Revamping Sampling-Based PGO with Context-Sensitivity and Pseudo-Instrumentation

Wenlei He
Meta Inc.
USA
wenlei@meta.com

Hongtao Yu
Meta Inc.
USA
hoy@meta.com

Lei Wang
Meta Inc.
USA
wlei@meta.com

Taewook Oh
Meta Inc.
USA
twoh@meta.com

***Abstract*—The ever increasing scale of modern data center demands more effective optimizations, as even a small percentage of performance improvement can result in a significant reduction in data-center cost and its environmental footprint. However, the diverse set of workloads running in data centers also challenges the scalability of optimization solutions. Profile-guided optimization (PGO) is a promising technique to improve application performance. Sampling-based PGO is widely used in data-center applications due to its low operational overhead, but the performance gains are not as substantial as the instrumentation-based counterpart. The high operational overhead of instrumentation-based PGO, on the other hand, hinders its large-scale adoption, despite its superior performance gains.**

In this paper, we propose CSSPGO, a context-sensitive sampling-based PGO framework with pseudo-instrumentation. CSSPGO offers a more balanced solution to push sampling-based PGO performance closer to instrumentation-based PGO while maintaining minimal operational overhead. It leverages pseudo-instrumentation to improve profile quality without incurring the overhead of traditional instrumentation. It also enriches profile with context-sensitivity to aid more effective optimizations through a novel profiling methodology using synchronized LBR and stack sampling. CSSPGO is now used to optimize over 75% of Meta's data center CPU cycles. Our evaluation with production workloads demonstrates 1%-5% performance improvement on top of state-of-the-art sampling-based PGO.

***Index Terms*—Profile Guided Optimization, Feedback Directed Optimization, Sampling, Instrumentation, Context-sensitive Profiling, Compiler**

I. Introduction

Data centers and their compute capacity in particular, have become an integral part of modern world infrastructure. They underpin social networks, search engines, AI tools and many other services that people around the world use on a daily basis. Due to the broad spectrum of services they power and the vast population they serve, data centers nowadays have reached unprecedented scale [1]. It's imperative to optimize data-center applications to minimize both capital expenditure for service providers as well as their environment footprint on power consumption.

Profile-guided optimization (PGO) is a compiler technology widely used to optimize data-center applications. However, the use of PGO often involves a trade-off between operational overhead and peak performance. This is because, while instrumentation-based PGO provides best performance, it requires special setup and dedicated profiling, hence can be prohibitive for large-scale adoption. Sampling-based PGO, on the other hand, has low entry barriers but it does not deliver the same performance as instrumentation-based PGO.

Context-sensitive sampling-based PGO with pseudo-instrumentation (CSSPGO) proposed in this paper provides an alternative solution with better performance than traditional sampling-based PGO while maintaining low operational overhead.

A. Motivation

Large data centers often run a diverse set of workloads. Given that most of the workloads are compute-bound, optimizing CPU performance with compiler optimizations and PGO in particular has proven to be very effective. Within Meta, there are thousands of different back-end services running to serve its users. This requires optimizations to be service-agnostic and have low-operational cost to be used across the entire server fleet, and PGO is no exception. While the most performant variant of PGO is instrumentation PGO, it comes with significant operational complexity. Instrumentation adds non-trivial run-time overhead, so profiling instrumented binary requires special setup for each service, and the instrumented binary usually cannot be run in production environment. Such limitation significantly hinders its adoption.

With sampling-based PGO, profiling can be performed directly in the production environment, and the collected profile can be fed to compilation continuously, making it a low-overhead, easily scalable PGO solution for data-center applications. AutoFDO [2] is an example of sampling-based PGO that is widely used to optimize data-center applications. It often delivers double-digit percentage of performance improvement. Unfortunately, profile data from sampling is less accurate than the ones from instrumentation, and as a result its performance also lags behind the instrumentation counterpart.

To achieve better performance on top of sampling-based PGO, post-link optimizers like BOLT [14] and Propeller [16] have been explored and adopted. Unlike instrumentation-based PGO, BOLT and Propeller are also sampling-based so they can profile directly in production environments. However, they still require an additional profiling and build iteration which lengthen release pipelines and hinders adoption. At Meta, most of the compute-bound services are optimized with sampling-based PGO, while only 1/7 of them are also optimized by

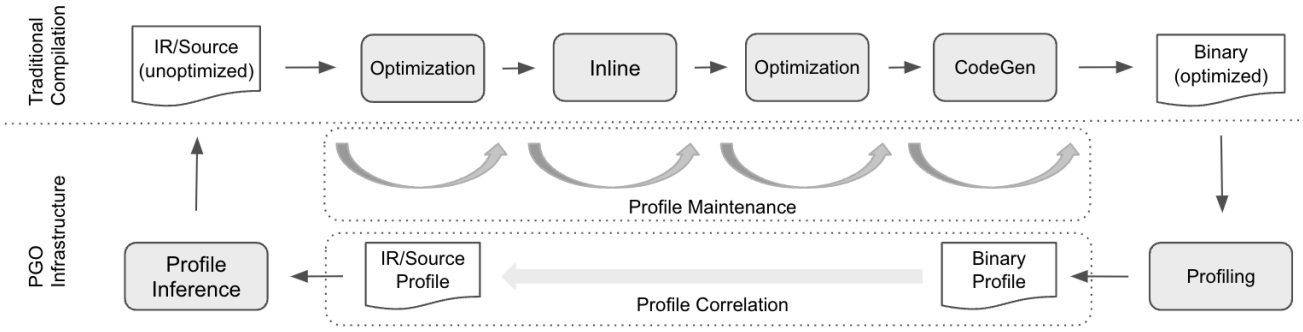


Fig. 1: Components of PGO compilation that produce and consume profile data

BOLT. There is only one service optimized by instrumentation-based PGO, and the operational overhead has been a major pain point that motivates the service owner to also explore migrating to sampling-based PGO.

With sampling-based PGO being the most leveraged optimization tool, there is strong interest in improving its capability so it can approach instrumentation-based PGO's performance. This motivated a systemic investigation into current sampling-based PGO's weakness, which in turn led to the introduction of an enhanced sampling-based PGO with context-sensitive profiling and pseudo-instrumentation (CSSPGO).

B. Contribution

In this paper, we analyze different approaches to PGO, with a focus on the trade-off between profile quality, peak performance and operational overhead. We mitigate profile quality issues, which is the biggest weakness of sampling-based PGO, by introducing flexible pseudo-instrumentation. We also take advantage of sample profiling to achieve context-sensitive profiling with no additional overhead. With pseudo-instrumentation and context-sensitive profiling combined, we demonstrate that CSSPGO can approach instrumentation-based PGO performance. Our main contributions are:

- We evaluate performance and profile quality of sampling-based PGO and instrumentation-based PGO on large server workloads.
- We introduce pseudo-instrumentation as a flexible solution for low overhead, high quality profile correlation mechanism.
- We introduce a sampling-based profiler to enrich profile with context-sensitivity, and a context-sensitive pre-inliner to achieve more selective inlining and better post-inline profile quality.
- We propose CSSPGO, an alternative sampling-based PGO system using pseudo-instrumentation and context-sensitive profiling and demonstrated it can achieve better profile quality and better performance over state-of-the-art sampling-based PGO.

II. Background

With PGO, optimizations can often improve program performance by additional double-digit percentage. Accurate control flow profile enables profile-guided optimizations to be more effective. For instance, the inliner can differentiate hot function calls from cold function calls and aggressively inline the hot ones only without bloating overall code size. Branch instructions can be inverted to maximize streamlined execution, and code layout can be improved to favor locality. Many other optimizations, such as loop unrolling or vectorization, can be better informed when they make decision on performance versus code-size trade-off.

