9 Month Assessment Report

Iti Shree

June 13, 2023

Abstract

The aim of my PhD is to develop an improved framework for Profile Guided Optimisation (PGO) by better understanding what kind, and how many profiles are needed to achieve more optimal performance than existing PGO tools such as BOLT. In this preliminary report, I have presented an analysis of Facebook's BOLT, a post-link binary optimiser which is now part of the LLVM project. I have conducted an experimental investigation to discern how the use of varied types of profiling information influences the performance of the optimised binary. The performance metrics gathered in this study are derived from a relatively small test set due to resource constraints. Expanding the scope of the test set to obtain more comprehensive performance data forms part of the future work of this research.

1 Introduction

While Profile Guided Optimisation (PGO) has been an active area of research for some time, it remains a field with space for further exploration. This report delves into the effects of varying profiling methods on the performance of newly created potentially optimised binaries. Additionally, it investigates the potential for using a hybrid of these profiling methods to develop an enhanced PGO framework.

I start with doing background research on BOLT, the best known existing PGO tool. Despite its extensive use by large corporations for complex applications, even relatively simple experiments uncover surprising interactions between profiling and performance.

My PhD ultimately aims to determine which profiling methods would prove most effective and feasible in terms of enhancing the performance of PGO tools further, starting with BOLT. As a starting point, I conducted experiments using existing profiling techniques on the Clang[6] compiler binary, comparing against a baseline Clang binary. Future work will extend the scope of these experiments to include a broader, more diverse range of files to collect and benchmark the profiling data.

Methodology

BOLT[4] is a post-link static binary optimiser that enhances the code layout by utilizing sample-based profiling data at a binary level. This study investigates how much performance difference can be achieved by providing different profiling sampling data into BOLT to optimise the binary. This report discusses optimising manually-compiled Clang binary (CLANG 16) using BOLT by feeding it three types of sample profiling data. All the experiments are conducted using **Intel(R) Xeon(R) CPU E3-1270 v5 @ 3.60GHz**. I used three types of sample profiles (referred to as "modes" throughout the report): No Last Branch Record

(NO LBR) which is the default `perf record` collected sample, Last Branch Record (LBR), and Intel-PT (intel-pt or PT); these are explained in background section. I have presented a simple table later in report which indicates that compared to baseline binary, Clang built with NO LBR profile (NO LBR mode Clang) performs 18.45% better, LBR mode Clang performs 19.82% better and PT mode Clang performs 22.1% better on single run of a test file. Later we evaluate the performance of a different version of Clang running over 100 times, taking mean values along with 99% confidence interval. The baseline binary has been trained on four C programs (training set), and baseline along with optimised binaries were benchmarked on the same set of files (testing set).

2   Background

Modern compilers have integrated a lot of compiler optimisation techniques, such as using basic block frequencies and branch target information to perform function inlining. But they don't have runtime information and can't differentiate hot code path (executed a lot) from cold code paths (executed rarely) . This often leads compilers to optimise cold code paths at the expense of hot ones. Sampling data can help us collect hot code path (at least most of the times) since they show up more often.

Profile-guided Optimisation (PGO) improves application performance by shrinking code size, reducing branch mispredictions, and reorganizing code layout to reduce instruction-cache problems. PGO provides information to the compiler about areas of an application that are most frequently executed

PGO[1], helps in optimising a program's runtime performance by reducing branch mispredictions, reducing instruction-cache problems by reordering code layout with help of function splitting, and shrinking code size.Traditional methods such as instrumentation-based profilers are tedious and have significant overhead of memory and performance for collecting profiling information. Additionally, they might not end up capturing accurate information. PGO is easily adoptable in production environment such as by continuous feeding of dynamic profiling data to rebuild the binary. This report focuses experiments using Facebook's BOLT, a post-link FDO tool which uses sample based profiling information. BOLT accepts specific format to rebuild the binary. It doesn't matter how we capture the sample profile as long as we can convert it into BOLT friendly format before rebuilding the binary. I have used linux's tool-chain perf. Not just BOLT provides tool to convert perf data into BOLT friendly data but perf also let us capture different type of profiling information very easily.

2.1   Profiling Techniques

Perf

Perf[6], also commonly known as perf_event, is a lightweight linux tool-chain that offers an abstraction over hardware-specific capabilities. It is primarily used for performance measurements by instrumenting events, these events are unified interference from different parts of the kernel. Perf uses performance counters to capture events per-thread, per-process and per-cpu or system-wide and collect sample on top of these events.

