Properties of Profile-Guided Compiler Optimizations with GCC and LLVM

By

©2022
Soma Pal

Submitted to the Department of Electrical Engineering and Computer Science and the
Faculty of the Graduate School of the University of Kansas
in partial fulfillment of the requirements for the degree of

Master of Science

Dr. Prasad Kulkarni, Chairperson

Committee members

Dr. Tamzidul Hoque

Dr. Mohammad Alian

Date defended: December 7, 2022

The Dissertation Committee for ©2022
Soma Pal certifies
that this is the approved version of the following dissertation:

Properties of Profile-Guided Compiler Optimizations with GCC and LLVM

Dr. Prasad Kulkarni, Chairperson

Date approved: December 7, 2022

Abstract

Profile-guided optimizations (PGO) are a class of sophisticated compiler transformations that employ information regarding the profile or execution-time behavior of a program to improve program performance, typically speed. PGOs for popular language platforms, like C, C++ and Java, are generally regarded as a mature and mainstream technology and are supported by most standard compilers. Consequently, properties and characteristics of PGOs are assumed to be established and known, but have rarely been systematically studied with multiple mainstream compilers.

The goal of this work is to explore and report some important properties of PGOs in mainstream compilers, specifically GCC and LLVM in this work. We study the performance delivered by PGOs at the program and function-level, impact of different execution profiles on PGO performance, and compare relative PGO benefit delivered by different mainstream compilers. We also describe the experimental framework that we built to conduct this research. We expect that our research will help focus future developmental work and research in PGOs and assist in building frameworks to field PGOs in actual systems.

Acknowledgements

I would like to express my gratitude to my advisor Professor Prasad Kulkarni for his guidance, support and patience throughout the process. His expert comments and directions were immensely valuable for this thesis. I strongly believe that the knowledge I have garnered from him will guide me in my future endeavors beyond KU.

I would also like to thank my thesis committee member Professor Mohammad Alian and Professor Tamzidul Hoque for their valuable comments and suggestions. Moreover, I would like to thank the Department of EECS at KU for giving me the opportunity to explore this research horizon.

Contents

List of Figures

List of Tables

Chapter 1

Overview, Research Motivation and Research objective

1.1 Introduction

Computing is constantly shaped by technology, applications, and market trends. Cloud computing, IoT, and mobile computing have been major drivers of innovation recently and are expected to thrive in the upcoming decade. A plethora of newly developed technologies, such as machine learning, and blockchain, present a new set of challenges that necessitate increased performance from modern processors.

To adequately support these modern application trends, the processor hardware has evolved to integrate multiple core processors, changes to on-chip interconnect, caches, a variety of hardware accelerators, and a memory controller on a single chip. Modern processors employ pipelined and superscalar architectures that incorporate out-of-order execution. Performance improvement on such systems depends on the program ability to fully utilize the characteristics of the underlying architecture.

On the other hand, modern high-level languages are constantly evolving, and are becoming more abstract and complex. These trends are helping to reduce both software development time and bugs in the system and at the same time, are also boosting sustainability [14].

Unfortunately, high levels of abstraction are known to increase program run times. With high-level languages, the burden of optimization falls on the compiler.

Modern compilers are complex software that translate programs written in a high-level language into binaries that execute on the underlying hardware. This conversion includes pre-processing, lexical analysis, parsing, semantic analysis, code optimization, and finally creation of executable files. Compilers are not only responsible for constructing small executables but also for achieving the best possible execution-time performance.

Compilers can provide bad optimization decisions that do not actually improve the performance of the code or, even worse, degrade it. For instance, one important optimization a compiler performs is function inlining. The C/C++ compilers, such as GCC [10] and Clang[21], can inline a function if the caller size does not increase too much. Many function calls are expanded. However, this optimization is only beneficial when the calls happen frequently. As a side-effect, this optimization also: (a) increases the code size, (b) wastes space from instruction and unified caches, and (c) increases the app's working set size in memory. These side effects can also degrade program performance in many cases [22].

Often, optimizations lack reliable heuristics needed to make wise selections. Even with the degradation from some optimizations, compiler typically achieve on overall performance improvement for most programs. Adopting compiler heuristics to eliminate even the occasional degradation from individual optimizations may result in a further benefit to program performance.

The code might be optimized more effectively if we could guide the compiler on how the code would behave at run-time. Profiling is the process of acquiring program information during execution. A profile is a name given to the generated information. The compiler can process multiple profiles for each program with different inputs. Its optimizations can be guided by these profiles. Compiler transformations that employ profile information to benefit program performance are called profile-guided optimizations (PGO) [18].

Profile-guided optimizations is a promising technique to improve the performance of various programs. One example of profile information obtained by the compiler may be about the most frequently executed sections of an application. Compiler optimization can then be more selective and precise when optimizing code for an application by being aware of the most frequently run sections.

Profiling data can be collected through offline and online processes. Offline profiling uses additional prior runs of the program to generate profile information. Once the profile data is generated, the compiler can use this information to guide code optimization decisions. Static compilers such as GNU gcc/g++ and Clang, mainly use offline profiling techniques. Online profiling or dynamic profiling gathers profile information during the same program run, and is frequently used by more advanced language runtimes like the Java virtual machine. Many researchers have also developed static analysis techniques to estimate some run-time information for PGOs. In our experiments, we are using offline profiling to get the profile data and study the performance of PGOs in different mainstream open-source compilers.

PGO consists of three steps or phases [6] as shown in Figure 1.1.

1. The first phase is to instrument the program. In this step, the compiler creates and links an instrumented program from the source code and a special code from the compiler.

2. The second phase is to run the instrumented executable. Each time the instrumented code executes, the instrumented program generates a dynamic information file that is used in the final compilation.

3. The third phase is the compilation. The dynamic information files are merged into a summary file if the software is built for a second time. The profile information con-

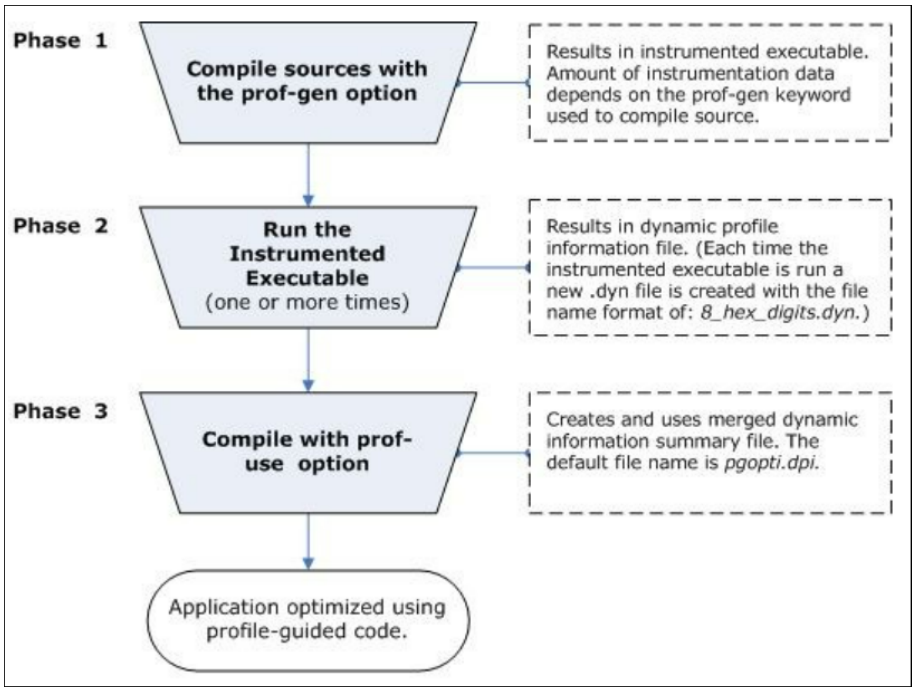

Phase 1
Compile sources with the prof-gen option
Results in instrumented executable. Amount of instrumentation data depends on the prof-gen keyword used to compile source.
Phase 2
Run the Instrumented Executable (one or more times)
Results in dynamic profile information file. (Each time the instrumented executable is run a new .dyn file is created with the file name format of: 8_hex_digits.dyn.)
Phase 3
Compile with prof-use option
Creates and uses merged dynamic information summary file. The default file name is pgopti.dpi.
Application optimized using profile-guided code.

Figure 1.1: Three phases of Profile Guided Optimizations (PGO)

tained in this file is used by the compiler to customize the executable to the provided program profile.

PGO has a number of benefits over traditional optimization techniques. For instance, with PGO, the compiler can optimize the program to specific processor architecture features, and maximize cache memory efficiency and branch prediction benefits. Likewise, the position of the spill code is optimized using profile data for register allocation. Selecting the most probable targets also offers superior branch prediction for indirect function calls.

PGO can also help the compiler decide on function inlining more wisely. PGO is the best for dealing with many commonly executed branches that are difficult to predict at compilation time. For instance, the codes with intensive error-checking ensure that the error conditions are false most of the time. The cold area is the part of code where infrequently executed code can be relocated. So, the branch is rarely predicted incorrectly. Minimizing cold code and incorporating it into the frequently executed (hot) code improves instruction cache behavior.

In this work, we thoroughly and methodically investigate the properties of PGOs for standard C/C++ compiler benchmark programs. We developed a detailed experimental framework and many techniques to answer the following questions:

1. What is the typical execution-time benefit from PGO with mainstream compilers?
2. How do different profiles affect performance of PGOs?
3. Do standard compilers deliver similar performance gains with PGOs?
4. What is the performance effect of PGOs at the finer-grained function level?

We also study auxiliary issues regarding program behavior and compiler comparison, including

(a) the fraction of execution time spent in the top 10 functions in the SPEC 2017 benchmarks,

(b) comparison of GCC and LLVM compile time, and performance characteristics such as executable size and run-time performance.

We believe that our experiments and observations shed further light on the functionality, traits, and efficacy of PGOs with mainstream compilers.

The rest of the paper is organized as follows. In Section 3, we provide a brief description of the background and some related works. In Section 4, we give a description of the experimental setup. In Section 5, we describe the standard compilers and tools used in the experiment. In Section 6, we describe our framework implementation with the GCC and LLVM compilers. In Section 7, we describe the results of our experiments. In Section 8, we conclude the paper.

Chapter 2

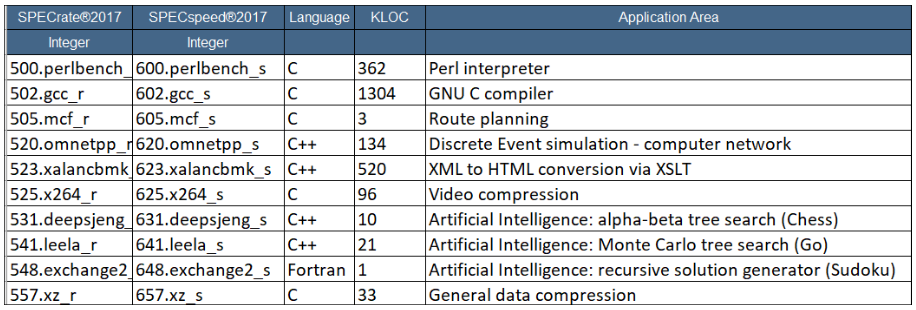
Background and Related Works

In this chapter, we describe some applications of profiling and a brief background on SPEC CPU2017. We also present prior work on the properties of profiling and PGOs, as well as a comparison of our current research goals with those of related previous studies.

In our current work, we study offline schemes to collect the program's execution time profiling data. This profile data is used by static compilers gcc/g++ and llvm. The selection of the compiler mainly depends on parameters like accessibility, support for hardware, and efficiency of compilers. Some of the most widely used compilers for C/C++ include the GNU Compiler Collection(GCC) and the LLVM Compiler Infrastructure(LLVM). As part of this study, we have assessed the efficacy of profiling and PGOs for these two compilers. Profile data has traditionally been employed to find the hot or frequently executed blocks or functions in a program. The hot program regions can be used to focus on compilation and optimization efforts. For illustration purposes, many online compilers only compile and apply PGOs to the hot region of a program to reduce overhead at run-time. This technique is called selective compilation [3]. Profile information is also used to perform some other optimization tasks. In this work, our goal is to analyze the effect of PGOs with the GCC and LLVM compilers for SPEC CPU2017 programs and also to study the performance in the *hotspot* regions of the programs.

Standardized Performance Evaluation Corporation (SPEC) is one of the most successful efforts in standardizing benchmark suites. SPEC CPU2017 is the most recent incarnation of a standard benchmark suite designed to stress a system's processor, memory subsystem, and compiler [1]. SPEC CPU2017 distributes as an ISO image that contains the following:
a) source code for each benchmark,
b) benchmark inputs,
c) a toolset for compiling, running, validating, and reporting on the benchmarks,
d) pre-compiled tools for a variety of operating systems,
e) source code for SPEC CPU2017 tools, for systems not covered by the pre-compiled tools, documentation, and configuration scripts that govern benchmark compilation process.
SPEC CPU2017 consists of 43 benchmarks that are divided into the following four categories :
(a) SPECspeed2017 floating-point (fp_speed),
(b) SPECspeed2017 integer (int_speed),
(c) SPECrate2017 floating-point(fp_rate),
(d) SPECrate2017 integer (int_rate).

