-flow paths of the target process via Intel's LBR mechanism (Section II-A). We feed this information into the BOLT optimizer, as discussed next.

Running BOLT. We provide a quick summary of how BOLT operates here to keep this paper self-contained. More details can be found in the BOLT papers [76, 77].

First, we use the perf2bolt utility to extract the LBR information recorded by perf into an internal format that BOLT can consume more easily. Armed with the extracted LBR information and the binary corresponding to the target process, BOLT runs a series of optimization passes (most notably basic-block and function reordering, see Section II) to produce a new, optimized binary.

Efficient Code Copying. To provide direct copying of code from the optimized binary into the target process, we launch the target process with an LD_PRELOAD library. LD_PRELOAD is a Linux feature that allows a user-specified shared library to be loaded, alongside a program's required shared libraries, when a process is launched. We use LD_PRELOAD to add some functions for code replacement into the address space of the target process. We then use ptrace to transfer control to our code, which reads in the optimized binary and copies its relevant contents into place. While ptrace can also perform memory copies into the target process, they are prohibitively slow since each copy requires a system call and several context switches. Performing this memory copy from within the target process is much more efficient and helps minimize the stop-the-world time.

A. BAM: Batch Accelerator Mode

For programs with short running times, OCOLOS's fixed optimization costs cannot be effectively amortized. If these programs are executed frequently, as is common in data centers, it may still be worthwhile to optimize them. To address this problem, we have developed an alternative deployment mode for OCOLOS called BATCH ACCELERATOR MODE or BAM. As the name implies, BAM is focused on batch workloads where the same binary is invoked repeatedly. Early invocations of the binary can be profiled and fed into BOLT, so that subsequent invocations can use the BOLTed binary instead and see improved performance. BAM performs its optimization online as the batch workload runs, so it does not suffer from stale profiles, stale binary mapping issues, or require any profile management – all of which can hinder the use of offline PGO systems like BOLT.

BAM is a Linux shared library that is attached to a command, *e.g.*, with LD_PRELOAD=bam.so make. BAM additionally needs to be told, via a configuration file, the binary to optimize. The BAM library makes use of another LD_PRELOAD feature which is transparent interception of calls to functions in any shared library. In particular, BAM intercepts libc's exec* calls and, if it finds an invocation of the target binary, adjusts the exec arguments to launch the binary with perf's profiling enabled. BAM also attaches its shared library to child processes to find invocations of the target binary no matter where they occur in the process tree.

Once BAM has collected a (configurable) number of profiles of the target binary's execution, it runs BOLT in a background process to produce the BOLTed binary. Once the BOLTed binary is available, BAM rewrites exec calls to use the BOLTed binary instead of the original binary, leading to an automatic performance boost for the remainder of the batch workload.

Similarly to OCOLOS's single-process mode (Section IV), BAM automatically profiles a workload as it runs, avoiding the challenges of stale profiling data and storing and retrieving profiles at scale. BAM's highly-compatible LD_PRELOAD-based design is also similar in spirit to OCOLOS, in that no application changes are required to use BAM. In the make example above, no changes are required to the Makefiles, application source code, make program, or the compiler toolchain.

One unique feature of BAM compared to OCOLOS is that BAM does not replace the code of a running process; it requires instead a subsequent exec call to allow the optimized binary to run. There is thus no stop-the-world component to BAM, and the overhead of switching from the original binary to the optimized one is essentially zero.

We see BAM being especially useful for accelerating Continuous Integration (CI) builds of large software projects. These CI builds are always done from scratch to ensure the software builds correctly on a fresh system [14]. So long as the software build is long enough for BAM to obtain useful profiling and run BOLT, BAM can transparently accelerate compiler invocations for the latter part of the build. BAM is complementary to build optimization techniques like distributed build caches [19, 20, 30, 37, 83, 84]. While a build cache can avoid some compiler invocations, BAM accelerates those compiler invocations that remain. BAM is also simpler to deploy than a build cache as BAM does not need any remote web services to be provisioned – BAM is purely local to each build.

VI. EVALUATION

In our evaluation of OCOLOS, we set out to demonstrate that OCOLOS can provide significant performance improvements for programs that suffer from processor front-end bottlenecks. To demonstrate OCOLOS's robustness, we evaluate it across a range of benchmarks, from complex, multithreaded programs such as the MySQL relational database to compute-bound, single-threaded workloads like the Verilator chip simulator and batch workloads like building the Clang compiler.

A. Experimental Setup

We run our experiments on a 2-socket Intel Broadwell Xeon E5-2620v4 server with 8 cores and 16 threads per socket (16 cores and 32 threads total) running at 2.1GHz. Each core has a 64-entry iTLB, a 1536-entry L2 TLB, a 32KiB L1i, a 32KiB L1d, a 256KiB L2 cache, and access to a shared 20MiB L3 cache and 128 GiB of RAM. The server runs Linux version 4.18. We use commit `88c70afe` of the Lightning BOLT system [77] from its GitHub repository [21].

For our benchmarks, we use MySQL version `8.0.28`, driven by inputs from Sysbench version `1.1.0-ead2689`. We use MongoDB version `6.0.0-alpha-655-gea6cea6`, driven by inputs from YCSB. We use Memcached version `1.6.12`, driven by inputs from memaslap version `1.0`. For MongoDB and Memcached the input names show the mix of operations, *e.g.* read95 insert5 means 95% of operations are reads and the other 5% are inserts. We use Verilator version 3.904, simulating an in-order rv64imafdc RISC-V single-core processor generated from RocketChip [5], with the processor running a set of RISC-V benchmarks [91]. All benchmarks are compiled with their default optimization level: -O3 for MySQL and Verilator, and -O2 for MongoDB and Memcached. We measure Verilator's performance as the throughput of iterations of the main Dhrystone loop or iterations over the input array for median and vvadd. We evaluate BAM on a build of Clang version 14.0.

