SPEC CPU2017: Performance, Event, and Energy Characterization on the Core i7-8700K

Ranjan Hebbar S R
Electrical and Computer Engineering,
The University of Alabama in Huntsville;
rr0062@uah.edu

Aleksandar Milenković
Electrical and Computer Engineering,
The University of Alabama in Huntsville;
milenka@uah.edu

ABSTRACT

Computer engineers in academia and industry rely on a standardized set of benchmarks to quantitatively evaluate the performance of computer systems and research prototypes. SPEC CPU2017 is the most recent incarnation of standard benchmarks designed to stress a system's processor, memory subsystem, and compiler. This paper describes the results of measurement-based studies focusing on characterization, performance, and energy-efficiency analyses of SPEC CPU2017 on the Intel's Core i7-8700K. Intel and GNU compilers are used to create executable files utilized in performance studies. The results show that executables produced by the Intel compilers are superior to those produced by GNU compilers. We characterize all the benchmarks, perform a top-down microarchitectural analysis to identify performance bottlenecks, and test benchmark scalability with respect to performance and energy. Findings from these studies can be used to guide future performance evaluations and computer architecture research.

CCS CONCEPTS

• **General and reference → Measurement; Evaluation; Performance;** *Metrics*; • **Computer systems organization → Multicore architectures** • **Hardware** → *Energy metering*

KEYWORDS

Benchmarks, Energy-efficiency, Microarchitectural analysis.

ACM Reference format:

R. Hebbar and A. Milenković. 2019. SPEC CPU-2017: Performance, Event and Energy Characterization on the Core i7-8700K. In *Proceedings of 10th ACM/SPEC International Conference on Performance Engineering*, Mumbai, India, April 2019, 8 pages. DOI: http://dx.doi.org/10.1145/3297663.3310314

1 INTRODUCTION

Computing has been constantly evolving as technology, applications, and markets continue to change and advance. The demise of Moore's and Dennard's Laws that for long described the semiconductor scaling is accompanied by perhaps even more dramatic changes in markets and applications. Mobile, IoT, and cloud computing promise to be major drivers of innovations in years to come. Still, processors that power contemporary laptop, desktop, and server computers remain one of the most important components in computing ecosystems. Understanding their performance and limitations is important for application developers, system analysts, and computer designers alike.

Benchmarking is the most widely used technique for measuring and comparing performance across different architectures [13]. We rely on it to evaluate current systems for bottlenecks and proposed enhancements in future systems. It is thus of utmost importance to have standardized benchmarks that are representative of real-life applications. Standardized Performance Evaluation Corporation (SPEC) is one of the most successful efforts in standardizing benchmark suites and SPEC CPU benchmarks have been consistently used by industry and academia to evaluate performance of modern processors [13]. Studies conducted on previous generations of SPEC CPU have been useful tools for evaluating the improvements in computer systems. Each new generation of SPEC CPU benchmarks has been more complex with larger input sets, spanning more diverse application domains than its predecessors [8]. Characterization studies on CPU2000 and CPU2006 have shown that they match real-life application trends [4] [5].

SPEC CPU2017 is the most recent incarnation of standard benchmarks designed to stress a system's processor, memory subsystem, and compiler. It includes four benchmark suites organized in floating-point and integer *speed* suites, used for comparing time for a computer to complete a single task, and floating-point and integer *rate* suites used to measure the throughput or work per unit time. Reflecting a shift in computing from single-core to multicores, CPU2017 *speed* suites include a significant number of benchmarks that are multithreaded.

There have been several academic studies of the SPEC CPU2017. Studies from the University of Texas [7] and the University of Arizona [6] focus on reducing the working set by using statistical techniques to identify redundant benchmarks and input sets. Both studies used the GNU's C/C++ and Fortran compilers in their analyses. However, almost all reportable runs on the SPEC website rely on Intel's C/C++ and Fortran compilers. Other studies have explored expanding the workloads of the benchmarks [1].

This paper aims to provide a comprehensive SPEC CPU2017 performance analysis and characterization using a workstation with a recent Intel's i7 8700K processor. The paper encompasses the following aspects of performance evaluation. (a) We perform performance comparison of CPU2017 executables created by Intel and GNU compilers using SPEC derived performance met-

ICPE '19, April 7-11, 2019, Mumbai, India.

ACM ISBN 978-1-4503-6239-9/19/04...$15.00.
DOI: http://dx.doi.org/10.1145/3297663.3310314

rics (Section 3). (b) We perform a general top-view characterization of all the individual benchmarks by counting the instructions, branch instructions, memory reads and writes, as well as misses in branch predictor structures and cache hierarchy (Section 4). (c) We perform an in-depth performance analysis using Intel's Top-down Microarchitectural Analysis Method to identify bottlenecks in Core i7-8700K (Section 5). (d) Finally, we perform analysis of scalability in the context of speed, throughput, and a combined energy delay metric, while varying the number of threads for *speed* benchmarks and the number of copies for *rate* benchmarks (Section 6). Using speedup metrics, defined in Section 2.4, that capture the scalability of individual *speed* and *rate* benchmarks as a function of the number of threads and copies, respectively, the benchmarks are classified into those that "scale very well," "scale moderately," and those that "scale poorly".

The main insights from these studies are as follows. (a) We find that Intel compilers produce CPU2017 executables that are faster than the equivalent GNU executables, on average ~65% for the floating-point suites and ~25% for the integer suites, mainly due to the efficient utilization of advanced vector extensions in the instruction set. (b) The floating-point benchmarks are largely bounded by stalls in memory hierarchy and limited memory bandwidth. The integer benchmarks are bounded by front-end stalls and memory bound stalls. (c) We find that performance of a few floating-point *speed* benchmarks scales very well with an increase in the number of threads, whereas a significant number of floating-point and integer *rate* benchmarks scales very well with an increase in the number copies. Good performance scaling typically results in increased energy-efficiency. Benchmarks that are bounded by limited memory bandwidth scale poorly in performance and energy and should not be run in multithreaded or multi-copy configurations.

2 BACKGROUND, MOTIVATION AND GOALS

2.1 SPEC CPU2017

The SPEC CPU2017 contains 43 benchmarks, organized into four suites as shown in Table 1. The *fp_speed*/*fp_rate* include benchmarks with predominantly floating-point data types designed to stress speed and throughput of modern computers, respectively, whereas *int_speed*/*int_rate* include benchmarks with predominantly integer data types. The benchmarks written in C, C++, and Fortran programming languages are derived from a wide variety of application domains.

A single copy of a *speed* benchmark (name ending with a suffix "_s"), SB_i, is run on a test machine using the reference input set; the *SPECspeed*(SB_i) metric reported by the running script is calculated as the ratio of the benchmark execution times on the reference machine [14] and the test machine, $T(Ref)/T(Test)$. 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 multithreaded execution. Multiple copies (N_C) of a *rate* benchmark (name ending with a suffix "_r"), RB_i, are typically run on a test machine, and the *SPECrate*(RB_i, N_C) metric is defined as the ratio of the execution times of a single-copy on the reference machine and N_C-copy on the test machine, multiplied by the number of copies: $N_C \cdot T(Ref,1)/T(Test,N_C)$.