The benchmark programs are written in C, C++, and Fortran programming languages that are derived from a wide variety of application domains. The fp_speed and fp_rate suites consist of benchmarks that work with floating-point data types, whereas the int_speed and int_rate suites contain benchmarks that perform computation on integer data types as shown in Figure 2.1.
Characterization studies performed on the previous SPEC CPU2000 and SPEC CPU2006 showed that these benchmarks match the evolution of real-life workloads [13]. We have



SPECrate®2017	SPECspeed®2017	Language	KLOC	Application Area
Integer	Integer			
500.perlbench_	600.perlbench_s	C	362	Perl interpreter
502.gcc_r	602.gcc_s	C	1304	GNU C compiler
505.mcf_r	605.mcf_s	C	3	Route planning
520.omnetpp_r	620.omnetpp_s	C++	134	Discrete Event simulation - computer network
523.xalancbmk	623.xalancbmk_s	C++	520	XML to HTML conversion via XSLT
525.x264_r	625.x264_s	C	96	Video compression
531.deepsjeng_	631.deepsjeng_s	C++	10	Artificial Intelligence: alpha-beta tree search (Chess)
541.leela_r	641.leela_s	C++	21	Artificial Intelligence: Monte Carlo tree search (Go)
548.exchange2_	648.exchange2_s	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)
557.xz_r	657.xz_s	C	33	General data compression

Figure 2.1: Rate and Speed Benchmarks in SPECCPU2017

used speed suits for our experiments. Speed suits are used to evaluate how fast the tested computer can execute a benchmark and evaluate the performance of the executions.

There are several works that study the effectiveness of PGOs on various compilers. Different studies have shown the effectiveness of compiler optimizations on multi-cores and found that -O3 level optimization provides the best performance across applications [2]. Thus, we have compared performance parameters with PGO for each benchmark over -O3 level optimization.

The usefulness and impacts of sampling-based profilers on adaptive tasks have been compared in several prior research projects. If a known profile is provided, it may be used to compare the correctness of any supplied profile data directly [4, 9]. Researchers have tested whether their profile is capable of appropriately guiding the dependent adaptive task using causality analysis when the right profile itself cannot be developed or is unknown [15, 17, 23]. Part of this study examines how profiles created from various plausible program inputs might describe the program execution for the current run rather than judging the profiler's accuracy.

In our experiments, we have evaluated the performance of SPEC CPU2017 benchmarks with two open-source compilers GCC and LLVM. We have considered four leading areas in the evaluation of compilers in PGO technique :

(a) Understanding the impact of profile data on the effectiveness of PGOs.

(b) The effect of cross-input profile information that is available using offline profiling techniques.

(c) Effectiveness of PGOs in self/standard inputs on standard compilers such as GCC/g++ and LLVM.

(d) Performance analysis of top ten hot functions for each benchmark.

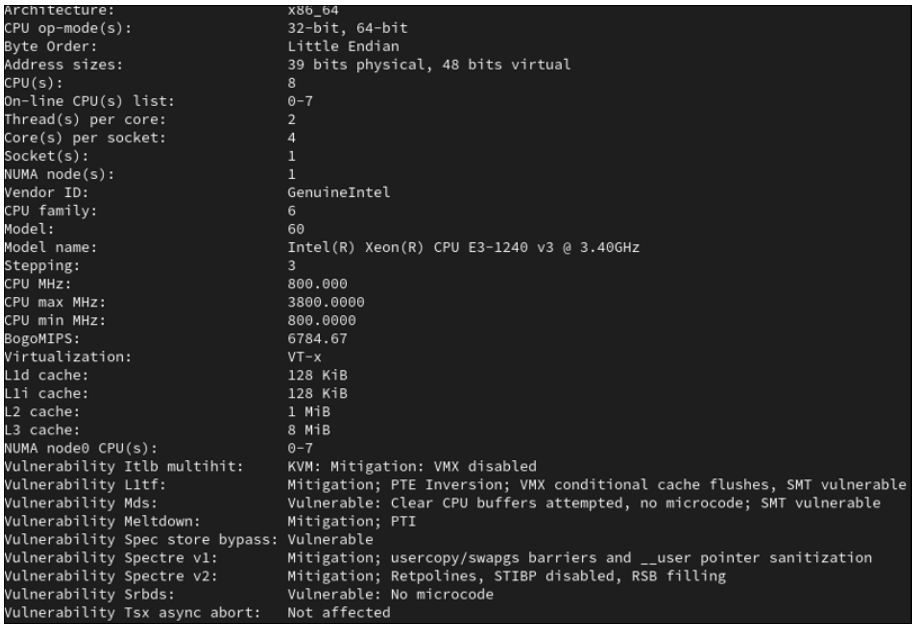
Chapter 3

Experimental Setup, Experiments and Data Collection

In this chapter, we provide a brief background on the properties of the compilers gcc/g++ and llvm/Clang. We also describe the installation of benchmarks that are relevant to this work and provide details of our experimental setup. Our initial experiments are performed on a desktop machine with intel(R) Xeon(R) CPU E3-1240 v3 @3.40GHz. It is based on the x86_64 architecture and is manufactured using Intel's 22nm technology node and architecture Haswell S. Its base clock speed is 3.40 GHz. However, a single core turbo boost frequency can reach up to 3.80. Properties of the processor used to gather evaluation results as obtained by Linux are presented in Figure 3.1.

3.1 Compilers

In this section we introduce the compilers used in this experiment. The choice of compilers is made with respect to the usage of SPEC benchmarks, availability, and known performance.

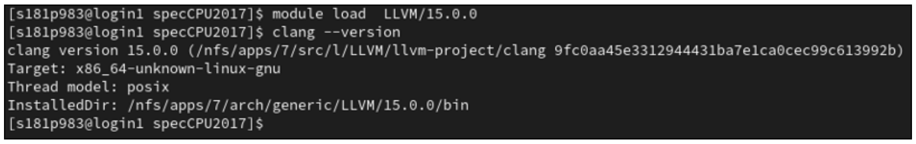


```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           60
Model name:                      Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40GHz
Stepping:                        3
CPU MHz:                         800.000
CPU max MHz:                     3800.0000
CPU min MHz:                     800.0000
BogoMIPS:                        6784.67
Virtualization:                  VT-x
L1d cache:                       128 KiB
L1i cache:                       128 KiB
L2 cache:                        1 MiB
L3 cache:                        8 MiB
NUMA node0 CPU(s):               0-7
Vulnerability Itlb multihit:     KVM: Mitigation: VMX disabled
Vulnerability L1tf:              Mitigation; PTE Inversion; VMX conditional cache flushes, SMT vulnerable
Vulnerability Mds:               Vulnerable: Clear CPU buffers attempted, no microcode; SMT vulnerable
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Spec store bypass: Vulnerable
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Retpolines, STIBP disabled, RSB filling
Vulnerability Srbds:             Vulnerable: No microcode
Vulnerability Tsx async abort:   Not affected
```

Figure 3.1: Specification of Intel processor using in experiment

3.1.1 Clang/LLVM Compilers

"The LLVM Compiler Infrastructure Project is a "collection of modular and reusable compiler and toolchain technologies" [11]. LLVM can be used to develop a front end for any programming language and back end for any instruction set architecture. LLVM is mainly designed around a language-independent Intermediate Representation(IR) that serves as a high-level, portable assembly language and can be optimized with a variety of transformations over multiple passes. LLVM is written in C++ and it is designed for compile-time, link-time, run-time, and idle-time optimization of target programs written in arbitrary programming languages. Clang is a compiler front end for the C and C++ programming languages and uses the LLVM compiler infrastructure as its backend. Clang inherits many features, such as link-time optimization, pluggable optimizer, and Just-in-Time compilation from LLVM. Several companies are using LLVM in their customized software development tools, including AMD, NVIDIA, Apple, and Sony [11, 5, 19]. In our research, we create binary executables for benchmarks using the LLVM compiler, and then we analyze the performance of those executables. For our tests, we are utilizing the most recent version 15.0.0 of LLVM and clang as shown in Figure 3.2.

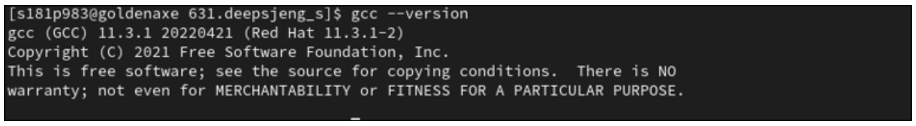


```
[s181p983@login1 specCPU2017]$ module load  LLVM/15.0.0
[s181p983@login1 specCPU2017]$ clang --version
clang version 15.0.0 (/nfs/apps/7/src/l/LLVM/llvm-project/clang 9fc0aa45e3312944431ba7e1ca0cec99c613992b)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /nfs/apps/7/arch/generic/LLVM/15.0.0/bin
[s181p983@login1 specCPU2017]$
```

Figure 3.2: LLVM/Clang version used in our experiments

3.1.2 GCC Compilers

The GNU compiler collection is an open-source and optimizing compiler produced by the GNU Project supporting various programming languages, hardware architecture, and operating systems. The GCC compiler was originally written to be a compiler for the GNU operating system [10]. It is widely used across various target architectures and is distributed along with the Linux operating systems. GNU compiler collection includes front ends for many programming languages and libraries from these languages. Unlike LLVM, GNU compilers are monolithic, where each language has a specific program to read the source code and produce machine code. High-level optimizations in GNU are limited. GCC is the primary component of the GNU toolchain and the standard compiler for most projects related to GNU and Linux kernel. GCC is the one of the biggest free programs [16, 20]. It plays an important role in the growth of free software as a tool. For our study, we are using the GCC version gcc (GCC) 11.3.1 10220421 (Red Hat 11.3.1-2) as shown in Figure 3.3.

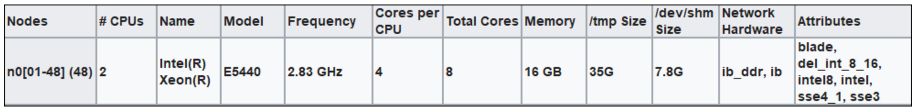


```
[s181p983@goldenaxe 631.deepsjeng_s]$ gcc --version
gcc (GCC) 11.3.1 20220421 (Red Hat 11.3.1-2)
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Figure 3.3: GCC version used in our experiments

3.2 Hardware Cluster

We utilized the ITTC research cluster at the Advanced Computing Facility (ACF) for our investigation. Only 2GB of RAM was allotted to our local workstation, which was inadequate to execute the benchmarks. Additionally, we ran nine benchmarks in parallel. Therefore, nine nodes were made accessible throughout a concurrent benchmark run. As a result, we made use of the ITTC research cluster's High-Performance Computing (HPC) facilities. The cluster uses the open-source, fault-tolerant, and highly scalable *Slurm* workload manager in addition to a job scheduling system for Linux clusters[12]. It also includes a range of hardware types with core counts ranging from 8 to 20, and offers a huge memory system with up to 512 GB of RAM. We utilized the hardware listed in the Figure 3.4 to run our experiments.



Nodes	# CPUs	Name	Model	Frequency	Cores per CPU	Total Cores	Memory	/tmp Size	/dev/shm Size	Network Hardware	Attributes
n0[01-48] (48)	2	Intel(R) Xeon(R)	E5440	2.83 GHz	4	8	16 GB	35G	7.8G	ib_ddr, ib	blade, del_int_8_16, intel8, intel, sse4_1, sse3

Figure 3.4: Specification of Cluster hardware

3.3 Benchmark Setup

We executed our experiments with C and C++ programs from the SPEC cpu2017 benchmark suite. We have worked with nine-speed benchmarks that consist of SPECspeed2017 floating-point (fp_speed) and SPECspeed2017 integer (int_speed). System requirements needed to install SPEC CPU2017 [1] are as follows

System requirements

(1) 16 GB main memory to run SPECspeed(per copy).

(2) 250 GB disk space is recommended to run a few benchmarks. However, a minimal

installation needs 10 GB
(3) C, and C++ compilers to be installed in the workstation as our target is to execute C/C++ benchmarks only.

3.3.1 Steps for Installation

SPEC CPU2017 is distributed as an ISO image, which needs to be mounted for installation. To install an ISO image, we need root privilege of the system. Here are the steps:
(1) Firstly, we set up the directory to mount ISO image and map it with the SPEC directory.
(2) ISO-mounted image contains the installation file. We mention the path where we install the benchmark. Then we run the installation file named *install.sh* which is provided by SPEC CPU2017.
(3) The SPEC directory contains the files' structure as follows:

1. *benchspec*: Some suite-wide files.
2. *CPU*: Benchmarks.
3. *bin*: Provides tools to run and report on the suit.
4. *config*: Config files to run each benchmark.
5. *Docs*: Some HTML documents and plain text documents.
6. *Results*: Contains a result that provides some log files and reports.
7. *temp*: Some temporary folders and files.
8. *tools* : Sources for the CPU 2017 tools.

3.3.2 Benchmark Setup

This section describes how to compile a benchmark using tools provided by SPEC. Here are the steps:

(1) Each benchmark has a build folder where all the binaries are created.

(2) Data files include three different forms of input data:

1. ref: The timed datasets.
2. test: Information for a straightforward test of an executable's functionality.
3. train: Information for Feedback-Directed Optimization(FDO).

(3) docs: Folder for documentations of benchmarks

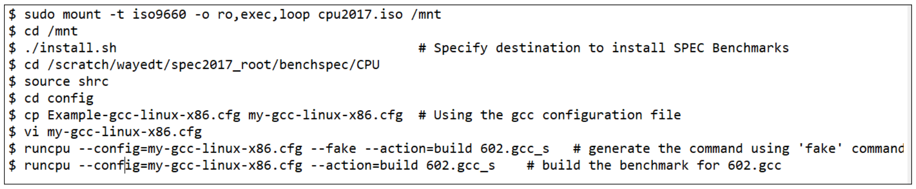
(4) exe: Compiled version of benchmarks

(5) run: Benchmarks are run here and all the logs files and error files are created

(6) spec: Metadata for the benchmarks' SPEC information

(7) src: It offers the benchmarks' sources

We opened the configuration files for each benchmark. Each benchmark includes a few sample configuration files. Based on our operating system, we selected a configuration file. We used the configuration file *my-gcc-linux-x86.cfg* as an example. To build a benchmark, we next transferred the configuration file to our own configuration file. We utilized the *runcpu tool*, which is a feature of the SPEC CPU2017. With a fake command, we ran the configuration file using *runcpu tool* to generate all the configuration instructions required. Then, emulated the SPEC-provided environment to run all build instructions produced by *runcpu tool*. A certain number of configuration and environmental variables were set for each benchmark in this simulation. We first set up the SPEC environment to build a benchmark using *source shrc*. Following that, we employed produced instructions using *fake* command for each benchmarks, copying the commands generated into our configuration file. To automate the benchmark build procedure, we built scripts. To create the benchmark, we modified several options and used the *runcpu* command to run the configuration file. The steps are as shown in Figure 3.5.