There are two common approaches to obtaining program profile for PGO: instrumentation-based and sampling-based. Instrumentation-based PGO inserts instructions (probes) that increment pre-allocated counters to track accurate execution counts for different paths of a program during run time. Sampling-based PGO, on the other hand, uses hardware support to sample the value of the instruction pointer to approximate execution frequencies for different parts of a program.

Quality of profile is key to the effectiveness of PGO. As the compiler optimizations form a sequential pipeline, each optimization not only consumes profile data but also produces modified profile data according to the transformations it makes for the next optimization. Therefore, maintaining profile accuracy throughout the optimization pipeline is as important as obtaining high-quality profile initially. Fig. 1 shows a componentized view of PGO, and how each component affects profile quality as profile flows through.

PGO would reach theoretical upper-bound performance if the input profile to every optimization is accurate. In reality, this is not achievable due to system limitations and engineering maturity. Different variants of PGO attempt to get closer to that upper-bound with mitigation, often accompanied with significant cost on operational overhead and usability. In the remainder of this section, we describe the two most important components contributing to high quality profile, 1) profile correlation, 2) profile maintenance, with their respective challenges and mitigation from PGO variants.

A. Profile Correlation

To capture the program behavior in production environments, profile needs to be collected from an optimized binary. Therefore, the first step of PGO is associating profile data from the optimized binary to the un-optimized IR, so that the following optimization passes can consume it. Such translation is termed

"profile correlation". Sampling-based PGO uses debug information (such as DWARF [23]) as correlation anchors, so it neither prevents any optimizations nor adds run-time overhead to the profiling binary. However it suffers from inaccurate correlation with aggressive optimizations, as they often fail to maintain the debug information accurately. Aggressive optimizations can adversely impact the accuracy of profile correlation. The challenge is that when two blocks are merged, for example, execution count from merged binary block cannot be accurately correlated back to the two pre-merge IR blocks. At the cost of a slower instrumented binary, instrumentation-based PGO mitigates profile correlation inaccuracy with instrumented probes serving as optimization barriers, therefore control flow optimizations involving code merge are largely prevented since blocks with probes incrementing different counters cannot be merged. On top of less aggressive optimizations, counter increments during run time also further slow down instrumented binary, often by 50%+ and to the point that it alters the workflow by requiring special profiling setup, which incur significant operational overhead.

After profile is correlated onto un-optimized IR, there is often a profile inference stage which aims to fix profile inaccuracies to the extent possible. Profile inference is particularly useful for sampling-based PGO as it can smooth out the profile inconsistencies caused by hardware sampling [9]. Advanced profile inference [10] can also help mitigate profile correlation issues caused by aggressive optimizations, but the resulting profile quality is usually still not as good as that of instrumentation-based PGO.

B. Profile Maintenance

Once the profile correlation and inference is done, the profile will be annotated in the IR and used by optimizations sequentially. After each optimizations, profile needs to be adjusted to reflect control flow graph changes if any: if a loop is unrolled by a factor of 4, profile counts for loop body will need to be scaled down by 4 as well. While profile maintenance for optimizations like loop unroll, vectorization is more of a mechanic update and quality of such maintenance mostly depends on engineering maturity of an implementation, profile maintenance for inlining remains a challenging problem.

After inlining, the accurate profile of an inlinee is the slice of the original callee profile along the inlined calling context. Maintaining accurate post-inline profile requires the input profile to be context-sensitive. Ball et al. [7], [11] proposed a profiling scheme to obtain context-sensitive profile through instrumentation. However such technique is not used by PGO of mainstream compilers due to its complexity and extra run-time overhead. Most PGO implementations do not use context-sensitive profiling, hence they suffer from inferior post-inline profile quality, which in turn makes post-inline PGO optimizations like register allocation and code layout less effective. Some implementations rely on the inlining of the profiling build to generate a partial context-sensitive profile [2]. However, this technique requires inlining decisions to be identical between the profiling build and the optimizing build, which is hardly achievable in practice. Hence the context-sensitivity it brings into profile is limited.

Motivated by the profile quality issue after inlining, Panchenko et al. developed BOLT [14], a post-link binary optimizer, which managed to achieve better performance on top of compiler PGO for server workloads. Propeller [16] is another attempt at binary optimizer with similar motivation. There are also efforts to use a separate late-stage profile after inlining from within the compiler [20]. Most solutions used today involve a late-stage profile either by the compiler itself or by a post-link optimizer, which require an additional profiling and re-optimize iteration that can be prohibitive for large-scale adoption.

III. Sampling-based context-sensitive PGO with pseudo-instrumentation

This section describes two innovations aimed at improving quality and optimization effectiveness for sampling-based PGO, 1) pseudo-instrumentation, 2) context-sensitive sampling-based profiling, and an enhanced sampling-based PGO system, CSSPGO. CSSPGO leverages the two innovations to tackle the challenges mentioned above for profile correlation and profile maintenance, and to achieve better performance while maintaining low operational overhead.

We first introduce pseudo-instrumentation as an alternative profile-correlation mechanism that can achieve better profile quality than the debug-location-based correlation mechanism used by today's sampling-based PGO. While our attempt focuses on getting the best possible profile quality with near-zero run-time cost, we emphasize that pseudo-instrumentation also provides a flexible framework for PGO implementation to achieve the desired balance between run-time overhead and profile-correlation quality.

We also describe a novel sampling-based context-sensitive profiling methodology to obtain context-sensitive binary profile without requiring additional late stage profiling. The new profiler takes advantage of synchronized LBR (Last Branch Record) [24] and stack sampling to recover and synthesize calling context for each profiled execution. On top of context-sensitive profiler, we propose a pre-inliner that can work with modern scalable LTO [12], [13] to make global context-sensitive inline decisions. The combination of new context-sensitive profiler and inliner achieves better post-inline profile quality which helps late optimizations, in addition to driving more selective inlining decisions.

Since both pseudo-instrumentation and context-sensitive profiling are transparent to users, unlike instrumentation-based PGO or post-link optimizers, CSSPGO is as low cost as today's sampling-based PGO. When deployed to production, CSSPGO shares the exact same workflow and setup as widely adopted sampling-based PGO solutions like AutoFDO.

A. Pseudo-instrumentation

An essential component of sampling-based PGO pipeline is profile correlation. As shown in Fig. 1, it is responsible for mapping samples on fully optimized binary back to source or