Table 1: CPU2017 Benchmarks [14]

SPECrate 2017 Floating Point	SPECspeed 2017 Floating Point	SPECrate 2017 Integer	SPECspeed 2017 Integer
503.bwaves_r	603.bwaves_s	500.perlbench_r	600.perlbench_s
507.cactuBSSN_r	607.cactuBSSN_s	502.gcc_r	602.gcc_s
508.namd_r		505.mcf_r	605.mcf_s
510.parest_r			
511.povray_r			
519.lbm_r	619.lbm_s	520.omnetpp_r	620.omnetpp_s
521.wrf_r	621.wrf_s		
526.blender_r		523.xalancbmk_r	623.xalancbmk_s
527.cam4_r	627.cam4_s	525.x264_r	625.x264_s
	628.pop2_s	531.deepsjeng_r	631.deepsjeng_s
538.imagick_r	638.imagick_s		
544.nab_r	644.nab_s	541.leela_r	641.leela_s
549.fotonik3d_r	649.fotonik3d_s	548.exchange2_r	648.exchange2_s
554.roms_r	654.roms_s	557.xz_r	657.xz_s

2.2 System Under Test

The studies are performed on a workstation built around an Intel's 8th generation processor Core i7-8700K. It is based on Coffee Lake architecture and is manufactured using Intel's 14nm++ technology node [15]. The processor includes six 2-way hyperthreaded physical cores for a total of twelve logical processor cores. Each processor core includes separate 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 4.70 GHz. The workstation includes 32 GiB DDR4 2400MHz RAM memory. The integrated memory controller is configured as dual-channel with a maximum bandwidth of 41 GiB/s. The workstation runs Ubuntu 16.04 LTS with Linux kernel 4.4.0. The native frequencies of the processor and memory are not altered, allowing the frequency governor to change frequency as required.

2.3 Tools and Evaluation Methods

The measurements performed in this study rely on SPEC utilities to report execution times and SPEC CPU composite performance metrics. In addition, a set of tools for event-based sampling and profiling is used, including Linux utilities *perf* [16] and *likwid* [9], as well as *Intel VTune Amplifier* [17]. These tools interface and gather information from on-chip performance monitoring units (PMUs) that are a part of modern processors' fabric. *Perf* and *likwid-perfctr* are used to collect important events such as the number of clock cycles, instructions retired, retired branches, and others over complete benchmark runs. *Likwid-powermeter,* a tool that accesses RAPL counters for measuring power and energy, is used in power profiling [11]. Similar to the *runcpu* utility, all the benchmarks are run three times and the results from the median execution time are reported in this paper. We find the differences between runs to be negligible (~2%).

Intel VTune Amplifier can be used to locate or determine aspects of the code and system, such as hot-spots in the application; 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. In this study we use *General Exploration* analysis to understand how efficiently the code passes through the core pipeline. During the *General Exploration* analysis, *Intel VTune Amplifier* collects a complete list of events for analyzing a typical application. It calculates a set of predefined ratios and facilitates identifying hardware-level performance problems. For modern microarchitectures starting with *Ivy Bridge*, *General Exploration* is based on the *Top-down Microarchitecture Analysis Method* (TMAM) [12].

Superscalar processors can be conceptually divided into the front-end and the back-end. The front-end is where instructions are fetched and decoded into micro-operations that constitute them. The back-end is where the required computation is performed. Each clock cycle, each processor core in Core i7-8700K can fill up to five of its pipeline slots with useful micro-operations. Therefore, for any time interval, it is possible to determine the maximum number of pipeline slots that could have been filled in and issued. The TMAM analysis performs this estimate and breaks up all pipeline slots into four categories: (i) Pipeline slots containing useful work that are issued and retired (*Retired*); (ii) Pipeline slots containing useful work that are issued and canceled (*Bad Speculation*); (iii) Pipeline slots that could not be filled with useful work due to problems in the front-end (*Front-End Bound*); and (iv) Pipeline slots that could not be filled with useful work due to structural and data hazards in the back-end (*Back-End Bound*).

2.4 Goals and Metrics

This paper aims to provide a comprehensive evaluation of the SPEC CPU2017 benchmarks when executed on the most recent Intel Core i7-8700K processor. Specifically, we focus on answering the following questions.

1. What is the impact of compilers on performance metrics? The SPEC CPU2017 benchmarks are compiled using the Intel Parallel Studio XE 18.0.1 and GNU compilers 5.5.0 with standard optimization parameters similar to those in the configuration files provided by SPEC (using –O3 optimization level).

2. What are the main characteristics of the SPEC benchmarks? To answer this question, we use the Linux *perf* and *likwid* tools to determine the number of instructions retired, the opcode mix (branch, load, stores) as well as main parameters capturing the behavior of branch predictor structures and cache hierarchy.

3. What are performance bottlenecks? Each benchmark is analyzed using the *Intel's Top-down Microarchitectural Analysis* (TMAM) [12]. Pipeline and clock-cycle views of each benchmark are used to determine their bottlenecks.

4. How do benchmarks' performance scale? As many *speed* benchmarks are multithreaded, a performance scalability study is performed by measuring benchmark execution times while varying the number of threads. To capture scalability of *speed* benchmarks when running with N_T threads, we use a metric called $S(SBi, N_T)$ which is calculated as shown in Eq. (1), where *T(SB*$_i$*, 1)* and *T(SB*$_i$*, N*$_T$*)* are the execution times for the benchmark *SBi* when run with a single and N_T threads, respectively. The speedup metric for an individual SPEC *rate* benchmark, *RBi*, when run with N_C copies, $S(RBi, N_C)$ is calculated as shown in Eq. (2), where *T(RB*$_i$*, 1)* is the execution time when a single copy of the benchmark is run, and *T(RB*$_i$*, N*$_C$*)* is the execution time when N_C copies of the benchmark are run on the test machine.

$$S(SBi\ , N_T) = \quad T(SBi, 1)/T(SBi, N_T) \tag{1}$$

$$S(RBi\ , N_C) = \quad (N_C \cdot T(RBi, 1))/T(RBi, N_C) \tag{2}$$

5. How do benchmarks scale when both performance and energy are considered? A combined metric called *PE* is used to capture both performance and energy efficiency. The *PE* metric for an individual SPEC *speed* benchmark, *SBi*, running with N_T threads, *PE(SBi, N*$_T$*)* is defined as shown in Eq. (3), where *E(SBi, N*$_T$*)* is the processor energy in Joules needed to complete execution of the benchmark *SBi* when running with N_T threads. To evaluate the performance and energy efficiency of a benchmark run with N_T threads relative to the run with a single thread, an improvement metric defined as shown in Eq. (4) is used. A *PE.I* greater than one means that runs with N_T threads are desirable. The *PE* metric for an individual SPEC *rate* benchmark, *RBi*, running with N_C copies, *PE(RBi, N*$_C$) is defined as shown in Eq. (5). To evaluate the performance and energy efficiency of an N_C-copy benchmark run with respect to a single-copy run, an improvement metric as shown in Eq. (6) is used.