```
$ sudo mount -t iso9660 -o ro,exec,loop cpu2017.iso /mnt
$ cd /mnt
$ ./install.sh                                   # Specify destination to install SPEC Benchmarks
$ cd /scratch/wayedt/spec2017_root/benchspec/CPU
$ source shrc
$ cd config
$ cp Example-gcc-linux-x86.cfg my-gcc-linux-x86.cfg  # Using the gcc configuration file
$ vi my-gcc-linux-x86.cfg
$ runcpu --config=my-gcc-linux-x86.cfg --fake --action=build 602.gcc_s   # generate the command using 'fake' command
$ runcpu --config=my-gcc-linux-x86.cfg --action=build 602.gcc_s    # build the benchmark for 602.gcc
```

Figure 3.5: Steps for building a benchmarks

Using the aforementioned processes as a guide, we created nine C/C++ benchmarks and ran our experiments with the GCC and Clang/LLVM compilers to gauge performance. The nine benchmarks are listed in Table 3.1.

Nine SPEC Benchmark List		
Benchmark Name	Type	Area
600.perlbench_s	Integer	Perl interpreter.
602.gcc_s	Integer	GNU C Compiler.
605.mcf_s	Integer	Route Planning.
623.xalancbmk_s	Integer	XML to HTML conversion via XSLT.
631.deepsjeng_s	Integer	Artificial Intelligence: alpha-beta tree search(chess).
641.leela_s	Integer	Artificial Intelligence: Monte Carlo tree search(Go).
657.xz_s	Integer	General data compression.
644.nab_s	Float	Molecular dynamics.
619.lbm_s	Float	Fluid dynamics.

Table 3.1: SPEC cpu2017 benchmarks used in this used.

3.4 Tools

In this section we describe the other important tools that we used for this work.

3.4.1 runcpu

The primary tool for SPEC CPU2017 is *runcpu*. Each benchmark is built and executed using a Unix shell[1]. The benchmark configuration files are used for its operation. The configuration files for the two compilers are nearly identical to the ones used in the SPEC examples. For all benchmarks, we utilized the best and most conservative optimization level, *-O3*, with no aggressive optimization. *runcpu* is used repeatedly to construct the benchmark. As *base matrix* may be utilized to customize the optimization for each benchmark independently, we employ them rather than a peak model.

3.4.2 perf Tool

Perf is a strong Linux-based program that can measure CPU performance counters, tracepoints, and kprobed and uprobes (dynamic tracing)[8, 7]. A simple command-line tool, the Linux perf tool is used to profile and track CPU performance on Linux computers. Although it is a relatively straightforward tool, it can analyze performance in depth. The perf tool has a few subcommands for gathering, tracking, and analyzing CPU event data.

Performance counters are CPU hardware registers that count hardware events such as instructions executed, cache-misses suffered, or branches mispredicted. They form the basis for profiling applications to trace dynamic control flow and identifying hotspots. *perf tool* provides rich, generalized abstractions over hardware-specific capabilities. Among others, it provides counters per task, per CPU, and per-workload, sampling, and source code event annotation.

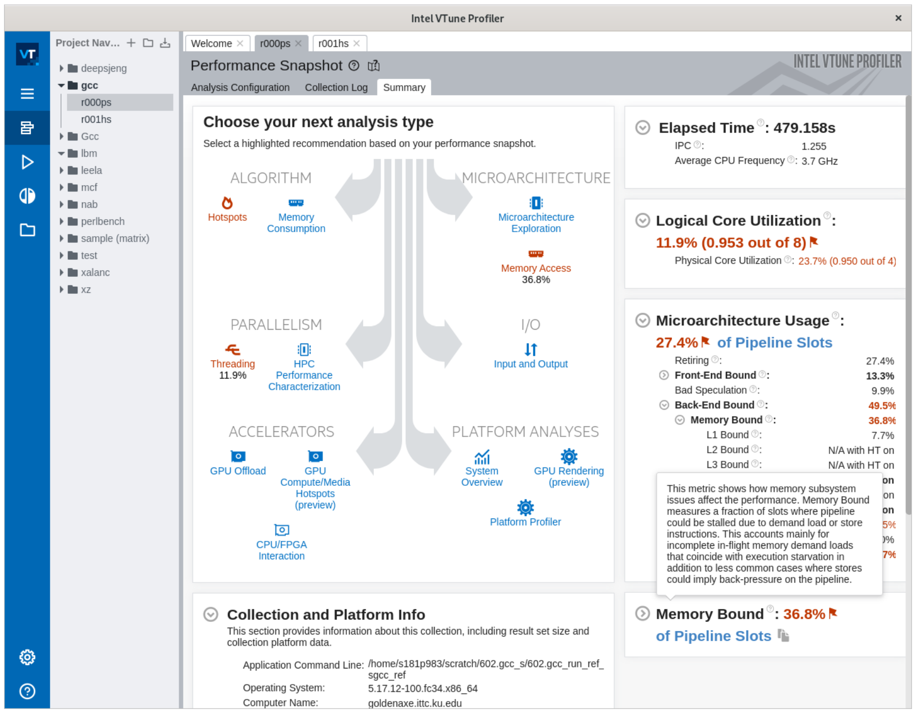
Tracepoints are instrumentation points placed at logical locations in code, for system calls, TCP/IP events, file system operations, etc. These have negligible overhead when not in use, and can be enabled by the perf command to collect information including timestamps and stack traces. The userspace perf command presents a simple-to-use interface with commands like:

1. *perf stat:* It collects event totals.
2. *perf record:* It documents occurrences for reporting in the future.
3. *perf report:* It segments events by function, process, etc.
4. *perf annotate:* It includes event counts in annotations of the source code or assembly.
5. *perf top:* It shows live event count.
6. *perf bench:* It executes several microbenchmarks for the kernel.

3.4.3 VTune Profiler

VTune Profiler(formerly VTune Amplifier) is a performance analysis tool for x86 based machines running Linux or Microsoft Windows operating systems[24]. Many features work on both Intel and AMD hardware, but advanced hardware-based sampling requires an Intel-manufactured CPU. VTune Profiler is available for free as a stand-alone tool or as part of the Intel oneAPI Base Toolkit. It supports several languages such as C, C++, Data Parallel C++ (DPC++), C, Fortran, Java, Python, Go, OpenCL, assembly and any mix. It also supports other native languages that follow standards and can also be profiled. VTune supports local and remote performance profiling. It can be run as an application with a graphical interface, as a command line, or as a server accessible by multiple users using a web browser.

On 32-bit and 64-bit x86 processors, the Intel VTune profiler is a performance analysis tool that depends on the underlying hardware counters to get the run-time characteristics of the program under evaluation. The profiler provides thorough details on the amount of time spent in each function and the amount of time the hardware spends executing code, stalls, etc. Hotspots in the program (the time-consuming routines), hardware-related problems with the code (such as data sharing, cache misses, branch misprediction, and others), thread activity, and system transitions (such as migrations and context shifts) may all be found or determined using this information. The summary of the tool's analysis is shown in Figure 3.6.



Choose your next analysis type

Select a highlighted recommendation based on your performance snapshot.

Elapsed Time: 479.158s

IPC:	1.255
Average CPU Frequency:	3.7 GHz

Logical Core Utilization: 11.9% (0.953 out of 8)

Physical Core Utilization: 23.7% (0.950 out of 4)

Microarchitecture Usage: 27.4% of Pipeline Slots

Retiring:	27.4%
Front-End Bound:	13.3%
Bad Speculation:	9.9%
Back-End Bound:	49.5%
Memory Bound:	36.8%
L1 Bound:	7.7%
L2 Bound:	N/A with HT on
L3 Bound:	N/A with HT on

This metric shows how memory subsystem issues affect the performance. Memory Bound measures a fraction of slots where pipeline could be stalled due to demand load or store instructions. This accounts mainly for incomplete in-flight memory demand loads that coincide with execution starvation in addition to less common cases where stores could imply back-pressure on the pipeline.

Memory Bound: 36.8% of Pipeline Slots

Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Application Command Line:	/home/s181p983/scratch/602.gcc_s/602.gcc_run_ref_sgcc_ref
Operating System:	5.17.12-100.fc34.x86_64
Computer Name:	goldenaxe.ittc.ku.edu

Figure 3.6: Intel Vtune Amplifier

3.5 Implementation and Experiment

The experimentation was concentrated on nine benchmarks. The configuration files for the two compilers, GCC and LLVM, resemble those in the SPEC examples. Shell scripts were used to develop the configuration files to compile each benchmark. We utilized the *runcpu* command together with the *fake* keyword to generate all the necessary instructions that helped to build the benchmark. We added each generated command to shell scripts to create complete configuration files. The preset shell scripts were first run without any optimization, and the resultant binary was then collected. Then, we compiled the same file repeatedly while utilizing the default and top optimization setting, -O3. Additionally, this run generated a binary file with the -O3-level optimization setting. We carried out the same procedure for every benchmark and created them all with the appropriate folder structures for two distinct compilers, such as GCC and LLVM. The three inputs (test, train, and ref) are distinct for each benchmark. The purpose of these three inputs are:

