Battle of Compilers: An Experimental Evaluation Using SPEC CPU2017

Ranjan Hebbar S.R
Dept. of Electrical and Computer Eng.
The University of Alabama in Huntsville
Huntsville, AL, 35899 U.S.A
rr0062@uah.edu

Mounika Ponugoti
Dept. of Electrical and Computer Eng.
The University of Alabama in Huntsville
Huntsville, AL, 35899 U.S.A
mp0046@uah.edu

Aleksandar Milenković
Dept. of Electrical and Computer Eng.
The University of Alabama in Huntsville
Huntsville, AL, 35899 U.S.A
milenka@uah.edu

***Abstract*—In order to meet growing application demands, modern processors are constantly evolving. Today they integrate multiple processor cores, an on-chip interconnect, large shared caches, specialized hardware accelerators, and memory controllers on a single die. Compilers play a key role in bridging the gap between abstract high-level source code used by software developers and the advanced hardware structures. This paper reports the results of a measurement-based study that evaluates three most prevalent compilers used in industry and academia. We compare the effectiveness of the Intel Parallel Studio XE-19 (IPS), the LLVM Compiler Infrastructure project, and the GNU Compiler Collection using the SPEC CPU2017 benchmark suite that is widely used to evaluate performance of modern computer systems. We quantitively evaluate the compilers with respect to metrics such as benchmark build times, executable code sizes, and execution times. The benchmarks are compiled using comparable optimization levels and they are run on an Intel 8th generation Core i7-8700K. The results show that LLVM creates the smallest executables, GNU has the lowest build times, and IPS has the best performance.**

Keywords—SPEC CPU2017, Benchmarking, Compilers, Performance Evaluation

I. Introduction

Computing is constantly evolving shaped by technology, applications, and markets trends. Mobile, IoT, and cloud computing have been major drivers of innovations recently and are likely to remain so in the next decade. The end of semiconductor scaling and demise of Moore's and Dennard's laws indicate that future performance improvements will have to come from architectural improvements and new computing structures rather than from getting smaller and faster transistors. Emerging new applications in domains such as machine learning, and block chain present a new set of challenges requiring more performance and increased energy-efficiency of modern processors. Market trends continue to shrink time-to-market.

To match application trends, modern x86 processors have evolved to be extremely complex hardware structures integrating multiple processor cores, on chip interconnect, uncore caches, myriad of hardware accelerators, and a memory controller on a single chip. Each processor core is a highly pipelined, superscalar machine incorporating out-of-order execution, speculative execution, prefetching, and various performance enhancing architectural features. A programmer's understanding of the underlying architecture to exploit the full potential of these features is critical for improving performance and/or for improving energy-efficiency.

On the other side modern high-level languages are constantly evolving, becoming more expressive and abstract. These trends help reduce software development time, reduce occurrence of software bugs, and improve code readability and maintainability. Unfortunately, higher levels of abstraction are known to increase program runtimes. With high-level languages the burden of optimization falls on compilers.

Modern compliers are extremely complex software that translate programs written in high-level languages into binaries that execute on the underlying hardware. The translations include a multitude of tasks, including preprocessing, lexical analysis, parsing, semantic analysis, and code optimization, and finally creating executable files. Compilers are not only expected to produce small executables that achieve the best possible performance, but also to do so in the least amount of time. Large software projects might involve multiple subsystems, code written in multiple programming languages, may incorporate matured libraries, and may span millions of lines of code. The predominant use of template programming requires the compiler to have multiple passes to create object files. Thus, the productivity of the software developer is directly affected by the build times of the program due to multiple iterations in the "edit-compile-debug" development flow. Thus, faster compilers are crucial to achieve higher productivity of software developers.

The selection of a compiler depends on parameters such as accessibility, support for hardware, efficiency of compiler, and backward compatibility. The most widely used modern compilers for software development are the Intel Parallel Studio XE (IPS), the GNU Compiler Collection (GNU), and the LLVM Compiler Infrastructure (LLVM). The goal of this paper is to evaluate effectiveness of these three compilers using the SPEC CPU2017 benchmark suites as a workload.

The SPEC CPU2017 benchmark suites are the latest in a series of standard benchmarks designed to stress a modern computer system's processor, memory, and compilers [1] [2]. Characterization studies performed on the previous SPEC CPU2000 and SPEC CPU2006 showed that these benchmarks match the evolution of the real-life workloads [3][4]. Reflecting recent changes in applications, SPEC CPU2017 includes four different benchmark suites organized in floating-point and integer *speed* suites and floating-point and integer *rate* suites.

978-1-7281-0137-8/19/$31.00 ©2019 IEEE

The *speed* suites are used to evaluate how fast the tested computer can execute a benchmark, wherein, the performance analyst has the option to choose the number of execution threads. The *rate* suites are used to test the throughput of a computer system, wherein, the performance analyst can select how many copies of a benchmark can be run concurrently [5].

There have been several studies focusing on effectiveness of compilers. A study by Machado et al., examines the effectiveness of compiler optimizations on multicores and finds that -O3 optimization level provides the best improvements across applications [6]. Plotnikov et al. designs tools to tune the optimization flags of the compiler to get the best possible performance [7]. Another study explores concurrent optimizations for both performance and energy-efficiency and find it is a challenging proposition [8] [9]. Earlier studies have compared IPS with Microsoft VC++ using workloads such as SPEC CPU 2000, and SPEC CPU 2006 [10] [11]. All the studies found that IPS is a better choice. Aldea et al. compares the sequential and parallel code generated by IPS, GNU, and Sun compiler for SPEC CPU 2006v1.1 benchmarks [12]. Lattner and Adve designed LLVM compilation framework. They compared the performance of LLVM-based C/C++ compiler and GNU on SPEC 2000 [13] [14]. The results show that LLVM performs better than GNU. However, all these studies rely on previous generation of the benchmarks, older versions of the compilers, and older hardware. To the best of our knowledge, there is no recent work which evaluates the latest versions of the compilers using the most recent SPEC CPU2017 benchmark suite.