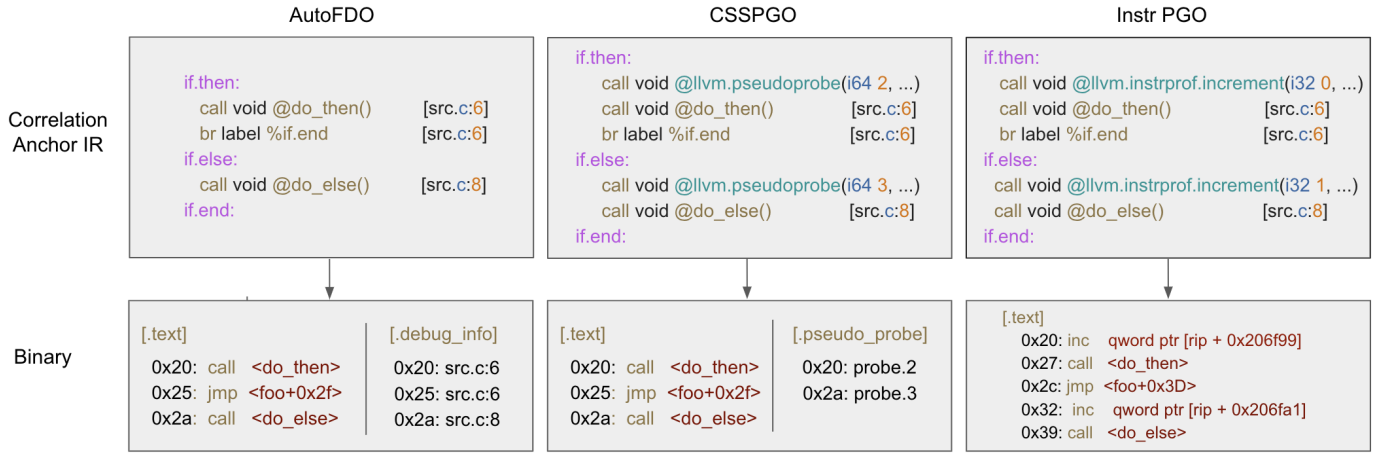


Fig. 2: Profile correlation anchors comparison for different PGO

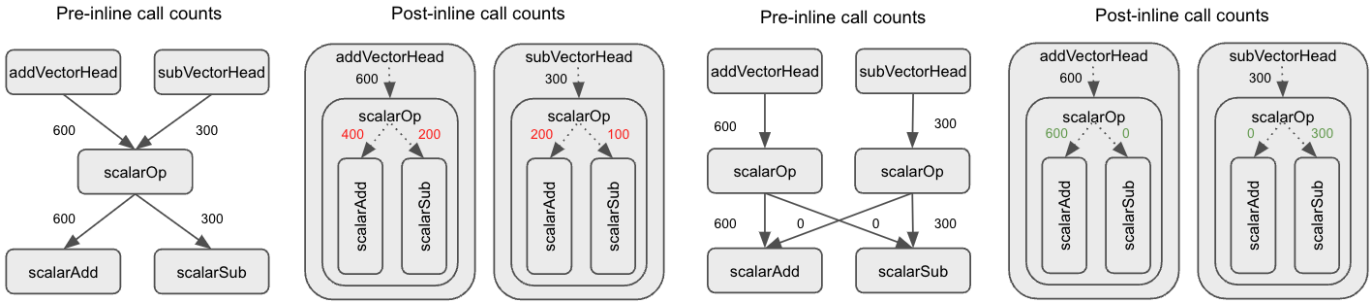
un-optimized IR locations so that they can be consumed by the compiler in the following compilation. Profile correlation used by today's sampling-based PGO is based on symbolization with debug information. While this approach has the benefit of reusing existing debug info infrastructure, it is also the main source of profile inaccuracies due to degraded quality of debug info through a fully-loaded optimization pipeline [2]. Source drifting is another cause of profile inaccuracies, where a minor change to source code can cause profile to be unusable.

To mitigate the aforementioned issues, we present pseudo-instrumentation, a flexible, low-overhead instrumentation technique that aims to be more resilient to compiler optimizations and minor source changes. It instruments the program with pseudo-probes that serve the purpose of a profile-correlation anchor for mapping a sampled execution to the IR. Its resilience to optimization is achieved by its representation as a standalone intrinsic instruction in compiler IR, as opposed to debug information attached to instructions. Such representation makes it behave like traditional instrumentation and helps preserve original control flow for accurate correlation. However, it is "pseudo" because unlike traditional instrumentation where the instrumented probes are translated to machine instructions that increment profile counter at run time, pseudo-probes do not result in any additional machine instructions. This makes it significantly lower-overhead than traditional instrumentation. In addition, the intrinsic representation offers flexibility for finer tuning to achieve a desired balance between overhead and accuracy. If an implementation can tolerate higher run-time overhead, it can choose to make pseudo-probe a stronger optimization barrier to better preserve original control flow and vice versa.

A pseudo-probe is inserted as an intrinsic into each basic block of the control-flow graph at an early stage of the optimization pipeline before any aggressive transformations. This is to minimize the effect of compiler optimization so instrumentation can be done on a stable IR. After insertion, pseudo-probes live in the form of intrinsic instructions throughout the whole compilation, at the end of which they are materialized as metadata against the location of the physical instruction next to it in the generated binary. Such metadata will be used later on during profile generation as correlation anchors, so that samples collected at those associated physical instructions can be used on the IR. The metadata is self-contained, i.e., no relocation references to or from the rest part of the binary, therefore it can be split out of the object files to minimize the impact on linking large binaries. The metadata will also not be loaded at run time, so it should have little impact on application performance.

We use a simple example to demonstrate how pseudo-probe works as correlation anchor in comparison with existing methodology. Fig. 2 lists the IR correlation anchors used by different PGO and their generated code in the LLVM compiler for an if-else construct. With traditional instrumentation-based PGO, user program is instrumented with a special intrinsic instruction `llvm.instrprof.increment`, which translates to a physical machine instruction `inc`. The intrinsic takes a counter identifier as its parameter along with other parameters omitted from the example. Note that counter increment at run time can slow down program execution significantly. On the contrary, existing sampling-based PGO does not need any instrumentation at all, instead it relies on debug info in both IR and binary to perform profile correlation. Therefore it is zero-overhead but at the cost of inferior profile quality. CSSPGO's pseudo-instrumentation uses intrinsic instructions to represent correlation anchors on the IR but materializes them as metadata in binary, so it can achieve better profile quality while maintaining low overhead.

Even if pseudo-probe does not translate into any machine instruction, it may still incur run-time cost because it may prevent some optimizations. In our implementation, we design pseudo-probe to block as little optimizations as possible to prioritize low overhead over high profile accuracy, by implementing it as a memory intrinsic with a semantics of accessing memory locations that are inaccessible by the user code. Therefore it does not interfere with user code which the optimizer is free to optimize. The implementation also



(a) Inaccurate post-inline counts w/ regular profile

(b) Accurate post-inline counts w/ context-sensitive profile

Fig. 3: Relationship between context-sensitivity and post-inline profile accuracy

```
Elem AddVectorHead(Vector V1, Vector V2) {
  return scalarOp(V1[0], V2[0], OpAdd);
}
Elem subVectorHead(Vector V1, Vector V2) {
  return scalarOp(V1[0], V2[0], OpSub);
}
Elem scalarOp(Elem E1, Elem E2, Opcode Op) {
  switch (Op) {
  case OpAdd:
    return scalarAdd(E1, E2);
  case OpSub:
    return scalarSub(E1, E2);
  }
}
```

Fig. 4: Code showing the benefit of context-sensitive profile

disallows the optimizer to move pseudo-probe around or drop it in a way that its execution frequencies will change. While such semantics typically does not affect optimizations such as code motion, inlining or loop vectorization, it may still block control-flow optimizations such as if-convert and tail merge. To achieve near-zero cost, we fine-tune a few critical optimizations, including if-convert, machine sink and instruction scheduling, to be unblocked by pseudo-probe.

We now discuss how pseudo-instrumentation can mitigate profile inaccuracies caused by common compiler optimizations. In general, there are two types of optimizations that can damage profile quality:

a) Code Merge: Optimizations that perform code merge, such as tail merge, can cause profile inaccuracy because there is no reasonable way to distribute merged profile counts back to the original program locations. Such optimizations can be blocked by pseudo-probes due to their different signatures for different blocks, hence the original control flow can be preserved.

b) Code Duplication: A statement in source program can compile into multiple binary instructions. When optimizations, such as loop invariant code motion, move some of them into a colder code region, those instructions will have different execution frequencies. Given that optimization tends to move instructions into colder region instead of hotter region, correlation techniques using debug info take the maximum execution frequency from those instructions. Unfortunately, such heuristic cannot handle optimizations involving code duplication where the frequency of each duplicated instruction should be added together. Dwarf discriminators [23] can be used to mark certain code duplication as a mitigation. However, it's not scalable since inserting annotation for all possible code duplication in compiler is not practical. Unlike the one-to-many mapping between the source location and the generated instructions, pseudo-probes maintains one-to-one mapping. This allows adding frequencies of copied probes to derive accurate original frequency.