(a) Test : Utilizing the test workload, we built up and executed each benchmark. We verified that they provided the correct answers. The test workloads are only executed as an extra layer of assurance that the resulting executables are functioning correctly. Their times are not reported and do not contribute to the overall metrics. Therefore, multiple benchmarks can be run simultaneously.

(b) Train : Performed similar steps for the train workload, for the same reasons, with the same verification, non-reporting, and parallelism.

(c) Ref : We have taken the base files for each benchmark and set up the directories for all the benchmarks. We ran refspeed (6xx) workload for our experiment.

3.5.1 Automated framework to run benchmark

To conduct our experiments correctly and objectively for this study, we implemented an automated framework to build and run the benchmarks. In this section, we describe these engineered frameworks.

We developed an automated framework that can build and run the benchmark. Shell scripts were used in the development of this automated framework. We deployed the automated framework on the ITTC cluster computer. The ITTC cluster machine is a gateway to access several nodes with huge memory and high-performance processing units. A batch job is used to carry out the automated process on the cluster machine. To execute nine benchmarks concurrently, we requested nine parallel nodes with the necessary setup. The framework retrieves all executable binaries for each benchmark from the build folder and executes each benchmark with both self-input and cross-inputs. With cross-inputs, it creates performance measurements for each benchmark. Running this task manually was quite time-consuming and highly mundane. This automated system made the experiment more fast and efficient. A few benchmarks required running for extended periods of time, such as 15 to 16 hours or even a whole day. Therefore, using this program to build and run the nine benchmarks minimized experimental time and human effort. Using a batch job, which is a special feature of the cluster computer, we ran this automated framework on the ITTC cluster system. This batch job generated results for nine benchmarks on the cluster. We deployed this process with both the GCC and LLVM compilers.

3.5.2 Experiments with the GCC Compiler

The folder structure was established, with the typical inputs from SPEC benchmarks. Each benchmark typically uses three standard inputs, such as test, train, and ref inputs. Our investigation was to determine if employing profile-guided optimization(PGO) in source code results in any performance gains over -O3 optimization. We used the build configuration file with -O3 level optimization to accomplish our objective. Initially, the original source program was optimized using the traditional O3 level optimization technique. PGO uses optimization flags to direct its operations. It does not rebuild the program completely, but it enables the compiler to identify sections of code that require optimization based on how well the program performs.

The instrumented compilation was the initial stage of profile-guided optimization(PGO). The main output of this instrumented compilation was an executable binary file containing instrumented code. We created the instrumented profile using the flag *-fprofile-generate* with the GCC compiler. This run then created profile data with the suffix *.gcda in the build folder of the benchmark. Then, in order to create an executable binary, we used the generated profile data and then ran the same build configuration file with the flag *-fprofile-use*. To create three distinct executables using the three standard inputs offered by SPEC, we executed the final executable binary with standard inputs such as test, train, and ref. The Table 3.2 shows lists of PGO flags used with the GCC compiler.

Profile Guided Optimization flags for GCC		
Flags	Action	Generated files
-fprofile-generate	This flag adds instrumented code in the source code	*.gcda.
-fprofile-use	This flag uses the instrumented code(* .gcda) and generate profile	Binary executable.

Table 3.2: List of PGO flags with GCC compiler.

The automated framework we created received the executable binaries and inputs after all the executable binaries such as test, train, ref, and O3, were generated. With cross-input, this automated system conducted twelve runs for each binary and produced outputs to a file.This framework internally employs the *perf stat* command offered by the Linux operating system which generates several performance metrics including task-clock(msec), page faults, cycles, instructions per cycle, and branches. Another script that we created exports data to a file that retrieves only task-clock(msec) for each cycle of twelve runs from a previously generated file. Then we computed the standard deviation as well as

average CPU consumption with PGO over O3-level optimization.This staistics helped us to analyze the performance of each benchmark on two compilers.
An identical procedure was then carried out for cumulative binary. To create the cumulative binaries, we integrated all of the test, train, and ref binaries. We executed cumulative binaries with ref inputs since we thought they would perform better. We used the same procedure to get the data and export them to a file.The statistics for each benchmark were then shown as bar graphs, along with an error bar that displayed the standard deviation.

3.5.3 Experiment with the Clang/LLVM Compiler

Our primary objective was to analyze the behavior of SPEC CPU benchmark programs with the GCC and LLVM with both traditional optimization and profile-guided optimization. In the case of the LLVM/Clang compiler, we followed a similar process to that of the GCC.
We compiled the build configuration files of the benchmark with the LLVM compiler. All configuration files were similar to those of the GCC compiler. We modified the configuration file to be compatible with the Clang/LLVM compiler. Then, we ran the build file using O3-level optimization, created the executable binary with standard O3-level optimization, and stored it in the build folder of the benchmark. Then, we again compiled the same configuration file utilizing PGO flags to generate the executable binary file with profile instrumented code. We used the *-fprofile-instr-generate* flag provided by LLVM to generate the instrumented code. During program execution, it captures the profile information of a program and saves it as an executable binary in the benchmark's build folder.
We ran the instrumented binary with train input to ensure that the resulting executable functions correctly. Then, we compiled the instrumented binary file with the reference input of the benchmark. During the execution of the binary, it collected all the profile information

of programs and created a *default.profraw* file in the run folder of the benchmark. This execution can created multiple **.profraw* files. These files contained all *raw* profile data. We combined all the raw instrumented **.profraw* files and converted them to *code.profdata* files using the *merge* command of the *LLVM-profdata* tool, which is provided by the LLVM compiler.

We built the merged instrumented code, *code.profdata*, using the flag *-fprofile-instr-use* to specify the collected profile data. This run created an executable binary file with profile data and stored it in the benchmark's build folder. Then, we ran the executable binary with the test, train, and ref inputs. For each benchmark, we followed the same steps and created executable binaries with profile information. The PGO flags used with the LLVM/Clang compiler are shown in the below Table 3.3.