In this study we evaluate effectiveness of Intel's IPS (commercial compiler) and two open-source compilers (GNU and LLVM), using SPEC CPU2017 benchmark suites as a workload. We consider three important metrics in evaluating compilers as follows: (a) the size of the executables (code sizes), (b) the execution times for *speed* benchmarks and throughput for *rate* benchmarks (performance), and (c) the total time needed to compile benchmarks (build times). The experiments are conducted on a recent Intel processor, Core i7-8700K. As the machine under test includes six processor cores, the experiments are conducted while varying the number of threads for speed benchmarks and the number of copies for rate benchmarks.

The main findings from this study are as follows.

- In terms of code size LLVM is a better choice as it creates executables of the smallest size. LLVM generates executables which are ~1.26x and ~5.92x smaller than the corresponding ones created by IPS and GNU, respectively.
- Regarding build times, the GNU compiler collection is the best choice. When a single processor is used to build executables, GNU build times are ~11.17x and ~1.02x shorter than the corresponding ones for LLVM and IPS, respectively. When six processor cores are used to build benchmarks, GNU build times are ~10.36x (LLVM) and 3.22x (IPS) shorter.
- Regarding benchmark performance, IPS is the clear winner with its ability to exploit hardware features of the x86 ISA. Considering the geometric mean of the SPEC ratios of all the benchmarks, we find that IPS executables run ~28% faster than the corresponding ones for LLVM and ~34% faster than the GNU executables.

The rest of the paper organized as follows. Section II gives a brief description of the SPEC CPU2017 benchmark suites and SPEC related metrics. Section III describes the compilers that we have used, and Section IV describes the tools and the experimental environment. Finally, Section V describes the results from our experiments and Section VI concludes the paper.

II. SPEC CPU2017

Standardized Performance Evaluation Corporation (SPEC) is one of the most successful efforts in standardizing benchmark suites. SPEC CPU benchmarks are a great tool to stress the processor, memory, and compiler. The most recent incarnation of the SPEC CPU, CPU2017, has been in development for many years and is expected to be a cornerstone in performance evaluation of desktop and server computers in the years to come. SPEC CPU2017 consists of benchmarks source codes, benchmark inputs, configuration scripts that govern benchmark compilation process, and running scripts that are designed to streamline performance studies. SPEC CPU2017 contains 43 benchmarks, organized into four groups, namely: (a) SPECspeed2017 floating-point (*fp_speed*), (b) SPECspeed2017 integer (*int_speed*), (c) SPECrate2017 floating-point (*fp_rate*), and (d) SPECrate2017 integer (*int_rate*). The benchmarks written in C, C++, and Fortran programming languages are derived from a wide variety of application domains. The *fp_speed* and *fp_rate* suites contain benchmarks that perform computations on floating-point data types (Table I), whereas the *int_speed* and *int_rate* suites contain benchmarks that perform computation on predominantly integer data types (Table II).

TABLE I. SPEC CPU2017 FLOATING-POINT BENCHMARKS

SPECrate 2017 Floating Point	SPECspeed 2017 Floating Point	Language	Application Area
503.bwaves_r	603.bwaves_s	Fortran	Explosion modeling
507.cactuBSSN_r	607.cactuBSSN_s	C++, C, Fortran	Physics: relativity
508.namd_r	-	C++	Molecular dynamics
510.parest_r	-	C++	Biomedical imaging
511.povray_r	-	C++, C	Ray tracing
519.lbm_r	619.lbm_s	C	Fluid dynamics
521.wrf_r	621.wrf_s	Fortran, C	Weather forecasting
526.blender_r	-	C++, C	3D rendering and animation
527.cam4_r	627.cam4_s	Fortran, C	Atmosphere modeling
-	628.pop2_s	Fortran, C	Wide-scale ocean modeling
538.imagick_r	638.imagick_s	C	Image manipulation
544.nab_r	644.nab_s	C	Molecular dynamics
549.fotonik3d_r	649.fotonik3d_s	Fortran	Computational Electromagnetics
554.roms_r	654.roms_s	Fortran	Regional ocean modeling

A single copy of a *speed* benchmark (name ending with a suffix "_s"), *SBi*, is run on a test machine using the reference input set; the *SPECspeed (SBi)* metric reported by the running script is calculated as the ratio of the single-thread benchmark execution time on the reference machine and the benchmark execution time on the test machine with N_T threads, as shown in Eq. 1.

$$SPECspeed(SBi, N_T) = T(Ref, 1)/T(Test, N_T) \quad (1)$$

A composite single number is also reported for an entire suite; it is calculated as the geometric mean of the individual *SPECspeed* ratios of all benchmarks in that suite. When running speed benchmarks, a performance analyst has an option to specify the number of OpenMP threads, N_T, as many of benchmarks support multi-threaded execuion.

Multiple copies (N_C) of a rate benchmark (name ending with a suffix "_r"), *RBi*, are typically run on a test machine, and the *SPECrate* (*RBi*, N_C) metric is defined as the ratio of the execution time of a single-copy of the benchmark on the reference machine and the execution time when N_C copies are run concurrently on the test machine, multiplied by the number of copies, as shown in Eq. 2

$$SPECrate(RBi, N_C) = N * T(Ref, 1)/T(Test, N_C) \quad (2)$$

TABLE II. SPEC CPU 2017 INTEGER BENCHMARKS

SPECrate 2017 Integer	SPECspeed 2017 Integer	Language	Application Area
500.perlbench_r	600.perlbench_s	C	Perl interpreter
502.gcc_r	602.gcc_s	C	GNU C compiler
505.mcf_r	605.mcf_s	C	Route planning
520.omnetpp_r	620.omnetpp_s	C++	Discrete Event simulation: computer network
523.xalancbmk_r	623.xalancbmk_s	C++	XML to HTML conversion via XSLT
525.x264_r	625.x264_s	C	Video compression
531.deepsjeng_r	631.deepsjeng_s	C++	AI: alpha-beta tree search (Chess)
541.leela_r	641.leela_s	C++	AI: Monte Carlo tree search (Go)
548.exchange2_r	648.exchange2_s	Fortran	AI: recursive solution generator (Sudoku)
557.xz_r	657.xz_s	C	General data compression

III. COMPILERS

This section introduces the compilers used in the study. The choice of compilers is made with respect to usage, availability, and known performance.