Besides compiler optimizations, source code changes can also introduce profile inaccuracies for the existing sampling-based PGO. For instance, a minor change in the source code such as adding or removing a program comment, can cause location of subsequent code to shift. Consequently, profile corresponding to previous source may become obsolete to the new source. While the original AutoFDO work [2] also attempts to mitigate such source code drift issue by using line offset instead of the control flow for profile correlation, the AutoFDO profile can contain partially correct profile information. In practice, we have observed minor source drift causing 8% performance loss for a server workload. This problem is mitigated with pseudo-instrumentation where a checksum reflecting the shape of the IR control-flow graph (CFG) is computed and persisted in the profile. A CFG change would be detected as a mismatch between the profile checksum and the IR checksum, but any changes not altering CFG, such as adding comments, can be handled transparently.

B. Context-sensitive Sampling-based PGO

Context-sensitive profile can provide fine grain insights into control flow of a function based on different calling context. This is especially useful in maintaining accurate post-inline profile and in driving selective inline decisions. As mentioned in section II, post-inline profile quality is critical for the effectiveness of late stage optimizations like register allocation and code layout. In this section, we first demonstrate how context-sensitive profile can help maintain accurate post-inline profile with a simple example. Then we describe a novel context-sensitive sample profiler. Unlike existing solutions, our approach does not require separate post-inline profile, nor does it rely on inlining of profiling build. Lastly, we introduce an inlining scheme that can work with modern scalable LTO, like

ThinLTO [12] in LLVM, to fully leverage richer input profile and achieve better inline decision as well as more accurate post-inline profile in coordination with context-sensitive profiling.

Fig. 3 illustrates post-inline profile accuracy difference with and without context-sensitive profile using a simple example shown in Fig. 4. We can see that `scalarAdd` can only be called from `addVectorHead->scalarOp` path, and `scalarSub` can only be called from `subVectorHead->scalarOp` path. However, without context-sensitive profile, such insight won't be available. When looking at context-insensitive profile for `scalarOp`, we only know that it calls both `scalarAdd` and `scalarSub`. As a result, if we end up inlining all function calls, we can only scale the call counts for `scalarOp`, which is inaccurate as shown in Fig. 3a. Such inaccurate post-inline profile can mislead later optimizations, potentially causing sub-optimal spill placement and more branches on hot paths.

If context-sensitive profile is available, compiler would be able to see two different profiles for `scalarOp` based on the calling context, i.e. whether it's called by `addVectorHead` or `subVectorHead`. This enables the inliner to maintain accurate post-inline counts like what's shown in Fig. 3b. As evident by the success of BOLT [14], accurate post-inline can improve performance through more effective late stage optimizations.

a) Context-sensitive Sample Profiler: We present a low-cost sample profiling methodology to obtain context-sensitive profile. With LBR (Last Branch Record) profiling, an interrupt is triggered each time the PMU (Performance Monitoring Unit) underflows, and we take a snapshot of LBR, consisting of 16 or 32 pairs of source and target addresses for consecutive taken branches, from which we can derive a sequence of linear execution paths. By accumulating the linear execution paths from all samples, we can then construct control-flow profile for functions, which is the input for today's sampling-based PGO like AutoFDO. On top of LBR sampling, our proposed profiling method also enables synchronized stack sampling to obtain calling context for each LBR sample, so each time when PMU underflow triggers an interrupt, in addition to taking a snapshot of LBR, a stack sample is also recorded. Fig. 5 shows an example of synchronized LBR and call-stack sample.

Given that stack sample and LBR sample are collected at the same time, for the last branch of an LBR sample, the collected stack sample naturally identifies its calling context. If all branches in an LBR sample are from the same frame, i.e. the LBR sample does not contain any call or return, they would share the same calling context identified by the stack sample. If an LBR sample contains calls or returns, we need to process the stack sample to unwind the call or return from LBR in reverse order to recover the accurate calling context for branches before the call or return. We also need to take implicit call and return from inlining into account when recovering accurate calling context. Specifically, for a linear execution path without branches, we need to check for change of inlined stack and adjust context accordingly. The process of recovering calling context for each LBR branch is illustrated in Algorithm 1. Note that LBR branches are processed in reverse execution order.

Algorithm 1 Reconstruct context from LBR and stack sample

Input: $LBRBranches, Stack$
Output: $RangesWithContext$

for $CurrBranch$ in $LBRBranches$ **do**
  if $CurrBranch$ is CALL **then**
    PopLeafFrames($Stack$, 1)
  else if $CurrBranch$ is RETURN **then**
    $Frame \leftarrow$ GetFrameFromAddr($CurrBranch.Source$)
    $Stack \leftarrow$ PushLeafFrames($Stack$, $Frame$)
  else if $PrevBranch$ is NULL **then**
    $PrevBranch \leftarrow CurrBranch$
    continue
  end if
  $Context \leftarrow$ ExpandInlinedFrames($Stack$)
  $BeginAddr \leftarrow EndAddr \leftarrow CurrBranch.Target$
  $EndLimit \leftarrow$ NextInstrAddr($PrevBranch.Source$)
  while $EndAddr <= EndLimit$ **do**
    $BeginFrames \leftarrow$ GetInlinedFrames($BeginAddr$)
    $EndFrames \leftarrow$ GetInlinedFrames($EndAddr$)
    $SameContext \leftarrow BeginFrames$ equal $EndFrames$
    if $SameContext$ and $EndAddr$ is not $EndLimit$ **then**
      $EndAddr \leftarrow$ NextInstrAddr($EndAddr$)
      continue
    end if
    $Context \leftarrow$ PushLeafFrames($Context$, $BeginFrames$)
    $RangesWithContext$[$BeginAddr$, $EndAddr$-1] = $Context$
    PopLeafFrames($Context$, $BeginFrames.size$)
    $BeginAddr \leftarrow EndAddr$
    $EndAddr \leftarrow$ NextInstrAddr($EndAddr$)
  end while
  $PrevBranch \leftarrow CurrBranch$
end for

In order to recover accurate context, special care is needed to mitigate a few challenges in a practical implementation:

- Synchronizing LBR and stack sample: The reconstruction of calling context is based on the assumption that the sampled stack is perfectly aligned with the frame that the last branch's target points to. In practice, this is not guaranteed even if LBR and stack sample are taken from the same interrupt. Due to sampling skid [3], we observed that stack sample can sometimes lag behind LBR sample by one frame. Fortunately, PEBS (Precise Event-Based Sampling) [4] can be used to eliminate the skid so both stack sample and LBR sample are always synchronized. With linux perf [22], using `perf record -g --call-graph fp -e br_inst_retired.near_taken:upp ...` can enable level-2 precision and achieve synchronous LBR and stack sampling.
- Reliable stack sampling: In order to obtain accurate stack samples, frame pointer is needed. However, some optimizations can destroy frame chain. The most common ones are frame pointer omission (FPO) and tail call elimination (TCE). FPO is commonly disabled in production environment, because having accurate stack sample is critical not only for PGO but also for debugging. TCE can also cause missing frames for immediate caller of a tail call. As a mitigation, a missing frame inferrer is introduced to recover caller frames of consecutive tail calls whenever possible. The idea is to build a dynamic

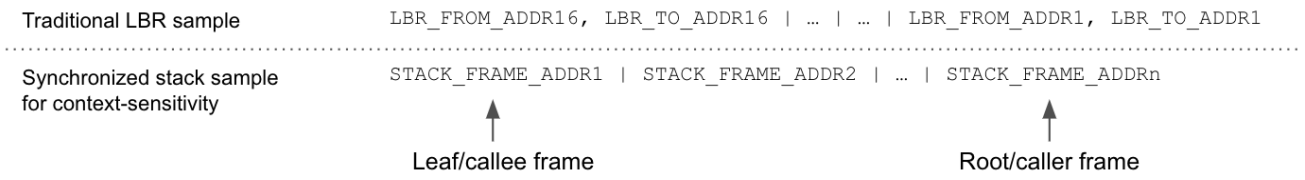


Fig. 5: Synchronized LBR and call stack sample

call graph that consists of only tail call edges constructed from LBR samples and do a DFS-search on that graph to find a unique path for a given pair of parent and child frame. The unique tail-call path is then used to fill in the missing frames between the source and target frame. Note that there could be multiple tail-call paths available for a give pair of frames, in which case the inference will fail. In practice it is observed that more than two-thirds of the missing tail call frames can be recovered.