Profile Guided Optimization flags for LLVM		
Flags	Action	Generated files
-fprofile-instr-generate	This flag adds instrumented code in the source code	default.profraw and code.profdata.
-fprofile-instr-use	This flag uses the instrumented code(code.profdata) and generate profile	Binary executable.

Table 3.3: List of PGO flags with LLVM compiler.

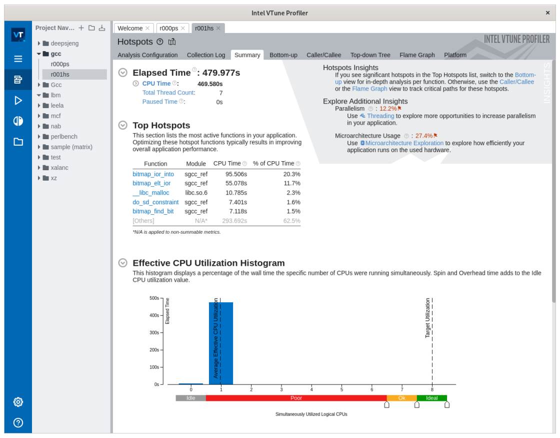
Once all the executable binaries were created for each benchmark, we ran each binary with all inputs. As an illustration, we executed the test binary with the test input along with the train and ref inputs. We developed an automated framework using shell scripts that submits a batch job to the cluster machine to execute the benchmark. This framework uses the *perf stat* command provided by the Linux perf tool to measure performance metrics such as *task-clock(msec)*, page faults, cycles, instructions per cycle, and branches. In our test, we ran all created executable binaries with the test, train, and ref inputs on the automated

framework. This framework helped in the execution of each benchmark twelve times and stored data in a file. We developed another script using python that fetches the performance parameter, *task-clock(msec)*, from the stored data and provides the standard deviation and the ratio between average CPU utilization using PGO and O3-level optimization.
We combined all PGO-test, PGO-train, and PGO-ref binaries to generate a cumulative binary, which we then executed using ref input. The same process was used to get data for the cumulative binary. We also gathered data for each benchmark using identical LLVM/Clang compilation procedures.

3.5.4 Experiment using Intel vTune Profiler

One important contribution of this work was an analysis of hotspots using the Intel vTune Profiler[19]. It provides predefined analysis configurations to address performance metrics. We used the hotspots configuration that investigates call paths and finds which part of the code takes up the maximum amount of time. This functionality of the vTune profiler provides CPU utilization data for each hotspot and identifies opportunities to tune the algorithm of a program.
Our studies employ a mechanism that we built to quantify the percentage improvement or degradation of the top ten functions for each benchmark using PGO over O3-level optimization. We provide executable binaries like PGO-test, PGO-train, PGO-ref, and O3 binaries and ref input to the Intel vTune profiler to analyze the hotspots for each benchmark. Our technique for measuring the performance metrics collects data on CPU utilization time for the top 10 most-used functions using PGO over No-PGO. The resulting ratio gives us a percentage measure of the improvement or degradation of the top 10 hotspots and provides the overall performance measure for each benchmark. When calculating the performance measure, we collect five recorded data values for each of the top 10 functions. To create a

measure of percentage CPU utilization, we compute an average of the total time spent in every method during the compilation of each benchmark using Intel Vtune Profiler. Then, we calculate the ratio of data using PGO over No-PGO O3-level optimization. The Intel vTune Profiler and the organization of the analysis are shown in Figure 3.7.



Intel VTune Profiler

Hotspots

Elapsed Time: 479.977s
CPU Time: 469.580s
Total Thread Count: 7
Paused Time: 0s

Hotspots Insights
If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee or the Flame Graph view to track critical paths for these hotspots.

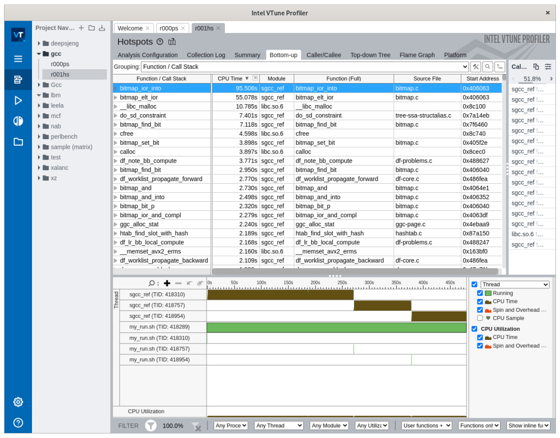
Explore Additional Insights
Parallelism: 12.2%
Use Threading to explore more opportunities to increase parallelism in your application.
Microarchitecture Usage: 27.4%
Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.

Top Hotspots
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
bitmap_ior_into	sgcc_ref	95.506s	20.3%
bitmap_elt_ior	sgcc_ref	55.078s	11.7%
__libc_malloc	libc.so.6	10.785s	2.3%
do_sd_constraint	sgcc_ref	7.401s	1.6%
bitmap_find_bit	sgcc_ref	7.118s	1.5%
[Others]	N/A*	293.692s	62.5%

*N/A is applied to non-summable metrics.

Effective CPU Utilization Histogram
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



Intel VTune Profiler

Hotspots

Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Module	Function (Full)	Source File	Start Address
bitmap_ior_into	95.506s	sgcc_ref	bitmap_ior_into	bitmap.c	0x406063
bitmap_elt_ior	55.078s	sgcc_ref	bitmap_elt_ior	bitmap.c	0x406063
__libc_malloc	10.785s	libc.so.6	__libc_malloc		0x8c100
do_sd_constraint	7.401s	sgcc_ref	do_sd_constraint	tree-ssa-structalias.c	0x7a14eb
bitmap_find_bit	7.118s	sgcc_ref	bitmap_find_bit	bitmap.c	0x7f6460
cfree	4.598s	libc.so.6	cfree		0x8c740
bitmap_set_bit	3.898s	sgcc_ref	bitmap_set_bit	bitmap.c	0x405f2e
calloc	3.897s	libc.so.6	calloc		0x8cec0
df_note_bb_compute	3.771s	sgcc_ref	df_note_bb_compute	df-problems.c	0x488627
bitmap_find_bit	2.950s	sgcc_ref	bitmap_find_bit	bitmap.c	0x406040
df_worklist_propagate_forward	2.770s	sgcc_ref	df_worklist_propagate_forward	df-core.c	0x486fea
bitmap_and	2.730s	sgcc_ref	bitmap_and	bitmap.c	0x4064e1
bitmap_and_into	2.498s	sgcc_ref	bitmap_and_into	bitmap.c	0x406352
bitmap_bit_p	2.320s	sgcc_ref	bitmap_bit_p	bitmap.c	0x406040
bitmap_ior_and_compl	2.279s	sgcc_ref	bitmap_ior_and_compl	bitmap.c	0x4063df
ggc_alloc_stat	2.240s	sgcc_ref	ggc_alloc_stat	ggc-page.c	0x4ebaa9
htab_find_slot_with_hash	2.189s	sgcc_ref	htab_find_slot_with_hash	hashtab.c	0x87a150
df_lr_bb_local_compute	2.168s	sgcc_ref	df_lr_bb_local_compute	df-problems.c	0x488247
__memset_avx2_erms	2.160s	libc.so.6	__memset_avx2_erms		0x163bf0
df_worklist_propagate_backward	2.109s	sgcc_ref	df_worklist_propagate_backward	df-core.c	0x486fea

Thread: sgcc_ref (TID: 418310), sgcc_ref (TID: 418757), sgcc_ref (TID: 418954), my_run.sh (TID: 418289), my_run.sh (TID: 418310), my_run.sh (TID: 418757), my_run.sh (TID: 418954)

Figure 3.7: Performance of gcc benchmark using Intel vTune Profiler

Chapter 4

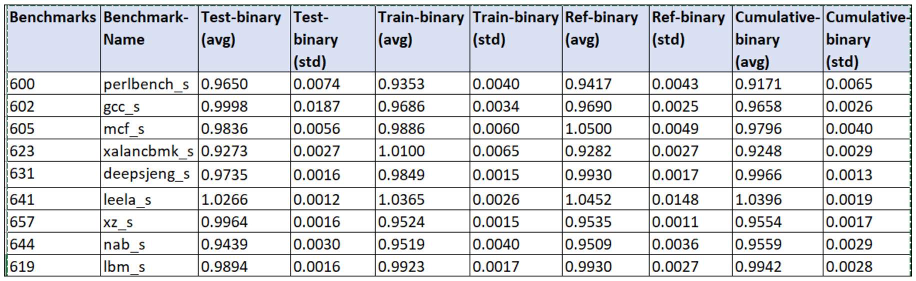
Results and Discussion

In this chapter, we describe the results of our experiments that investigate the characteristics of profile-guided optimizations with the GCC and LLVM. We conduct experiments to explore the questions in Section 2 and provide our findings.

4.1 Overall Observations

We run our experiment with two standard compilers, GCC and LLVM. Our first set of experiments is designed to evaluate how PGO with the GCC and LLVM/Clang compilers can utilize profile data to improve performance. We prepare configurations to compare the behavior and performance of each benchmark using PGO over No-PGO O3-level optimization. In all cases, we run each configuration 12 times and report their average and standard deviation. Figures 4.1 and 4.2 present the execution time with GCC and LLVM compilers with PGO over No-PGO for each benchmark. These two figures help us answer the first question of our experiments.

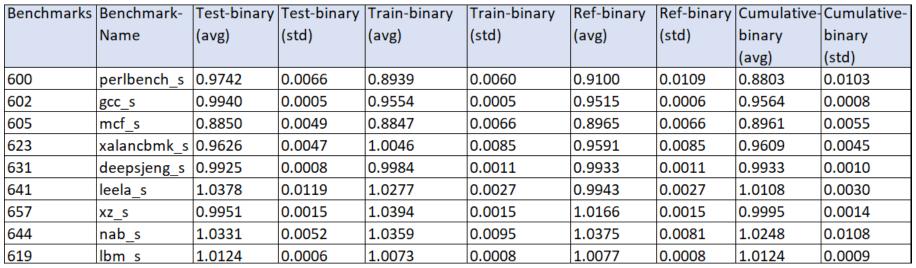
We anticipate that these two compilers will provide similar results in all benchmarks. In practice, however, this is not always the case. Figure 4.1 shows that a few benchmarks perform better with GCC compared to the LLVM compiler. For instance, the GCC compiler



Benchmarks	Benchmark-Name	Test-binary (avg)	Test-binary (std)	Train-binary (avg)	Train-binary (std)	Ref-binary (avg)	Ref-binary (std)	Cumulative-binary (avg)	Cumulative-binary (std)