A. Intel Parallel Studio XE-19 (IPS)

Intel Parallel Studio XE is a software development suite developed by the Intel Corporation to facilitate code development in C, C++, and Fortran for parallel computing [15]. The Intel C++ compiler, also known as *icc* is a group of C and C++ compilers and *ifort* is an Intel Fortran compiler. It supports the latest standards in C++ as well as OpenMP. The Intel compilers are available for free of charge for students and open source developers.

IPS is a hardware aware compiler and it is optimized for x86 instruction set architecture (ISA). However, the performance on other supported architectures can be modest. IPS supports three high level optimization techniques – inter-procedural optimization (IPO), profile-guided optimization (PGO), and high-level optimization (HLO). At –O3 optimization level, IPS enables auto-parallelization and vectorization to improve the performance of the serial code when possible and also enables more aggressive optimizations such as cache blocking, loop transformations, prefetching, and others [16]. IPS uses optimized math libraries to better use the underlying hardware.

B. The LLVM Compiler Infrastructure Project

The LLVM Compiler Infrastructure Project is a "collection of modular and reusable compiler and toolchain technologies" [17]. The main goal of the project is to build a set of compiler components which can be used across different compilers to reduce the time and cost to build a new compiler. 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. *Flang* is a Fortran front-end compiler designed to use LLVM as its backend and is fully interoperable with *Clang* C++ [18]. Several companies are using LLVM in their customized software development tools, including AMD [19], NVIDIA [20], Apple [21], and Sony.

C. GNU Compiler Collection

The GNU compiler collection is an open source compiler from the Free Software Foundation Inc. which was originally written to be a compiler for the GNU operating system. 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 compliers are monolithic, where each language has a specific program to read the source code and produce machine code. Whereas, IPS uses heavy optimizations by default, high-level optimizations in GNU are limited [22]. GNU compiler is designed to produce portable code rather than optimized code.

IV. TEST ENVIRONMENT & TOOLS

The studies are performed on a test system with Intel's 8th generation Core i7-8700K processor [23]. It is based on the Intel's Coffee Lake microarchitecture and is manufactured using Intel's 14nm++ technology node. The processor is a homogeneous hexa-core with two-way hyperthreading. Each processor core includes a private 8-way set-associative 32 KiB level 1 caches for instructions (L1I) and data (L1D) and a 4-way 256 KiB unified level 2 cache (L2). The last level cache (LLC) of 12 MiB is shared among all processor cores and is built as a 16-way set-associative structure. The processor's nominal clock frequency is 3.70 GHz; however, a single core turbo boost frequency can reach up to 4.70 GHz. To have uniform test conditions the processor core frequency was set to 4.30 GHz for all evaluations with no observed power throttling in any runs. The frequency governor is set to 'performance' mode.

The test system has 32 GiB of DDR4 2400MHz RAM memory. The integrated memory controller is configured as dual-channel with a maximum bandwidth of 39 GiB/s. The workstation runs Ubuntu 18.04.1 LTS with Linux kernel 4.14.0. The metrics of interests in this study are derived from SPEC utilities that report build times, execution times and SPEC

CPU2017 composite performance metrics. Hyper-threading is disabled to enhance performance monitoring event capturing accuracy.

A. *runcpu*

The configuration files for all three compilers are similar to the examples provided by SPEC. Optimization level is set to –O3 for all benchmarks and no aggressive optimizations are used. Repeatable runs using the *runcpu* are generated. The benchmarks with multiple inputs are combined together to report a single number. We use base metrics instead of a peak metric since the former can be used to tailor the optimizations for each benchmark separately.

B. *Intel VTune Amplifier*

The *Intel VTune Amplifier* is a performance analysis tool that relies on the underlying hardware counters to get run-time parameters of the application under test on 32 and 64-bit x86 machines [24]. The profiler gives detailed information on time spent in each function, time spent by the hardware in actual execution and stalls etc. This information can be used to locate or determine aspects of the code and system, such as hot-spots in the application (the most time-consuming functions); hardware-related issues in code such as data sharing, cache misses, branch misprediction, and others; and thread activity and transitions such as migrations and context-switches.

V. RESULTS

This section shows the results of experiments, specifically build time, executable code size, and SPEC speed and rate performance metrics. Note that we were unable to successfully compile and run all benchmarks across all compilers. The failed benchmarks are represented by "-" in the tables. The discussion involving an entire benchmark suite contains only the benchmarks that have results across all the compilers. Any benchmark that does not have results across all compilers is omitted from the summary view of the suite.

A. *Executable Sizes*

Table III shows the an individual benchmark's size in terms of kilo lines of code (KLOC), and the code size generated by each of the compilers. 3

fp_speed: In the case of the *fp_speed* benchmarks, the size of the code ranges from 1 KLOC to 991 KLOC. The size of benchmark executables varies widely for different compilers. The LLVM compilers consistently create the smallest executables for all the benchmarks. The GNU compilers produce the largest executables s for *627.cam4_s* and *638.imagick_s*. The IPS compilers creates the largest executables for *603.bwaves*, *619.lbm_s*, *649.fotonik3d_s*, and *654.roms_s*. Overall the IPS executables are ~1.35x larger than the LLVM executables, whereas the GNU executables are ~2.7x larger than the LLVM executables. GNU being a cross-platform compiler focuses on portability, hence the code generated by GNU is significantly larger.

int_speed: In the case of *int_speed* benchmarks, the size of the source code ranges from 1 KLOC to 1304 KLOC. Similar to floating-point benchmarks, the LLVM compilers create the executables of the smallest size. The GNU compilers create executables of the largest size, except for *605.mcf_s* and *648.exchange2_s*. Overall, IPS produces executables ~1.35x larger than that of LLVM and GNU produces code ~7x larger than that of LLVM.

fp_rate: In the case of *fp_rate* benchmarks, the code size ranges from 1 KLOC to 1577 KLOC. The size of the source code remains the same for benchmarks that have both *speed* and *rate* variants. The decreases in the size of executables relative to their speed counterparts come from compilation switches – the rate benchmarks are not multithreaded. IPS produced executables are ~1.11x smaller than that of the LLVM produced executable and GNU produced executable is ~5.6x smaller than that of LLVM executables.