- Scalability: Context-sensitive profile contains multiple versions of profile for a function when there are multiple possible calling contexts. So it is expected to be larger in size than context-insensitive profile. However, larger profile can lead to slower profile generation and slower PGO compilation as it takes longer to read/write and process. For programs with a dense dynamic call graph, profile size increase due to context-sensitivity can be on the order of 10x, therefore incurring significant overhead. Since cold functions are unlikely to be inlined, we mitigate the profile size increase by only keeping context-sensitive profile for hot functions and trim profiles for cold functions to be context-insensitive. Experiment shows that our mitigation can produce context-sensitive profile comparable in size to regular profile, without loosing its benefit.

The above profiling methodology can obtain a context-sensitive profile without a separate late stage profile, hence it is transparent to users. It also does not rely on inlining of previous build, making it a practical and effective solution.

b) Context-sensitive Pre-inliner: Context-sensitive profile can enable more selective inlining and more accurate post-inline profile as mentioned above. In practice, it needs a cooperating inliner that can fully leverage richer profile to reach its full potential. In this section, we describe a context-sensitive pre-inliner that can be retrofitted to compiler implementations without a proper profile-guided inliner, and is also effective in mitigating challenges imposed by modern scalable whole-program optimization framework.

In order to have accurate profile for a function, a compiler needs to take into account the inline decisions for all calls to this function. If it is inlined at one call site, context-sensitive profile along that call path should be excluded in the function's base profile; otherwise, the corresponding context-sensitive profile should be merged back into that function's base profile. This implies two requirements for the inliner: 1) it needs to visit call sites and make inlining decisions in call graph's top-down order; 2) it needs to be able to move and merge context-sensitive profile in the call graph based on inlining decision. Meeting the two requirements can be challenging as we explain below.

Top-down order is important for profile-guided inlining because it allows specialization of inline decisions based on calling context and its associated context-sensitive profile. In the example shown in Fig. 3, given context-sensitive profile (Fig. 3b), a top-down inliner would be able to inline only `scalarAdd`, but not `scalarSub` into `scalarOp` if the top level inliner is `addVectorHead`. This leads to more selective inlining. However, it is not achievable with a bottom-up inliner because the decision to inline `scalarAdd` or `scalarSub` into `scalarOp` is made before the decision to inline `scalarOp` into `addVectorHead` or `subVectorHead`; additionally, once the inlining decision for `scalarOp` is made, both `addVectorHead` and `subVectorHead` will have to use the same `scalarOp` as it does not support specialization based on calling context.

For many compiler implementations, initially they are not built with profile-guided inlining as a priority, hence they do not have a top-down inliner. LLVM is an example where inlining happens in bottom-up order as part of its call graph (CGSCC) pass. For a mature compiler, it is difficult to make fundamental changes to inlining order as inliner is often coupled with other optimizations. AutoFDO mitigated this problem by adding an early inliner for profile-guided top-down inlining. However, early inliner operates on less optimized IR, so it is often challenged by inaccurate inlining cost estimation and needs to be conservative.

Modern whole-program or cross-module optimization frameworks like ThinLTO [12] or LIPO [13] pose another challenge to profile-guided inlining. They make LTO compilation scalable by isolating the compilation of different modules or clusters so they can be parallelized and distributed. They perform whole-program analysis on a summary or index, and they pass analysis results and optimization decisions to parallelized compilations for execution. Such frameworks greatly speed up LTO compilation, but they make profile adjustment across modules based on inlining decision impossible.

To mitigate the aforementioned challenges, and to maximize the benefit of context-sensitive profiling and inlining, we present a context-sensitive pre-inliner that runs before compilation as part of offline profile generation to make global top-down inline decisions with profile and inline cost estimates. The pre-inline adjusts profile based on its own inline decision and persists its decision in the generated profile so it can be passed to compiler, which will try to honor the decision made by pre-inline when

possible. With inlining decision and profile adjustment done in pre-inliner, the limitation imposed by ThinLTO can be worked around so we can still have accurate post-inline profile. The workflow of pre-inliner including adjustment for context-sensitive profile is illustrated by Algorithm 2. For inliner, the most important inputs to its heuristic are: 1) call site hotness, 2) cost of inlining, usually proxied by size increase estimate after inlining a call site. Pre-inliner takes the whole-program profile and the profiled binary as input. It uses the input profile to build up call graph top-down order and to get call site hotness. For cost of inlining, it retrieves the actual function size from the previously built profiling binary as proxy, and it also differentiates function sizes among different inlined copies using a trie, which is usually more accurate than cost estimate on early-stage IR. Algorithm 3 details how function size is extracted from profiling binary. We observed that extracted size can often accurately tell the pre-inliner that certain functions will eventually be fully optimized away.

Algorithm 2 Top-down pre-inliner w/ context-sensitive profile

```
for Function in GetTopDownOrder(ProfiledCallGraph) do
    Profile ← GetBaseProfileFor(Function)
    for ContextProfile in GetContextProfilesFor(Function) do
        if not ContextIsInlined(ContextProfile) then
            MoveContextProfileToBase(Profile, ContextProfile)
        end if
    end for
    FuncSize ← GetFuncSize(Profile)
    Candidates ← GetCallees(Profile)
    while Candidates not empty and FuncSize < Limit do
        Candidate ← PopMostBeneficial(Candidates)
        CandidateSize ← GetFuncSize(Candidate.Profile)
        Hotness ← GetCallHotness(Candidate)
        if ShouldInline(CandidateSize, Hotness) then
            MarkContextInlined(Candidate.Profile)
            FuncSize ← FuncSize + CandidateSize
            NewCandidates ← GetCallees(Candidate.Profile)
            Enqueue(Candidates, NewCandidates)
        end if
    end while
end for
```

Algorithm 3 Compute context-sensitive inline cost

```
Input:  Binary
Output: FuncSizeForContext
for Function in Binary do
    Addr ← Function.StartAddr
    while Addr <= Function.EndAddr do
        InlinedFrames ← GetInlinedFrames(Addr)
        InstrSize ← GetInstrSizeForAddr(Addr)
        FuncSizeForContext[InlinedFrames] += InstrSize
        PopLeafFrames(InlinedFrames, 1)
        while InlinedFrames not empty do
            Size ← FuncSizeForContext[InlinedFrames]
            if Size is unknown then
                FuncSizeForContext[InlinedFrames] = 0
            end if
            PopLeafFrames(InlinedFrames, 1)
        end while
        Addr ← Addr + InstrSize
    end while
end for
```

With context-sensitive profiling and the pre-inliner combined, CSSPGO is able to further improve performance while reducing code size for server workloads. We will discuss the results in the next section.

