Experiments with SPEC CPU 2017: Similarity, Balance, Phase Behavior and SimPoints

Shuang Song, Qinzhe Wu, Steven Flolid,
Joseph Dean, Reena Panda, Junyong Deng,
Lizy K. John
songshuang1990, qw2699, stevenflolid,
jd45664, reena.panda, jd47372@utexas.edu, ljohn@ece.utexas.edu

Laboratory for Computer Architecture
Department of Electrical and Computer Engineering
The University of Texas at Austin

TR-180515-01

This work was partially supported by National Science Foundation (NSF)
under grant numbers 1725743 and 1745813, the Texas Advanced Computing Center (TACC)
and by an Intel unrestricted gift. Any opinions, findings,
conclusions or recommendations expressed in this material are those of
the authors and do not necessarily reflect the views of NSF or other sponsors.

CONTENTS

I Introduction 3

II CPU2017 Benchmarks: Overview & Characterization 3
- II-A Benchmark Overview . . . 4
- II-B Performance Characterization . . . 4
- II-C Performance Bottleneck Analysis . . . 5
- II-D Scalability . . . 5

III Methodology 6

IV Redundancy in CPU2017 Benchmark Suite 7
- IV-A Subsetting the CPU2017 Benchmarks . . . 7
- IV-B Evaluating Representativeness of Subsets . . . 8
- IV-C Selecting Representative Input Sets . . . 8
- IV-D Are Rate and Speed Benchmarks Different? . . . 9
- IV-E Benchmark Classification based on Branch and Memory Behavior . . . 10
- IV-F Difference Between Benchmarks from Same Application Area . . . 10

V Balance in the SPEC CPU2017 Benchmark Suites 11
- V-A Comparing Performance Spectrum of CPU2017 & CPU2006 Suites . . . 11
- V-B Comparison of Application Domains . . . 12
- V-C Comparing Power Consumption . . . 12
- V-D Case Study on EDA Applications . . . 12
- V-E Case Study on Database Applications . . . 13
- V-F Case Study on Graph Applications . . . 13
- V-G Sensitivity of CPU2017 Programs to Performance Characteristics . . . 13

VI Large Scale Phase Analysis 14
- VI-A Phase-level Variability . . . 14
- VI-B Simulation Points . . . 16

VII Related Work 16

VIII Conclusion 16

IX Acknowledgements 18

References 18

Appendix 19
- A Supplemental Graphs and Tables . . . 19

***Abstract*—The recently released SPEC CPU2017 benchmark suite has already started receiving a lot of attention from both industry and academic communities. However, due to the significantly high size and complexity of the benchmarks, simulating all the CPU2017 benchmarks for design trade-off evaluation is likely to become extremely difficult. Simulating a randomly selected subset, or a random input set, may result in misleading conclusions. This paper analyzes the SPEC CPU2017 benchmarks using performance counter based experimentation from seven commercial systems, and uses statistical techniques such as principal component analysis and clustering to identify similarities among benchmarks. Such analysis can reveal benchmark redundancies and identify subsets for researchers who cannot use all benchmarks in pre-silicon design trade-off evaluations. The benchmarks in the CPU2017 suite are compared to others in the suite and also with CPU2006. Additionally, to evaluate the balance of CPU2017 benchmarks, we analyze the performance characteristics of CPU2017 workloads and compare them with emerging database, graph analytics and electronic design automation (EDA) workloads. One of the major changes in the new suite is the addition of multithreaded benchmarks, and hence we analyze the scalability of the multithreaded programs. We also study the phase behavior and variability of the programs, and identify large scale phases and simulation points that researchers can use if the whole program cannot be used for the experimentation.**

I. Introduction

Since its formation in 1988, SPEC has carefully chosen benchmarks from real world applications and periodically distributed these benchmarks to the semiconductor community. The last SPEC CPU benchmark suite was released in 2006 and has been widely used by industry & academia to evaluate the quality of processor designs. In the last 10+ years, the processing landscape has undergone a significant change. For instance, the size of processor components (caches, branch predictors, TLBs, etc.) and memory have increased significantly. To keep pace with technological advances and emerging application domains, the 6^{th} generation of SPEC CPU benchmarks have just been released.

The SPEC CPU2017 suite [1] consists of 43 benchmarks, separated into 4 sub-suites, corresponding to "rate" and "speed" versions of the integer and floating point programs (summarized in Table I). Benchmarks in CPU2017 have up to $\sim$10X higher dynamic instruction counts than those in CPU2006; such an increase in the program size is bound to exacerbate the simulation time problem on detailed performance simulators [2], [3], [4], [5], [6]. To keep the simulation times manageable, researchers often use a subset of the benchmarks. However, arbitrarily selected subsets can result in misleading conclusions. Understanding program behavior and their similarities can help in selecting benchmarks to represent target workload spaces. In this paper, we first conduct a detailed characterization of the CPU2017 benchmarks using performance counter based experimentation from several state-of-the-art systems and extract critical insights regarding the micro-architectural bottlenecks of the programs. Next, we leverage statistical techniques such as Principal Component Analysis (PCA) and clustering analysis to understand the (dis)similarity of benchmarks and identify redundancies in the suite. We demonstrate that using less than one-third of the benchmarks can predict the performance of the entire suite with $\geq$93% accuracy.