TABLE III. EXECUTABLE SIZE (LOWER IS BETTER)

Benchmarks	***KLOC***	***Executable Size [KB]***		
		ips	***llvm***	***gnu***
fp_speed				
603.bwaves_s	1	1,060	96	184
619.lbm_s	1	136	27	64
621.wrf_s	991	29,966	-	59,503
627.cam4_s	407	12,035	11,953	31,775
628.pop2_s	338	153,447	-	166,408
638.imagick_s	259	3,875	2,329	9,974
644.nab_s	24	-	260	1,224
649.fotonik3d_s	14	2,329	770	827
654.roms_s	210	3,071	1,517	2,298
int_speed				
600.perlbench_s	362	3,951	2,469	10,918
602.gcc_s	1304	15,529	10,900	60,514
605.mcf_s	3	141	48	134
620.omnetpp_s	134	3,732	2,679	29,039
623.xalancbmk_s	520	8,499	7,128	72,053
625.x264_s	96	1,284	667	3,446
631.deepsjeng_s	10	244	112	528
641.leela_s	21	278	222	3,870
648.exchange2_s	1	1,059	231	158
657.xz_s	33	348	228	1,160
fp_rate				
503.bwaves_r	1	993	72	103
508.namd_r	8	1,269	902	4,769
510.parest_r	427	8,220	13,723	121,743
511.povray_r	170	2,511	1,299	7,469
519.lbm_r	1	146	26	59
521.wrf_r	991	31,116	-	58,456
526.blender_r	1,577	26,172	18,766	104,064
527.cam4_r	407	10,378	11,876	31,168
538.imagick_r	259	3,608	2,195	9,792
544.nab_r	24	-	247	1,186
549.fotonik3d_r	14	2,243	-	797
554.roms_r	210	3,082	1,489	2,274
int_rate				
500.perlbench_r	362	3,951	2,469	10,918
502.gcc_r	1304	15,492	10,900	60,414
505.mcf_r	3	140	48	133
520.omnetpp_r	134	3,732	2,697	29,039
523.xalancbmk_r	520	8,499	7,128	72,052
525.x264_r	96	1,283	667	3,446
531.deepsjeng_r	10	244	112	528
541.leela_r	21	278	222	3,870
548.exchange2_r	1	1,055	235	158
557.xz_r	33	339	223	1,144

int_rate: In the case of *int_rate* benchmarks, the code size ranges from 1 KLOC to 1304 KLOC. The code size is the same for each of the *int_rate* benchmarks as their *speed* counterparts.

B. Build Times

SPEC CPU2017 configuration files that govern the process of compiling benchmarks (compiler and libraries used, optimization switches, and others) allow us to specify the number of processor cores that can be utilized during compilation. Thus, we consider build times for all the benchmarks when using one processor and when using six processor cores. The build times are shown in Table IV.

TABLE IV. BUILD TIMES (LOWER IS BETTER)

Benchmarks	***Build Time (1-CPU) [s]***			***Build Time (6-CPU) [s]***		
	ips	***llvm***	***gnu***	***ips***	***llvm***	***gnu***
fp_rate						
603.bwaves_s	11	23	3	9	7	1
619.lbm_s	7	5	2	6	4	1
621.wrf_s	1,972	-	897	1795	-	436
627.cam4_s	266	1,671	148	244	392	46
628.pop2_s	118	-	124	102	-	49
638.imagick_s	82	606	54	75	165	17
644.nab_s	-	51	6	-	12	2
649.fotonik3d_s	25	160	8	23	101	5
654.roms_s	95	395	30	83	94	7
int_speed						
600.perlbench_s	55	575	49	48	133	13
602.gcc_s	241	2,957	222	202	560	48
605.mcf_s	2	11	1	2	5	1
620.omnetpp_s	97	834	99	57	143	20
623.xalancbmk_s	175	2,116	250	86	379	48
625.x264_s	39	346	29	35	82	9
631.deepsjeng_s	4	26	4	4	8	2
641.leela_s	8	96	13	5	21	4
648.exchange2_s	8	136	4	7	135	4
657.xz_s	6	54	6	4	11	2
fp_rate						
503.bwaves_r	3	12	2	3	5	1
508.namd_r	55	256	23	53	75	7
510.parest_r	228	4,405	397	92	828	79
511.povray_r	41	307	29	33	59	8
519.lbm_r	6	4	1	7	4	1
521.wrf_r	1,997	-	895	1772	-	434
526.blender_r	242	3,493	258	174	889	57
527.cam4_r	124	1,699	142	104	392	46
538.imagick_r	77	583	53	71	162	17
544.nab_r	-	50	5	-	12	2
549.fotonik3d_r	24	-	9	23	-	6
554.roms_r	100	394	30	89	95	6
int_rate						
500.perlbench_r	48	575	50	41	133	13
502.gcc_r	221	2,962	220	184	562	47
505.mcf_r	2	11	2	2	4	1
520.omnetpp_r	98	832	97	57	144	20
523.xalancbmk_r	176	2,121	251	86	379	48
525.x264_r	37	346	29	34	82	10
531.deepsjeng_r	4	26	4	3	8	2
541.leela_r	8	96	15	5	21	4
548.exchange2_r	7	135	4	7	136	4
557.xz_r	7	53	6	4	11	2

fp_speed: Build time for the *fp_speed* benchmarks show that GNU build times are consistently lower than that of LLVM and IPS. LLVM build times are excessively long. Thus, IPS is ~5.88x faster than LLVM and GNU is ~11.67x faster than LLVM for single-core builds (1-CPU). When 6-CPUs are used to build the benchmarks, we see that LLVM build times reduce significantly to match closely the build times of IPS and GNU. While using 6-CPUs a significant improvement in build time is observed for both GNU (~3.18) and LLVM (~3.74), whereas IPS sees a marginal improvement (~1.10). IPS is still ~1.73x faster than LLVM and GNU is ~9.91x faster than LLVM.