IV. Evaluation

This section evaluates our implementation of CSSPGO in LLVM [17] using Meta's production server workloads. We first compare CSSPGO performance against AutoFDO and show that CSSPGO can improve performance by 1-5% on LTO optimized server workloads while generating smaller code. In the case of HHVM workload [25], it can bridge 60% of the performance gap between instrumentation-based PGO and AutoFDO. We then demonstrate that pseudo-instrumentation has negligible run-time overhead, in contrast with the 73% slowdown from instrumentation on HHVM. Next, we conduct profile quality analysis to quantify profile quality improvements from CSSPGO. Additionally, we evaluate CSSPGO on an client workload, the open-source Clang compiler [17], to demonstrate that the improvements from CSSPGO is workload agnostic and not limited to Meta's server workloads.

A. Performance and Code Size

All performance evaluations are done on Skylake DE servers with 64GB RAM. The compiler we used is a fork of LLVM-15, with CSSPGO implemented on top. The server workloads we used are among the most compute-intensive workloads that consume about one-third of total CPU cycles of Meta's server fleet, namely, AdRanker, AdRetriever, AdFinder, HHVM and HaaS, all optimized with ThinLTO [12]. AdRanker, AdRetriever and AdFinder are backend services for Ads retrieval and ranking. HHVM [25] is a JIT compiler for Hack and PHP that powers the Facebook website. HaaS is a JavaScript remote-execution service based on Hermes [26]. We evaluate the performance of CSSPGO in a production environment, with live traffic being duplicated and flow through two different systems built from identical source but using different PGO variants on top of ThinLTO. We use the same production setup for profile collection as well. For HHVM, CPU utilization is controlled as an invariant by a load driver, so RPS (requests per second) alone is used as the performance metric. For other workloads, performance improvements can manifest as either CPU utilization drop or throughput increase or both, so a synthesized performance metric is used to capture both CPU and throughput changes. The synthesized metric represents the total CPU utilization improvements with throughput delta translated to equivalent CPU utilization changes.

PGO performance is affected by both profile quality and the optimization pipeline, as a result different PGO variants make different tweaks and tunings for their optimizations to accommodate their own profile quality. For the purpose of measuring the impact of CSSPGO, pseudo-instrumentation and context-sensitivity specifically, we try to align the optimization pipeline to the extent possible for fair comparison. Since CSSPGO by default uses Profi [10], an advanced profile inference component, we also turned on Profi for AutoFDO in our experiments. Additionally, we enabled function splitting, Ext-TSP block layout [15] for all variants of PGO we

tested. However, we acknowledge that discrepancies in the optimization pipelines still exist, with the most notable one being value-profile-based optimizations, which is an advantage of instrumentation-based PGO.

The AutoFDO baseline setup we used represents the best results we can get with AutoFDO. Note that we didn't use FS-AutoFDO [21] in our baseline. FS-AutoFDO is a recent enhancement to AutoFDO that achieves separate late stage profile annotation via multiplexing a single input profile using discriminator encoding techniques, and it can improve AutoFDO performance when profile and code generation is very stable between iterations. We didn't use FS-AutoFDO because in production environment, such stability requirement often cannot be met, in which case its late stage profile annotation may degrade profile quality. For our production workloads, we found that FS-AutoFDO enhancement led to regression for this reason, so we evaluated AutoFDO without such enhancement for the best baseline results.

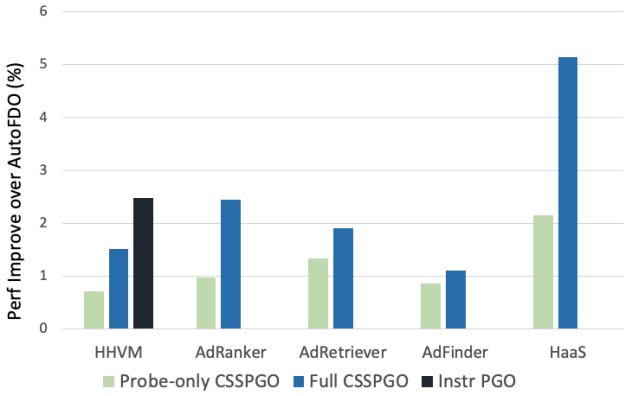


Fig. 6: CSSPGO performance comparison with AutoFDO and Instr PGO

Fig. 6 shows the performance improvements from CSSPGO using AutoFDO as the baseline. For the 5 server workloads we evaluated, CSSPGO delivers an additional 1-5% performance on top of AutoFDO. We also tried to compare CSSPGO performance against instrumentation-based PGO. On HHVM, CSSPGO is able to deliver 1.5% extra performance while instrumentation-based PGO delivers 2.4% performance on top of AutoFDO. Unfortunately, we were not able to get instrumentation-based PGO numbers for other workloads. Due to the overhead from instrumentation, profiling runs were too slow, which triggered internal health checks to fail and processes being terminated before they were able to serve live traffic. The incomplete data for instrumentation-based PGO is also a reflection of the challenges for its large-scale adoption in production due to its overhead. On HHVM, the only workload with instrumentation-based PGO data, CSSPGO bridged over 60% of the performance gap between instrumentation-based PGO and AutoFDO without any additional overhead on profile collection.

Fig. 6 also shows a breakdown of CSSPGO performance. While full CSSPGO leverages both pseudo-instrumentation and context-sensitivity, probe-only CSSPGO is a variant of CSSPGO that only uses pseudo-instrumentation. Experiments show that pseudo-instrumentation contributes to 38-78% of CSSPGO's total performance gain, with the rest coming from context-sensitivity.

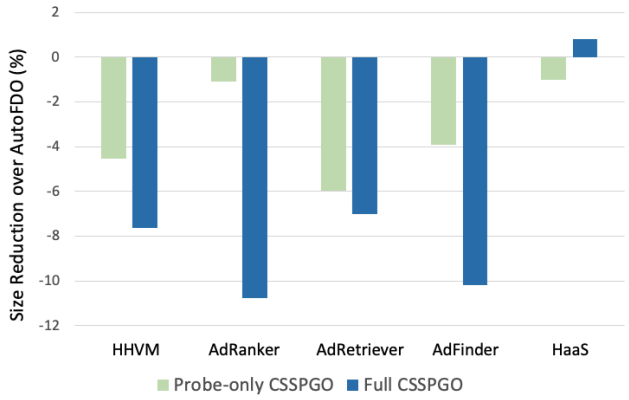


Fig. 7: CSSPGO code size comparison with AutoFDO

Note that CSSPGO also produces smaller code compared to AutoFDO on the workloads we tested, with the exception of HaaS. Fig. 7 compares code size between AutoFDO, probe-only CSSPGO and full CSSPGO. We can see that CSSPGO produces noticeably smaller code across 4 of the 5 workloads, however code size from probe-only CSSPGO is bigger than full CSSPGO. This is explained by more selective inlining from context-sensitive profile and the new pre-inliner only available in full CSSPGO. For HaaS, the code size changes are small, all within 1%. Given that HaaS sees the biggest performance improvement from CSSPGO, there may be opportunity to fine tune CSSPGO and its pre-inliner in particular to use the code size savings to achieve more aggressive optimization and better overall performance.

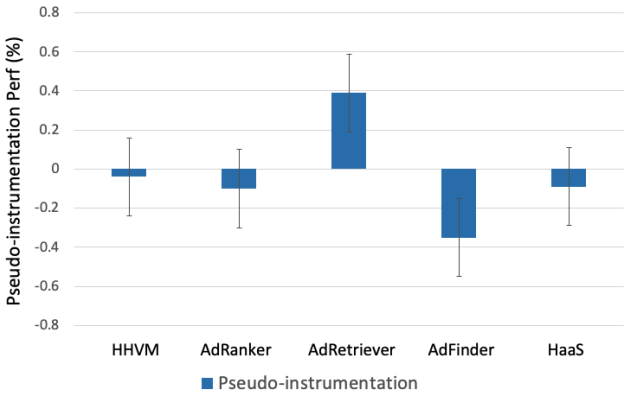


Fig. 8: Performance of pseudo-instrumentation

B. *Profiling Overhead*

Earlier in the paper, we described pseudo-instrumentation as a flexible framework for low overhead, high-quality profiling. In this section, we measure the profiling overhead of our implementation that prioritizes low overhead over high accuracy. With experiments in production environment, we demonstrate

that the overhead from pseudo-instrumentation is negligible on server workloads. Fig. 8 shows a performance comparison between pseudo-instrumentation enabled and disabled, using the same production setup mentioned above. The performance delta is within the P95 confidence interval shown by the error bar for three workloads. Surprisingly, AdRetriever's performance is slightly better with pseudo-instrumentation. This can happen when the inserted pseudo-probes block undesirable optimizations. In contrast, we measured 73% slow down when profiling instrumented HHVM as shown in Table I.