Measurable events includes:

- PMU Hardware events: Performance Monitoring Unit or PMU contains measurable events such as

---

[1]Sometimes known as Feedback-Driven Optimisation (FDO): both terms refer to the same concept.

number of cycles, instruction retired, L1 cache and so on. They may vary a bit depending on processor and model type.

- Software events: low level events such as page-faults, cpu-clock and so, based on kernel counters.

- User Statically Defined Tracing: static trace points for user level program.

- Kernel Tracepoint Events: hardcoded kernel-level instrumentation point.

- Dynamic Tracing: for dynamically instrumenting software by creating event(s) in any location.

- Timed Profiling: commonly used for capturing snapshot by using custom timer interrupt events and invoked using `perf record -FHz`

The perf_event tool is part of linux-tools-common package and can either be installed via `apt install linux-tools-generic` or be build from source. For this report, I have used in-built perf on my OS.

perf can instrument in following modes: a) Counting, using `perf stat` which keeps count of running process for supported events. Unless specified `perf stat` runs in per-thread mode. Once all occurrence of events are aggregated the summary is presented on the standard output b) Sampling mode, using `perf record/report/annotate`. With `perf record` event data is written to the kernel and then read at an asynchronous rate to write to the perf.data file. This file is then analysed by the perf `report` or `script` commands.

Another way to instrument is by using BPF program on events which can then execute custom user-defined programs in kernel space, and perform filters and summaries on the data. But I won't be covering that in my experiments.

How is a sample event recorded?

An event sample is recorded when sampling counter overflows, which means counters go from 2^{64} to 0. Since 64-bit hardware counters aren't implemented in any PMU, perf emulated it in software. Hence, when using perf on 32-bit system perf silently truncates the period. On every counter overflow, the sample contains record of program execution with some additional information. The content of recorded data depends on what user specified for, but contains instruction pointer by default, irrespective of event type.

Sample period and rate

By default perf records sample at **both** user and kernel levels using cycles events with average rate of 1000 samples / sec or 4000 events / sec. During Sampling `perf record` asks PMU to count some events, for example cpu-cycles, and generate an overflow interrupt after every *nth* event (e.g a million). On every interrupt perf_event records fields such as : PID/TID (Process/Thread ID), timestamp, command being executed, and instruction pointer to the ring buffer and reset the counter to a new value.

The frequency (the average rate of collecting samples / sec) and period (the number of occurrences of the event) i.e collecting sample every *nth* occurrence of an event by passing, can be adjusted but has a maximum limit. Frequency can be adjusted by passing `-F` and period can be adjusted by passing `-c` to `perf record`. Once the ring buffer is filled, perf will dump the data to `perf.data` file, the size of file will depends upon frequency and period set.

LBR

By default perf only records basic performance counter and cannot follow indirect function call, i.e it can only know where the target is at the runtime, this leads to incomplete control flow. Intel has additional support in their CPUs called Last Branch Record (LBR)[1], a hardware register to record additional information in branch stack (or brstack) about branch instruction such as "from" and "to" address of each branch with some additional metadata (timing, branch mispredictions and so). The retired branches are captured in the rotating ring buffer. The stack depth of the ring buffer (usually 8, 16, or 32 frames) varies according to the Intel CPU model. Using LBR can also help in building better control flow graph with information about hot code paths, the profile generated can be used for finding basic block frequency or do profile-guided optimisation.

PT

PT is a hardware based tracing technique integrated into Intel's CPU hardware[2]. It traces branch execution which theoretically can be used to build exact control flow since it can know all executed code path.

This makes PT to capture huge amount of data which is not the case with standard processing such as PMU. Thus it uses additional auxiliary buffer(AUX) which is associated with perf's ring buffer.

We can use PT with perf's record command as:

```
perf record -e intel-pt//u ls
```

Buffer Handling

PT produces hundreds of megabytes of data per second per CPU and sometimes it often encounters two types of errors: **trace data-loss** (generating data at faster rate than it can be recorded to file) and **overflow packet** (unable to record data to memory).

Passing larger auxtrace **mmap** size seems to help in resolving the trace data loss, which can be done by passing the `-m` flag. So, passing `-m, 64M` means we can set trace buffer size to 64MiB per CPU. We would need to be careful with choosing auxtrace mmap size. For instance, if we have 8 CPUs we are essentially setting total 512MiB of trace buffer size.