$$PE(SBi, N_T\ ) = \ 1/(T(SBi, N_T) \cdot E(SBi, N_T)) \tag{3}$$

$$PE.I(SBi, N_T) = PE(SBi, N_T\ )/PE(SBi, 1) \tag{4}$$

$$PE(RBi, N_C) = \ 1/(T(RBi, N_C) \cdot E(RBi, N_C)) \tag{5}$$

$$PE.I(RBi, N_C) = (N_C{}^2 \cdot PE(RBi, N_C))/PE(RBi, 1) \tag{6}$$

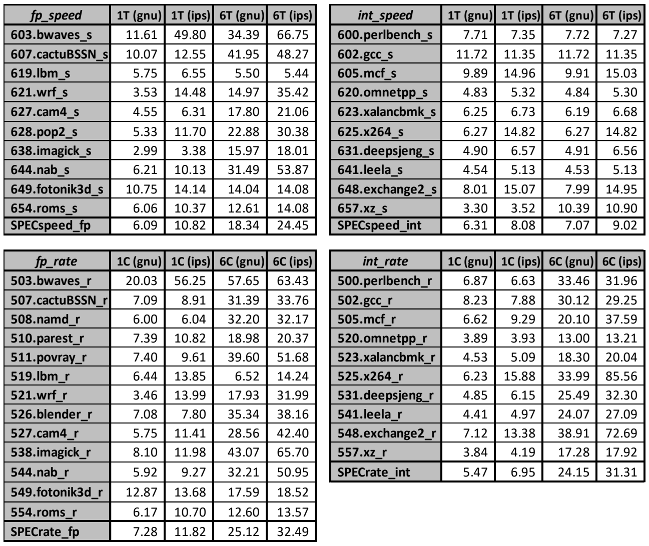
3 COMPILERS COMPARISON

Table 2 shows the *SPECspeed_fp* and *SPECspeed_int* metrics for the *speed* benchmarks and the *SPECrate_fp* and *SPECrate_int* metrics for the *rate* benchmarks compiled by the Intel Parallel Studio (IPS) and GNU compilers and executed on the test machine with the number of threads/copies set to 1 and 6 (N_T, N_C=1, N_T, N_C=6). These are higher is better (HB) metrics, a higher speed metric means that less time is needed to run a benchmark, and a higher rate metric means that more work is done in unit time

The results for *fp_speed* benchmarks (Table 2 top, left) show that the Intel compilers produce executables that run significantly faster than those produced by the GNU compilers. For single-threaded benchmark runs (N_T=1) significant performance improvements are observed in all benchmarks, most notably for *603.bwaves_s* and *621.wrf_s* with over 4 times improvement and for *628.pop2_s* with more than 2 times improvement. When the number of threads is set to 6 (N_T=6), thus matching the number of physical processor cores, performance of the Intel compiled executables still exceeds the performance of the corresponding GNU executables in all the benchmarks, except for *619.lbm_s* and *649.fotonik3d_s*. Looking at the performance improvements when increasing the number of threads from 1 to 6, several benchmarks see significant improvements regardless of the compiler used, e.g. *644.nab_s* (speedup relative to N_T=1 is ~5 times for both compilers sets). The composite *SPECspeed_fp* is 18.34 for the 6-threaded vs. 6.09 for single-threaded GNU executable runs (~3 times improvement), whereas it is 24.45 for the 6-threaded and 10.82 for single-threaded Intel executable runs (~2.25 times improvement). Thus, the composite *SPECspeed_fp* for IPS is over 75% higher than for the GNU compilers for single-threaded runs and over 30% for six-threaded runs.

Regarding the *int_speed* benchmarks, the results show that the Intel compilers provide significant performance improvements for several benchmarks, such as *605.mcf_s*, *625.x264_s*, and *648.exchange2_s*. For other benchmarks, the differences are relatively modest in favor of the IPS executables, except for *600.perlbench_s* and *602.gcc_s* where the GNU executables run slightly faster. As all the integer benchmarks are single-threaded except *657.xz_s*, increasing N_T does not significantly change *SPECspeed_int*. The composite *SPECspeed_int* for single-threaded runs is 6.31 for GNU and 8.08 for IPS (~25% improvement).

Table 2. *SPECspeed{fp,int}* and *SPECrate{fp,int}* for IPS and GNU executables with 1 and 6 threads/copies



fp_speed	1T (gnu)	1T (ips)	6T (gnu)	6T (ips)
603.bwaves_s	11.61	49.80	34.39	66.75
607.cactuBSSN_s	10.07	12.55	41.95	48.27
619.lbm_s	5.75	6.55	5.50	5.44
621.wrf_s	3.53	14.48	14.97	35.42
627.cam4_s	4.55	6.31	17.80	21.06
628.pop2_s	5.33	11.70	22.88	30.38
638.imagick_s	2.99	3.38	15.97	18.01
644.nab_s	6.21	10.13	31.49	53.87
649.fotonik3d_s	10.75	14.14	14.04	14.08
654.roms_s	6.06	10.37	12.61	14.08
SPECspeed_fp	6.09	10.82	18.34	24.45

int_speed	1T (gnu)	1T (ips)	6T (gnu)	6T (ips)
600.perlbench_s	7.71	7.35	7.72	7.27
602.gcc_s	11.72	11.35	11.72	11.35
605.mcf_s	9.89	14.96	9.91	15.03
620.omnetpp_s	4.83	5.32	4.84	5.30
623.xalancbmk_s	6.25	6.73	6.19	6.68
625.x264_s	6.27	14.82	6.27	14.82
631.deepsjeng_s	4.90	6.57	4.91	6.56
641.leela_s	4.54	5.13	4.53	5.13
648.exchange2_s	8.01	15.07	7.99	14.95
657.xz_s	3.30	3.52	10.39	10.90
SPECspeed_int	6.31	8.08	7.07	9.02

fp_rate	1C (gnu)	1C (ips)	6C (gnu)	6C (ips)
503.bwaves_r	20.03	56.25	57.65	63.43
507.cactuBSSN_r	7.09	8.91	31.39	33.76
508.namd_r	6.00	6.04	32.20	32.17
510.parest_r	7.39	10.82	18.98	20.37
511.povray_r	7.40	9.61	39.60	51.68
519.lbm_r	6.44	13.85	6.52	14.24
521.wrf_r	3.46	13.99	17.93	31.99
526.blender_r	7.08	7.80	35.34	38.16
527.cam4_r	5.75	11.41	28.56	42.40
538.imagick_r	8.10	11.98	43.07	65.70
544.nab_r	5.92	9.27	32.21	50.95
549.fotonik3d_r	12.87	13.68	17.59	18.52
554.roms_r	6.17	10.70	12.60	13.57
SPECrate_fp	7.28	11.82	25.12	32.49

int_rate	1C (gnu)	1C (ips)	6C (gnu)	6C (ips)
500.perlbench_r	6.87	6.63	33.46	31.96
502.gcc_r	8.23	7.88	30.12	29.25
505.mcf_r	6.62	9.29	20.10	37.59
520.omnetpp_r	3.89	3.93	13.00	13.21
523.xalancbmk_r	4.53	5.09	18.30	20.04
525.x264_r	6.23	15.88	33.99	85.56
531.deepsjeng_r	4.85	6.15	25.49	32.30
541.leela_r	4.41	4.97	24.07	27.09
548.exchange2_r	7.12	13.38	38.91	72.69
557.xz_r	3.84	4.19	17.28	17.92
SPECrate_int	5.47	6.95	24.15	31.31

Considering single-copy *fp_rate* benchmark runs, the Intel compilers improve performance significantly for several benchmarks such as *503.bwaves_r*, *519.lbm_r*, *521.wrf_r*, and *527.cams_r*. The only benchmark where no significant improvement is observed is *508.namd_s*. The composite *SPECrate_fp* metric shows that the Intel compilers outperform the GNU compilers by 60% in single-copy runs and by 30% for six-copy runs.

Considering single-copy *int_rate* benchmark runs, we find that the Intel executables run significantly faster than the GNU executables for several benchmarks, e.g., *525.x264_r* and *548.exchange_r*. However, for *500.perlbench_r* and *502.gcc_r* the GNU executables perform better. The composite metric *SPECrate_int* for the Intel executables is 27% higher than for the ones generated by the GNU compilers with single-copy runs and 29% for six-copy runs. By analyzing opcode mix of these executables we find that the main reason for superior performance of the Intel compilers relative to the GNU compilers is that they take better advantage of the advanced vector instruction set extensions. These findings are not a surprise, reportable runs available on the SPEC CPU2017 page use almost exclusively the Intel compilers. The rest of our analysis is performed using executables produced by the Intel compilers.