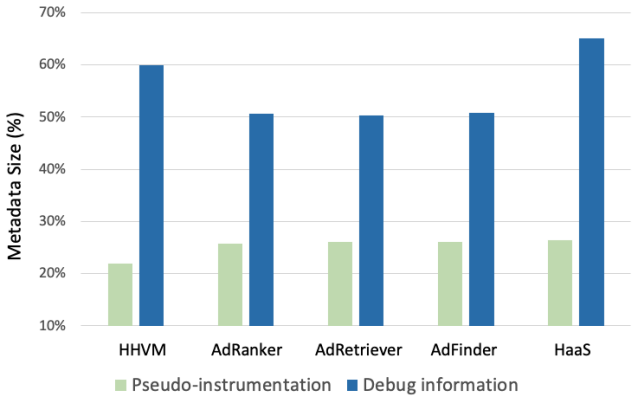


Fig. 9: Size overhead of metadata

Fig. 9 lists the size overhead of the pseudo-instrumentation metadata for each server workload, expressed as a percentage of the total binary size including the debug information generated under the -g2 option. The size overhead of the debug information for the same binary is also shown in the figure for comparison. On average, the pseudo-instrumentation metadata takes about 25% of the binary size. As discussed earlier, the metadata is self-contained, i.e., no relocation references to or from the rest part of the binary, therefore it can be split out of the binary to minimize the concern on binary size if needed. The metadata won't be loaded at run time, so it won't impact application's performance.

C. Profile Quality

The performance improvements from CSSPGO shown above can partly be attributed to the profile-quality improvements that CSSPGO brings. In this section, we take a quantitative approach to analyze profile quality and demonstrate its direct improvements. We evaluate CSSPGO's profile quality using a block-overlap metric with instrumentation-based PGO profile as the ground truth and compare CSSPGO against AutoFDO [2]. The block-overlap metric is commonly used to compare the similarity of two profiles with respect to a common control-flow graph.

Let V be the set of basic blocks in a flow graph of a function. We define block overlap degree for the function as

$$D(V) = \sum_{v \in V} \min\Big(\frac{f(v)}{\sum_{v \in V} f(v)}, \frac{gt(v)}{\sum_{v \in V} gt(v)}\Big),$$

where $gt(v)$ is the ground-truth count, i.e, the instrumentation-based PGO count for a block v, and $f(v)$ is the CSSPGO or sampling-based PGO count for the same block. The block overlap degree for a program is then defined as a weighted aggregation over all functions:

$$D(P) = \sum_{V \in P} D(V) \frac{\sum_{v \in V} f(v)}{\sum_{V \in P} \sum_{v \in V} f(v)}.$$

We only use HHVM to measure the profile quality since it is the only workload we were able to optimize with instrumentation-based PGO successfully for reasons mentioned earlier. Table I shows block overlap and profiling overhead for HHVM with AutoFDO, CSSPGO and instrumentation-based PGO. The block overlap degree for CSSPGO is 92.3%, while it is 88.2% for AutoFDO. CSSPGO pushes the profile quality a step closer to the instrumentation-based PGO. Also note that CSSPGO achieved better profile quality while keeping profiling overhead near-zero, which is significantly lower than the 73% overhead from instrumentation-based PGO.

TABLE I: HHVM profile quality and profiling overhead

	AutoFDO	CSSPGO	Instr PGO
Block overlap	88.2%	92.3%	100%
Profiling overhead	0%	0.04%	73.06%

D. Client Workloads

We now evaluate CSSPGO with client workloads to demonstrate that the improvements from CSSPGO are workload-agnostic, even though CSSPGO was developed with a focus on Meta's production server workloads. We choose the well-known open-source Clang compiler [17] and its release/17.x branch as the benchmark and measure its bootstrapping performance.

We first built an initial version of Clang for training purpose, with the same setup used in server workloads evaluation earlier. Next, the training compiler was launched to build Clang again with the default options. The collected profile data was then used in a subsequent build to produce PGO optimized Clang. Lastly, we used the PGO optimized Clang to build Clang itself again for performance evaluation.

Using AutoFDO as the baseline, we have measured a 2.8% performance improvement from CSSPGO with a 5.5% code size reduction, and a 6.6% improvement from instrumentation-based PGO with a 34% code size reduction. Compared to server workloads, there's a larger gap between sampling-based PGO and instrumentation-based PGO on client workloads, which has to do with the limitation of sampling-based profiling itself. Server workloads usually have long running steady state which enables sampling-based profiling to have decent coverage on all executed code path. On the other hand, with the same training setup, sampling-based profiling may cover a much smaller portion of executed code on client workloads comparing to instrumentation-based profiling. This can lower the ceiling for sampling-based PGO performance on client workloads. However, despite the unique challenge for client workloads,

CSSPGO is still able to provide a meaningful performance uplift.

V. Related Work

Profile-guided optimization has been used extensively to improve program performance. Over the past decades, research in this area has focused on lowering profiling and training cost, improving profile accuracy and augmenting code optimizations. Ball et al. [18], [19] lowered instrumentation's run-time cost by optimizing probe placement using minimal spanning tree. This technique dramatically reduced the run time overhead and is widely used by commercial compilers nowadays. Unfortunately as discussed throughout the paper, the cost is still unacceptable in some circumstances. Cho et al. [6] proposed a novel instrumentation framework to reduce profiling cost by combining dynamic instrumentation and sampling. They reported an average 3% to 6% run time slowdown which is still undesirable for production deployment. In addition, it is unclear how profile correlation from binary counters to the source could be done and what level of profile accuracy can be achieved. As hardware performance monitoring units (PMU) evolve and feature more reliable and precise sampling, they have been used to achieve zero-cost profiling on modern processors and drive profile guided compiler optimization [2], [3], [5], [27]–[32]. Our system also takes advantage of hardware PMU to be highly-efficient while achieving a higher profile accuracy.

While hardware PMUs help reduce profiling overhead, due to their sampling nature and hardware limitations, they often do not provide consistent samples along all program paths thus results in inferior profile accuracy. Research has been done in this area to mitigate the problem. Levin et al. [9] developed a profile inference framework based on minimal cost flow (MCF) to smooth sample profile inconsistencies. Advanced profile inference [10] has also been developed to further mitigate the issue. Apart from hardware related profile inconsistencies, these techniques also help mitigate profile inaccuracies caused by compiler optimizations. Liu et al. [8] studied sample inconsistencies from hardware point of view. They unveiled that a higher profile accuracy can be achieved by using multiple hardware event profiles. FS-AutoFDO [21] is a recent enhancement to sampling-based PGO that improves profile accuracy by using hierarchical discriminators. As discussed previously, FS-AutoFDO needs a stable profile and code generation between iterations to achieve its peak performance. All works mentioned above can be combined with CSSPGO to further boost profile accuracy.