All performance measurements, unless otherwise noted, show steady-state performance. For OCOLOS, we measure performance after code replacement is complete, except in Figure 7 where we show MySQL's performance before, during, and after code replacement. OCOLOS and BOLT results are based on 60 seconds of profiling unless otherwise noted. Unless otherwise noted, we show averages of 5 runs with error bars indicating the standard deviation.

B. Performance and Characterization

Figure 5 shows the throughput improvement OCOLOS provides across our set of benchmarks. We compare OCOLOS to four 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; *PGO oracle input* uses the same profiling file as *BOLT oracle input* but feeds it to clang's builtin PGO pass [62]. 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 5 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. Clang's PGO generally falls short of BOLT, similar to the results from the BOLT paper [76] though our benchmarks are different, likely due to the challenges of mapping PCs back to the source code [36]. Aggregating profiling information across inputs is worse than using just the oracle profile of the input being run, as different inputs tend to exhibit contradictory control-flow biases that cancel each other out.

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 (Section IV-B). In some cases like MySQL delete and write only, use of code pointers that continue to refer to unoptimized C_0 code results in a non-trivial performance gap (18 and 13 percentage points, respectively).

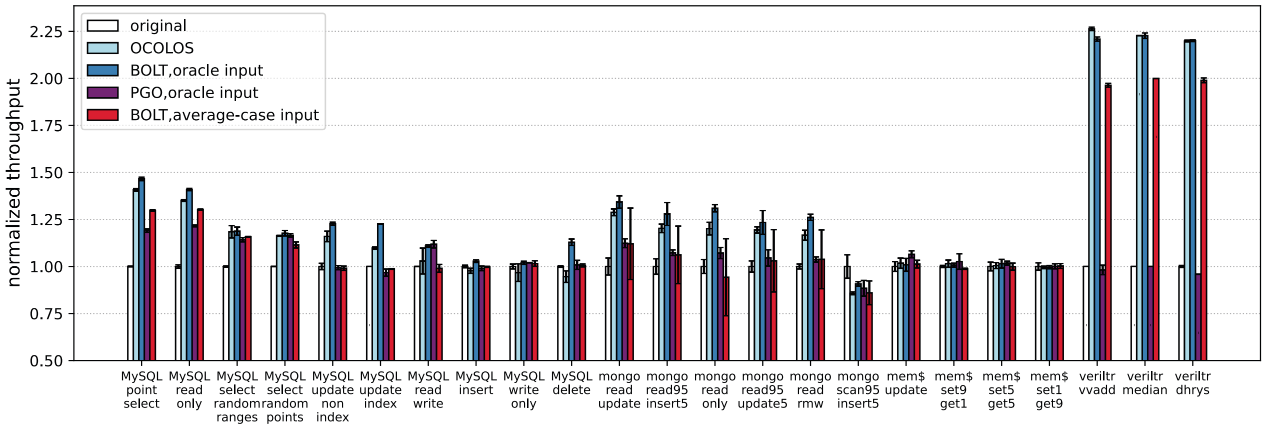

Fig. 5: Performance of OCOLOS (light blue bars) compared to BOLT using an oracle profile of the input being run (dark blue bars), Clang PGO using the same oracle profile (purple bars) and BOLT using an average-case profiling input aggregated from all inputs (pink bars). All bars are normalized to original non-PGO binaries (white bars).

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. This shows that OCOLOS's efficient design enables dynamic code optimizations with almost the same performance gains as PGO while providing additional benefits such as guaranteeing the accuracy of profiling information and easy mapping to the target binary, and a simple deployment model that avoids the need for profiles to be stored and queried.

MongoDB scan95 insert5 is an odd case where conventional static compilation outperforms all of the profile-guided techniques (*e.g.*, OCOLOS is 14% slower than original). To understand this behavior better, we applied Intel's TopDown [109] performance measurement methodology which can identify the root microarchitectural cause of low IPC. TopDown classifies pipeline slots in each cycle to one of four top-level cases: Retiring (useful work), Front-End Bound (L1i, iTLB, and decoder bottlenecks), Back-End Bound (L1d or functional unit bottlenecks) or Bad Speculation (branch or memory aliasing mispredictions). With scan95 insert5, in all of the BOLT-based configurations (OCOLOS, BOLT oracle and BOLT average-case) the workload shifts from being front-end bound to back-end bound, with many memory accesses in particular stalled waiting for DRAM, suggesting that poor memory controller scheduling may be the root cause of the slowdown. The PGO version of scan95 insert5 has very similar TopDown metrics to original, so the cause of its slowdown is unclear.

Table I shows characterization data for our benchmarks, such as code size metrics, the average number (across inputs) of functions that are reordered by BOLT, on the call stack when code replacement occurs, and direct call sites that are patched. We also report memory consumption in terms of maximum resident set size, which is the peak amount of physical memory allocated to a process, when running the original binary, BOLT, and OCOLOS on MySQL oltp_read_only, mongodb read_update, Memcached set10 get90, and Verilator dhrystone. OCOLOS requires a modest amount of extra memory, only 208 MiB for mongodb and much less for other benchmarks. OCOLOS's memory consumption is affected primarily by binary size, and does not scale up with larger or longer-running inputs. Note also that OCOLOS's memory consumption is not an ongoing cost, but is incurred during code replacement and can be deallocated afterwards.

OCOLOS's storage requirements are under 200 MiB for each benchmark, chiefly for profiling data and the optimized binary, which does not produce a significant amount of disk I/O. Note that these files are also transient: after the optimized binary is produced they can be deleted.

	MySQL	Mongo	Mem$	Verilator
functions	33,170	69,807	374	406
v-tables	3,812	6,165	0	10
.text section (MiB)	24.6	50.0	0.142	2.3
avg funcs reordered	963.6	2,364.2	74.2	83.2
avg funcs on stack	79	100.6	10	5
avg call sites changed	31,677.2	30,9297.8	496.6	251.2
max RSS (MiB)				
original	397.4	1434.4	67.8	263.4
BOLT	398.0	1432.8	67.9	263.7
OCOLOS	438.5	1640.5	69.8	265.4

TABLE I: Benchmark characterization data

C. MySQL Case Study

Next we present an in-depth case study of MySQL, using it to illustrate different aspects of OCOLOS's performance. We focus on MySQL because it is a complex workload and it has the widest variety of inputs among our benchmarks.