For the first time, SPEC has also provided separate "speed" and "rate" versions of benchmarks (see Table I, 5nn.benchmark_r for the rate version and 6nn.benchmark_s for the speed version) in their CPU suite. SPECspeed always runs one copy of each benchmark, and SPECrate runs multiple concurrent copies of each benchmark. We observe that the CPU2017 speed benchmarks have up to 8x higher instruction counts than their rate equivalents. SPEC's web page indicates that such benchmarks differ in terms of the workload sizes, compilation flags, etc. However, are they truly different in the performance spectrum? Our analysis indicates that most benchmarks (except a few cases, e.g., *imagick*, $fotonik3d$) have very similar performance characteristics between the rate and speed versions.

SPEC CPU2006 benchmarks [7] have long been the de facto benchmark for studying single-threaded performance. The SPEC CPU2017 benchmark suite has replaced many of the benchmarks in the SPEC CPU2006 suite with larger and more complex workloads; compared to the CPU2006 programs, it is not known whether the CPU2017 workloads have different performance demands or whether they stress machines differently. How much of the performance spectrum is lost due to benchmark removal? Do the newly added benchmarks expand the performance spectrum? We perform a detailed comparison between the two suites to identify key differences in terms of performance and power consumption.

While CPU2017 suite has introduced or expanded several application domains (e.g., artificial intelligence), many application domains have been removed (e.g., speed recognition, electronic design automation) or not included (e.g., graph analytics). We further investigate the application domain balance and coverage of the CPU2017 benchmarks using statistical techniques. Specifically, we explore whether the CPU2017 workloads have performance features that can exercise computer systems in a similar manner as emerging data-serving and graph analytics workloads.

The rest of this paper is organized as follows. Section II gives an overview of the CPU2017 benchmarks and analyzes their micro-architectural performance. This section also discusses the scalability of the CPU2017 benchmarks since that is one of the primary difference with the earlier CPU2006 suite. Section III discusses the methodology used to measure program (dis)similarity. Section IV proposes representative subsets and input sets of the programs. Section V evaluates balance in the CPU2017 suite. Section VI presents the phase behavior and simulation points for the CPU2017 suite. Finally, we discuss related work and conclude in Sections VII and VIII, respectively.

II. CPU2017 Benchmarks: Overview & Characterization

In this section, we will first provide an overview of the CPU2017 benchmarks. We will also characterize their micro-architectural behvaior, while focusing on single-core CPU performance. This characterization is performed on an Intel Skylake machine (3.4 GHz, i7-6700 processor, 8MB last-level cache) running Ubuntu 14.04. Benchmarks are compiled using

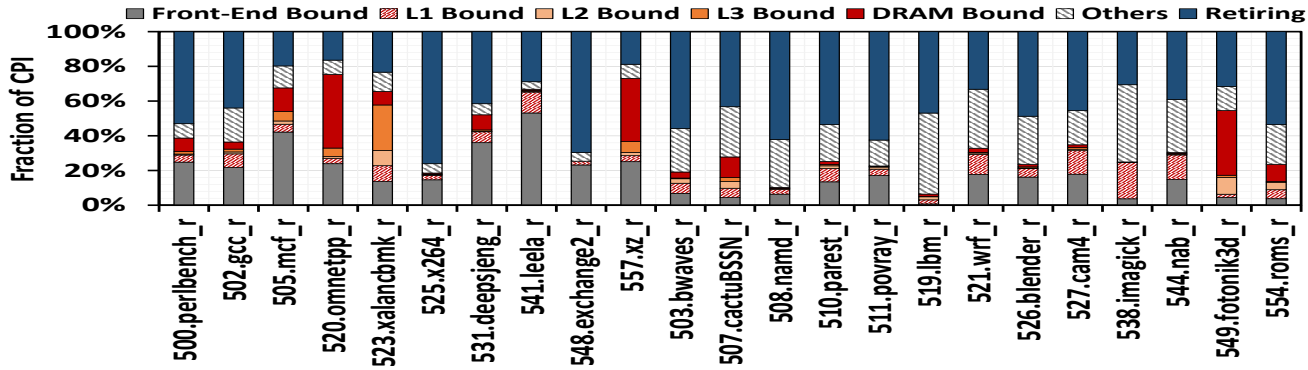


Fig. 1: Cycles per instruction (CPI) stack of CPU2017 rate benchmarks.

gcc compiler with SPEC recommended optimization flags. The performance counter measurements are carried out using the Linux perf [8] tool.

A. Benchmark Overview

Unlike its predecessors, the CPU2017 suite [1] is divided into four categories: speed integer (SPECspeed INT), rate integer (SPECrate INT), speed floating point (SPECspeed FP) and rate floating point (SPECrate FP), as shown in Table I. The SPECspeed INT, SPECspeed FP and SPECrate INT groups consist of 10 benchmarks each, while the SPECrate FP group consists of 13 benchmarks. In addition, the CPU2017 benchmarks are still written in C, C++ and Fortran languages.