int_speed: Build times for the *int_speed* benchmarks that for GNU and IPS are comparable when using a single processor core (1-CPU). LLVM consistently takes longer to generate executables for all the benchmarks. With 1-CPU builds, IPS is ~11.26x faster than LLVM and GNU is ~10.56x faster than LLVM. When 6-CPUs are used to build the benchmarks, we see that LLVM build times reduce significantly to match closely with IPS and GNU. While using 6-CPUs a significant improvement in build time is observed for both GNU (~4.48) and LLVM (~4.84), whereas IPS sees a marginal improvement (~1.41). IPS is still ~3.28x faster than LLVM and GNU is about ~9.78x faster than LLVM.

fp_rate: Build times for 1-CPU builds with LLVM is ~12.73x and 11.92x slower than IPS and GNU respectively. While using 6-CPUs a compiler benefits for parallelization which reflects in improvement in build times for both GNU (~4.44) and LLVM (~4.21), whereas IPS sees a marginal improvement (~1.39). IPS is still ~4x faster than LLVM and GNU is ~11.30x faster than LLVM.

int_rate: When we look at the int_rate benchmarks we see no noticeable difference between the *int_speed* and *int_rate* for all compilers. Unlike for the *fp_speed* benchmarks which use OpenMP, the *int_speed* suites do not use OpenMP making them the same as *int_rate* benchmarks.

In summary, though LLVM has smaller executable size, it has significantly longer build times in comparison with GNU and IPS. This is especially true for floating-point benchmarks. Though the GNU compilers produce executables that are significantly larger in size, the build times are shorter than the build times of LLVM. IPS on the other hand produces executables as small as LLVM and it does that in build times that are comparable to the GNU build times. The number of CPUs used in building the benchmarks play a significant role in build times for GNU and LLVM, however the IPS does not appear to benefit much when using multiple cores in the building process.

C. Performance

Fig. 1, Fig. 2, Fig. 3, and Fig. 4 show *SPECspeed* metrics defined in Eq. 1 for the *speed* benchmarks and in Eq. 2 for the *rate* benchmarks run with 1, 2, 4, and 6 threads/copies, respectively.

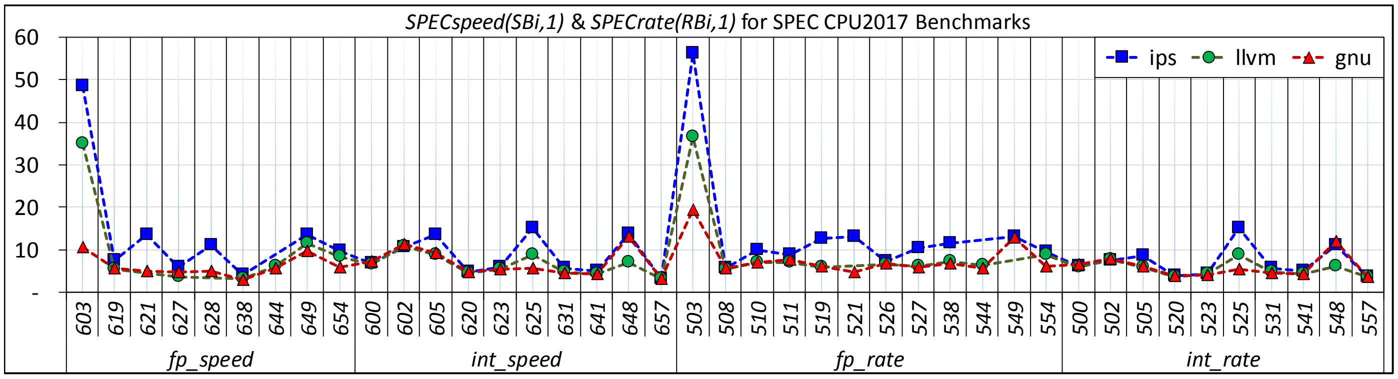

FIG. 1. SPEC RATIO FOR SINGLE THREAD/COPY (HIGHER IS BETTER)

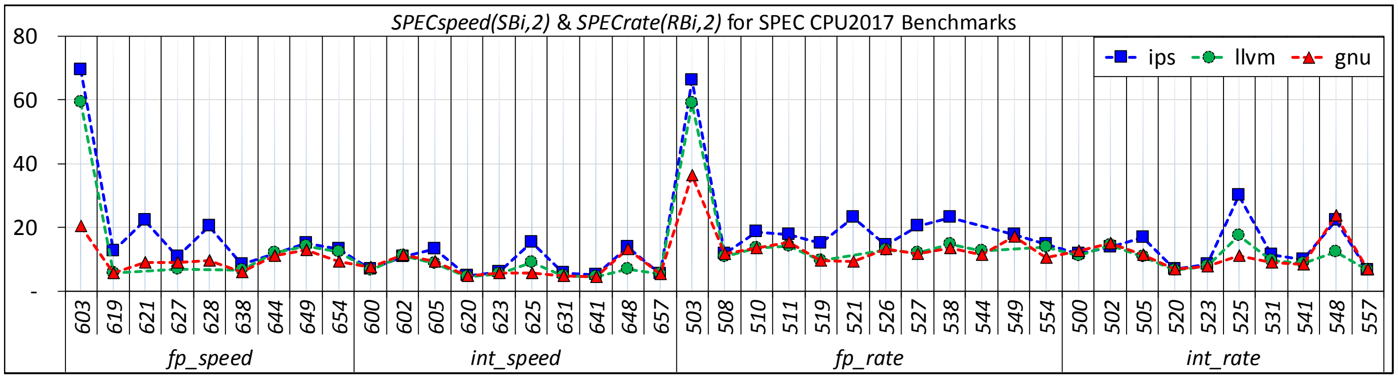

FIG. 2. SPEC RATIO FOR 2-THREADS/COPIES (HIGHER IS BETTER)