600	perlbench_s	0.9650	0.0074	0.9353	0.0040	0.9417	0.0043	0.9171	0.0065
602	gcc_s	0.9998	0.0187	0.9686	0.0034	0.9690	0.0025	0.9658	0.0026
605	mcf_s	0.9836	0.0056	0.9886	0.0060	1.0500	0.0049	0.9796	0.0040
623	xalancbmk_s	0.9273	0.0027	1.0100	0.0065	0.9282	0.0027	0.9248	0.0029
631	deepsjeng_s	0.9735	0.0016	0.9849	0.0015	0.9930	0.0017	0.9966	0.0013
641	leela_s	1.0266	0.0012	1.0365	0.0026	1.0452	0.0148	1.0396	0.0019
657	xz_s	0.9964	0.0016	0.9524	0.0015	0.9535	0.0011	0.9554	0.0017
644	nab_s	0.9439	0.0030	0.9519	0.0040	0.9509	0.0036	0.9559	0.0029
619	lbm_s	0.9894	0.0016	0.9923	0.0017	0.9930	0.0027	0.9942	0.0028

Figure 4.1: PGO vs NonPGO in GCC

outperforms LLVM for a few benchmarks such as 623.xalanbmk_s, 657.xz_s, 644.nab_s and 619.lbm_s.



Benchmarks	Benchmark-Name	Test-binary (avg)	Test-binary (std)	Train-binary (avg)	Train-binary (std)	Ref-binary (avg)	Ref-binary (std)	Cumulative-binary (avg)	Cumulative-binary (std)
600	perlbench_s	0.9742	0.0066	0.8939	0.0060	0.9100	0.0109	0.8803	0.0103
602	gcc_s	0.9940	0.0005	0.9554	0.0005	0.9515	0.0006	0.9564	0.0008
605	mcf_s	0.8850	0.0049	0.8847	0.0066	0.8965	0.0066	0.8961	0.0055
623	xalancbmk_s	0.9626	0.0047	1.0046	0.0085	0.9591	0.0085	0.9609	0.0045
631	deepsjeng_s	0.9925	0.0008	0.9984	0.0011	0.9933	0.0011	0.9933	0.0010
641	leela_s	1.0378	0.0119	1.0277	0.0027	0.9943	0.0027	1.0108	0.0030
657	xz_s	0.9951	0.0015	1.0394	0.0015	1.0166	0.0015	0.9995	0.0014
644	nab_s	1.0331	0.0052	1.0359	0.0095	1.0375	0.0081	1.0248	0.0108
619	lbm_s	1.0124	0.0006	1.0073	0.0008	1.0077	0.0008	1.0124	0.0009

Figure 4.2: PGO vs NonPGO in CLANG

On the other hand, Figure 4.2 shows the execution-time benefit from PGO with the LLVM/Clang compiler. We assume that both compilers will use similar PGO techniques to improve the performance of a program, as both compilers are similar C/C++ compilers. They do not, however, show comparable outcomes. Figure 4.2 depicts that LLVM outperforms the GCC compiler in a few benchmarks. For example, the LLVM compiler performs better than GCC on benchmarks such as 600.perlbench_s, 602.gcc_s, and 605.mcf_s.

Figures 4.3 and 4.4 represent the program performance using profile-guided optimization(PGO) over No-PGO standard -O3 level optimization with the GCC and LLVM compilers. The first bar for each benchmark in Figures represent the ratio of the program run-times with the PGO-test binary and O3 binary with the standard ref input. The second bar compares the ratio of the program runtime with the PGO-train binary and O3 binary with ref input, The third bar represents the ratio of the program runtime with the PGO-ref binary and O3 binary with ref input, while the last bar compares the program runtime PGO-cumulative binary and O3 binary with ref input. Every comparison gives an estimation of the performance gain/loss due to profiling the benchmark program with the GCC and LLVM compilers.

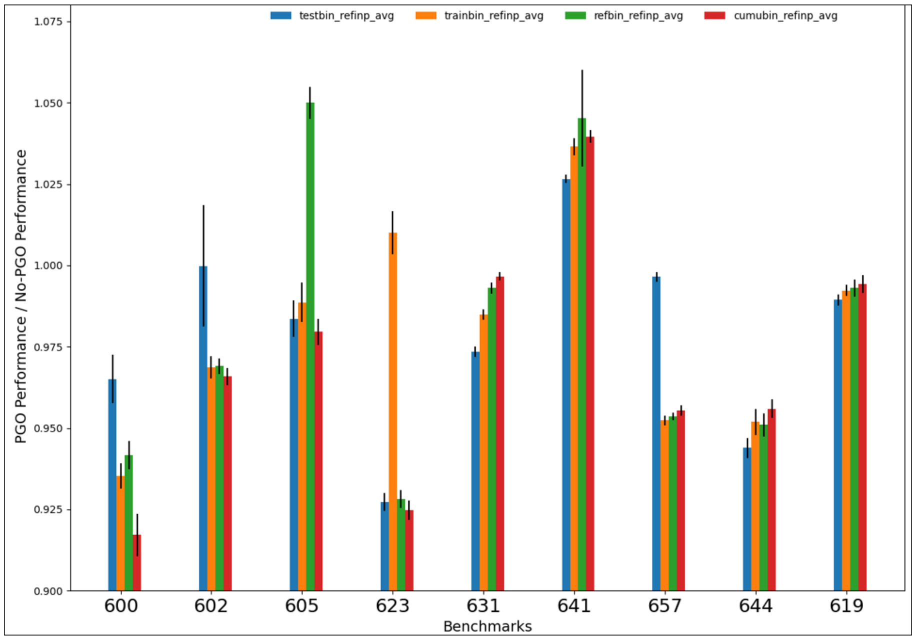


Figure 4.3: PGO vs NonPGO with ref inputs in gcc compiler

In this study, we evaluated the effectiveness of profiling the program behavior with standard ref input and then using that profile data to guide PGOs during an offline evaluation run with the ref input. One training run and one evaluation run are performed for each benchmark-input pair. The training run uses one benchmark-input pair and collects profile data and method compilation orders. The evaluation run executes the benchmark using the ref input and profile data. We use the automated framework, which is discussed in Section 6.1. The program run-time is recorded for 12 runs.

Theoretically, we expect the PGO-generated binary that is customized with the same input as used in the evaluation run to deliver the best run-time performance results. In practice, however, we found this to not always be the case. That is why, we obtain the best

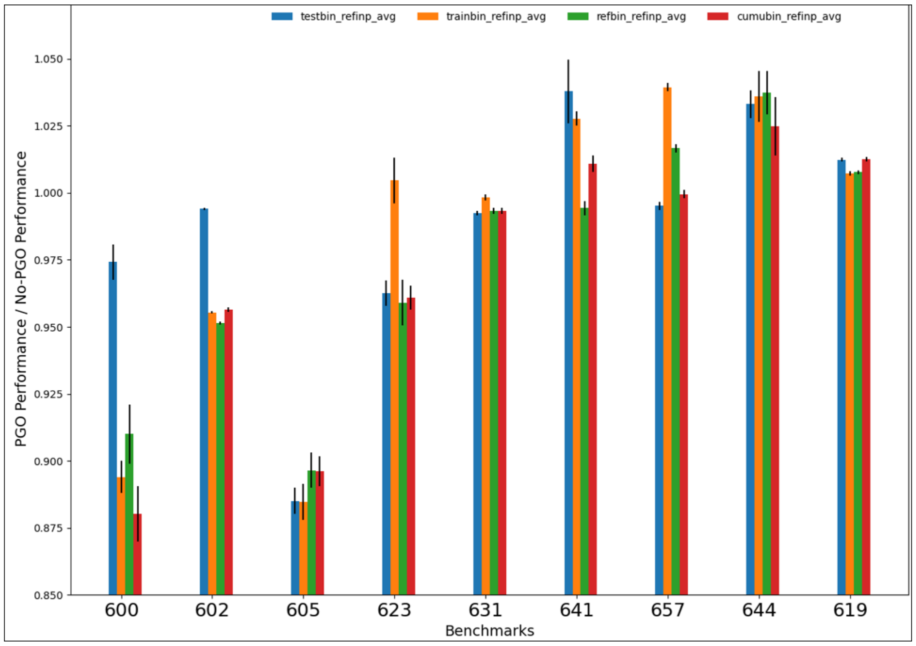


Figure 4.4: PGO vs NonPGO with ref inputs on CLANG compiler

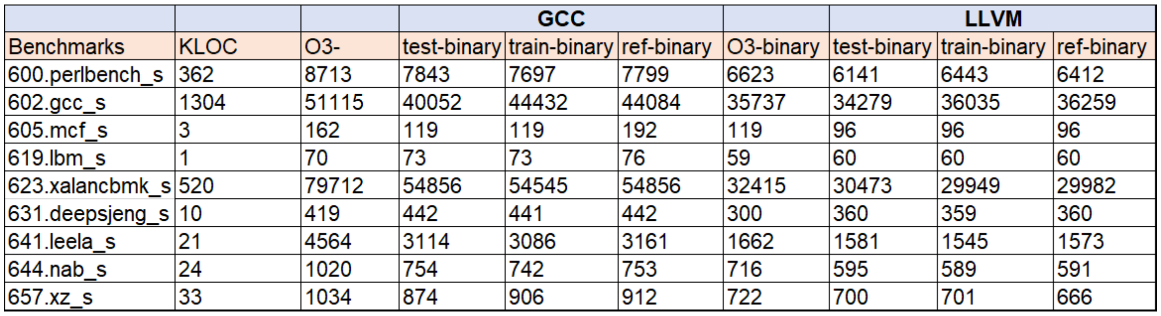
performance metrics by utilizing cross-binaries, such as PGO-test, PGO-train, and PGO-cumulative binaries with the ref input. In this study, we identify how different profiles affect the performance of PGOs.

For each benchmark with the GCC compiler, Figure 4.3 compares the run-time reported by the offline profiling to the run-time using O3-level optimization. We observe that few benchmarks provide the performance we expect. For instance, 631.deepsjeng_s, 657.xz_s, and 644.nab_s produce experimental findings that are consistent with the theory. These benchmarks show better performance with the refbinary-refinput pair. We also discover that using profiling data from PGO-cumulative binary with ref input provides performance that is within 4-9% of that obtained using profile data from O3-level binary. A number of benchmarks, for example, 600.perlbench_s (9%), 623.xalanbmk_s (8%), and 605.mcf_s(4%) provide a significant performance improvement over the No-PGO binary. In one instance, such as 641.leela_s, we detect significant performance degradation, whereas employing PGO has no performance impact on 631.deepsjeng_s.

For each benchmark, Figure 4.4 depicts the performance with the Clang /LLVM compiler using PGO over No-PGO. A few benchmarks,like 602.gcc_s, 623.xalabmk_s, 641.leela_s, and 619.lbm_s generate experimental results that confirm the profile-guided-optimization(PGO) hypothesis. These benchmarks perform better in refbinary-refinput pairs. We also observe from the results that a few benchmarks,like 600.perlbench_s and 605.mcf_s show significant performance improvement using PGO. In fact, the benchmark 600.perlbench_s shows a 12% improvement, and 605.mcf_s has an 11% improvement in performance using offline profiling. Figure 4.4 also exhibits that a few benchmarks, such as 641.leela_s, 644.nab_s, and 619.lbm_s have considerable performance degradation during the execution of the benchmark programs using PGO. There might be several causes for performance deterioration, but finding causes is outside the scope of this work. We also discover that a few benchmarks,such as 631.deepsjeng_s and 657.xz_s, have a negligible impact on PGO performance when compared to No-PGO.

Profile-Guided-Optimization(PGO) is a very well-known technique for improving the execution time of programs. It employs a few well-known techniques to improve performance. For instance, function inlining is the most common and popular PGO technique employed by standard compilers. As we have already stated before, GCC and LLVM both being similar C/C++ compilers, our assumption is similar PGO techniques are applied to improve the performance of the program, resulting in similar output when using PGO. In practice, however, they do not provide similar results. Therefore, they might not apply similar PGO techniques to improve the performance of the benchmark program. Compilers do not adhere to any typical trends to demonstrate performance increases by utilizing PGO. We observe that both compilers, GCC and LLVM, have different performance impacts on individual benchmarks. When GCC and LLVM performance is compared, there are few benchmarks that perform better in GCC, and not necessarily the same benchmarks perform better in LLVM. For instance, a few benchmarks, such as 600.perlbench_s, 602.gcc_s, 605.mcf_s, 623.xalanbmk_s, 657.xz_s, and 644.nab_s, have better performance with the GCC compiler using PGO. On the other hand, the 600.perlbench_s, 602.gcc_s, and 605.mcf_s benchmarks do better with the LLVM.