To resolve overflow packets error in addition of setting AUX mmap size, re-running the workload helps, or reducing MTC (Mini Time Counter) packets in PT (which I have elaborated on later in this report). This will capture lesser data(a couple of MiB) but is helpful to remove "Instruction trace errors" in captured data.

It is important that we get rid of any "Instruction trace errors" as BOLT fails to convert data into in a acceptable format if it encounters trace errors in `.data` file.

BOLT

BOLT is a static post link binary optimiser that is build on top of LLVM compiler infrastructure.

It uses sample based profiling data and relocations mode (by passing `-emit-relocs` flag) to recompile the binary. BOLT reorder function blocks (or basic blocks) to split hot/cold code blocks with the goal of improving code locality. For function discovery, BOLT relies on ELF symbol table and function frame information (to get function boundaries)[5].

To recompile the binary, BOLT requires a specific format data. The fields that are required to get BOLT friendly data from profiling information are: thread ID (TID) or process ID (PID), instruction pointer (IP), and either event or branch stack information (brstack). So, it is possible to collect profile from any method as long as it can be fed into BOLT's profile conversion tool. However, for easier workflow using perf is recommended since a) it's easy to record different mode of samples b) BOLT has a utility tool to convert perf data directly into BOLT friendly data.

3   Optimising Clang with BOLT

I decided to replicate the tutorial to optimise Clang binary provided as an example in the BOLT's github[2] repository and followed it with some additional experiments and benchmarking.

Experiment Setup

For benchmarking, I used Intel(R) Xeon(R) CPU E3-1270 v5 @ 3.60GHz, on Debian OS and baseline CLANG 16 (without any optimisation). The files were executed hundred times using multitime [7], which is extension of linux times to run file multiple times and present mean, standard deviation, mins, medians, and maxs values on standard output. I used the sample mean and standard deviation values. Later I converted sample standard deviation value into 99% confidence interval to plot the graphs in this report. All the scripts to replicate the experiments can be found on my github repository.[3]

NO LBR vs LBR vs PT

In this section, I present my experiment where I have collected profiles in three modes namely NO LBR, LBR and PT and used the sample profiles captured to recompile Clang binary.

I conducted small experiment to compare performance of baseline Clang with Clang build using NO LBR mode profile and LBR mode profiling data. I used two distinct sets: a training set to generate the profiles for constructing the new version of Clang, and a testing set[4] to procure the sample mean wall-clock time over hundred runs and the 99% confidence interval[5].

Testing file	Baseline	NO LBR	LBR
format.c	0.907 ± 0.0008	0.735 ± 0.0005	0.723 ± 0.0005
tty.c	0.557 ± 0.0008	0.449 ± 0.0005	0.442 ± 0.0005
window-copy.c	0.880 ± 0.0008	0.715 ± 0.0008	0.704 ± 0.0008
extsmaild.c	0.251 ± 0.0005	0.209 ± 0.0005	0.205 ± 0.0005

Table 1: The first column presents the testing set files, while the succeeding columns represent performance of Clang variants, each build with profiles collected during extsmaild.c compilation

The above experiment demonstrates that a Clang build with an LBR mode profile is approximately 1.66% faster than a Clang build using a NO LBR mode profile. I intended for the PT data to run the same optimisation as BOLT when it is fed LBR mode profiling data. However, BOLT does not support the processing of PT data and only handles NO LBR and LBR data formats. By making some adjustments to BOLT's data aggregator source code, we can enable it to use of PT-collected profile information.

---

[2]https://github.com/llvm/llvm-project/blob/main/bolt/docs/OptimizingClang.md
[3]https://github.com/nmdis1999/experiment_setter
[4]Both the training and testing are same: extsmaild.c, tty.c, window-copy.c and format.c
[5]Files used are from following repository: https://github.com/ltratt/extsmail https://github.com/tmux/tmux

BOLT only requires three fields for optimisation in LBR mode: process ID (pid)/thread ID (tid), instruction pointer (ip), and branch stack (brstack). Therefore, extracting these should be sufficient. The main challenge was obtaining branch flags, which contains mispredictions. This flag is not available in PT mode, so I was unable to extract the brstack field from perf data directly. Fortunately, the perf team added a feature that allows modification of the configuration option[6]. This configuration change helps setting the mispred flag on all branches by altering the ∼/.perfconfig file.