Several new benchmarks and application domains have been added in the CPU2017 suite. In the FP category, nine new benchmarks have been added: *parest* implements a finite element solver for biomedical imaging; *blender* performs 3D rendering; *cam*4, *pop*2 and *roms* represent the climatology domain; *imagick* is an image manipulation application; *nab* is a floating-point intensive molecular modeling application representing the life sciences domain; *fotonik3d* and *cactuBSSN* represents the physics domain. In the INT category, the most notable enhancement has been made in the artificial intelligence domain with three new benchmark additions (*deepsjeng*, *leela* and *exchange*2). Two other compression-related benchmarks, *x*264 (video compression) and *xz* (general data compression) have also been added. We will analyze the application domain coverage of CPU2017 suite in detail in Section IV.

B. Performance Characterization

Table I shows the dynamic instruction count, instruction mix, and CPI of each CPU2017 benchmark. The dynamic instruction count of the benchmarks is in the order of trillions of instructions. In general, the speed benchmarks have significantly higher dynamic instruction count than the rate benchmarks. The ratio of dynamic instruction count in speed to rate categories is ∼8x (avg) for the floating-point benchmarks and ∼2x (avg) for the integer benchmarks. Compared to the CPU2006 FP benchmarks, the CPU2017 FP benchmarks have ∼10x higher dynamic instruction count. This steep increase in instruction counts will further exacerbate the problem of benchmark simulation time on most state-of-the-art simulators [2], [3], [5].

TABLE I: Dynamic Instr. Count, Instr. Mix and CPI of the 43 SPEC CPU2017 benchmarks (Intel Skylake).

Benchmark	Icount (Billion)	Loads (%)	Stores (%)	Branches (%)	CPI
SPECspeed Integer — 10 benchmarks					
600.perlbench_s	2696	27.20	16.73	18.16	0.42
602.gcc_s	7226	40.32	15.67	15.60	0.58
605.mcf_s	1775	18.55	4.70	12.53	1.22
620.omnetpp_s	1102	22.76	12.65	14.55	1.21
623.xalancbmk_s	1320	34.08	7.90	33.18	0.86
625.x264_s	12546	37.21	10.27	4.59	0.36
631.deepsjeng_s	2250	19.75	9.37	11.75	0.55
641.leela_s	2245	14.25	5.32	8.94	0.80
648.exchange2_s	6643	29.61	20.22	8.67	0.41
657.xz_s	8264	13.34	4.73	8.21	1
SPECrate Integer — 10 benchmarks					
500.perlbench_r	2696	27.20	16.73	18.16	0.42
502.gcc_r	3023	34.51	16.64	14.96	0.59
505.mcf_r	999	17.42	6.08	11.54	1.16
520.omnetpp_r	1102	22.10	12.27	14.12	1.39
523.xalancbmk_r	1315	34.26	8.07	33.26	0.86
525.x264_r	4488	23.03	6.47	4.37	0.31
531.deepsjeng_r	1929	19.61	9.10	11.61	0.57
541.leela_r	2246	14.28	5.33	8.95	0.81
548.exchange2_r	6644	29.62	20.24	8.69	0.41
557.xz_r	1969	17.33	3.87	12.24	1.22
SPECspeed Floating-point — 10 benchmarks					
603.bwaves_s	66395	31.00	4.42	13.00	0.34
607.cactuBSSN_s	10976	43.87	9.50	1.80	0.68
619.lbm_s	4416	29.62	17.68	1.40	0.87
621.wrf_s	18524	23.20	5.80	9.48	0.77
627.cam4_s	15594	20	14	10.92	0.68
628.pop2_s	18611	21.71	8.41	15.13	0.48
638.imagick_s	66788	18.16	0.46	9.30	1.17
644.nab_s	13489	23.49	7.51	9.55	0.68
649.fotonik3d_s	4280	33.99	13.89	3.84	0.78
654.roms_s	22968	32.02	8.02	7.53	0.52
SPECrate Floating-point — 13 benchmarks					
503.bwaves_r	5488	34.92	4.77	9.51	0.42
507.cactuBSSN_r	1322	43.62	9.53	1.97	0.69
508.namd_r	2237	30.12	10.25	1.75	0.41
510.parest_r	3461	29.51	2.50	11.49	0.48
511.povray_r	3310	30.30	13.13	14.20	0.42
519.lbm_r	1468	28.35	15.09	1.05	0.53
521.wrf_r	3197	22.94	5.93	9.48	0.81
526.blender_r	5682	36.10	12.07	7.89	0.53
527.cam4_r	2732	19.99	8.37	11.06	0.56
538.imagick_r	4333	22.55	7.97	10.94	0.90
544.nab_r	2024	23.70	7.46	9.65	0.69
549.fotonik3d_r	1288	39.12	v12.07	2.52	0.96
554.roms_r	2609	34.57	7.57	6.73	0.48