When comparing GCC and LLVM performance, we also observe that there is a difference in the size of the source code of each benchmark in terms of Kilo Lines of Code(KLOC) and executable size generated by each of the compilers. Figure 4.5 represents the comparison of executable sizes produced by the GCC and LLVM compilers. We can better understand the third research question by using these comparisons.

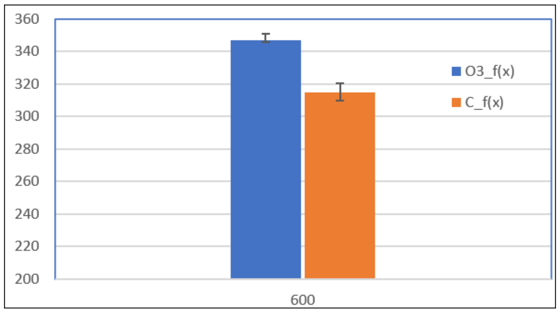


			GCC				LLVM		
Benchmarks	KLOC	O3-	test-binary	train-binary	ref-binary	O3-binary	test-binary	train-binary	ref-binary
600.perlbench_s	362	8713	7843	7697	7799	6623	6141	6443	6412
602.gcc_s	1304	51115	40052	44432	44084	35737	34279	36035	36259
605.mcf_s	3	162	119	119	192	119	96	96	96
619.lbm_s	1	70	73	73	76	59	60	60	60
623.xalancbmk_s	520	79712	54856	54545	54856	32415	30473	29949	29982
631.deepsjeng_s	10	419	442	441	442	300	360	359	360
641.leela_s	21	4564	3114	3086	3161	1662	1581	1545	1573
644.nab_s	24	1020	754	742	753	716	595	589	591
657.xz_s	33	1034	874	906	912	722	700	701	666

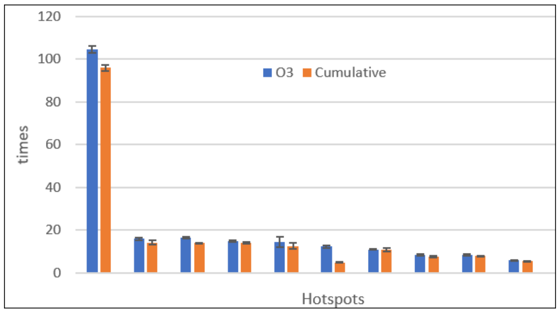
Figure 4.5: Executable Sizes of each benchmark with the GCC and LLVM

In experiments, the size of the code ranges from 1KLOC to 1304 KLOC[1]. The size of benchmark executables varies widely for the two compilers. We find that LLVM consistently creates smaller executables for all benchmarks compared to GCC. The GCC compiler produces the largest executables for 623.xalanbmk_s and 602.gcc_s. Overall, the executables generated by GCC are about 1.8x larger than LLVM executables. GNU being a cross-platform compiler focuses on probability, and this may be why the code generated by GCC is significantly larger.

The second set of experiments focuses on the performance effect of PGOs at the fine-grained function level. Using the Intel vTune profiler, we analyze performance at the function level for individual benchmark with binaries built using GCC and LLVM compilers.



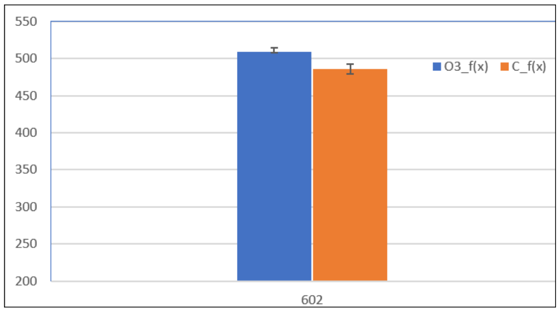
(a) Performance of PGO



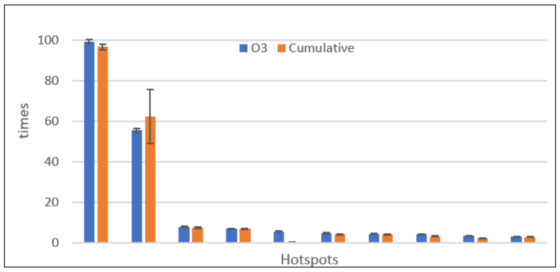
(b) Performance of top hotspots

Figure 4.6: Performance of 600.perlbench benchmark by Intel vTune Profiler

Figures 4.6 to 4.14 represent the results of the second set of experiments with the binaries produced by the GCC compiler. The Intel vTune Profiler analyze the previously profiled data with ref input. Results show the total execution time for each benchmark with both options - PGO is on and PGO off and evaluate the benefits of optimizations for the top ten hot functions.



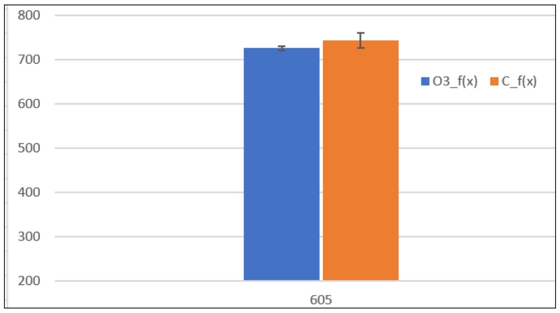
(a) Performance of PGO



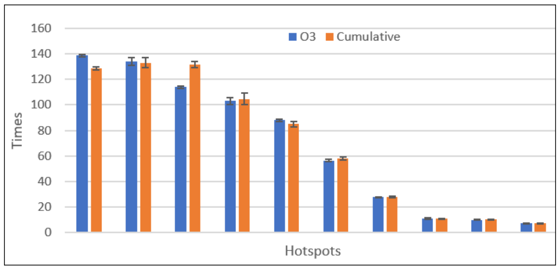
(b) Performance of top hotspots

Figure 4.7: Performance of 602.gcc benchmark by Intel vTune Profiler

Figures 4.8 and 4.11 provide a detailed performance analysis of benchmarks, 605.mcf_s and 641.leela_s, that are executed with binaries built using GCC compiler and standard ref input provided by the SPEC. The total execution time of benchmarks using PGO and No-PGO are depicted in Figures 4.8(a) and 4.11(a). For both benchmarks, the program outperforms No-PGO, indicating that PGO has a considerable negative impact on performance. To analyze the reason for the performance drop, Figures 4.8(b) and 4.11(b) repre-

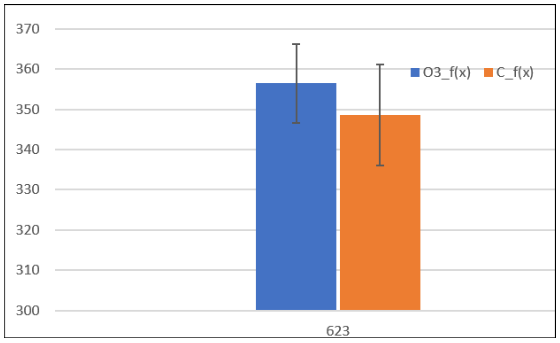


(a) Performance of PGO

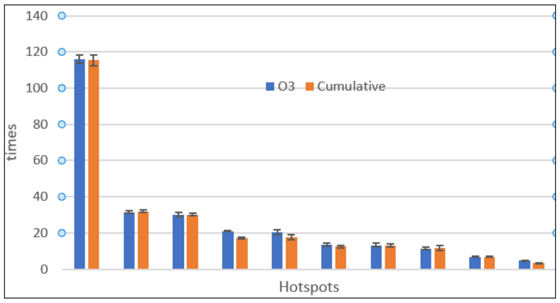


(b) Performance of top hotspots

Figure 4.8: Performance of 605.mcf benchmark by Intel vTune Profiler



(a) Performance of PGO



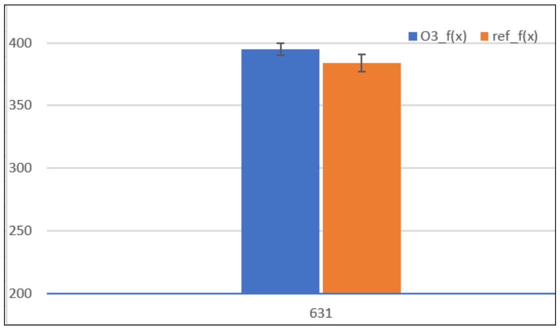
(b) Performance of top hotspots

Figure 4.9: Performance of 623.xalancbmk benchmark by Intel vTune Profiler

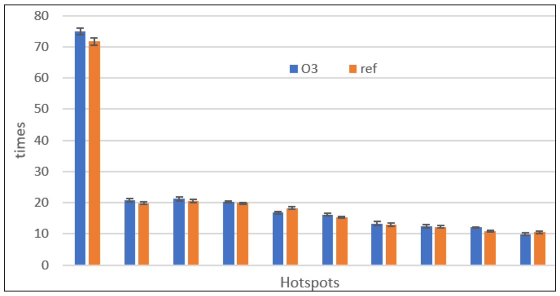
sent the finer-grained performance analysis of the top ten hot functions during the program execution. We observe that the third and fourth functions of 605.mcf_s and the first three consecutive functions of 641.leela_s take longer execution time with PGO than the No-PGO traditional approach.

On the other hand, Figures 4.15 to 4.23 demonstrate the outcomes of the tests with binaries generated by the Clang/LLVM compiler. The Intel vTune Profiler examines earlier profiled data with standard ref input. Results reflect the performance impact of PGOs on the top ten hot functions and display the overall execution time for each benchmark with and without PGO.

We notice that a few benchmarks, such as 641.leela_s, 644.nab_s, and 619.lbm_s, have performance degradation with binaries produced by the Clang/LLVM compiler. Figures

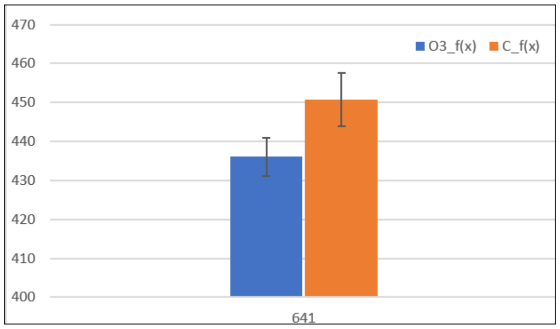


(a) Performance of PGO

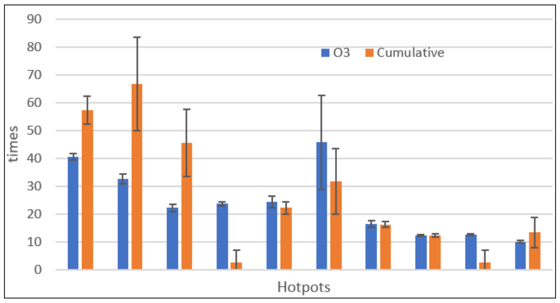


(b) Performance of top hotspots

Figure 4.10: Performance of 631.deepsjeng benchmark by Intel vTune Profiler



(a) Performance of PGO

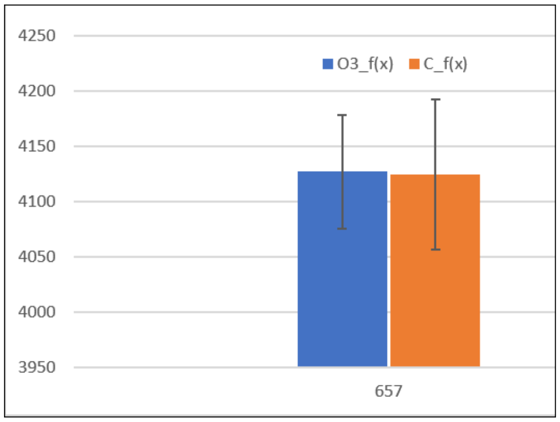


(b) Performance of top hotspots

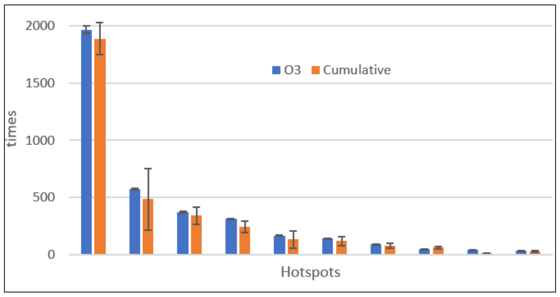
Figure 4.11: Performance of 641.leela benchmark by Intel vTune Profiler

4.20, 4.22, and 4.23 display the total execution time and the top ten most used functions of the benchmarks. We observe that three functions in 641.leela_s and two functions in 644.nab_s take more execution time than No-PGO. In case of the benchmark, 619.lbm_s, the first function takes the maximum time of execution of the benchmark, whereas the other functions are barely noticeable. However, the fraction of execution time devoted to remaining functions has a significant impact on total execution time. In this instance, every subsequent function requires a little longer than the first one. Consequently, the overall performance declines.

Additionally, we also collect data to compare the GCC and LLVM performance in terms of executable sizes of each benchmark and build-time with two compilers. Executable size of O3 level binary and build times of benchmark are shown in Figure 4.24.

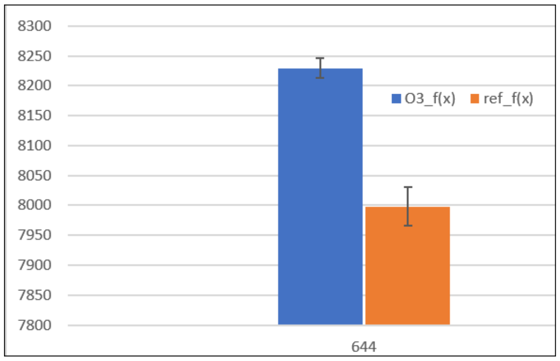


(a) Performance of PGO

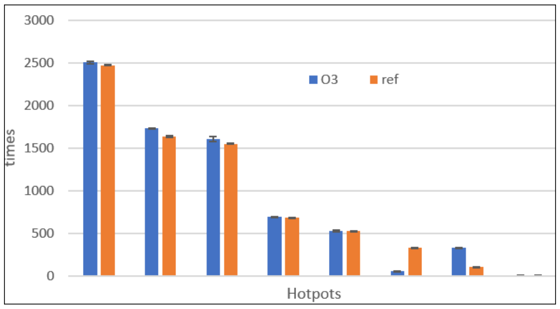


(b) Performance of top hotspots

Figure 4.12: Performance of 657.xz benchmark by Intel vTune Profiler



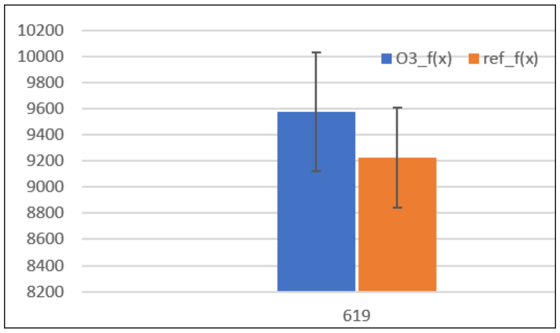
(a) Performance of PGO



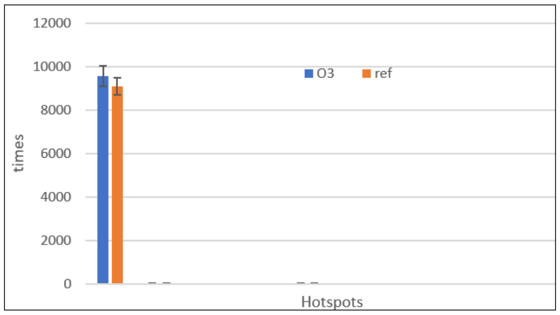
(b) Performance of top hotspots

Figure 4.13: Performance of 644.nab benchmark by Intel vTune Profiler

We observe that LLVM has a smaller executable size, but it has significantly longer built times in comparison with the GCC compiler. Though the GCC compiler produces executables that are considerably large in size, the build times are shorter than the build times of LLVM.

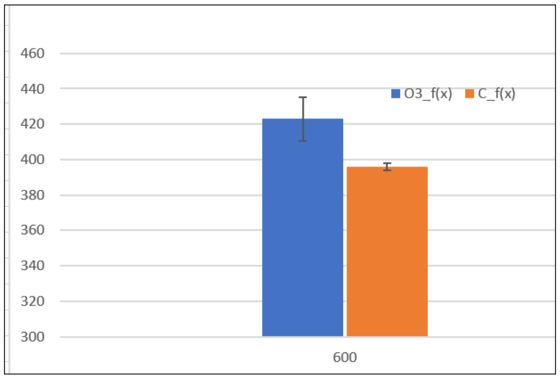


(a) Performance of PGO

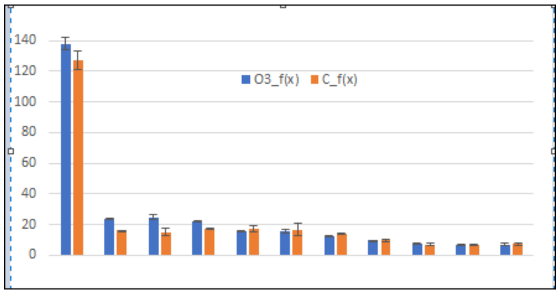


(b) Performance of top hotspots

Figure 4.14: Performance of 619.lbm benchmark by Intel vTune Profiler

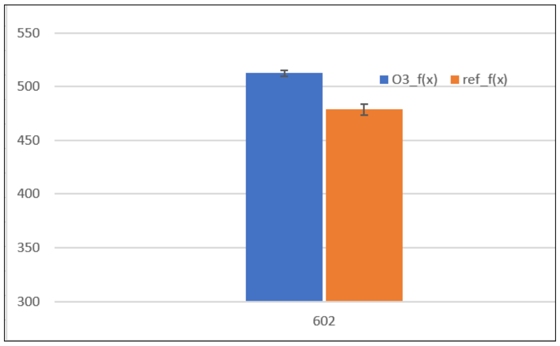


(a) Performance of PGO

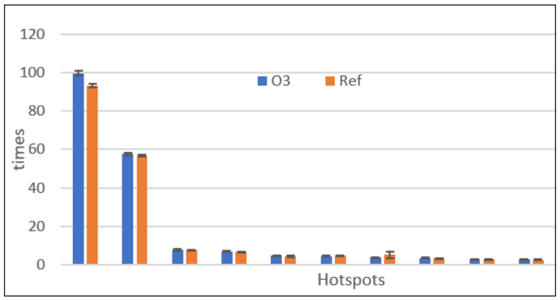


(b) Performance of top hotspots

Figure 4.15: Performance of 600.perlbench benchmark by Intel vTune Profiler

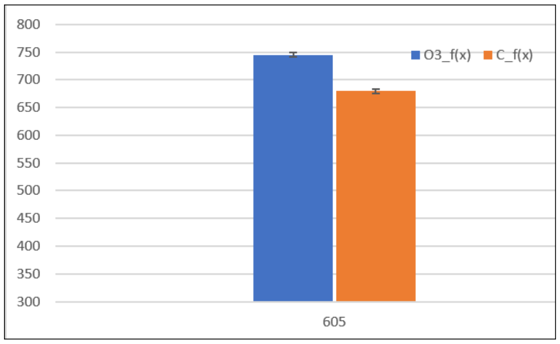


(a) Performance of PGO

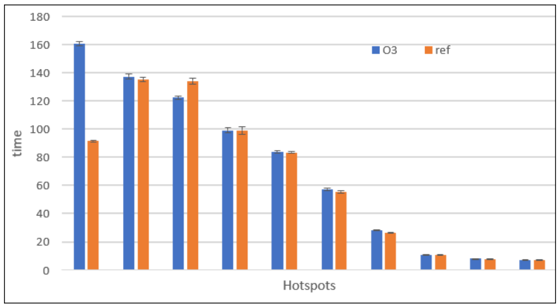


(b) Performance of top hotspots

Figure 4.16: Performance of 602.gcc benchmark by Intel vTune Profiler

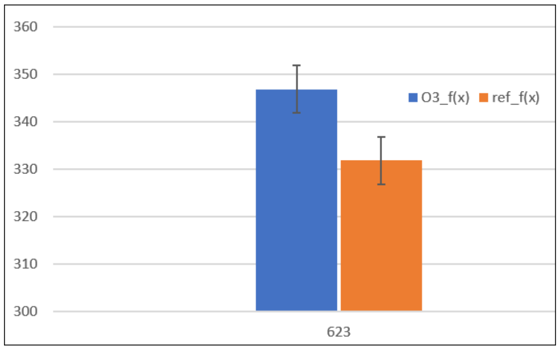


(a) Performance of PGO

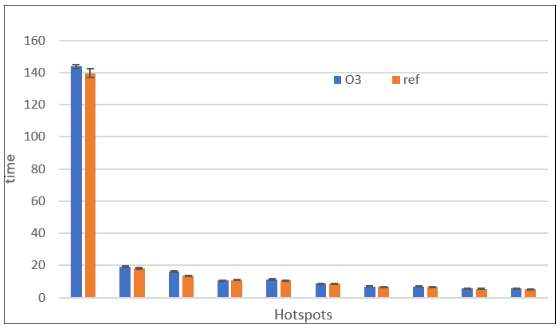


(b) Performance of top hotspots

Figure 4.17: Performance of 605.mcf benchmark by Intel vTune Profiler

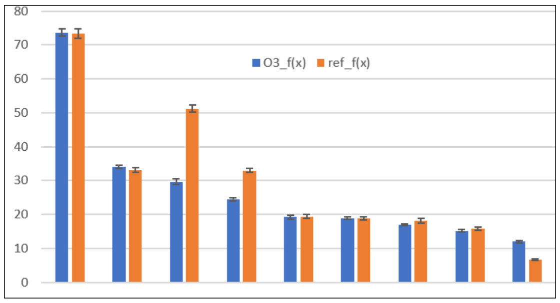


(a) Performance of PGO

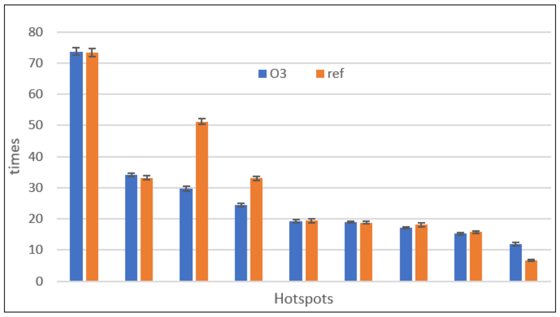


(b) Performance of top hotspots

Figure 4.18: Performance of 623.xalancbmk benchmark by Intel vTune Profiler

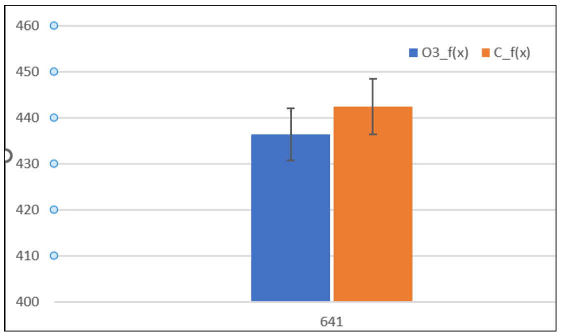


(a) Performance of PGO