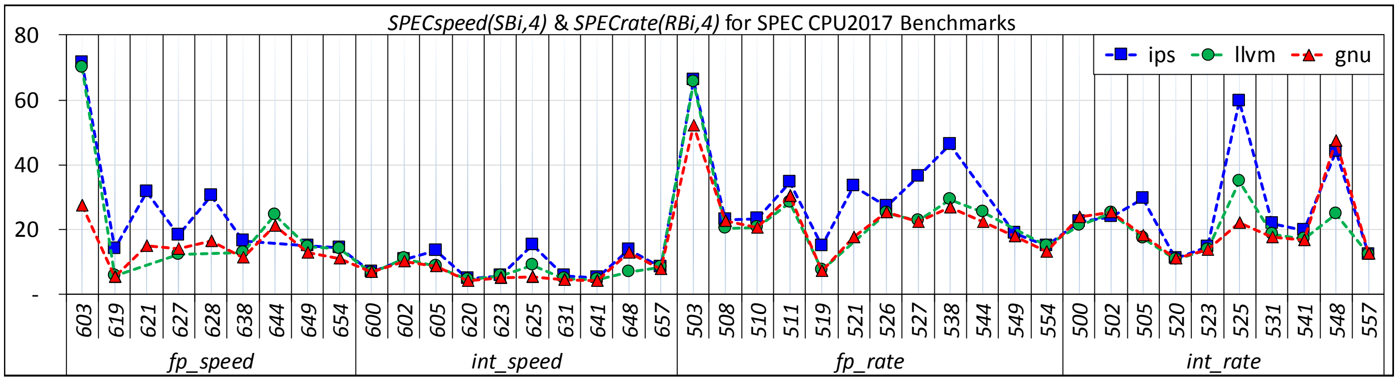

FIG. 3. SPEC RATIO FOR 4-THREADS/COPIES (HIGHER IS BETTER)

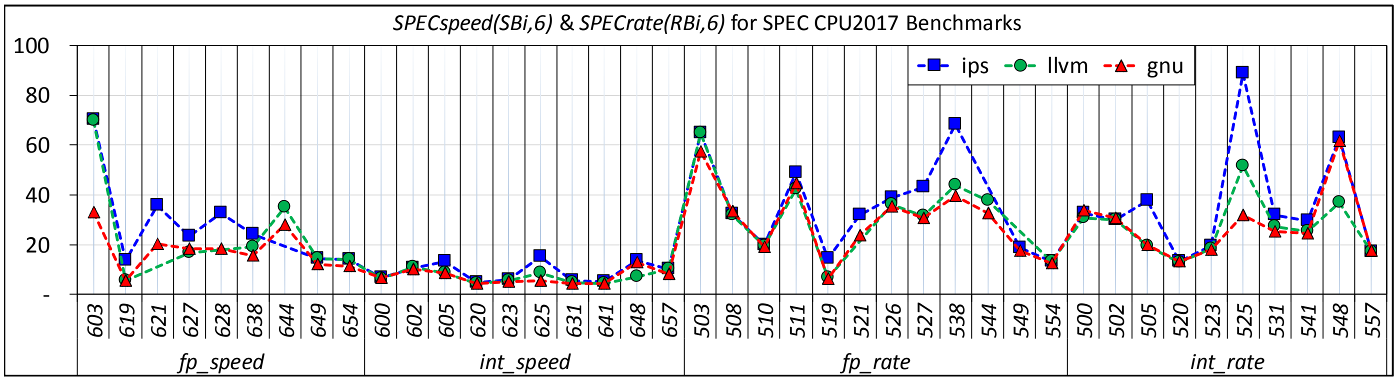

FIG. 4. SPEC RATIO FOR 6-THREADS/COPIES (HIGHER IS BETTER)

fp_speed: Considering single threaded *fp_speed* benchmarks, executables created by IPS outperform the executables created by LLVM and GNU for all the benchmarks. IPS outperforms LLVM by ~32% and GNU by ~70%. Similar performance trends are observed when the threads increases. When N_T=6, IPS still has gains of ~29% over LLVM and ~57% over GNU.

The reason for IPS to perform better over LLVM and GNU is its vectorization capability. By analyzing benchmark runs using Intel's VTune Amplifier tool, we find that IPS executables efficiently utilize vector extensions when performing single precision and double precision operations. In the interest of space, we have not included the results obtained from Intel VTune in this paper.

int_speed: Looking at the *int_speed* benchmark suites the results are more consistent across compilers with many benchmarks having similar execution times. The exceptions are *605.mcf_s* and *625.x264_s,* wherein, IPS outperforms LLVM and GNU. The reason for these exceptions is because of the small number of load instructions, store instructions, and L1 cache misses in *605.mcf_s* and vectorization in *625.x264_s*. The performance of LLVM is less comparable to GNU in most of the cases with the exception of *625.x264_s* where GNU is far better. For this benchmark, number of executed instructions, number of loads and stores are almost double compared to GNU.

Overall, IPS executables perform better than LLVM and GNU ones. The LLVM executables are the slowest for *int_speed* benchmarks. Only one of the integer benchmarks, *657.xz_s*, is parallelizable and its performance scalability is comparable across threads and compilers.

fp_rate: In the case of the *fp_rate* benchmarks, when single copy runs are executed, we observe that the benchmark behavior is similar to the *speed* counterparts. IPS outperforms LLVM by ~40% and GNU by ~54%. When executing 6 copies IPS has a performance improvement of ~22% over LLVM and ~26% over GNU.

int_rate: The performance of the int_rate benchmarks are comparable to their *int_speed* counterparts with *557.xz_r* being an exception as it has a smaller workload in the *rate* suite. IPS outperforms LLVM by ~21% and GNU by ~17%. With 6-copy execution we see that the performance ratios remain stable, where IPS outperforms LLVM by ~22% and GNU by ~17%.

In summary IPS executables outperform those created by LLVM and GNU for all benchmarks. The performance of LLVM and GNU are comparable with LLVM doing better for floating-point benchmarks and GNU showing slightly better performance for the integer benchmarks.