In terms of instruction mix, we can make several interesting observations. For the integer benchmarks (rate and speed), the fraction of branch instructions is roughly ≤15%, with several benchmarks (e.g., 625.*x*264_*s*, 641.*leela_s*, 525.*x*264_*r*) having ≤8% branch instructions. This behavior is in contrast

to the CPU2006 integer programs, which have an average of 20% branches in their dynamic instruction stream [9]. The *xalancbmk* benchmark, which is one of the four C++ programs in the INT category, has the highest fraction of branch instructions (33%). The other C++ programs (*omnetpp*, *leela* and *deepsjeng*) have ≤15% branches. For the FP categories, most benchmarks have much lower fraction of control instructions (≤9% on average) than the integer benchmarks, with several benchmarks having as low as 1% branches. The large dynamic basic block size of the FP programs can be an opportunity for the underlying micro-architectures to exploit higher degree of parallelism. In terms of memory operations, the CPU2017 benchmarks are memory-intensive, with several benchmarks (e.g., 602.*gcc_r*, 507.*cactuBSSN_r*) having ∼50% fraction of memory (load and store) instructions. Later in this section, we will show that a significant fraction of the execution time of these benchmarks is spent in servicing cache and memory requests, which limits their performance.

Table II shows the range of a few performance metrics of the CPU2017 benchmarks measured using hardware performance counters on the Skylake micro-architecture. The magnitude difference between the min and max values shows that there is a lot of diversity in the performance characteristics across different benchmarks. The older SPEC CPU benchmarks have often been criticized because they do not have sufficient instruction cache miss activity as some of the emerging cloud and big-data applications [10], [11]. Interestingly, many CPU2017 benchmarks do not suffer from high instruction cache miss rates, even though the workload sizes have increased significantly.

C. Performance Bottleneck Analysis

In this section, we conduct micro-architectural bottleneck analysis of the CPU2017 applications using cycle per instruction (CPI) stack statistics. A CPI stack breaks down the execution time of an application into different micro-architectural activities (e.g., accessing cache), showing the relative contribution of each activity. Optimizing the largest component(s) in the CPI stack leads to the largest performance improvement. Therefore, CPI stacks can be used to identify sources of micro-architecture inefficiencies. We follow the top-down performance analysis methodology to collect the CPI stack information [12]. Table I also shows the actual CPI numbers for the benchmarks.

Figure 1 shows the CPI stack breakdown of the CPU2017 rate applications (see Table I for the CPI values). The front-end bound category includes the instruction fetch and branch misprediction related stall cycles. The 'other' category includes resource stalls, instruction dependencies, structural dependencies, etc. Several interesting observations can be made from the CPI stack breakdown. In most cases, more than 50% of the total execution time is spent on various types of on-chip micro-architectural activities, with 505.*mcf_r* and 520.*omnetpp_r* having the highest CPI among all the benchmarks. Several benchmarks (e.g., 541.*leela_r*, 505.*mcf_r*, 557.*xz_r*) spend a significant fraction of their execution time on front-end stalls as they suffer from higher branch misprediction rates. The 505.*mcf_r* benchmark further suffers from high instruction cache miss rate, aggravating its front-end performance bottleneck. In general, the integer benchmarks suffer from higher branch misprediction rates than the floating-point benchmarks, leading to higher branch mis-speculation related stalls. In terms of back-end (cache and memory) performance, 520.*omnetpp_r*, 523.*xalancbmk_r*, 505.*mcf_r* and 549.*fotonik3d_r* benchmarks spend a significant fraction of their execution time servicing cache and memory requests. For 526.*blender_r* and 538.*imagick_r* benchmarks, high inter-instruction dependencies are the major cause of pipeline stalls. Most speed benchmarks (not shown here due to space limit) also have similar performance correlations.

TABLE II: Range of important performance characteristics of SPEC CPU2017 benchmarks.

	Rate INT	Speed INT	Rate FP	Speed FP
Metric	**Range (Min - Max)**			
L1D$ MPKI[1]	∼0 - 56	∼0 - 54.7	2 - 95.4	5.5 - 98.4
L1I$ MPKI	∼0 - 5.1	∼0 - 5.2	∼0 - 11.3	0.1 - 11.6
L2D$ MPKI	∼0 - 20.5	∼0 - 20.7	∼0 - 7	0.2 - 8.6
L2I$ MPKI	∼0 - 0.9	∼0 - 0.9	∼0 - 1.2	∼0 - 1.2
L3$ MPKI	∼0 - 4.5	∼0 - 4.6	∼0 - 4.3	∼0 - 5
Branch misp. per kilo inst.	0.9 - 8.3	0.5 - 8.4	0 - 2.5	0.01 - 2.5

[1]MPKI stands for Misses Per Kilo Instructions.

D. Scalability

The SPEC CPU benchmarks have traditionally been single-threaded, however, multithreading is introduced to the SPEC CPU benchmark suite for the first time in the 2017 suite. Hence, it would be interesting to explore the scalability of SPEC CPU 2017 benchmarks. Of the 47 benchmarks offered in SPEC 2017, only the SPECspeed FP benchmarks and the SPECspeed Integer benchmark, 657.*xz_s*, allow to define the number of threads to run on. This section summarizes the runtime of these benchmarks and provides a scalability analysis. This study is performed on an Intel Xeon machine with 6 cores and 12 thread slots.