4 BENCHMARK CHARACTERIZATION

This section gives the results of the characterization of the SPEC CPU2017 benchmarks. The single-threaded *speed* and the single-copy *rate* benchmarks, compiled by the Intel compilers, are run on the test machine. In case of benchmarks that use multiple input files (e.g., *600_perlbench_s*, *602_gcc_s*, *603_bwaves_s*, *657_xz_s*) the combined readings are provided.

Table 3 shows the main characteristics of the benchmarks, including: (a) the dynamic number of instructions retired (*IC* – instruction count); (b) the frequency of retired control-flow instructions (*Branches*); (c) the frequency of branch misses (*Branch misses*) that include events caused by misses in branch predictor structures (BTB, iBTB, RAS) or events where predictor structures provided incorrect branch target or branch outcome predictions [10]; (d) the frequency of memory reads and writes (*Loads* and *Stores*); as well as (e) cache misses across the cache hierarchy (*L1*, *L2*, and *L3 misses*). The instruction count is given in billions and all other metrics are expressed in units per 1,000 (kilo) retired instructions (PKI). *Branches* and *Branch misses* shed more light on the processor's front-end and its ability to provide a steady supply of decoded instructions to its backend. *L1 misses* counts the number of cache misses in L1D and L1I caches and match the number of references in the L2 cache. *L2 misses* counts the number of misses in the private per-core L2 cache, and *L3 misses* counts the number of misses in the shared uncore LLC cache.

The dynamic instruction count varies widely across suites and individual benchmarks within suites. E.g., the average IC is 14.1 trillion for *fp_speed* (0.96 to 69.1 trillion) and 2.4 trillion for *int_speed* benchmarks and significantly smaller for *fp_rate* and *int_rate* (1.68 and 1.52 trillion, respectively). The dynamic *IC*s reported in this study are notably lower than the corresponding ones reported in earlier studies that used GNU compilers [7][6]. This is especially true for a selected set of benchmarks, such as *603.bwaves_s*, *621.wrf_s*, *625.x264_s*, *628.pop2_s*, and *654.roms_s.*

Several observations can be made regarding the frequency of branches and branch misses. (a) First, benchmarks in the floating-point suites have a rather low fraction of branches - only 62.9 in *fp_speed* and 81.7 PKI in *fp_rate* – and a very small fraction of them result in misses - the average miss rate is 0.64 (from 0 to 3.2) PKI for *fp_speed* and 1.54 (from 0 to 5.6) PKI for *fp_rate.* (b) The integer benchmarks have a significantly higher fraction of branches, ~180 PKI on average. The benchmarks with a significant fraction of branch misses are *605.mcf_s*/*505.mcf_r* and *641.leela_s*/*541.leela_r*, are good candidates for studies targeting front-end architectural improvements.

Benchmarks in the floating-point suites have a relatively high fraction of loads. Thus, the average number of loads is 431.9 PKI in *fp_speed* and 398.1 in *fp_rate*. The average fraction of memory writes is 94.3 in *fp_speed* and 91.4 in *fp_rate*. These relatively frequent memory accesses turn into misses in the cache hierarchy. The average fraction of *L1 misses* is 213.4 for *fp_speed* and 114 PKI for *fp_rate*, whereas the average fraction of *L2 misses* is 54 PKI for *fp_speed* and 27 for *fp_rate*. The average fraction of *L3 misses* is 31.7 for *fp_speed* and 14.1 for *fp_rate*. These results reflect the fact that the floating-point *rate* benchmarks have smaller working sets that fit in the cache hierarchies better than their *speed* counterparts. Several benchmarks (both *speed* and *rate* variants), such as *bwaves*, *lbm*, *fotonik3d*, and *roms*, have a significant portion of misses across all levels of caches.

Table 3: General Parameters for CPU2017 Benchmarks

Benchmarks	**IC**	**Branches**	**Branch misses**	**Loads**	**Stores**	**L1 misses**	**L2 misses**	**L3 misses**
	[Billion]	**[PKI]**	**[PKI]**	**[PKI]**	**[PKI]**	**[PKI]**	**[PKI]**	**[PKI]**
fp_speed								
603.bwaves_s	8,816.3	8.6	0.05	715.8	73.5	188.5	57.9	51.7
607.cactuBSSN_s	8,812.9	15.5	0.01	514.0	110.2	109.4	18.5	11.9
619.lbm_s	3,830.4	21.7	0.59	380.2	169.0	466.9	118.2	64.9
621.wrf_s	7,729.1	79.7	0.81	407.4	76.5	130.4	35.2	11.9
627.cam4_s	12,079.9	101.7	0.74	234.7	119.1	66.1	14.6	7.2
628.pop2_s	8,121.8	76.9	0.46	370.7	114.3	236.9	51.3	13.7
638.imagick_s	69,141.7	145.8	0.33	193.9	5.6	31.6	9.9	0.2
644.nab_s	13,489.8	108.2	3.17	369.9	81.7	23.4	4.9	1.4
649.fotonik3d_s	3,315.8	30.6	0.07	558.5	113.4	348.4	89.7	68.7
654.roms_s	5,868.0	40.1	0.19	574.0	79.4	531.9	140.3	85.4
int_speed								
600.perlbench_s	2,741.2	202.5	1.44	296.3	183.6	24.1	6.1	1.4
602.gcc_s	2,549.1	259.9	3.71	308.2	84.3	100.7	28.2	11.2
605.mcf_s	1,193.7	242.0	21.35	353.9	76.9	330.3	108.5	28.5
620.omnetpp_s	1,101.1	220.4	4.54	338.1	173.3	170.5	52.7	33.8
623.xalancbmk_s	964.7	238.6	0.93	283.4	59.3	243.4	88.0	5.1
625.x264_s	1,356.8	78.9	1.32	204.1	80.9	22.8	5.1	1.1
631.deepsjeng_s	1,777.0	129.6	6.01	240.9	104.8	18.8	3.6	3.5
641.leela_s	1,927.5	154.6	16.82	264.8	94.0	10.0	1.5	0.0
648.exchange2_s	2,062.6	113.1	3.37	363.7	224.0	0.2	0.0	0.0
657.xz_s	7,723.5	152.6	11.10	242.3	79.3	66.4	21.1	9.6
fp_rate								
503.bwaves_r	1,241.9	10.7	0.04	651.2	88.8	236.2	78.8	61.6
507.cactuBSSN_r	1,065.3	17.1	0.02	513.0	112.0	132.6	18.2	8.5
508.namd_r	1,959.1	20.7	0.97	373.9	111.6	31.6	1.7	0.6
510.parest_r	2,368.7	104.0	4.68	421.9	36.5	157.4	48.8	1.3
511.povray_r	2,609.4	159.5	1.09	382.6	129.3	58.3	4.4	0.0
519.lbm_r	567.3	20.4	0.03	408.1	131.5	221.3	45.4	52.7
521.wrf_r	1,343.1	78.1	0.89	411.3	75.8	135.1	36.1	11.6
526.blender_r	1,688.8	163.8	5.57	336.4	49.4	28.3	9.5	1.4
527.cam4_r	1,500.0	115.0	1.14	285.7	99.3	111.1	25.7	3.3
538.imagick_r	2,508.4	127.7	0.96	196.8	79.5	17.9	2.1	0.0
544.nab_r	1,383.0	108.4	4.40	348.6	95.3	30.1	5.2	1.2
549.fotonik3d_r	1,401.9	25.7	0.07	581.1	116.1	330.9	118.3	62.5
554.roms_r	730.0	39.8	0.10	591.1	82.4	512.8	153.5	49.6
int_rate								
500.perlbench_r	2,741.4	202.8	1.44	296.3	183.4	24.4	6.2	1.4
502.gcc_r	1,172.0	239.1	3.58	286.6	130.0	105.8	32.0	9.5
505.mcf_r	677.4	226.6	22.74	336.5	118.3	241.1	70.6	24.3
520.omnetpp_r	1,101.1	220.4	4.52	338.1	173.3	171.8	53.0	34.2
523.xalancbmk_r	964.2	238.5	0.95	283.4	59.3	246.5	88.7	5.2
525.x264_r	1,275.5	81.1	1.40	216.2	86.8	25.4	5.8	1.1
531.deepsjeng_r	1,525.9	129.7	6.25	239.9	104.1	15.5	2.8	2.3
541.leela_r	1,927.9	154.6	16.82	264.8	94.0	10.7	1.6	0.0
548.exchange2_r	2,062.6	113.1	3.37	363.7	224.0	0.2	0.0	0.0
557.xz_r	1,798.6	181.6	6.34	249.0	51.5	45.8	12.5	3.6