To see a small, concrete example of how OCOLOS can improve performance, we used the perf report and perf annotate utilities to examine the distribution of L1i misses in an execution of MySQL oltp_read_only. Under both BOLT with average-case input and Clang PGO, the MYSQLparse function is a common source of L1i misses - with BOLT average-case it actually has the most L1i misses of any function. MYSQLparse is auto-generated by Bison as the main parsing function for SQL queries, with over 176 KiB of binary code. perf reports frequent L1i misses in basic blocks dealing with backtracking and checking for additional tokens. It makes sense that the average-case input has a difficult time, as it is unable to specialize the parser code for the current query mix and oltp_read_only has only select queries. It is less clear why PGO performs poorly since it has the oracle profile, but it is likely due to problems mapping low-level PCs back to source code and LLVM IR. With both OCOLOS and BOLT oracle, MYSQLparse does not even appear on perf's radar as no L1i misses are sampled within it.

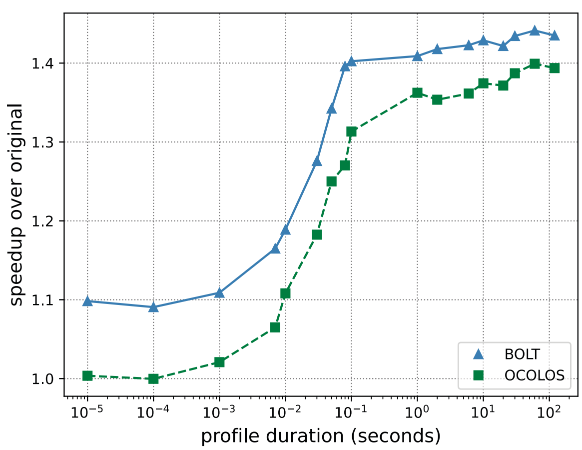

Fig. 6: The impact of profile duration on speedup for MySQL read_only

1) Profiling Duration: The amount of time that OCOLOS spends gathering profile information is configurable. While we use a default of 60 seconds for our current experiments, OCOLOS can still perform well with significantly less profiling information. Figure 6 shows the speedup over the original binary when varying the duration of profiling. The green squares show OCOLOS, and the blue triangles show BOLT to represent how well offline BOLT can optimize when given the same profiling information as OCOLOS. BOLT again provides a ceiling on OCOLOS's expected performance. Figure 6 illustrates that profiling for at least 1 second offers a good absolute speedup over the original binary and also achieves most of the benefits that offline BOLT does. Below 100 milliseconds, profile quality suffers significantly for both OCOLOS and BOLT.

2) Code Replacement Costs: To better understand the performance impact of OCOLOS's code replacement mechanism,

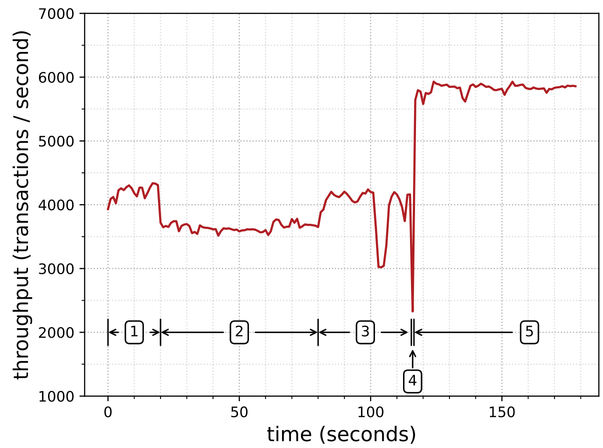

Fig. 7: Throughput of MySQL read_only before, during, and after code replacement. 95% tail latency degrades from 1.00ms to at most 1.55ms during code replacement.

we performed an experiment with MySQL read_only with Sysbench reporting the client's transaction throughput every second. Figure 7 shows the results. The first 20 seconds (region 1 of the graph) are a warm-up period, showing the performance of the original binary at around 4,200 transactions per second (tps). After this, perf profiling begins collecting LBR samples (region 2), reducing throughput to about 3,600 tps. In region 3, perf2bolt runs 4 background threads to translate the LBR samples into a format that BOLT can use, and then single-threaded BOLT generates the optimized binary. BOLT, in particular, is quite CPU-intensive, causing a reduction in throughput just after the 100-second mark. In region 4, OCOLOS performs code replacement which entails a brief single-threaded stop-the-world phase of 669 milliseconds (see Table I for other benchmarks). After that, in region 5, MySQL's parallel execution resumes with the optimized code in place lifting the performance to almost 6,000 tps. Sysbench also reports 95^{th} percentile tail latency for each 1-second window of execution. Analyzing a single representative run, the average 95% transaction latency during region 1 is 1.00 milliseconds, degrading to a worst-case of 1.55 ms during regions 3 and 4, and improving to 0.73 ms on average in region 5.

Figure 7 shows that the performance impact of OCOLOS is modest, even during code replacement. As we discussed in Section VI-C1, OCOLOS can still perform well with as little as 1 second of profiling. Although we are already using the Lightning BOLT system [77] which has been optimized for lower execution times, there likely exist further opportunities to reduce region 3 costs by shifting some of BOLT's work into an offline phase. Such optimizations do not matter for BOLT's original offline setting, however, they would be beneficial for OCOLOS. Finally, there is scope to reduce OCOLOS's pause time further by shifting more work to occur inside the target process via the OCOLOS LD_PRELOAD library and parallelizing the code replacement routines that currently execute serially.