TABLE III: Runtime of multi-threaded SPEC CPU2017 speed benchmarks (in seconds)

	Number of Threads				
Speed 2017 FP	1	2	4	6	12
603.bwaves_s	11684	6512	4360	3370	2804
607.cactuBSSN_s	4407	2379	1185	853	846
619.lbm_s	1962	1189	1135	1101	1049
621.wrf_s	6515	3474	1913	1405	1154
627.cam4_s	4906	2644	1468	1038	810
628.pop2_s	4852	2507	1378	978	808
638.imagick_s	30729	15661	7732	5181	2708
644.nab_s	4427	3535	1805	1214	879
649.fotonik3d_s	1539	940	849	827	826
654.roms_s	6053	3249	2031	1632	1494

	Number of Threads					
Speed 2017 FP	1	2	4	8	16	32
603.bwaves_s	9625	4795	3168	1868	1060	607
607.cactuBSSN_s	2458	1247	699	394	239	155
619.lbm_s	1553	790	459	266	186	158
621.wrf_s	4262	2325	1311	741	468	366
627.cam4_s	2909	1537	854	533	342	217
628.pop2_s	2972	1503	830	495	314	279
638.imagick_s	23047	11686	5917	3054	1623	866
644.nab_s	4250	2592	1355	677	341	191
649.fotonik3d_s	1284	671	371	224	168	149
654.roms_s	4192	2165	1097	569	305	191

TABLE IV: Speedup of multi-threaded SPEC CPU2017 speed benchmarks

	Number of Threads			
Speed 2017 FP	2	4	6	12
603.bwaves_s	1.794	2.679	3.467	4.166
607.cactuBSSN_s	1.852	3.718	5.166	5.209
619.lbm_s	1.650	1.728	1.782	1.870
621.wrf_s	1.875	3.405	4.637	5.645
627.cam4_s	1.855	3.340	4.637	5.645
628.pop2_s	1.935	3.521	4.961	6.056
638.imagick_s	1.962	3.974	5.931	11.347
644.nab_s	1.252	2.452	3.646	5.036
649.fotonik3d_s	1.637	1.812	1.860	1.863
654.roms_s	1.863	2.980	3.708	4.051

	Number of Threads				
Speed 2017 FP	2	4	8	16	32
603.bwaves_s	2.007	3.038	5.153	9.080	15.857
607.cactuBSSN_s	1.971	3.516	6.239	10.285	15.858
619.lbm_s	1.966	3.383	5.838	8.349	9.829
621.wrf_s	1.833	3.251	5.752	9.107	11.645
627.cam4_s	1.893	3.406	5.458	8.506	13.406
628.pop2_s	1.977	3.581	6.004	9.465	10.652
638.imagick_s	1.972	3.895	7.546	14.200	26.613
644.nab_s	1.640	3.137	6.278	12.463	22.251
649.fotonik3d_s	1.914	3.461	5.732	7.643	8.617
654.roms_s	1.936	3.821	7.367	13.744	21.948

TABLE V: Decreasing Order of Scalability of SPEC FP benchmarks (speed version)

2−threads	4−threads	6−threads	12−threads
638.imagick_s	638.imagick_s	638.imagick_s	638.imagick_s
628.pop2_s	607.cactuBSSN_s	607.cactuBSSN_s	627.cam4_s
621.wrf_s	628.pop2_s	628.pop2_s	628.pop2_s
654.roms_s	621.wrf_s	627.cam4_s	621.wrf_s
627.cam4_s	627.cam4_s	621.wrf_s	607.cactuBSSN_s
607.cactuBSSN_s	654.roms_s	654.roms_s	644.nab_s
603.bwaves_s	603.bwaves_s	644.nab_s	603.bwaves_s
619.lbm_s	644.nab_s	603.bwaves_s	654.roms_s
649.fotonik3d_s	649.fotonik3d_s	649.fotonik3d_s	619.lbm_s
644.nab_s	619.lbm_s	619.lbm_s	649.fotonik3d_s

2−threads	4−threads	8−threads	16−threads	32−threads
603.bwaves_s	638.imagick_s	638.imagick_s	638.imagick_s	638.imagick_s
628.pop2_s	654.roms_s	654.roms_s	654.roms_s	644.nab_s
638.imagick_s	628.pop2_s	644.nab_s	644.nab_s	654.roms_s
607.cactuBSSN_s	607.cactuBSSN_s	607.cactuBSSN_s	607.cactuBSSN_s	607.cactuBSSN_s
619.lbm_s	649.fotonik3d_s	628.pop2_s	628.pop2_s	603.bwaves_s
654.roms_s	627.cam4_s	619.lbm_s	621.wrf_s	621.wrf_s
649.fotonik3d_s	619.lbm_s	621.wrf_s	603.bwaves_s	628.pop2_s
627.cam4_s	621.wrf_s	649.fotonik3d_s	627.cam4_s	619.lbm_s
621.wrf_s	644.nab_s	627.cam4_s	619.lbm_s	649.fotonik3d_s
644.nab_s	603.bwaves_s	603.bwaves_s	649.fotonik3d_s	627.cam4_s

TABLE VI: Level of Scalability.