Benchmarks in the integer suites have a somewhat smaller fraction of memory reads (~289.5 PKI) and writes (~116 PKI) than those in the floating-point suites. The average fraction of *L1 misses* is 98.7 for *int_speed* and 88.7 PKI for *int_rate*, whereas the average fraction of *L2 misses* is 31 PKI for *int_speed* and 27 for *int_rate*. The average fraction of L3 misses is 9.4 for *int_speed* and 8.2 for *int_rate*. From these observations, we can conclude that benchmarks in the integer suites present fewer memory requests and fewer cache misses are observed. Two benchmarks that see a bit more cache misses are *mcf* and *omnetpp* (both the *speed* and *rate* variants).

5 TMAM ANALISYS

Figure 1 shows the results of TMAM for all the *speed* benchmarks executed with one thread (N_T=1) as well as the average instruction per cycle (IPC) on the secondary *y*-axis. With TMAM, the product of the number of pipeline slots (5 in Coffee Lake architecture) and the number of clock cycles needed to execute a benchmark constitutes 100% of possible pipeline slots. Each pipeline slot is then marked as either *Retiring* (orange), *Bad Speculation* (gray), *Front-End Bound* (yellow), or *Back-End Bound* stalls. The *Back-End Bound* stalls are further broken down into (i) *Core Bound* stalls (light blue) that are caused by pressures on execution units or lack of instruction-level parallelism, and (ii) *Memory Bound* stalls (royal blue) that are caused by stalls related to caches and memory subsystems. Memory latency and limited memory bandwidth are major factors contributing to a large number of *Memory Bound* slots.

For *fp_speed* benchmarks, the percentage of *Retiring* slots varies from as low as 23% in *654.rom_s* (IPC=0.81) to 92% in *638.imagick_s* (IPC=3.41). There is a strong correlation between the *Retiring* slots and IPC – the higher the percentage of *Retiring* slots, the higher IPC. Relatively small fractions of pipeline slots are wasted due to *Bad Speculation* or *Front-end* misses. The portion of *Back-End Bound* slots highly correlates with the number of cache misses. Thus, the *Back-End Bound* slots account for a significant portion of pipeline slots in several benchmarks such as *619.lbm_s*, *649.fotonik3d_s*, and *654.rom_s* (over 70%). An exception is *638.imagick_s* that has only 5% of *Back-End Bound* slots. For the *int_speed* benchmarks, the portion of slots marked as *Bad Speculation* and *Front-End Bound* is significantly higher than in *fp_speed* – the averages are around 15% and 16%, respectively. The percentage of *Retiring* slots varies from as high as 59% in *625.x264_s* (IPC=2.43) to as low as 17% in *620.omnetpp_s* (IPC=0.70). The percentage of *Back-End Bound* slots is on average 33% for the *int_speed* benchmarks, and it varies from 9% in *648.exchange2_s* to 70% in *623.xalancmbk_s*. Averaging across *fp_speed* benchmarks, the *Memory Bound* stalls account for 31% and *Core Bound* for 19% of the total slots. For *int_speed* benchmarks, the *Memory Bound* stalls account for 19% and *Core Bound* stalls account for 11%.

Whereas the *top-level view* describes pipeline slot utilization, it does not directly translate into clock cycles and how they are utilized. It is important to know where the stalls are as a precursor in finding ways to eliminate them through either software optimization or future enhancements in hardware. To address this issue, we consider the breakdown of benchmark execution using a *clock cycles view* where clock cycles are marked as either used (orange) or unused/stalled. A clock-cycle is considered unused when no micro-operation begins execution during that cycle across all ports. The unused clock cycles are further divided into L1 bound, L2 bound, LLC/L3 bound, DRAM bound, store bound, and other unused (e.g., functional units are not available). A fully optimized application should not waste any cycles. The unused cycle ratio represents the room for improvement in both software and hardware. Figure 2 shows the clock cycle view for the *speed* suites. Most of the benchmarks spend a significant portion of time in main memory (DRAM). Increasing memory speed and bandwidth could help mitigate this issue.