3) End-to-End Overheads: Table II shows OCOLOS's overheads for code replacement. The intervals between code replacements are configurable with OCOLOS; longer intervals amortize code replacement costs better but are less sensitive to application phases or input changes.

One way to evaluate OCOLOS's overheads in an end-to-end manner is to consider how long it takes OCOLOS to "recover" the ground lost during code replacement. Considering MySQL read_only as an example (Figure 7), the steady state throughput of the original program is about 4,200 transactions per second, which OCOLOS boosts to 5,850 tps after code replacement is complete. Taking the reduced throughput during code replacement into account, at 30 seconds after code replacement completes OCOLOS has processed as many transactions as if we had run the original binary the entire time. All execution after this point is a net gain for OCOLOS, so running for several minutes before the next code replacement is advisable in practice. With smaller speedups, OCOLOS must run for longer before performing code replacement again. More generally, if OCOLOS hurts performance by a factor of a during code replacement which lasts for s seconds, and then boosts performance by a factor of b after code replacement completes, we should run the optimized code for at least as/b seconds to recover the ground lost during code replacement.

	MySQL	Mongo	Mem$	Verilator
perf2bolt time (sec)	28.186	26.624	12.918	4.181
llvm-bolt time (sec)	8.237	17.882	0.1404	1.935
replacement time (sec)	0.669	1.221	0.020	0.146

TABLE II: Fixed costs of code replacement

4) Microarchitectural Impacts: Next we investigate the microarchitectural causes of OCOLOS's performance benefits. Figure 8 shows a variety of front-end performance counter measurements, each represented as events per 1,000 instructions. The MySQL inputs along the x-axis are sorted from highest (left) to lowest (right) speedup with OCOLOS to match the order in Figure 5. Moving from top to bottom in Figure 8, we see that OCOLOS is able to achieve significant reductions in L1i and iTLB MPKI. All MySQL inputs also show large reductions in the number of taken branches; fewer taken branches means less pressure on branch prediction resources which may reduces mispredicted branches as well. Across all of these front-end metrics, OCOLOS achieves results very similar to offline BOLT.

Somewhat surprisingly, the front-end metrics in Figure 8 often do not correlate particularly well with the speedup that OCOLOS provides. To overcome this, we again turned to Intel's TopDown [109] methodology. Using TopDown's *Front-End Latency* and *Retiring* percentages, a simple linear regression

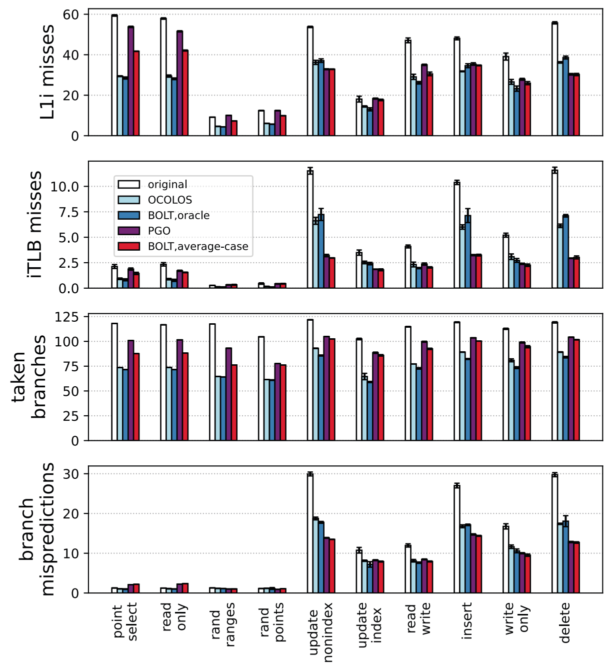

Fig. 8: Microarchitectural events (per 1,000 instructions) for MySQL inputs

can accurately determine workloads that will and won't benefit from OCOLOS (Figure 9). Moreover, with OCOLOS's online approach, even should identifying performance losses a priori prove challenging, we can always revert to C_0 code to at least recover the original performance.

D. Batch Accelerator Mode

In this section, we examine the impact of BAM on a from-scratch build of the Clang compiler. In a large software build, BAM profiles the initial compiler executions to generate an optimized compiler binary that is tuned to the source program being compiled. A full Clang build contains 2,624 compiler executions in all. The dashed red line near the top of Figure 10 illustrates the running time of the original Clang build, executing parallel jobs via make -j. For the dashed orange line at the bottom, we aggregate profiling information from the entire build and feed it to BOLT, and then measure a fresh build using the resulting BOLTed binary. This represents a lower bound on the running time that BAM can achieve.

The green triangles (with a polynomial curve fit to them) show the performance of BOLT when we profile only a limited number of compiler executions (given on the x-axis), generating an optimized binary while measuring the time of a fresh build using this binary. The cost of collecting profiles and running BOLT is excluded; the optimized binary is available at the start of the build. These results show how well an ideal

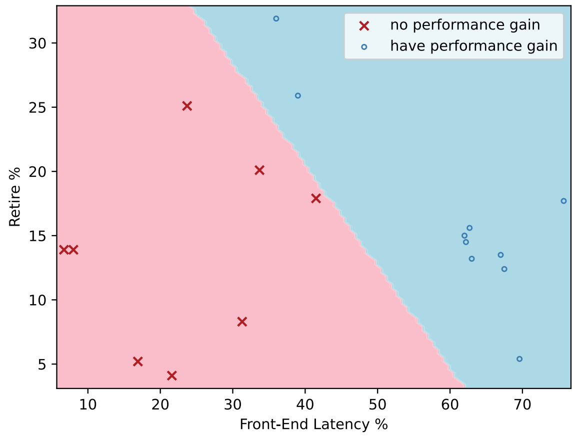

Fig. 9: TopDown's [109] Front-end Latency and Retire percentages allow us to accurately classify which workloads will benefit from OCOLOS and which won't.