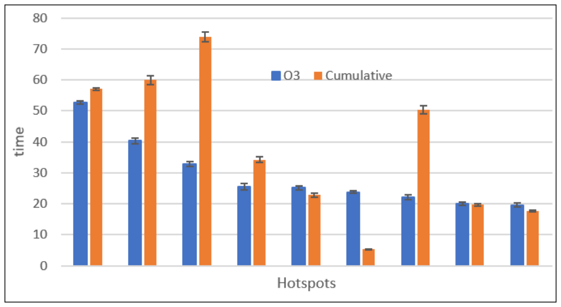


(b) Performance of top hotspots

Figure 4.19: Performance of 631.deepsjeng benchmark by Intel vTune Profiler

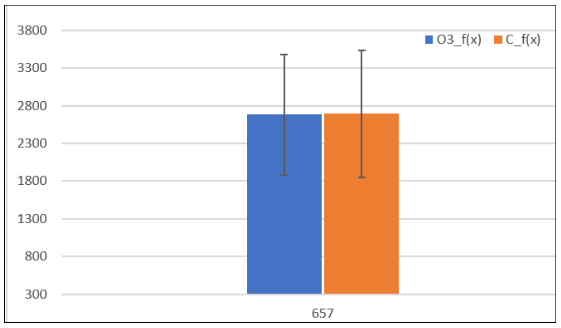


(a) Performance of PGO

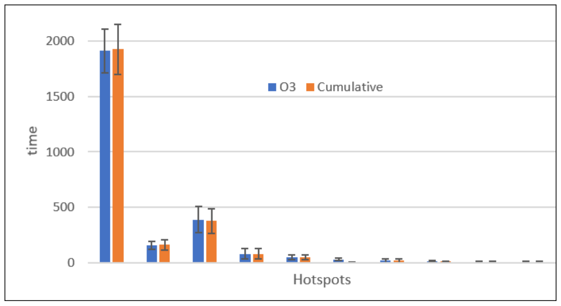


(b) Performance of top hotspots

Figure 4.20: Performance of 641.leela benchmark by Intel vTune Profiler

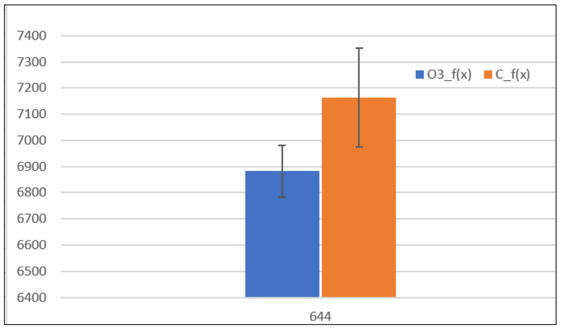


(a) Performance of PGO

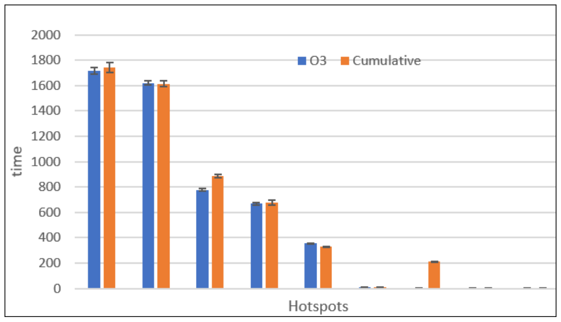


(b) Performance of top hotspots

Figure 4.21: Performance of 657.xz benchmark by Intel vTune Profiler

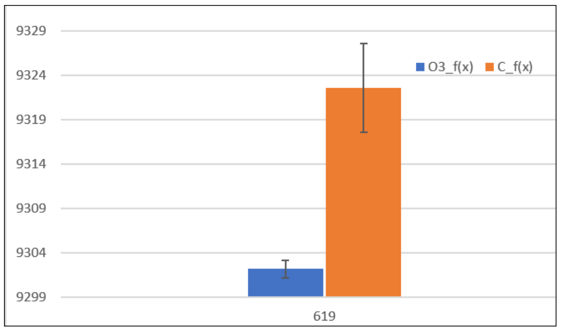


(a) Performance of PGO

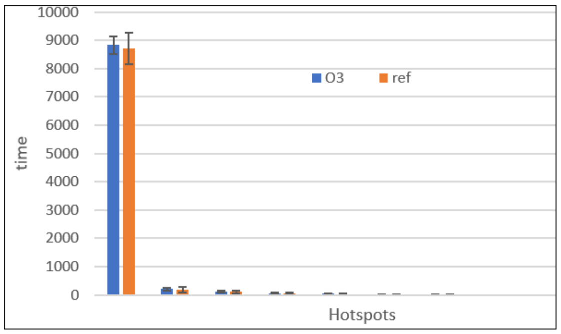


(b) Performance of top hotspots

Figure 4.22: Performance of 644.nab benchmark by Intel vTune Profiler

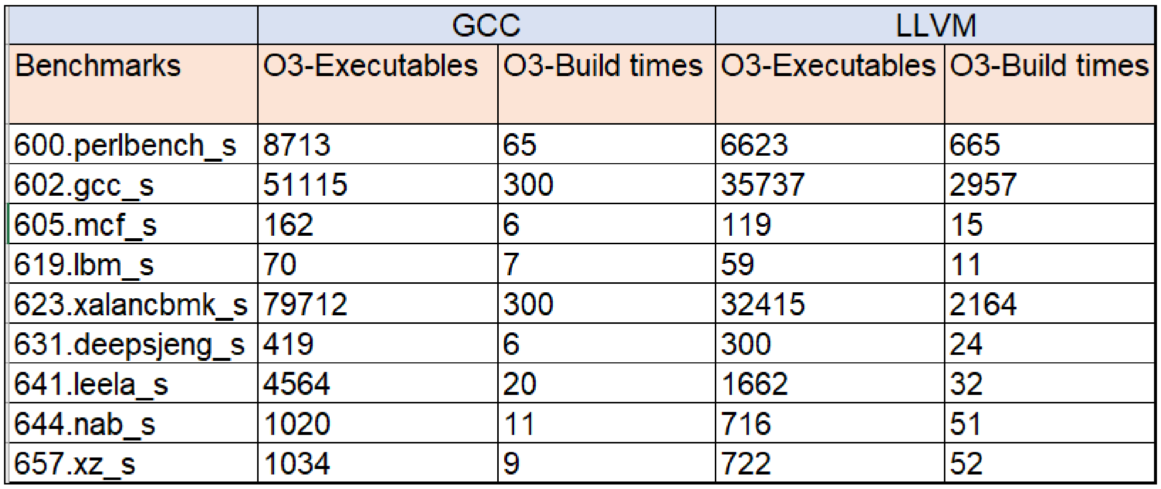


(a) Performance of PGO



(b) Performance of top hotspots

Figure 4.23: Performance of 619.lbm benchmark by Intel vTune Profiler



	GCC		LLVM	
Benchmarks	O3-Executables	O3-Build times	O3-Executables	O3-Build times
600.perlbench_s	8713	65	6623	665
602.gcc_s	51115	300	35737	2957
605.mcf_s	162	6	119	15
619.lbm_s	70	7	59	11
623.xalancbmk_s	79712	300	32415	2164
631.deepsjeng_s	419	6	300	24
641.leela_s	4564	20	1662	32
644.nab_s	1020	11	716	51
657.xz_s	1034	9	722	52

Figure 4.24: Executable Size and build time of each benchmark

Chapter 5

Conclusion and Future Works

5.1 Conclusion

We study the properties and evaluate the performance of PGOs in mainstream C/C++ compilers (GCC and LLVM) and standard benchmark programs (SPECcpu 2017). We explore four primary research questions: (a) What is the performance gain from PGO with mainstream compilers? (b) How do different program behavior profiles, corresponding to distinct program inputs, affect the efficacy of PGOs? (c) Based on the premise that most compiler optimizations (for C/C++) are mature and well-known, do different compilers deliver similar performance gains with PGOs?, and (d) What fraction of hot program functions see a significant benefit from PGOs? Additionally, we also compare the size and speed of the program binaries generated by the highest (-O3) optimization level in GCC and LLVM.

We make several notable, and some counter-intuitive, observations in this work. (a) The performance benefit delivered by binaries generated using PGOs (employed by the compilers used in this study) is often small. (b) While using a *different* profile/input during the training and evaluation runs can be expected to produce performance losses, we found that even using the *same* input can result in a performance loss in some cases. (c) Surprisingly, the *same* program profile does not always produce the best performance. (d) Surprisingly, we find that programs deliver widely different performance results when compiled using PGOs with GCC and LLVM. This observation suggests that different mainstream compilers seem to employ substantially different PGO optimizations and/or heuristics. (e) We explored the program-level performance trends in more detail by using the Intel vTune Profiler to measure the finer-grained function-level performance characteristics for PGOs. Again, we find the different functions are affected differently by PGOs with no overwhelming trend. (f) We also observe that binaries generated by LLVM are typically both significantly smaller and realize higher performance compared to corresponding binaries generated by GCC.

Finally, we believe that the frameworks we designed and constructed in this work, and our experimental results and observations can prove useful to other researchers and practitioners that wish to understand or use PGOs for their workloads.

5.2 Future Work

In the future, we plan to expand the scope of this work by studying other compilers and benchmarks. We will next use a white-box approach by studying the compiler code to better understand the causes of the performance results we observed in benchmarks with PGOs.

Bibliography

[1] SPEC CPU® 2017. Standard performance evaluation corporation. http://www.spec.org/cpu2017/index.html, November 2022. Cited on 7, 13, 17, 32

[2] IBM Corporation 2020. ”optimization levels”,. https://www.ibm.com/docs/en/aix/7.2?topic=implementation-efficient-program-design, August 2022. Cited on 8

[3] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P.F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005. Cited on 6

[4] Matthew Arnold and David Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *International Symposium on Code Generation and Optimization*, pages 51–62. IEEE, 2005. Cited on 8

[5] Intel Corporation. ”amd optimizing c/c++ compiler”,. https://developer.amd.com/amd-aocc/., August 2022. Cited on 11

[6] Intel Corporation. ”intel® c++ compiler classic developer guide and reference”,. https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming/profile-guided-optimization-pgo.html, August 2022. Cited on 3

[7] Intel Corporation. ”intel® c++ compiler classic developer guide and reference”,. https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming/profile-guided-optimization-pgo.html, August 2022. Cited on 17

[8] Paul J. Drongowski. Sand, software and sound. http://sandsoftwaresound.net/perf/perf-tutorial-hot-spots/, November 2013-2015. Cited on 17

[9] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. *ACM SIGARCH Computer Architecture News*, 28(5):202–211, 2000. Cited on 8

[10] Arthur Griffith. *GCC: The Complete Reference*. McGraw-Hill, Inc., USA, 1 edition, 2002. Cited on 2, 12

[11] 2018 Apple Inc. Llvm compiler overview. https://developer.apple.com/library/archive/documentation/CompilerTools/Conceptual/LLVMCompilerOverview/index.html, December 2012. Cited on 11

[12] KU ITTC. "cluster documentation",. https://help.ittc.ku.edu/Cluster_Documentation, 2021. Cited on 13

[13] Erik Johansson and Sven-Olof Nyström. Profile-guided optimization across process boundaries. *ACM SIGPLAN Notices*, 35(7):23–31, 2000. Cited on 7

[14] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 256–267, New York, NY, USA, 2017. Association for Computing Machinery. Cited on 1

[15] Cunning Plan. Programming language design and implementation, 1996. Cited on 8

[16] Víctor Rodríguez. "cutting edge toolchain (latest features in gcc/glibc)",. https://www.youtube.com., October 2019. Cited on 12

[17] Shai Rubin, Rastislav Bodik, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. *ACM SIGPLAN Notices*, 37(1):140–153, 2002. Cited on 8

[18] Michael D. Smith. Overcoming the challenges to feedback-directed optimization (keynote talk). *SIGPLAN Not.*, 35(7):1–11, jan 2000. Cited on 2

[19] Ranjan Hebbar SR, Mounika Ponugoti, and Aleksandar Milenković. Battle of compilers: An experimental evaluation using spec cpu2017. In *2019 SoutheastCon*, pages 1–8. IEEE, 2019. Cited on 11, 25

[20] Richard M Stallman et al. Using the gnu compiler collection. *Free Software Foundation*, 4(02), 2003. Cited on 12

[21] The Clang Team. "clang compiler user's manual",. https://clang.llvm.org/docs/UsersManual.html, August 2007-2022. Cited on 2

[22] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 977–989, New York, NY, USA, 2022. Association for Computing Machinery. Cited on 2

[23] April W Wade, Prasad A Kulkarni, and Michael R Jantz. Exploring impact of profile data on code quality in the hotspot jvm. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(6):1–26, 2020. Cited on 8

[24] Intel Developer Zone. “intel® vtunetm amplifier 2018 user's guide,”. `https://software.intel.com/en-us/vtune-amplifier-help-introduction`, March 2018. Cited on 18