Adding following lines to ∼/.*perfconfig* extracts brstack information out of sampled data collected with PT

```
1  [intel-pt]
2  mispred-all = on
```

After this I adjusted data aggregator source code to use `--itrace=i100usle` option in conjunction with perf scripts to filter brstack field data in PT mode. To check performance of Clang build in three mode I captured profile in all three mode from extsmaild.c and ran benchmark using multitime on the same file.

Below Table 2 shows the comparison of baseline, NO LBR mode, LBR mode and PT mode Clang.

Testing file	Baseline	NO LBR	LBR	Intel PT
format.c	0.907 ± 0.0008	0.735 ± 0.0005	0.723 ± 0.0005	0.701 ± 0.0008
tty.c	0.557 ± 0.0008	0.449 ± 0.0005	0.442 ± 0.0005	0.428 ± 0.0005
window-copy.c	0.880 ± 0.0008	0.715 ± 0.0008	0.704 ± 0.0008	0.680 ± 0.0008
extsmaild.c	0.251 ± 0.0005	0.209 ± 0.0005	0.205 ± 0.0005	0.201 ± 0.0005

Table 2: Compares Clang build with three modes to Baseline Clang. PT mode Clang on average performs 22.1% better than Baseline 4.5% better than the NO LBR mode and 2.9% better than the LBR mode

To probe into why PT mode Clang outperforms the others, I examined the number of functions that BOLT aims to reorder and the actual percentage of functions it ends up reordering in all three modes.

Analysis of the Number of Functions Recorded During Compilation, Number of Functions Captured in profile that BOLT identified, and Number of Functions Reordered by BOLT

I used the baseline Clang to gather profiles through the compilation of the extsmaild.c file and subsequently rebuilt the binary in three modes.Table 3 shows us number of functions BOLT intends to reorder and percentage of Clang's total number of functions it ends up reordering.

Mode	Reordering Aim	% of captured profile function reordered
NO LBR	606	81.68%
LBR	2638	80.14%
PT	5202	78.99%

Table 3: Number of functions identified and reordered by BOLT while processing profile captured from extsmaild.c compilation.

Out of 135363 functions in Clang binary, BOLT reorders 495 in NO LBR mode, 2114 functions in LBR mode and 4109 functions in PT mode i.e 0.37%, 1.56%, and 3.04% of total functions respectively.

---

[6]https://www.uwsg.indiana.edu/hypermail/linux/kernel/1509.3/03186.html

Clearly, LBR mode profile captures 4.2x more functions than NO LBR mode while PT mode profile captures almost 2x more function than LBR mode.

Does increase in number of functions captured in the profile results in faster Clang variant? Before answering the question above, I'll present a small hypothesis:

Hypothesis I: Increase in lines of code (LOC) in testing file is directly proportional to increased number of functions recorded by perf and thus captured BOLT

Profiling mode	extsmaild.c (1781)	tty.c (3026)	format.c (5188)	window-copy (5662)
NO LBR	606	1141	1550	1635
LBR	2638	3962	4981	4965
PT	5202	6678	7911	8045

Table 4: Profiled functions captured by BOLT using baseline Clang compilation of the following C programs. The column headers are C program I tested on and the number in bracket next to file name represents line of code(LOC) values.

Looking at values in Table 4, the number seems to hold for small set of test files. This might however, not be true when testing on bigger testing set.

Hypothesis II: The larger inputs helps capturing more functions using perf. This, when processed by BOLT, could potentially lead to the faster Clang builds.

Benchmarked on / Clang versions	Baseline Clang	Clang.extsmaild.c (1781)	Clang.tty.c (3026)	Clang.format.c (5188)	Clang.window-copy.c (5662)
format.c	0.907 ± 0.0008	0.737 ± 0.0061	0.739 ± 0.0074	0.739 ± 0.0089	0.735 ± 0.0005
tty.c	0.557 ± 0.0008	0.449 ± 0.0005	0.451 ± 0.0005	0.451 ± 0.0005	0.450 ± 0.0005
window-copy.c	0.880 ± 0.0008	0.715 ± 0.0008	0.718 ± 0.0005	0.717 ± 0.0008	0.717 ± 0.0005
extsmaild.c	0.251 ± 0.0005	0.209 ± 0.0005	0.209 ± 0.0005	0.209 ± 0.0005	0.209 ± 0.0005

Table 5: Clang build in NO LBR mode - The column header represents Clang build with BOLT using profile collected during compilation of training files. So, they are named as *clang.training_file (LOC)* and ordered in ascending order of LOC. Row names are files used for benchmarking the four optimised Clang binaries.