High	638.imagick_s, 654.roms_s, 607.cactuBSSN_s
Moderate	628.pop2_s, 644.nab_s, 619.lbm_s, 603bwaves_s
Minimal	649.fotonik3d_s, 621.wrf_s, 627.cam4_s,

High	638.imagick_s, 628.pop2_s, 607.cactuBSSN_s
Moderate	621.wrf_s, 654.roms_s, 627.cam4_s, 644.nab_s
Minimal	649.fotonik3d_s, 619.lbm_s, 603.bwaves_s

Table III illustrates the runtime of the multithreaded programs from the CPU 2017 suite with thread counts increasing to the maximum number of threads supported by the platform studied. The corresponding speedups are in Table IV. It is observed that not all SPECspeed 2017 FP benchmarks scale in the same manner. Benchmark 638.*imagick_s* has near-perfect scaling. 619.*lbm_s* and 649.$fotonik3d\_s$ gain almost no speedup from multi-threading. Apart from 638.*imagick_s*, 619.*lbm_s* and 649.$fotonik3d\_s$, other benchmarks show scalability up to 6-thread, and have small increase at 12-thread.

TABLE VII: Program characteristics for similarity analysis.

Characteristics	Metrics
Cache	L1I/D MPKI, L2I/D MPKI, L3 MPKI
TLB	L1I/D TLB MPMI[2], Last level TLB MPMI[3], Page Walks per MI
Branch predictor	Branch MPKI, Branch taken MPKI
Inst Mix	Percentage of Kernel, User, INT, FP Load, Store, Branch, SIMD
Power	Core, LLC and Memory Power

The 638.*imagick_s* is an image manipulation/image processing benchmark, which exhibits a large amount of parallelism. Hence, its high scalability is to be expected. In contrast, 649.$fotonik3d\_s$ is a computational benchmark with multiple sequential steps in its code, making parallel execution harder. Many other benchmarks scale almost linearly up to 6-threads, but only have minor performance improvement going to 12-threads. This can be explained with the resource sharing structure of the machine used for the experiment. The experimental machine has 6 physical cores, which supports up to 12 threads in the SMT (Simultaneous Multi-Threading) form. But executing in the 12-thread configuration leads to much more resource contention than the 6-thread one, because many resources are shared in the SMT mode. This is the main influence that prevents benchmarks from yielding a good scalability at 12-thread. Table V presents the benchmarks in decreasing order of scalability. The programs with the highest and lowest scalability are identified in Table VI.

III. Methodology

To perform a comprehensive analysis of the CPU2017 benchmark suite, we collect and use a large range of program characteristics, related to instruction and data locality, branch predictability, and instruction mix. The profiled characteristics are micro-architecture dependent, which can cause the results to be biased by features of a particular machine. Thus, in order to minimize this bias, measurements are collected on seven commercial machines with three different ISAs (machine details are summarized in Table VIII). The differences in micro-architecture, ISA, and compiler help to eliminate any micro-architectural dependency and allows to capture only the true differences among the benchmarks. The performance metrics used in any subsequent analysis are listed in Table VII. Some of the hardware performance counter data used in this study were measured by the authors, while other data were collected by various SPEC companies on their machines with advanced compilers.

As we perform measurements on seven different machines, we treat each performance counter-machine pair as a metric. Overall, we measure 20 performance-related metrics for each benchmark on every machine, leading to a total of 140 metrics. However, it is difficult to manually look at all the data and conduct meaningful analysis. Hence, we leverage the

[2]MPMI stands for Misses Per Million Instructions.

[3]Depends on the profiled machine, this can be unified or individual.

TABLE VIII: Hardware configurations of 7 machines (Intel, AMD, and Oracle) used in the experiments

Processor	ISA	L1(KB)	L2(KB)	LLC(MB)
Intel Core i7-6700	x86	2x32	256	8
Intel Xeon E5-2650 v4	x86	2x32	256	30
Intel Xeon E5-2430 v2	x86	2x32	256	15
Intel Xeon E5405	x86	2x32	2x6MB	N/A
SPARC-IV+ v490	SPARC	2x64	2MB	32
SPARC T4	SPARC	2x16	128	4
AMD Opteron 2435	x86	2x64	512	6

Principal Components Analysis (PCA) technique [13], [14] to first remove any correlations among the variables (e.g., when two variables measure the same benchmark property). PCA converts i variables X_1, X_2,...,X_i into j linearly uncorrelated variables Y_1, Y_2,...,Y_j, called Principal Components (PCs). Each PC is a linear combination of various features or variables with a certain weight, known as loading factor (see Equation 1).

$$Y_1 = \sum_{k=1}^{i} a_{1k}X_k \;\; ; Y_2 = \sum_{k=2}^{i} a_{2k}X_k \;\; ... \qquad (1)$$

PCA transformation has many interesting properties, the first PC covers most of the variance while other PCs cover decreasing variances. Dimensionality of the data-set can be reduced by removing components with lower variance values. We use the Kaiser Criterion to choose PCs, where only top few PCs are retained, with eigenvalues ≥ 1. After performing PCA, we use another statistical technique called hierarchical clustering to analyze the similarity among benchmarks. The similarity between benchmarks is measured using the Euclidean distance of program characteristics. The results produced by this clustering technique can be presented as a tree or dendrogram. Linkage distances shown in a dendrogram represent similarity between programs (e.g. Figure 2).