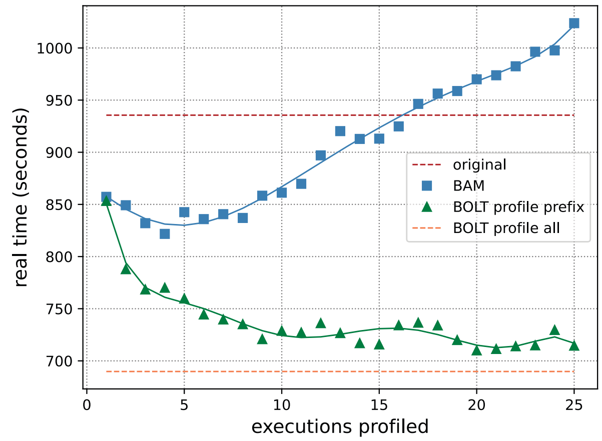

Fig. 10: The running time of a Clang build with the original compiler, and compilers optimized by BOLT and BAM.

BAM implementation can perform if it did not suffer from any profiling and optimizations overheads, revealing the marginal utility of extra profiling data.

Finally, the blue squares (and polynomial curve) show the performance achieved by BAM. We first observe that, even when profiling just one compiler execution, BAM provides a speedup of 1.09× over the original build. At first, profiling additional compiler executions leads to a speedup of up to 1.14×, as this profiling data is "worth the wait". However, after about 5 executions BAM suffers diminishing returns from additional profiling for two reasons. First, the value of that profiling data is relatively low as shown by the decreasing slope of the green line. Second, as BAM waits for more profile data, it starts the optimization process later, losing out on opportunities to use the optimized binary. This opportunity cost increases over time, causing the BAM running time to eventually surpass that of the original build.

Ultimately, our BAM investigation demonstrates that the amount of profiling data needed to run PGO effectively is quite low, mirroring our results from Section VI-C1. BAM is able to leverage this property to accelerate the Clang build, without any changes to Clang or the build infrastructure.

VII. RELATED WORK

The performance implications of front-end stalls have inspired computer architecture and compiler researchers to propose numerous techniques for improving instruction locality. We divide this work into three categories and qualitatively compare these techniques against OCOLOS to describe how OCOLOS addresses their shortcomings.

Instruction prefetching mechanisms. Computer architects primarily aim to solve the front-end stall problem via instruction prefetching [4, 24, 25, 28, 31, 33, 42, 46, 47, 57, 58, 59, 68, 70, 87, 88, 92, 93, 94, 96, 97]. A plethora of such techniques, ranging from simpler next-line [97] and discontinuity [42, 81, 100, 101] prefetchers to sophisticated temporal [24, 25, 46, 47] (or record-and-replay [6]) prefetching, aim to strike a balance between performance and high metadata storage overhead. Branch predictor-guided prefetchers [87, 88] are extremely effective [40, 41, 58, 59] and consequently, have been adopted in many recent processors [32, 78, 95, 102]. Nevertheless, these state-of-the-art prefetchers fall short when applications contain a large number of taken branch instructions that exhaust the capacity of the branch predictor and BTB [25, 48, 58, 99]. OCOLOS can convert taken branches into not-taken ones, easing pressure on the branch predictor (Figure 8) and improving overall performance.

Profile-guided code layout optimizations. Compiler techniques to address the front-end stall problem mainly focus on improving instruction locality via code layout optimizations [10, 29, 34, 36, 61, 63, 65, 76, 77, 79, 86, 113]. These techniques perform basic-block reordering [72, 80], function reordering [75], and hot/warm/cold code splitting [12] (also known as function splitting [76]) using profiles collected from previous executions [27, 38]. While these techniques are extremely effective at improving instruction locality [6] and therefore widely adopted in today's data centers [10, 76, 77], profile quality limits their ability to achieve close-to-optimal performance as we show in Section III-A. To address this limitation, OCOLOS always uses the best-quality profile from the current execution. Some managed language runtimes, like the HotSpot JVM [39], also perform PGO at run time, profiling the application running on the VM. While OCOLOS targets unmanaged languages instead, OCOLOS could complement a system like HotSpot by performing PGO on the running HotSpot binary itself.

Other systems [17, 26, 60] have also proposed run-time code optimization for unmanaged languages. ClangJIT [26] can perform C++ template specialization at run time, improving performance and avoiding the latency and code bloat of producing all template specializations at compile time. BinOpt [17] can lift, at run time, the machine code of selected functions to LLVM IR, perform optimizations and recompile to machine code, and then replace the machine code with the optimized code and resume execution of the program. BinOpt requires application code changes to use its API to identify functions to optimize, unlike OCOLOS which operates transparently to the application. While BinOpt does not currently utilize profiling information, it could do so in principle. However, BinOpt's use of LLVM IR as the optimization target would make it challenging to map machine-code-level profiling information to LLVM IR [36, 76], which is why tools like BOLT operate at the machine code level instead.

Static code layout optimizations. Evidence-based static code layout optimizations [7, 9, 15, 44, 66, 107] also aim to address the profile-sensitivity problem of profile-guided code layout optimizations. State-of-the-art static code layout optimizers mainly use machine learning techniques (*e.g.*, deep neural networks [66, 69, 85] or decision trees [9, 15]) to find an optimal code layout. Despite using sophisticated machine learning techniques, such techniques fall far short of the profile-guided techniques and provide only one-third of the speedups offered by the profile-guided code layout optimizers [69]. Therefore, in this work, we focus on improving the performance of these profile-guided techniques by applying them in an online manner with OCOLOS.