Table 5 represents Clang build using NO LBR mode profiles, it's easily noticeable that the range (Mean ± CI) for each run using Clang binaries on individual testing files exhibits significant overlap. The delta, in this case, isn't substantial enough to infer a relationship between LOC and performance.

Benchmarked on / Clang versions	Baseline Clang	Clang.extsmaild.c (1781)	Clang.tty.c (3026)	Clang.format.c (5188)	Clang.window-copy.c (5662)
format.c	0.907 ± 0.0008	0.736 ± 0.0082	0.731 ± 0.0070	0.725 ± 0.0093	0.723 ± 0.0005
tty.c	0.557 ± 0.0008	0.450 ± 0.0005	0.444 ± 0.0005	0.442 ± 0.0005	0.442 ± 0.0005
window-copy.c	0.880 ± 0.0008	0.716 ± 0.0008	0.710 ± 0.0008	0.705 ± 0.0008	0.704 ± 0.0008
extsmail.c	0.251 ± 0.0005	0.209 ± 0.0005	0.207 ± 0.0005	0.205 ± 0.0005	0.206 ± 0.0005

Table 6: Clang build in LBR mode - Benchmarked on four C source training files.

However, in the LBR mode (Table 6), using training file with higher LOC showed some improvement: exhibiting either best wall-clock time or closer value to best performing Clang. There is trend of diminishing returns over increasing LOC once the difference of in LOC is not too much. The two largest files in terms of LOC are format.c and window-copy.c with 5188 and 5662 LOC respectively. One example is, during benchmarking, Clang build using profile from format.c compilation sometimes beats Clang build using profile from window-copy.c compilation.

Benchmarked on / Clang versions	Baseline Clang	Clang.extsmaild.c (1781)	Clang.tty.c (3026)	Clang.format.c (5188)	Clang.window-copy.c (5662)
format.c	0.907 ± 0.0008	0.723 ± 0.006	0.709 ± 0.0009	0.701 ± 0.009	0.701 ± 0.0008
tty.c	0.557 ± 0.0008	0.441 ± 0.0005	0.430 ± 0.0005	0.429 ± 0.0005	0.428 ± 0.0005
window-copy.c	0.880 ± 0.0008	0.705 ± 0.0008	0.689 ± 0.0008	0.684 ± 0.0008	0.680 ± 0.0008
extsmail.c	0.251 ± 0.0005	0.205 ± 0.0005	0.203 ± 0.0005	0.201 ± 0.0005	0.202 ± 0.0005

Table 7: Clang build in PT mode - Benchmarked on four C source training files.

In the PT mode (Table 7), a similar trend like LBR is noticed - an increase in LOC tends to enhance performance, albeit with notably diminishing returns when two files with lesser delta in LOC are compared.

Could the observed relative increase in Clang's performance with respect to the line of code (LOC) in both LBR and PT modes be attributed to their capacity to record more functions during sampling? This brings me to the next hypothesis:

Hypothesis III: perf in PT mode records more functions in the samples than LBR mode to build better control flow.

`perf record` in LBR and PT mode has some additional information which is not available in NO LBR mode. More details can be fetched about branches such as FROM and TO addresses of each branch with some additional meta-data such as timing packets. LBR during each sample only logs last 8-32 branches while PT theoretically records every branch. There is a limitation that both LBR and PT feature of perf are currently only supported on Intel devices. Though there seem to be ongoing development to add support for LBR on AMD devices[3].

One interesting outcome to see would be aggregating profiles recorded in the LBR mode to get optimised Clang and compare Clang in PT mode using profile from single file compilation . My next two hypothesis capture this:

Hypothesis IV : Aggregating multiple profiles over n runs on the same training file results in building faster Clang binaries.

To test this hypothesis, I aggregated profiles (collected with perf) over *n* runs on the same training file and analyse the resulting performance variations. Since testing file consists of four files: extsmaild.c, tty.c, format.c, and window-copy.c. There are four bar graphs in Figure 1, each representing Clang versions benchmarked on the respective testing file.