IV. Redundancy in CPU2017 Benchmark Suite

A. Subsetting the CPU2017 Benchmarks

We discussed in Section II-B that the dynamic instruction counts of the CPU2017 benchmarks have increased up to 10x versus its predecessor. Such a significant increase in the runtime of benchmarks will make it virtually impossible to perform architectural analysis for the entire CPU2017 benchmark suite on detailed performance simulators in a reasonable time. If similar information can be obtained using a subset of the CPU2017 benchmark suite, it can help architects and researchers to make faster design trade-off analysis. In this section, we study the (dis)similarities between different benchmarks belonging to the SPECrate INT, SPECspeed INT, SPECrate FP and SPECspeed INT categories individually. Linkage distance is used to identify representative subsets of the CPU2017 sub-suites.

Figure 2 shows the dendrogram plot for the SPECspeed INT benchmarks (SPECrate INT, not shown due to space considerations, has a very similar dendrogram). Seven PCs that cover more than 91% of the variance are chosen based on the Kaiser criterion. The x-axis shows the linkage distance between different benchmarks (y-axis). Smaller linkage distance between any two benchmarks indicates that the benchmarks are close, and vice versa. The ordering of benchmarks on the y-axis has no special significance. We can observe that the 605.*mcf_s* and 505.*mcf_r* benchmarks have the most distinct performance features among all the INT benchmarks. The dendrogram plot shown in Figure 2 can be used to identify a representative subset of the SPECspeed INT suite. For instance, if a researcher wants to reduce his simulation time budget to only three benchmarks for the SPECspeed INT category, a vertical line drawn at a linkage distance of 17.5 in Figure 2 can yield a subset of three benchmarks (605.*mcf_s*, 623.*xalancbmk_s* and 641.*leela_s*). For clusters having more than two benchmarks, the benchmark with the shortest linkage distance is chosen as the representative benchmark. Such analysis can be done at varying linkage distances to select the appropriate number of benchmarks when simulation time is constrained. To subset the SPECrate INT benchmark category, we use a similar approach. Overall, only simulating the suggested subsets (summarized in Table IX) can reduce the total simulation time by 5.6× and 4.5× for SPECspeed INT and SPECrate INT suites, respectively.

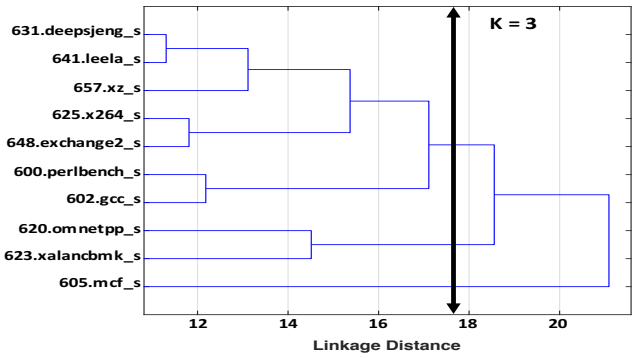


Fig. 2: Dendrogram showing similarity between SPECspeed INT benchmarks.

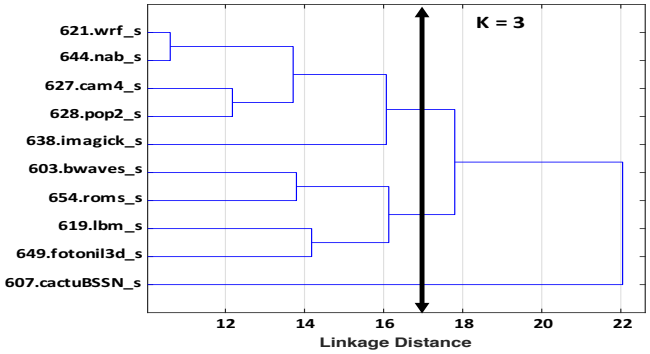


Fig. 3: Dendrogram showing similarity between SPECspeed FP benchmarks.

The dendrograms for the SPECspeed FP and SPECrate FP benchmarks are shown in Figures 3 and 4 respectively. The 607.*cactuBSSN_s* and 507.*cactuBSSN_r* benchmarks have the most distinctive performance characteristics among all the FP benchmarks. Further analysis into the performance characteristics of the two benchmarks reveals that they have unique behavior in terms of their memory and TLB performance. The two vertical lines drawn in Figures 3 and 4 show the points at which 3-benchmark subsets are formed for both the FP suites. Using the benchmark subsets summarized in Table IX reduces

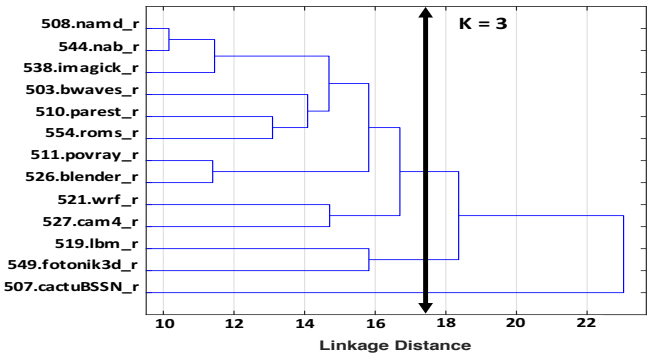


Fig. 4: Dendrogram showing similarity between SPECrate FP benchmarks.