VIII. 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, however, applied to a running process. By operating at run time, OCOLOS always profiles the most up-to-date and relevant behavior of the program, and avoids problems with mapping the profile to a target binary that can frustrate offline PGO. We describe how OCOLOS's design can perform run-time code replacement safely for unmanaged programs, with essentially only fixed costs paid at code replacement time. We evaluate OCOLOS on a range of workloads, from large multithreaded server applications to a single-threaded chip simulator and a large software build. We show that OCOLOS can provide speedups of up to 2.20×, all without requiring any changes to the applications being accelerated by OCOLOS.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful suggestions and feedback. This work was supported by generous gifts from Intel Labs, NSF/Intel joint grants #2010810 and #2011168, NSF #1942754, a Rackham Predoctoral Fellowship, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center cosponsored by SRC and DARPA. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies. We thank Maksim Panchenko and Guilherme Ottoni from Meta Inc. for helpful discussions about BOLT.

REFERENCES

[1] "Github - akopytov/sysbench: Scriptable database and system performance benchmark," https://github.com/akopytov/sysbench.

[2] José Nelson Amaral, Edson Borin, Dylan R Ashley, Caian Benedicto, Elliot Colp, Joao Henrique Stange Hoffmam, Marcus Karpoff, Erick Ochoa, Morgan Redshaw, and Raphael Ernani Rodrigues, "The alberta workloads for the spec cpu 2017 benchmark suite," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2018, pp. 159–168.

[3] Sundaram Ananthanarayanan, Masoud Saeida Ardekani, Denis Haenikel, Balaji Varadarajan, Simon Soriano, Dhaval Patel, and Ali-Reza Adl-Tabatabai, "Keeping master green at scale," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303970

[4] Ali Ansari, Fatemeh Golshan, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad, "Mana: Microarchitecting an instruction prefetcher," *The First Instruction Prefetching Championship*, 2020.

[5] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

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

[7] Thomas Ball and James R Larus, "Branch prediction for free," *ACM SIGPLAN Notices*, vol. 28, no. 6, pp. 300–313, 1993.

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

[9] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn, "Evidence-based static branch prediction using machine learning," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 1, pp. 188–222, 1997.

[10] Dehao Chen, David Xinliang Li, and Tipp Moseley, "Autofdo: Automatic feedback-directed optimization for warehouse-scale applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 2016, pp. 12–23.

[11] Hyoun Kyu Cho, Tipp Moseley, Richard Hank, Derek Bruening, and Scott Mahlke, "Instant profiling: Instrumentation sampling for profiling datacenter applications," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–10.

[12] Robert Cohn and P Geoffrey Lowney, "Hot cold optimization of large windows/nt applications," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE, 1996, pp. 80–89.

[13] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun, "{REPT}: Reverse debugging of failures in deployed software," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 17–32.

[14] Charlie Curtsinger and Daniel W. Barowy, "Riker: Always-Correct and fast incremental builds from simple specifications," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 885–898. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/curtsinger

[15] Veerle Desmet, Lieven Eeckhout, and Koen De Bosschere, “Using decision trees to improve program-based and profile-based static branch prediction,” in *Asia-Pacific Conference on Advances in Computer Systems Architecture*. Springer, 2005, pp. 336–352.

[16] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury, “Binary rewriting without control flow recovery,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 151–163.

[17] Alexis Engelke and Martin Schulz, “Robust Practical Binary Optimization at Run-time using LLVM,” in *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, 2020, pp. 56–64.

[18] Walter Erquinigo, David Carrillo-Cisneros, and Alston Tang, “Reverse debugging at scale,” https://engineering.fb.com/2021/04/27/developer-tools/reverse-debugging/.

[19] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula, “Cloudbuild: Microsoft’s distributed and caching build service,” in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 11–20.

[20] Facebook, “Buck: A high-performance build tool,” https://buck.build, 2021.

[21] Facebook BOLT team, “BOLT: Binary Optimization and Layout Tool,” https://github.com/facebookincubator/BOLT.

[22] Facebook BOLT team, “How to use bolt to optimize bin continuously in the production environment ,” https://github.com/facebookincubator/BOLT/issues/94#issuecomment-668985872.

[23] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi, “Clearing the clouds: a study of emerging scale-out workloads on modern hardware,” in *ACM SIGPLAN Notices*, vol. 47, no. 4. ACM, 2012, pp. 37–48.

[24] Michael Ferdman, Cansu Kaynak, and Babak Falsafi, “Proactive instruction fetch,” in *International Symposium on Microarchitecture*, 2011.

[25] Michael Ferdman, Thomas F Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos, “Temporal instruction fetch streaming,” in *International Symposium on Microarchitecture*, 2008.

[26] H. Finkel, D. Poliakoff, J. Camier, and D. F. Richards, “ClangJIT: Enhancing C++ with Just-in-Time Compilation,” in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2019, pp. 82–95. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/P3HPC49587.2019.00013

[27] Joseph A Fisher and Stefan M Freudenberger, “Predicting conditional branch directions from previous runs of a program,” *ACM SIGPLAN Notices*, vol. 27, no. 9, pp. 85–95, 1992.

[28] Nathan Gober, Gino Chacon, Daniel Jiménez, and Paul V Gratz, “The temporal ancestry prefetcher.”

[29] Google, “Propeller: Profile guided optimizing large scale llvm-based relinker,” https://github.com/google/llvm-propeller, 2020.

[30] Google, “Bazel,” https://bazel.build, 2022.

[31] Daniel A Jiménez Paul V Gratz and Gino Chacon Nathan Gober, “Barca: Branch agnostic region searching algorithm,” *The First Instruction Prefetching Championship*, 2020.

[32] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha *et al.*, “Evolution of the samsung exynos cpu microarchitecture,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 40–51.

[33] Vishal Gupta, Neelu Shivprakash Kalani, and Biswabandan Panda, “Run-jump-run: Bouquet of instruction pointer jumpers for high performance instruction prefetching,” *The First Instruction Prefetching Championship*, 2020.

[34] Stavros Harizopoulos and Anastassia Ailamaki, “Steps towards cache-resident transaction processing,” in *International conference on Very large data bases*, 2004.

[35] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao, “Optimizing Binary Translation of Dynamically Generated Code,” in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO ’15. USA: IEEE Computer Society, 2015, p. 68–78.