D. Benchmark Characteristcs

To understand the characteristics of the benchmarks, we run the executables produced by the compilers on Intel VTune which gives the comprehensive pipeline view of the benchmarks. Table V shows the main characteristics of the benchmarks such as the instructions per cycle (IPC) and the dynamic instructions executed (IC); the instruction count is given in billions.

fp_speed: The dynamic instruction-count executed by IPS ranges from ~3.3 trillion (*649.fotonik3d_s*) to ~29 trillion (*638.imagick_s*) with an average IPC of 1.64. Whereas LLVM instruction count ranges from ~3.8 trillion (*638.imagick_s*) to ~61.9 trillion (*638.imagick_s)* with an average IPC of 2.23, GNU instruction count is high, and it ranges from ~4.2 trillion (*619.lbm_s*) to ~73.7 trillion (*638.imagick_s)* with an average IPC of 2.53.

TABLE V. BENCHMARK CHARACTERISTICS

Benchmarks	***IPC***			***IC [Billion]***		
	ips	***llvm***	***gnu***	***ips***	***llvm***	***gnu***
fp_speed						
603.bwaves_s	1.69	2.60	2.68	8,792.4	18,832.6	64,906.9
619.lbm_s	1.26	1.12	1.09	3,752.4	4,509.8	4,234.3
621.wrf_s	1.84	-	1.58	7,742.1	-	18,833.7
627.cam4_s	1.90	1.92	1.86	12,384.7	19,813.1	15,003.1
628.pop2_s	1.74	-	2.14	7,891.7	-	22,263.2
638.imagick_s	2.00	3.29	3.39	29,684.8	61,922.1	73,748.0
644.nab_s	-	1.75	1.48	-	21,308.4	20,207.6
649.fotonik3d_s	1.16	1.14	1.51	3,308.2	3,864.6	6,222.6
654.roms_s	0.92	0.85	1.93	6,305.8	6,898.9	22,470.8
int_speed						
600.perlbench_s	2.38	2.54	2.48	2,699.2	3,003.0	2,664.7
602.gcc_s	1.52	1.55	1.45	2,452.0	2,425.3	2,244.1
605.mcf_s	0.80	0.86	0.78	1,204.9	1,997.9	1,745.7
620.omnetpp_s	0.68	0.71	0.72	1,001.9	1,068.5	1,066.6
623.xalancbmk_s	0.93	0.97	1.13	945.7	1,053.2	1,285.6
625.x264_s	2.43	2.72	3.17	1,215.1	2,338.4	4,420.7
631.deepsjeng_s	1.60	1.66	1.65	1,754.9	2,121.0	2,239.4
641.leela_s	1.23	1.25	1.21	1,776.5	2,078.8	2,084.8
648.exchange2_s	2.24	2.51	2.47	2,047.7	4,555.0	2,430.9
657.xz_s	0.92	0.99	1.01	7,754.0	8,371.0	8,536.7
fp_rate						
503.bwaves_r	1.62	2.35	1.99	1,242.1	2,791.9	4,492.5
508.namd_r	2.67	2.64	1.37	1,911.4	2,007.1	1,114.5
510.parest_r	2.08	1.60	2.71	2,328.3	2,555.1	1,961.4
511.povray_r	2.32	2.42	2.02	2,609.8	3,398.8	3,344.0
519.lbm_r	1.66	1.92	2.33	583.0	1,453.8	3,170.8
521.wrf_r	1.82	-	1.86	1,346.7	-	1,390.7
526.blender_r	2.13	1.73	1.57	1,934.8	1,700.4	3,285.0
527.cam4_r	2.11	1.86	1.72	1,513.2	2,292.9	1,717.4
538.imagick_r	2.53	2.67	1.87	2,343.9	3,887.0	2,464.6
544.nab_r	-	1.81	2.73	-	2,075.0	4,443.1
549.fotonik3d_r	1.13	-	1.52	1,427.1	-	2,006.2
554.roms_r	1.11	1.04	1.45	789.8	798.9	1,911.5
int_rate						
500.perlbench_r	2.42	2.54	2.23	2,699.2	3,003.0	2,580.4
502.gcc_r	1.37	1.43	1.36	1,135.4	1,124.7	1,066.0
505.mcf_r	0.84	0.94	0.84	679.8	1,101.0	972.0
520.omnetpp_r	0.68	0.72	0.71	1,002.0	1,102.1	1,082.6
523.xalancbmk_r	0.93	0.98	1.14	945.7	1,052.9	1,285.8
525.x264_r	2.43	2.71	3.14	1,215.1	2,338.4	4,419.6
531.deepsjeng_r	1.72	1.77	1.74	1,503.1	1,819.5	1,921.8
541.leela_r	1.23	1.25	1.20	1,776.5	2,078.8	2,084.7
548.exchange2_r	2.25	2.51	2.50	2,286.1	4,556.0	2,430.8
557.xz_r	1.39	1.51	1.55	1,804.1	1,969.1	2,027.4

int_speed: The dynamic instruction-count executed by IPS ranges from ~945 billion (*623.xalancbmk_s*) to ~7.7 trillion (*657.xz_s*) with an average IPC of 1.19. Whereas LLVM instruction count ranges from ~1.05 trillion (*623.xalancbmk_s*) to ~8.37 trillion (*657.xz_s*) with an average IPC of 1.33, GNU instruction count is high, and it ranges from ~1.06 trillion

(*620.omnetpp_s*) to ~8.53 trillion (*657.xz_s*) with an average IPC of 1.34.

fp_rate: The dynamic instruction-count executed by IPS ranges from ~582 billion (*519.lbm_r*) to ~2.6 trillion (*511.povray_r*) with an average IPC of 2.08. Whereas LLVM instruction count ranges from ~798 billion (*554.roms_s*) to ~3.8 trillion (*538.imagick_s)* with an average IPC of 2.06, GNU instruction count is high, and it ranges from ~1.1 trillion (*508.namd_r*) to ~4.4 trillion (*503.bwaves_r)* with an average IPC of 1.98. We observe that, the aggregate IC for *fp_rate* is ~6x (IPS), ~8x (LLVM), and ~11x (GNU) smaller than *fp_speed*.

int_rate: As observed earlier, the *int_rate* benchmarks are comparable to *int_speed* benchmark with minor changes in work loads. Thus, the aggregate IC for int_rate is ~1.5x smaller than that of *int_speed*.