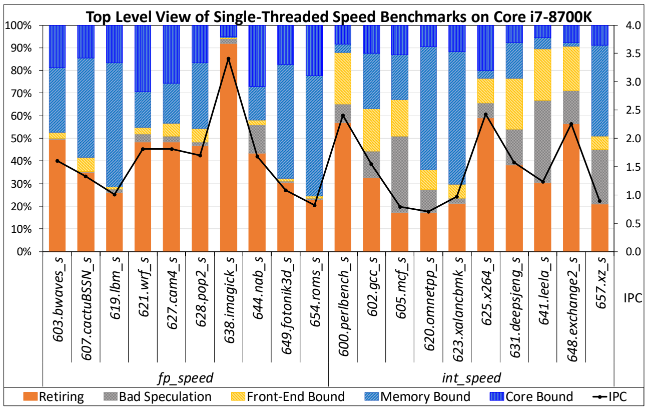


Figure 1: Top Level View of *Speed* Benchmarks

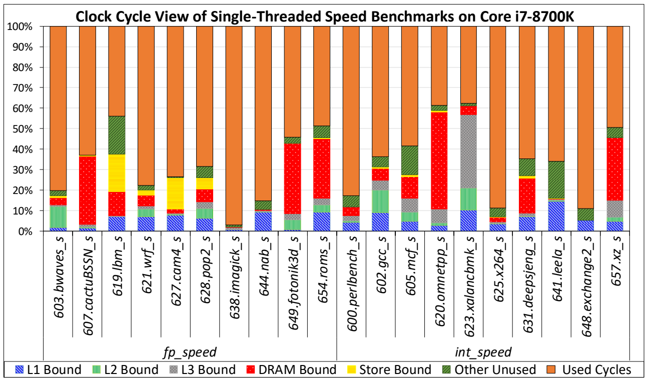


Figure 2: Clock Cycle view of *Speed* Benchmarks

Figure 3 shows the results of TMAM for all the *rate* benchmarks executed with one copy. Looking at *fp_rate*, the average IPC is 1.83, ranging from 1.01 (*554.roms_r*) to 2.70 (*508.namd_r*). *Retiring* slots average 49%, whereas *Front-End Bound* and *Bad Speculatio*n average 5% and 6%, respectively. However, the *Back-End Bound* accounts for 40% of slots. The correlation between *Retiring* slots and the IPC metric seen earlier for the *speed* benchmarks holds true for the *rate* benchmarks, too. Thus, *508.namd_r* has 75% of slots in *Retiring* which translates into IPC of 2.70. On the other side, *549.fotonik3d_r* and *554.roms_r* have only 28% of slots in *Retiring*, which translates into IPC of 1.04 and 1.01 respectively. Expectedly, the majority of benchmarks in the *fp_rate* suite is *Back-End Bound*, e.g., *549.fortonik3d_r* and *554.roms_r* has 70% of slots. With respect to *int_rate*, the average IPC is 1.53, ranging from 0.70 (*520.omnetpp_r*) to 2.46 (*525.x264_r*). With 49% of *Retiring* slots, *int_rate* has a higher percentage of *Bad Speculatio*n and *Front-End Bound* stalls at 16% for both. The *Back-End Bound* stalls account for 31%. Averaging across *fp_rate*, the *Memory Bound* stalls account for 22% and *Core Bound* for 18% of the total slots. For *int_rate* the *Memory Bound* stalls account for 19% and *Core Bound* stalls account for 11%. Figure 4 shows a clock cycle view of each of the *rate* benchmarks. For many *rate* benchmarks, the breakdown is similar to their *speed* counterparts. In some cases, the stalls in the back-end are less frequent than in their counterparts because inputs are smaller and thus result in fewer misses in caches.

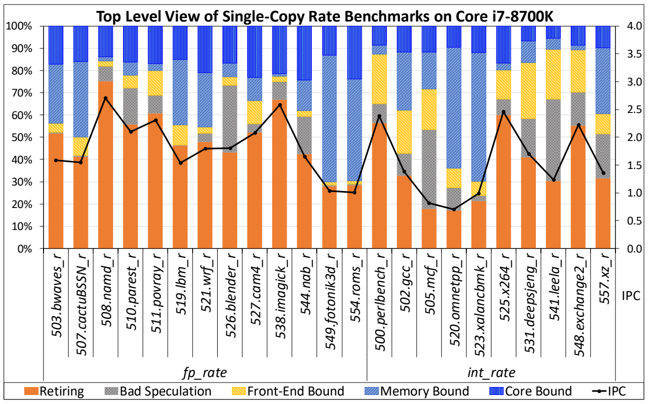


Figure 3: Top Level View of *Rate* Benchmarks

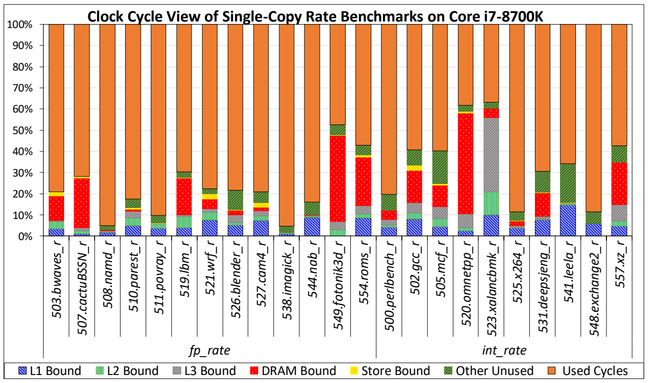


Figure 4: Clock Cycle View of *Rate* Benchmarks

6 SCALABILITY STUDY RESULTS

This section explores the impact of thread/copy count on performance and energy consumption [2]. To eliminate the impact of uneven distribution of turbo bins in the Intel's Turbo Boost technology, where the processor clock frequency increases are a function of the number of active processor cores, the processor clock frequency is set to 4.30 GHz across all the cores, regardless of the number of threads or copies [3]. Temperature and frequency monitoring show the peak package temperature to be ~57°C with no thermal throttling. The execution time is obtained directly from the SPEC CPU2017 *runcpu* utility when benchmarks are run with the reference data inputs. Table 4 shows the speedup, *S*, defined in Eq. (1) for *fp_speed* and *int_speed* and in Eq. (2) for *fp_rate* and *int_rate*, when the number of threads/copies is 1, 2, 4, 6, 8, 10, and 12. The benchmarks are grouped based on the speedup achieved when running with 6 threads/copies, into those that "scale very well", $S \geq 4$, "scale moderately", $2 \leq S < 4$, and those that "scale poorly" ($S < 2$). We also identify benchmarks that continue scaling when the number of threads/copies exceeds the number of physical cores.

***Speedup* for *fp_speed/int_speed*.** For *fp_speed* just two benchmarks can be classified as those that scale very well (*638.imagick_s* and *644.nab_s*). Their speedup continues increasing even when the number of threads exceeds the number of physical cores. Benchmarks that belong to the moderately scaling group (*607.cactuBSSN_s*, *621.wrf_s*, *627.cam4_s*, and *628.pop2_s*) see little or no benefits when the number of threads exceeds the number of physical cores. The remaining benchmarks scale poorly (*603.bwaves_s*, *619.lbm_s*, *649.fotonik3d_s*, and *654.roms_s*) and their performance degrades as the number of threads exceeds the number of physical cores. For *619.lbm_s* running with N_T>1 will hurt performance (S<1). In the case of a perfectly parallelizable application that is memory intensive, increase in the number of threads might end up hurting performance. The *fp_speed* benchmarks with poor scalability have a significant number of cache misses and are bounded by limited memory bandwidth. The integer benchmarks are not parallelized, except *657.xz_s* that scales moderately.

***Speedup* for *fp_rate*.** For *fp_rate* a number of benchmarks scales well, including *507.cactuBSSN_r, 508.namd_r, 511.povray_r, 526.blender_r, 527.cam4_r, 538.imagick_r,* and *544.nab_r*. These benchmarks scale up until the number of copies matches the number of logical cores and are not bounded by memory. Two benchmarks *510.parest_r* and *521.wrf_r* belong to the moderately scaling group. The remaining benchmarks are bounded by off-chip memory accesses and scale poorly (*503.bwaves_r, 519.lbm_r, 549.fotonik3d_r,* and *554.roms_r*).

***Speedup* for *int_rate*.** For *int_rate*, the speedup of 12-copy benchmarks is found to be the best, ranging from 3.57 (*520.omnetpp_r*) to 7.78 (*541.leela_r*). All benchmarks scale very well except *520.omnetpp_r* that scales moderately. All benchmarks except *505.gcc_r* and *505.mcf_r* continue to scale when the number of copies exceeds the number of physical cores.

Looking at the scalability results for the test machine, we see that the best performance is achieved when N_T=6 for *fp_speed* and N_T=12 for *int_speed*. For *fp_rate* and *int_rate* the best performance is achieved when N_C=12. Hence, for *PE.I* analysis we consider runs with the number of thread/copies set to 1, 6, and 12. Table 5 shows the execution time, energy, the speedups *S*, and the *PE.I* metrics, when the number of threads/copies is set to 1, 6, and 12. When *PE.I* > *S* >1 for a given benchmark, that means that multithreaded and multi-copy executions not only save time, but also reduce the overall energy consumed for computation. We retain the classification of benchmarks from above and discuss the changes in *PE.I*.