[36] Wenlei He, Julián Mestre, Sergey Pupyrev, Lei Wang, and Hongtao Yu, “Profile inference revisited,” *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–24, 2022.

[37] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu, *The Vesta approach to software configuration management*. Compaq. Systems Research Center [SRC], 2001.

[38] Urs Hölzle and David Ungar, “Optimizing dynamically-dispatched calls with run-time type feedback,” in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, 1994, pp. 326–336.

[39] HotSpot JVM team, “Jdk-6743900: frequency based block layout,” https://bugs.java.com/bugdatabase/view_bug.do?bug_id=6743900.

[40] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo, “Rebasing instruction prefetching: An industry perspective,” *IEEE Computer Architecture Letters*, 2020.

[41] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo, “Re-establishing fetch-directed instruction prefetching: An industry perspective,” *IEEE International Symposium on Performance Analysis of Systems and Software*, 2021.

[42] Quinn Jacobson, Eric Rotenberg, and James E Smith, “Path-based next trace prediction,” in *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 1997, pp. 14–23.

[43] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz, “Apt-get: profile-guided timely software prefetching,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 747–764.

[44] Bhargava Kalla, Nandakishore Santhi, Abdel-Hameed A Badawy, Gopinath Chennupati, and Stephan Eidenbenz, “A probabilistic monte carlo framework for branch prediction,” in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 651–652.

[45] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks, “Profiling a warehouse-scale computer,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15. New York, NY, USA: ACM, 2015, pp. 158–169. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750392

[46] Cansu Kaynak, Boris Grot, and Babak Falsafi, “Shift: Shared history instruction fetch for lean-core server processors,” in *International Symposium on Microarchitecture*, 2013.

[47] Cansu Kaynak, Boris Grot, and Babak Falsafi, “Confluence: unified instruction supply for scale-out servers,” in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 166–177.

[48] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci, “Twig: Profile-guided btb prefetching for data center applications,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 816–829.

[49] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci, “Dmon: Efficient detection and correction of data locality problems using selective profiling,” in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, ser. OSDI 2021. USENIX Association, Jul. 2021.

[50] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci, “I-spy: Context-driven conditional instruction prefetching with coalescing,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 146–159.

[51] Tanvir Ahmed Khan, Muhammed Ugur, Krishnendra Nathella, Dam Sunwoo, Heiner Litz, Daniel A Jiménez, and Baris Kasikci, “Whisper: Profile-guided branch misprediction elimination for data center applications,” in *Proceedings of the 55th International Symposium on Microarchitecture (MICRO)*, Oct. 2022.

[52] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci, “Ripple: Profile-guided instruction cache replacement for data center applications,” in *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, ser. ISCA 2021, Jun. 2021.

[53] Tanvir Ahmed Khan, Yifan Zhao, Gilles Pokam, Barzan Mozafari, and Baris Kasikci, “Huron: hybrid false sharing detection and repair,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 453–468.

[54] Hyesoon Kim, M Aater Suleman, Onur Mutlu, and Yale N Patt, "2d-profiling: Detecting input-dependent branches with a single input data set," in *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 2006, pp. 11–pp.

[55] Andi Kleen and Beeman Strong, "Intel processor trace on linux," *Tracing Summit*, 2015.

[56] Andy Kleen, "An introduction to last branch records," https://lwn.net/Articles/680985/.

[57] Aasheesh Kolli, Ali Saidi, and Thomas F Wenisch, "Rdip: return-address-stack directed instruction prefetching," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2013, pp. 260–271.

[58] Rakesh Kumar, Boris Grot, and Vijay Nagarajan, "Blasting through the front-end bottleneck with shotgun," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 30–42, 2018.

[59] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan, "Boomerang: A metadata-free architecture for control flow delivery," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 493–504.

[60] Michael A Laurenzano, Yunqi Zhang, Lingjia Tang, and Jason Mars, "Protean code: Achieving near-free online code transformations for warehouse scale computers," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 558–570.

[61] David Xinliang Li, Raksit Ashok, and Robert Hundt, "Lightweight feedback-directed cross-module optimization," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010, pp. 53–61.

[62] LLVM Team, "How To Build Clang and LLVM with Profile-Guided Optimizations," https://llvm.org/docs/HowToBuildWithPGO.html.

[63] C-K Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney, "Ispike: a post-link optimizer for the intel/spl reg/itanium/spl reg/architecture," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 15–26.

[64] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[65] Chi-Keung Luk and Todd C Mowry, "Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors," in *International Symposium on Microarchitecture*, 1998.

[66] Yonghua Mao, Junjie Shen, and Xiaolin Gui, "A study on deep belief net for branch prediction," *IEEE Access*, vol. 6, pp. 10 779–10 786, 2017.

[67] Gabriel Marin, Alexey Alexandrov, and Tipp James Moseley, "Break dancing: low overhead, architecture agnostic software branch tracing," in *22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '21)*, 2021.

[68] Pierre Michaud, "Pips: Prefetching instructions with probabilistic scouts," in *The First Instruction Prefetching Championship*, 2020.

[69] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira, "Vespa: static profiling for binary optimization," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–28, 2021.

[70] Tomoki Nakamura, Toru Koizumi, Yuya Degawa, Hidetsugu Irie, Shuichi Sakai, and Ryota Shioya, "D-jolt: Distant jolt prefetcher," *The First Instruction Prefetching Championship*, 2020.

[71] Nicholas Nethercote and Julian Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 89–100. [Online]. Available: https://doi.org/10.1145/1250734.1250746

[72] Andy Newell and Sergey Pupyrev, "Improved basic block reordering," *IEEE Transactions on Computers*, vol. 69, no. 12, pp. 1784–1794, 2020.

[73] Guilherme Ottoni, "Hhvm jit: A profile-guided, region-based compiler for php and hack," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 151–165.

[74] Guilherme Ottoni and Bin Liu, "Hhvm jump-start: Boosting both warmup and steady-state performance at scale," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 340–350.