VI. Conclusion

The complexity of underlying hardware has grown exponentially to meet software development trends. As software abstraction grows along with hardware complexity the burden of optimization falls more on the compilers to bridge the gap between hardware and software. Compilers are expected to use the existing hardware features to get the best possible performance. In this measurement-based evaluation, we look at the latest stable versions of three widely used compilers (IPS, LLVM, and GNU) using build time, code size, and SPEC Ratio as metrics on the Core i7-8700K. We use the SPEC CPU2017 benchmark suite which represents the current application trends.

Considering code size, we find that LLVM is the optimal choice, with build time being the tradeoff. When build time is considered, GNU is preferred with code size being the tradeoff. However, when performance is considered as the metric, IPS outperforms LLVM and GNU due to better exploitation of hardware resources (eg. vectorization, prefetching, cache optimization).

IPS is architecture bound and is optimized for x86. On the other hand, GNU is cross compatible with a focus on portability. This evaluation is limited to an Intel processor leaving the room for further research on how these compilers would perform on processors from other architectures and from different venders. Another field of interest would be to evaluate the effect of their compiler on energy consumption.

References

[1] “SPEC CPU® 2017.” [Online]. Available: https://www.spec.org/cpu2017/. [Accessed: 19-Mar-2018].

[2] R. Hebbar Seethur Raviraj, “Spec CPU2017: Performance, Energy and Event Characterization on Modern Processors,” M.S.E., The University of Alabama in Huntsville, United States -- Alabama, 2018.

[3] R. Panda, S. Song, J. Dean, and L. K. John, “Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 271–282.

[4] A. Limaye and T. Adegbija, “A Workload Characterization of the SPEC CPU2017 Benchmark Suite,” in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018, pp. 149–158.

[5] R. Hebbar and A. Milenković, “SPEC CPU2017: Performance, Event, and Energy Characterization on the Core i7-8700K,” in *Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering*, Mumbai, India, 2019.

[6] R. S. Machado, R. B. Almeida, A. D. Jardim, A. M. Pernas, A. C. Yamin, and G. G. H. Cavalheiro, “Comparing Performance of C Compilers Optimizations on Different Multicore Architectures,” in *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, 2017, pp. 25–30.

[7] D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, R. Zhuykov, and J.-H. Lee, “Automatic Tuning of Compiler Optimizations and Analysis of their Impact,” *Procedia Computer Science*, vol. 18, pp. 1312–1321, 2013.

[8] S. M. F. Rahman *et al.*, “Studying the impact of application-level optimizations on the power consumption of multi-core architectures,” in *Proceedings of the 9th conference on Computing Frontiers - CF ’12*, Cagliari, Italy, 2012, p. 123.

[9] M. Valluri and L. K. John, “Is Compiling for Performance — Compiling for Power?,” in *Interaction between Compilers and Computer Architectures*, G. Lee and P.-C. Yew, Eds. Boston, MA: Springer US, 2001, pp. 101–115.

[10] S. T. Gurumani and A. Milenkovic, “Execution Characteristics of SPEC CPU2000 Benchmarks: Intel C++ vs. Microsoft VC++,” in *Proceedings of the 42Nd Annual Southeast Regional Conference*, New York, NY, USA, 2004, pp. 261–266.

[11] T. K. Prakash and L. Peng, “Performance Characterization of SPEC CPU2006 Benchmarks on Intel Core 2 Duo Processor,” *ISAST Transactions on Computer Software Engineering*, p. 6, 2008.

[12] S. Aldea, D. R. Llanos, and A. González-Escribano, “Using SPEC CPU2006 to evaluate the sequential and parallel code generated by commercial and open-source compilers,” *The Journal of Supercomputing*, vol. 59, no. 1, pp. 486–498, Jan. 2012.

[13] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, San Jose, CA, USA, 2004, pp. 75–86.

[14] C. Lattner, “Introduction to the LLVM Compiler System,” 04-Nov-2008.

[15] admin, “Home | Intel® Parallel Studio XE,” 07:00:00 UTC. [Online]. Available: https://software.intel.com/en-us/parallel-studio-xe. [Accessed: 03-Feb-2019].

[16] “Intel® C++ Compiler 19.0 Developer Guide and Reference.” [Online]. Available: https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference. [Accessed: 15-Feb-2019].

[17] “The LLVM Compiler Infrastructure Project.” [Online]. Available: https://llvm.org/. [Accessed: 03-Feb-2019].

[18] *Flang is a Fortran language front-end designed for integration with LLVM.: flang-compiler/flang*. flang-compiler, 2019.

[19] “AMD Optimizing C/C++ Compiler,” *AMD*. [Online]. Available: https://developer.amd.com/amd-aocc/. [Accessed: 14-Feb-2019].

[20] “CUDA LLVM Compiler,” *NVIDIA Developer*, 07-May-2012. [Online]. Available: https://developer.nvidia.com/cuda-llvm-compiler. [Accessed: 14-Feb-2019].

[21] “LLVM Compiler Overview.” [Online]. Available: https://developer.apple.com/library/archive/documentation/CompilerTools/Conceptual/LLVMCompilerOverview/index.html. [Accessed: 13-Feb-2019].

[22] “Using the GNU Compiler Collection (GCC): Optimize Options.” [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. [Accessed: 14-Feb-2019].

[23] “Intel® Core^{TM} i7-8700K Processor Product Specifications,” *Intel® ARK (Product Specs)*. [Online]. Available: https://tinyurl.com/ybcw5vc8. [Accessed: 24-Mar-2018].

[24] “Intel® VTune^{TM} Amplifier 2018 User’s Guide,” *Intel Developer Zone*. [Online]. Available: https://software.intel.com/en-us/vtune-amplifier-help-introduction. [Accessed: 28-Mar-2018].