***PE.I* for *fp_speed/int_speed*.** The benchmarks that scale very well in performance (*638.imagick_s* and *644.nab_s*) also reduce the overall energy, so *PE.I* > 8 for N_T=6. They also take advantage of hyper-threading when the number of threads exceeds the number of physical cores, especially *644.nab_s* that reaches *PE.I* of 12.95 when N_T=12. In a group of moderately scaling benchmarks some of them (*607.cactuBSSN_s*, *627.wrf_s*) provide energy savings when running with 6 and 12 threads (*PE.I* > *S*) and some of them do not (*621.wrf_s*, and *628.pop2_s*). Finally, the last group with poorly scaling benchmarks results in energy losses when increasing the number of threads (PE.I < S). *PE.I* for these benchmarks falls far below 1, indicating that multithreaded runs of these benchmarks are inferior to single-threaded runs when both performance and energy are considered together. In essence, memory intensive benchmarks scale poorly when the required memory bandwidth goes beyond the maximum available bandwidth, causing significant losses in energy.

Table 4: *Speedup*, S, of CPU2017 Benchmarks (HB metric)

# Threads		1 T	2 T	4 T	6 T	8 T	10 T	12 T
fp_speed	603.bwaves_s	1.00	1.44	1.36	1.41	1.41	1.38	1.36
	607.cactuBSSN_s	1.00	1.92	3.16	3.57	3.66	3.95	4.34
	619.lbm_s	1.00	0.93	0.89	0.87	0.85	0.84	0.81
	621.wrf_s	1.00	1.74	2.37	2.43	2.49	2.56	2.61
	627.cam4_s	1.00	1.89	2.89	3.38	3.54	3.55	3.56
	628.pop2_s	1.00	1.45	2.58	2.58	2.44	2.39	2.32
	638.imagick_s	1.00	2.13	3.94	5.44	5.46	5.77	5.97
	644.nab_s	1.00	2.10	3.75	5.33	5.83	6.44	7.12
	649.fotonik3d_s	1.00	1.09	1.06	1.04	1.03	1.02	0.98
	654.roms_s	1.00	1.21	1.45	1.39	1.39	1.37	1.22
int_speed	657.xz_s	1.00	1.85	2.67	3.34	3.64	3.69	3.71
# Copies		1 C	2 C	4 C	6 C	8 C	10 C	12 C
fp_rate	503.bwaves_r	1.00	1.16	1.17	1.16	1.14	1.15	1.10
	507.cactuBSSN_r	1.00	1.91	3.19	4.23	4.33	4.50	4.71
	508.namd_r	1.00	2.17	4.00	5.74	6.38	6.52	6.78
	510.parest_r	1.00	1.97	2.30	2.09	1.90	1.85	1.63
	511.povray_r	1.00	2.12	4.01	5.73	5.97	6.02	6.49
	519.lbm_r	1.00	1.19	1.19	1.17	1.14	1.17	1.10
	521.wrf_r	1.00	1.90	2.52	2.41	2.28	2.30	2.07
	526.blender_r	1.00	2.10	3.77	5.38	5.86	6.03	6.43
	527.cam4_r	1.00	2.05	3.42	4.04	3.67	3.60	3.30
	538.imagick_r	1.00	2.16	4.00	5.62	5.89	5.95	6.20
	544.nab_r	1.00	2.15	4.01	5.75	6.64	6.70	7.23
	549.fotonik3d_r	1.00	1.32	1.41	1.40	1.34	1.30	1.22
	554.roms_r	1.00	1.56	1.53	1.45	1.30	1.29	1.04
int_rate	500.perlbench_r	1.00	2.03	3.64	5.00	5.52	5.72	6.03
	502.gcc_r	1.00	1.97	3.19	4.03	4.12	3.85	3.58
	505.mcf_r	1.00	2.02	3.43	4.51	4.55	4.52	4.48
	520.omnetpp_r	1.00	1.82	2.88	3.37	3.49	3.46	3.57
	523.xalancbmk_r	1.00	2.01	3.23	4.21	4.63	4.70	4.76
	525.x264_r	1.00	2.23	3.97	5.65	6.21	6.40	6.91
	531.deepsjeng_r	1.00	2.11	3.78	5.29	6.01	6.15	7.15
	541.leela_r	1.00	2.13	3.98	5.73	6.43	7.10	7.98
	548.exchange2_r	1.00	2.11	3.99	5.92	6.04	6.25	6.54
	557.xz_r	1.00	1.98	3.46	4.70	5.27	5.80	6.10

***PE* for *fp_rate*.** For *fp_rate*, all the benchmarks that scale well provide significant energy savings when running with 6 and 12 copies. The moderately scaling benchmarks have *PE.I* < *S* indicating that performance improvements come at the cost of increased overall energy. Finally, the poorly scaling benchmarks increase the total energy consumed resulting in *PE.I* to be significantly lower than *S*. Resource and memory contention degrades performance and increases energy overhead to such an extent that running N_C concurrent copies of these benchmarks is inferior to running a single copy N_C times sequentially.

***PE* for *int_rate*.** The *int_rate* benchmarks all see energy savings when running with 6 and 12 copies (*PE.I* > *S*). These benchmarks are not memory intensive and do not cause significant contention on shared resources. As noted in the text above *505.gcc_r* and *505.mcf_r* has *S* and *PE.I* degraded when increasing the number of copies from 6 to 12.

Table 5: *PE* Analysis for CPU2017 Benchmarks

Benchmarks	{NT | NC}=1		{NT | NC}=6		{NT | NC}=12		S [6]	PE.I [6]	S [12]	PE.I [12]
	Time	Energy	Time	Energy	Time	Energy				
	[s]	[J]	[s]	[J]	[s]	[J]				
fp_speed										
603.bwaves_s	1,230	22,783	873	35,415	904	39,716	1.41	0.91	1.36	0.78
607.cactuBSSN_s	1,498	24,003	419	19,454	345	19,370	3.57	4.41	4.34	5.38
619.lbm_s	844	15,403	976	39,944	1,041	43,748	0.87	0.33	0.81	0.29
621.wrf_s	996	19,333	411	19,974	381	21,301	2.43	2.35	2.61	2.37
627.cam4_s	1,544	27,724	457	22,245	434	24,430	3.38	4.21	3.56	4.04
628.pop2_s	1,105	22,565	429	22,971	477	27,359	2.58	2.53	2.32	1.91
638.imagick_s	4,711	84,870	866	53,047	789	53,612	5.44	8.71	5.97	9.46
644.nab_s	1,887	30,193	354	18,575	265	16,608	5.33	8.66	7.12	12.95
649.fotonik3d_s	667	13,247	644	26,687	679	29,563	1.04	0.51	0.98	0.44
654.roms_s	1,593	32,096	1,147	52,324	1,311	63,978	1.39	0.85	1.22	0.61
int_speed										
600.perlbench_s	247	5,897	244	5,848	243	5,823	1.01	1.02	1.02	1.03
602.gcc_s_a	352	7,715	352	7,718	352	7,716	1.00	1.00	1.00	1.00
605.mcf_s	328	7,263	328	7,164	330	7,230	1.00	1.02	0.99	1.00
620.omnetpp_s	335	7,083	336	7,085	336	7,014	1.00	1.00	1.00	1.01
623.xalancbmk_s	211	4,401	210	4,372	212	4,413	1.00	1.01	0.99	0.99
625.x264_s	131	2,778	130	2,793	130	2,776	1.00	1.00	1.01	1.01
631.deepsjeng_s	240	5,412	241	5,433	241	5,421	1.00	0.99	1.00	1.00
641.leela_s	334	7,503	335	7,525	334	7,506	1.00	0.99	1.00	1.00
648.exchange2_s	196	4,613	197	4,608	196	4,602	1.00	1.00	1.00	1.01
657.xz_s	1,984	30,328	593	21,942	534	20,054	3.34	4.62	3.71	5.62
fp_rate										
503.bwaves_r	178	3,740	922	37,596	1,942	88,150	1.16	0.69	1.10	0.56
507.cactuBSSN_r	158	2,661	224	11,727	403	25,743	4.23	5.76	4.71	5.84
508.namd_r	171	3,034	179	11,756	303	23,347	5.74	8.88	6.78	10.57
510.parest_r	262	5,117	754	36,144	1,936	90,892	2.09	1.77	1.63	1.10
511.povray_r	264	5,302	276	19,526	488	40,574	5.73	9.34	6.49	10.18
519.lbm_r	84	1,951	429	20,912	910	50,929	1.17	0.66	1.10	0.51
521.wrf_r	173	3,368	430	21,359	1,002	52,018	2.41	2.28	2.07	1.61
526.blender_r	218	3,872	243	14,150	407	26,578	5.38	8.83	6.43	11.24
527.cam4_r	169	3,247	251	14,743	615	37,504	4.04	5.34	3.30	3.43
538.imagick_r	228	4,164	244	14,719	442	31,273	5.62	9.54	6.20	9.90
544.nab_r	197	3,144	205	11,381	327	22,318	5.75	9.53	7.23	12.22
549.fotonik3d_r	296	5,649	1,271	51,361	2,920	127,853	1.40	0.92	1.22	0.65
554.roms_r	163	3,371	675	31,174	1,885	92,145	1.45	0.94	1.04	0.46
int_rate										
500.perlbench_r	263	4,715	315	18,756	523	37,994	5.00	7.55	6.03	8.98
502.gcc_r	194	3,192	288	14,329	648	34,657	4.03	5.39	3.58	3.96
505.mcf_r	190	2,964	253	12,953	509	27,548	4.51	6.19	4.48	5.79
520.omnetpp_r	348	5,595	619	27,808	1,169	57,652	3.37	4.07	3.57	4.16
523.xalancbmk_r	226	3,670	321	14,728	568	29,890	4.21	6.30	4.76	7.02
525.x264_r	126	2,272	134	8,145	219	16,366	5.65	9.45	6.91	11.51
531.deepsjeng_r	206	3,671	234	13,203	346	24,137	5.29	8.83	7.15	13.05
541.leela_r	363	6,341	380	21,749	546	37,447	5.73	10.02	7.98	16.21
548.exchange2_r	212	3,861	215	13,503	389	28,173	5.92	10.16	6.54	10.76
557.xz_r	307	4,786	391	18,571	604	34,701	4.70	7.27	6.10	10.09