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

[76] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni, "Bolt: a practical binary optimizer for data centers and beyond," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Press, 2019, pp. 2–14.

[77] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni, "Lightning bolt: powerful, fast, and scalable binary optimization," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, 2021, pp. 119–130.

[78] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala *et al.*, "The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc," *IEEE Micro*, vol. 40, no. 2, pp. 53–62, 2020.

[79] Larry L Peterson, "Architectural and compiler support for effective instruction prefetching: a cooperative approach," *ACM Transactions on Computer Systems*, 2001.

[80] Karl Pettis and Robert C Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990, pp. 16–27.

[81] Jim Pierce and Trevor Mudge, "Wrong-path instruction prefetching," in *International Symposium on Microarchitecture*, 1996.

[82] A. Omar Portillo-Dominguez, Philip Perry, Damien Magoni, Miao Wang, and John Murphy, "Trini: An adaptive load balancing strategy based on garbage collection for clustered java systems," *Softw. Pract. Exper.*, vol. 46, no. 12, p. 1705–1733, dec 2016. [Online]. Available: https://doi.org/10.1002/spe.2391

[83] GNU Project, "Gnu autoconf," https://www.gnu.org/software/autoconf/, 2021.

[84] GNU Project, "Gnu automake," https://www.gnu.org/software/automake/, 2021.

[85] Easwaran Raman and Xinliang David Li, "Learning branch probabilities in compiler from datacenter workloads," *arXiv preprint arXiv:2202.06728*, 2022.

[86] Alex Ramirez, Luiz André Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P Geoffrey Lowney, and Mateo Valero, "Code layout optimizations for transaction processing workloads," *ACM SIGARCH Computer Architecture News*, 2001.

[87] Glenn Reinman, Todd Austin, and Brad Calder, "A scalable front-end architecture for fast instruction delivery," *ACM SIGARCH Computer Architecture News*, vol. 27, no. 2, pp. 234–245, 1999.

[88] Glenn Reinman, Brad Calder, and Todd Austin, "Fetch directed instruction prefetching," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1999, pp. 16–27.

[89] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt, "Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE micro*, vol. 30, no. 4, pp. 65–79, 2010.

[90] Manman Ren and Shane Nay, "Improving iOS Startup Performance with Binary Layout Optimizations," 2019, [Online; accessed 25-Oct-2019]. [Online]. Available: https://www.facebook.com/atscaleevents/videos/664302790740440/

[91] RISCV Team, "riscv-tests," https://github.com/riscv-software-src/riscv-tests.

[92] Alberto Ros and Alexandra Jimborean, "The entangling instruction prefetcher," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 84–87, 2020.

[93] Alberto Ros and Alexandra Jimborean, "A cost-effective entangling prefetcher for instructions," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 99–111.

[94] Eric Rotenberg, Steve Bennett, and James E Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE, 1996, pp. 24–34.

[95] J Rupley, "Samsung exynos m3 processor," *IEEE Hot Chips*, vol. 30, 2018.

[96] André Seznec, "The fnl+ mma instruction cache prefetcher," in *The First Instruction Prefetching Championship*, 2020.

[97] Alan Jay Smith, "Sequential program prefetching in memory hierarchies," *Computer*, no. 12, pp. 7–21, 1978.

[98] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A Jiménez, Heiner Litz, and Baris Kasikci, "Thermometer: profile-guided btb replacement for data center applications," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 742–756.

[99] Niranjan K Soundararajan, Peter Braun, Tanvir Ahmed Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney, "Pdede: Partitioned, deduplicated, delta branch target buffer," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 779–791.

[100] Lawrence Spracklen, Yuan Chou, and Santosh G Abraham, "Effective instruction prefetching in chip multiprocessors for modern commercial applications," in *International Symposium on High-Performance Computer Architecture*, 2005.

[101] Viji Srinivasan, Edward S Davidson, Gary S Tyson, Mark J Charney, and Thomas R Puzak, "Branch history guided instruction prefetching," in *International Symposium on High-Performance Computer Architecture*, 2001.

[102] David Suggs, Mahesh Subramony, and Dan Bouvier, "The amd "zen 2" processor," *IEEE Micro*, vol. 40, no. 2, pp. 45–52, 2020.

[103] David Terei and Amit Levy, "Blade: A data center garbage collector," 2015. [Online]. Available: https://arxiv.org/abs/1504.02578

[104] Muhammed Ugur, Cheng Jiang, Alex Erf, Tanvir Ahmed Khan, and Baris Kasikci, "One profile fits all: Profile-guided linux kernel optimizations for data center applications," *ACM SIGOPS Operating Systems Review*, vol. 56, no. 1, pp. 26–33, Jun. 2022.

[105] April W Wade, Prasad A Kulkarni, and Michael R Jantz, "Aot vs. jit: impact of profile data on code quality," in *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2017, pp. 1–10.

[106] David W Wall, "Predicting program behavior using real or estimated profiles," *ACM SIGPLAN Notices*, vol. 26, no. 6, pp. 59–70, 1991.

[107] Youfeng Wu and James R Larus, "Static branch frequency and program profile analysis," in *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994, pp. 1–11.

[108] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu, "Can we trust profiling results? understanding and fixing the inaccuracy in modern profilers," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 284–295.

[109] Ahmad Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 35–44.

[110] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen, "On the precision of precise event based sampling," in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2020, pp. 98–105.

[111] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale N Patt, "Branchnet: A convolutional neural network to predict hard-to-predict branches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 118–130.

[112] Stephen Zekany, Daniel Rings, Nathan Harada, Michael A Laurenzano, Lingjia Tang, and Jason Mars, "Crystalball: Statically analyzing runtime behavior via deep sequence learning," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016.

[113] Jingren Zhou and Kenneth A Ross, "Buffering databse operations for enhanced instruction cache performance," in *International conference on Management of data*, 2004.