Looking at Figure 1, it's quite evident that the performance of Clang versions, created with 5, 10, and 15 aggregated LBR profiles, matches or even surpasses the performance of the Clang generated with an PT mode profile The same trends seems to follow in other three graphs. Albeit on small test set, this seems to be proving my hypothesis correct.
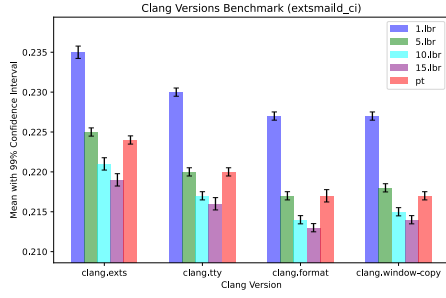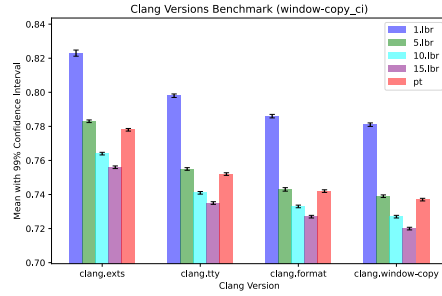
(a) Benchmark of Clang versions on extsmaild.c
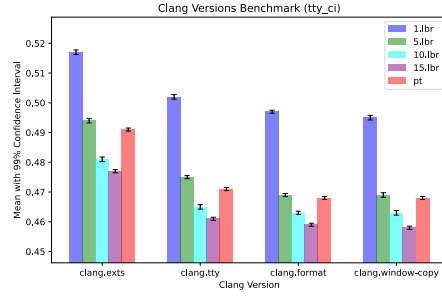
(b) Benchmark of Clang versions on tty.c
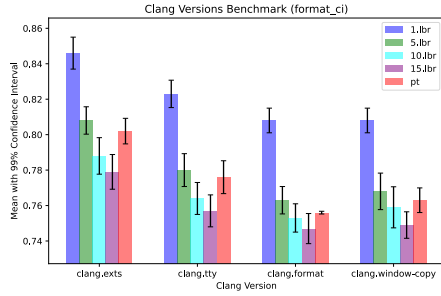
(c) Benchmark of Clang versions on format.c

(d) Benchmark of Clang versions on window-copy.c

Figure 1: On the graph, the x-axis represents the Clang versions build with different training files, and the y-axis depicts the mean along with the 99% confidence interval. The color, situated at the top right, indicates `number.mode` - 'number' signifies the counts of profiles that have been aggregated and 'mode' pertains to the profiling mode used to build the optimised Clang variant. And the black lines on top of colored bar are range(±) of confidence interval. In most cases, clang.*filename*.15.lbr surpasses clang.*filename*.pt

Hypothesis V: Aggregating profiles from different training files results in building faster Clang binary.

Next, instead of aggregating profiles from run on the same file, I captured profiles from runs on different files to aggregate and feed them into BOLT.

Clang versions/ benchmarked on	extsmaild.c	tty.c	format.c	window-copy.c
Clang.all-prog-aggr-lbr	0.220 ± 0.0005	0.473 ± 0.0005	0.773 ± 0.0007	0.754 ± 0.0007
Clang.exts.pt	0.223 ± 0.0005	0.488 ± 0.0007	0.794 ± 0.0005	0.776 ± 0.0007
Clang.tty.pt	0.220 ± 0.0005	0.470 ± 0.0005	0.771 ± 0.0005	0.752 ± 0.0007
Clang.format.pt	0.216 ± 0.0005	0.466 ± 0.0005	0.756 ± 0.0007	0.741 ± 0.0007
Clang.window-copy.pt	0.217 ± 0.0005	0.467 ± 0.0005	0.736 ± 0.0007	0.759 ± 0.0007

Table 8: When comparing a Clang build in LBR mode, which uses aggregated profiles from different file compilations, with the Clang build in PT mode that uses a profile from a single file compilation, there doesn't seem to be much advantage.

Table 8 represents Clang trained with two modes LBR and PT, where in LBR mode I aggregated profiles collected from 4 C Source files to build optimised Clang and for PT mode I used these files individually to collect and build new Clang. Later I have benchmarked sample mean and 99% confidence interval on the same files I collected profiles from.

Figure 2 clearly indicates that aggregating four profiles captured from different file compilations is not enough to beat all the Clang builds with PT mode profiles.