Research has also been done in the area of extending profile content to strengthen compiler optimizations for better code quality. Ammons et al. [11] proposed an instrumentation-based approach to achieve a flow- and context-sensitive profile. However the advantage of CSSPGO is that its sampling-based profile collection incurs little run time overhead. Recent effort also uses a separate late-stage post-inline profile to improve the context-sensitivity of instrumentation-based PGO [20]. It requires an extra iteration of profiling and optimization, thus can be prohibitive for large-scale adoption. Additionally, unlike our work, the context-sensitivity achieved through separate post-inline profile cannot be used to drive better inline decision.

VI. Conclusion

The ever increasing scale of modern data centers demands more effective PGO solutions, while the diverse set of workloads running in those data centers challenges the scalability of PGO. In this paper, we present CSSPGO, a balanced solution that can deliver better performance than state-of-the-art sampling-based PGO while keeping profiling and operational overhead at minimum. CSSPGO consists of two components, pseudo-instrumentation and context-sensitive profiler accompanied with a pre-inliner. Pseudo-instrumentation provides a flexible framework for PGO implementations to achieve a desired balance between profiling overhead and profile quality. We demonstrated that pseudo-instrumentation can achieve better profile quality without additional overhead. The novel context-sensitive profiler we present offers a zero-cost way to obtain context-sensitive profile without needing separate profiling, and new pre-inliner mitigates the challenges imposed by modern scalable LTO to fully leverage context-sensitive profile for better inlining.

We evaluated our implementation of CSSPGO in LLVM on Meta's production server workloads. We demonstrated CSSPGO can deliver 1-5% additional performance on top of AutoFDO, a mature implementation of sampling-based PGO, bridging over 60% of the performance gap between AutoFDO and instrumentation-based PGO without additional overhead. The balance between performance and usability makes CSSPGO a practical solution for large-scale adoption. As of today, CSSPGO is used to optimize over 75% of Meta's data-center CPU cycles. Although our study focuses on Meta's server workloads, we believe CSSPGO is workload agnostic and can be applied to improve performance of other workloads too. Future work may explore a different overhead and performance balance with CSSPGO to further approach instrumentation-based PGO performance.

Acknowledgment

We would like to thank Sergey Pupyrev and Julian Mestre for their contribution in integrating advanced profile inference with CSSPGO. We also thank Mark Santaniello and Jordan Rome for their work in Meta's profiling infrastructure to support CSSPGO. Last, we thank Guilherme Ottoni and the anonymous reviewers for their valuable feedback on earlier drafts of this paper.

References

[1] https://datacenters.atmeta.com/

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

[3] D. Chen, N. Vachharajani, R. Hundt, S.-w. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng. Taming hardware event samples for fdo compilation. In Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '10, pages 42–52, New York, NY, USA, 2010. ACM.

[4] Intel Corporation. Volume 3B: System Programming Guide, Part 2. Intel 64 and IA-32 Architectures Software Developer's Manual, 2016.

[5] Ren, G., Tune, E., Moseley, T., Shi, Y., Rus, S. and Hundt, R., 2010. Google-wide profiling: A continuous profiling infrastructure for data centers. IEEE micro, 30(4), pp.65-79.

[6] H. K. Cho, T. Moseley, R. Hank, D. Bruening and S. Mahlke, "Instant profiling: Instrumentation sampling for profiling datacenter applications," Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Shenzhen, China, 2013, pp. 1-10, doi: 10.1109/CGO.2013.6494982.

[7] Ball, Thomas, and James R. Larus. "Efficient path profiling." In Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29, pp. 46-57. IEEE, 1996.

[8] Liu, X.H., Peng, Y. and Zhang, J.Y., 2016. A sample profile-based optimization method with better precision. In Proc. Int. Conf. Artif. Intell. Comput. Sci (pp. 340-346).

[9] Levin, R., Newman, I. and Haber, G., 2008. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In High Performance Embedded Architectures and Compilers: Third International Conference, HiPEAC 2008, Göteborg, Sweden, January 27-29, 2008. Proceedings 3 (pp. 291-304). Springer Berlin Heidelberg.

[10] He, W., Mestre, J., Pupyrev, S., Wang, L. and Yu, H., 2022. Profile inference revisited. Proceedings of the ACM on Programming Languages, 6(POPL), pp.1-24.

[11] Ammons, G., Ball, T. and Larus, J.R., 1997. Exploiting hardware performance counters with flow and context sensitive profiling. ACM Sigplan Notices, 32(5), pp.85-96.

[12] Johnson, T., Amini, M. and Li, X.D., 2017, February. ThinLTO: scalable and incremental LTO. In 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (pp. 111-121). IEEE.

[13] X. D. Li, R. Ashok, and R. Hundt, Lightweight feedback-directed cross-module optimization. Proceedings of the In-ternational Symposium on Code Generation and Optimization, pp. 53–61, 2010.

[14] Panchenko, M., Auler, R., Nell, B. and Ottoni, G., 2019, February. Bolt: a practical binary optimizer for data centers and beyond. In 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (pp. 2-14). IEEE.

[15] Newell, A., and Pupyrev, S., 1 Dec. 2020, Improved Basic Block Reordering. IEEE Transactions on Computers, vol. 69, no. 12, pp. 1784-1794.

[16] Shen, H., Pszeniczny, K., Lavaee, R., Kumar, S., Tallam, S. and Li, X.D., 2023, January. Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (pp. 617-631).

[17] Lattner, C. and Adve, V., 2004, March. LLVM: A compilation framework for lifelong program analysis and transformation. In International symposium on code generation and optimization, 2004. CGO 2004. (pp. 75-86). IEEE.

[18] Ball, T. and Larus, J.R., 1994. Optimally profiling and tracing programs. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(4), pp.1319-1360.

[19] Goldberg, A.J., 1991. Reducing overhead in counter-based execution profiling. Computer Systems Laboratory, Stanford University.

[20] [PGO] context sensitive PGO, https://reviews.llvm.org/D54175.

[21] [SampleFDO] Flow Sensitive Sample FDO (FSAFDO) profile loader, https://reviews.llvm.org/D107878.

[22] https://perf.wiki.kernel.org/.

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

[24] Rajwar, R., Lachner, P., Knauth, L.A. and Lai, K.K., Intel Corp, 2013. Processor with last branch record register storing transaction indicator. U.S. Patent 8,479,053.

[25] Adams, K., Evans, J., Maher, B., Ottoni, G., Paroski, A., Simmers, B., Smith, E. and Yamauchi, O., 2014, October. The hiphop virtual machine. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages, and Applications (pp. 777-790).

[26] https://engineering.fb.com/2019/07/12/android/hermes/

[27] Wicht, B., Vitillo, R.A., Chen, D. and Levinthal, D., 2014. Hardware counted profile-guided optimization. arXiv preprint arXiv:1411.6361.

[28] Novillo, D., 2014, November. SamplePGO-the power of profile guided optimizations without the usability burden. In 2014 LLVM Compiler Infrastructure in HPC (pp. 22-28). IEEE.

[29] Friberg, S., 2004. Dynamic profile guided optimization in a VEE on IA-64 (Doctoral dissertation, Master's thesis, KTH–Royal Institute of Technology, 2004. IMIT/LECS-2004-69).

[30] Ramasamy, V., Yuan, P., Chen, D. and Hundt, R., 2008. Feedback-directed optimizations in gcc with estimated edge profiles from hardware event sampling.

[31] Dean, J., Hicks, J.E., Waldspurger, C.A., Weihl, W.E. and Chrysos, G., 1997, December. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In Proceedings of 30th Annual International Symposium on Microarchitecture (pp. 292-302). IEEE.

[32] Merten, M.C., Trick, A.R., George, C.N., Gyllenhaal, J.C. and Hwu, W.M.W., 1999, May. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In Proceedings of the 26th annual international symposium on Computer architecture (pp. 136-147).