7 CONCLUSIONS

This paper includes a number of experimental studies performed on a test machine with an Intel's Core i7-8700K processor when running CPU2017 benchmarks. The findings of the study are as follows. (a) Compiler comparison: We determine that Intel compilers produce executables that run faster than those produced by GNU compilers, mainly due to better utilization of advanced vector extensions in the Intel64 ISA. (b) Characterization: We characterize the benchmarks in terms of their resource requirements by providing a top-view that includes SPEC metrics, execution time, and frequency of events with a significant impact on performance. This characterization may help computer architecture researchers to identify benchmarks that are a suitable target for their architectural enhancements. (c) TMAM Analysis: Intel's Top-down Microarchitecture Analysis Method shows an in-depth analysis of the utilization of internal processor resources. These results reveal effectiveness of the i7-8700K pipeline and bottlenecks exposed by the CPU2017 benchmarks, including a breakdown to memory bound, core-bound, and front-end bound pipelines slots. We find that the floating-point benchmarks are mainly back-end bound (memory and core), whereas the integer benchmark are bound by the front-end and memory. (d) Scalability: We analyze the impact of increasing the number of threads/copies on performance, energy, and a combined metric called *PE.I* that considers the scalability of benchmarks when both performance and energy efficiency are considered together. The results reveal how CPU2017 benchmarks scale as we increase the number of threads/copies in i7-8700K and how to perform trade-offs between energy and power. Considering the size and complexity of the SPEC CPU2017 benchmark suits, the scalability findings can be used to guide future architectural simulations by selecting suitable benchmarks and their running parameters.

REFERENCES

[1] J. N. Amaral, E. Borin, D. R. Ashley, C. Benedicto, E. Colp, J. H. S. Hoffmam, M. Karpoff, E. Ochoa, M. Redshaw, and R. E. Rodrigues. 2018. The Alberta Workloads for the SPEC CPU 2017 Benchmark Suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 159–168. DOI:https://doi.org/10.1109/ISPASS.2018.00029

[2] A. Dzhagaryan and A. Milenković. 2014. Impact of Thread and Frequency Scaling on Performance and Energy in Modern Multicores: A Measurement-based Study. In *Proceedings of the 2014 ACM Southeast Regional Conference* (ACM SE '14), 14:1–14:6. DOI:https://doi.org/10.1145/2638404.2638473

[3] H. Esmaeilzadeh, T. Cao, X. Yang, S. M. Blackburn, and K. S. McKinley. 2011. Looking Back on the Language and Hardware Revolutions: Measured Power, Performance, and Scaling ∗. *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems* (March 2011), 319–332.

[4] J. L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (September 2006), 1–17. DOI:https://doi.org/10.1145/1186736.1186737

[5] A. Kejariwal, A. V. Veidenbaum, A. Nicolau, X. Tian, M. Girkar, H. Saito, and U. Banerjee. 2008. Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the intel® Core^{TM} 2 Duo processor. In *International Conference on Embedded Computer Systems: Architectures, Modeling*, 132–141. DOI:https://doi.org/10.1109/ICSAMOS.2008.4664856

[6] A. Limaye and T. Adegbija. 2018. A Workload Characterization of the SPEC CPU2017 Benchmark Suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 149–158. DOI:https://doi.org/10.1109/ISPASS.2018.00028

[7] R. Panda, S. Song, J. Dean, and L. K. John. 2018. Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 271–282. DOI:https://doi.org/10.1109/HPCA.2018.00032

[8] T. K. Prakash and L. Peng. 2008. Performance Characterization of SPEC CPU2006 Benchmarks on Intel Core 2 Duo Processor. In *ISAST Trans. Comput. Softw. Eng.* (1), 36–41.

[9] J. Treibig, G. Hager, and G. Wellein. 2010. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *2010 39th International Conference on Parallel Processing Workshops*, 207–216. DOI:https://doi.org/10.1109/ICPPW.2010.38

[10] V. Uzelac and A. Milenkovic. 2009. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*, 207–217. DOI:https://doi.org/10.1109/ISPASS.2009.4919652

[11] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore. 2012. Measuring Energy and Power with PAPI. In *2012 41st International Conference on Parallel Processing Workshops*, 262–268. DOI:https://doi.org/10.1109/ICPPW.2012.39

[12] A. Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 35–44. DOI:https://doi.org/10.1109/ISPASS.2014.6844459

[13] 2010. White-Paper Using SPEC CPU2006 Benchmark Results to Compare the Compute Performance of Servers.

[14] SPEC CPU® 2017. Retrieved March 19, 2018 from https://www.spec.org/cpu2017/

[15] Intel® Core^{TM} i7-8700K Processor Product Specifications. *Intel® ARK (Product Specs)*. Retrieved March 24, 2018 from https://tinyurl.com/ybcw5vc8

[16] Perf: Linux profiling with performance counters. *Perf Wiki*. Retrieved March 19, 2018 from https://perf.wiki.kernel.org/index.php/Main_Page

[17] Intel® VTune^{TM} Amplifier 2018 User's Guide. Retrieved June 1, 2018 from https://tinyurl.com/y76ondwo