I will present another hypothesis to analyse why in Hypothesis III, Clang build with aggregating LBR mode
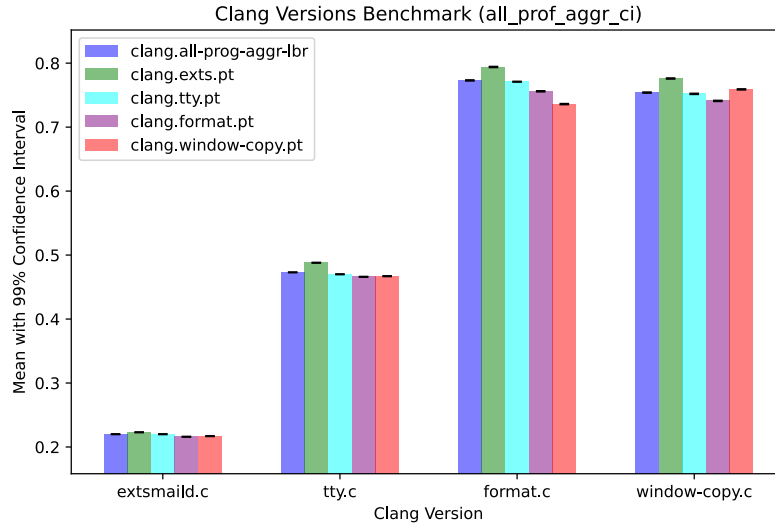
Figure 2: Comparison of Clang build with aggregated LBR profiles to Clangs' build with PT mode profiles. Where *clang.all-prof-aggr-lbr* is build using profile from compiling 4 separate C source files.

profiles collected over 5 or more runs on same file, first converges to the performance of PT mode Clang, and later even beats it.

Hypothesis VI: LBR mode collected profiles has finer-grained data than PT mode collected profiles

Linux Perf's wiki states that "perf with LBR can provide finer timing than PT." [6] This might explain why in Figure 1 Clang build with LBR aggregated data outperforms Clang build using PT profile. However, it is not clear whether the statement holds true when we tune cycle-accurate mode on in PT mode profiling. I intend to verify this as future work, by collecting profiling data for specific set of address, both in LBR and PT (cycle-accurate) mode.

In PT, logs are packetised. PT traces uses three types of timing packets: Packet stream Boundary (PSB) with Time Stamp Counter (TSC) updates, Mini Time Counter (MTC), and Cycle (CYC). PSB although being coarse grained, is the basic synchronization of the trace and provides a full TSC time stamp. The MTC packet has finer and regular timing updates from the 24/25Mhz ART (Always Running Timer). Lastly, Cycle accurate mode (CYC) packets are updated relative to the last send MTC packet.

By default, perf when operating in PT mode uses MTC packets with period of 3. This means there is an ART timing update ever $2^3 = 8$ times, so roughly every 300ns. This is quite coarse, so I used CYC packet in conjunction with MTC. However, the data generated by perf command increases in size with decrease in MTC period's value and thus gets difficult to processes. For this experiment, I didn't alter MTC period's value.

In case modification is needed in MTC period, the following command can be used:

```
perf record -e intel_pt/cyc,cyc_thres=value,mtc_period=value/u -
```
where:

- cyc: cycle accurate mode timing packets

- cyc_thres: minimum number of cycles which must pass before the next CYC packet is sent.

- mtc_period: flag to set value for MTC period

Clang versions	Mean ± CI
Clang.exts.15.lbr	6.879 ± 0.0167
Clang.exts.pt	7.158 ± 0.0167
Clang.exts.1.pt	7.140 ± 0.0178
Clang.exts.5.pt	6.777 ± 0.0167
Clang.exts.10.pt	6.699 ± 0.0201
Clang.exts.15.pt	6.633 ± 0.0183

Table 9: Clang build with LBR, PT and PT with cycle accurate mode benchmarked on bootstrapped single source gcc.c file.

Taking best performing Clang from Hypothesis IV (Figure 1a) i.e Clang.exts.15.lbr, where Clang binary was optimised by aggregating profile collected from 15 consecutive runs of extsmaild.c compilation, I compare it with Clang build using single PT mode profile (no cycle accurate mode), single PT mode profile (with cycle accurate mode), 5, 10, and 15 aggregated profiles from PT mode profiling. All optimised Clang version were run/benchmarked on gcc.c which was bootstrapped from gcc source code into single C file.

Figure 3 shows that when aggregating profiles in PT (cycle accurate mode) from n consecutive runs starting from Clang.exts.5.pt, the performance of PT cycle accurate mode Clang when aggregated is better than Clang build with Clang build with LBR aggregated profiles.
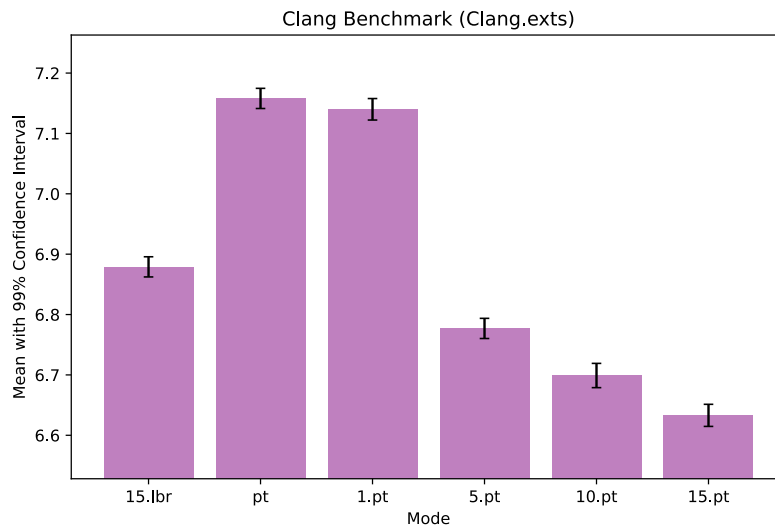

Figure 3: Here we compare a Clang version built with aggregated LBR profiles to a Clang build using PT mode profiles.'n.pt' refers to a Clang build conducted in cycle-accurate mode.

Future Work

Results of Hypothesis VI indicate that when profiles are aggregated both in LBR and PT mode they tend to give better wall-clock timings. However, there is a trade-off for using one profiling mode. While LBR gives

finer-timing by default it has very small number of branches logged. And although in cycle-accurate mode PT mode, Clang beats the best LBR mode Clang (Figure 1) the data captured is 3 times larger than that in LBR mode.

Finding a hybrid approach, which combines two or more profiling method such that we can use both LBR's capability to record finer timing for branch instructions and PT's capability to capture detailed program execution in profiles can create a more accurate and comprehensive profiling technique. Going forward, I plan to investigate how to improve PGO tools with such hybrid profiling technique.

The next steps are:

1. Expand the experiments from this report to a larger training and testing set.

2. Determine the cost of the hybrid. profiling technique using LBR and PT mode profiles with BOLT

3. Test the hybrid profiling technique with other PGO tool(s).

Timeline

Year	Month	Goals
2023	July-December	Expanding the scale of experiment, analysing the data, and modifying BOLT to adopt to hybrid profiling information
2024	January-June	Literature review on PGO tools. Forming the research question. Benchmarking the results to see whether I can work on top of BOLT to add additional capabilities or should I build a new tool/framework.
2024	July-December	Complete first version of experimental PGO framework.
2025	January-June	Benchmarking and debugging of the new framework. Drafting the initial findings.
2025	July-December	Identifying areas which can be expanded on the new framework, receive feedback and do revisions.

References

[1] The Linux Kernel Developers. Intel Last Branch Record, Linux kernel documentation. `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf`, 2016. Accessed: 2023-06-13.

[2] The Linux Kernel Developers. Intel Processor-Trace, Linux kernel documentation. `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf`, 2016. Accessed: 2023-06-13.

[3] Perf Mailing List. Amd supports lbrextv2. `https://lwn.net/Articles/904482/`, 2022. Accessed: 2023-06-13.

[4] Bill Nell and Guilherme Ottoni Maksim Panchenko, Rafael Auler. Bolt: A practical binary optimizer for data centers and beyond. 2019.

[5] Laith Sakka and Guilherme Ottoni Maksim Panchenko, Rafael Auler. Lightning bolt: Powerful, fast, and scalable binary optimization. 2021.

[6] Linux Perf. Perf wiki. `https://perf.wiki.kernel.org/index.php/Main_Page`, 2023. Accessed: 2023-06-13.

[7] Laurence Tratt. Multitime. `https://tratt.net/laurie/src/multitime/`, 2012. Accessed: 2023-06-13.