TABLE IX: Representative subsets of the CPU2017 sub-suites.

SPECspeed INT Subset of 3 Benchmarks	605.mcf_s, 641.leela_s, 623.xalancbmk_s
SPECrate INT Subset of 3 Benchmarks	505.mcf_r, 523.xalancbmk_r, 531.deepsjeng_r,
SPECspeed FP Subset of 3 Benchmarks	607.cactuBSSN_s, 621.wrf_s 654.roms_s
SPECrate FP Subset of 3 Benchmarks	507.cactuBSSN_r, 549.fotonik3d_r 544.nab_r

the simulation time by 4.5× and 6.3× for the SPECspeed and SPECrate FP sub-suites, respectively. It is interesting to observe that the chosen subsets contain several newly added benchmarks such as, 544.*nab_r*, 507.*cactuBSSN_r*, 654.*roms_s*, and 607.*cactuBSSN_s*. It should be noted that although this subsetting approach can identify reduced subsets in terms of hardware performance characteristics, it does not guarantee a coverage of all the different application domains of the benchmark suite.

B. Evaluating Representativeness of Subsets

Next, we evaluate the usefulness of the subsets (identified in the last section) to estimate the performance of the CPU2017 benchmark suites on commercial systems, whose results are already published on SPEC's web page.

For this analysis, we record the performance of different benchmarks on different commercial computer systems' (speedup over a *ref* machine) from SPEC's database. Then, we compute the overall performance score (geometric mean) of the benchmark subsets and compare it against the performance score (geometric mean) of all the benchmarks in that sub-suite. For example, for the SPECspeed INT category, we compute the average performance score using the 3-benchmark subset and compare it against the average performance score using all 10 benchmarks belonging to the SPECspeed INT category. Since CPU2017 suite is released very recently, very few companies have submitted the results for all speed and rate categories. Therefore, the different commercial systems used for validating the four benchmark categories are not exactly identical. But, we include all the submitted results obtained from SPEC's web page.

Figure 5 shows the validation results for the SPECspeed INT and SPECrate INT sub-suites. The average error for the SPECspeed INT category is $\leq$1% across 4 systems. For the SPECrate INT category, using a subset of 3 benchmarks

TABLE X: Accuracy comparison among proposed subsets and random subsets.

	Identified subsets	Rand set1	Rand set2
SPECspeed INT	<1%	28.2%	23.4%
SPECrate INT	7%	22.4%	21.7%
SPECspeed FP	3%	49.7%	25.6%
SPECrate FP	4.5%	39.1%	27.1%

achieves an average error of 7% (maximum 12.9%) in terms of speedup as compared to using all the benchmarks. Figure 6 shows similar validation results for the FP categories. Using 3 out of the 10 SPECspeed FP benchmarks produces an average error of 3%, and 3 out of the 13 SPECrate FP benchmarks leads to a 4.5% speedup estimation error. To further evaluate the effectiveness of the proposed subsets, we compare their speedup estimation accuracy with respect to two randomly selected subsets. Results are shown in Table X, where random sets 1 and 2 result in an average error of 34.85% and 24.45% respectively.

The above analysis shows that the identified subsets can accurately predict the performance speedup of the entire benchmark suite. Including more benchmarks in the subset can reduce the prediction error, but will also increase the simulation time significantly. However, only a third of the benchmark suite can be used to predict the performance of the entire benchmark suite reasonably well.

C. Selecting Representative Input Sets

Similar to CPU2006 benchmarks, many CPU2017 benchmarks have multiple input sets. For example, 502.*gcc_r* and 525.*x*264*_r* benchmarks have five and three different input sets, respectively. For a reportable run of such benchmarks, SPEC requires aggregating results across all the different input sets. However, simulating all possible input sets for a benchmark for design trade-off studies can take a prohibitive amount of time. In this section, we want to systematically evaluate differences among the performance characteristics of different input sets belonging to the same benchmark. Such analysis can help researchers to select representative input sets of each benchmark for their evaluation studies, rather than choosing an input set in an ad hoc manner.

Figure 7 shows the dendrogram plot for different INT benchmarks and their input sets. Benchmarks having a single input set are represented by their original names, while benchmarks with multiple input sets are numbered based

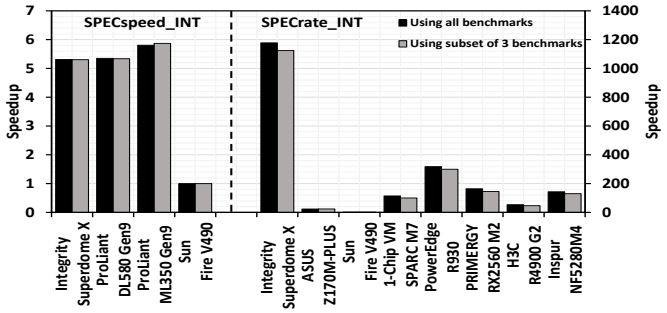


Fig. 5: Validation of SPECspeed and SPECrate INT subsets using performance scores of commercial systems from SPEC's web page.

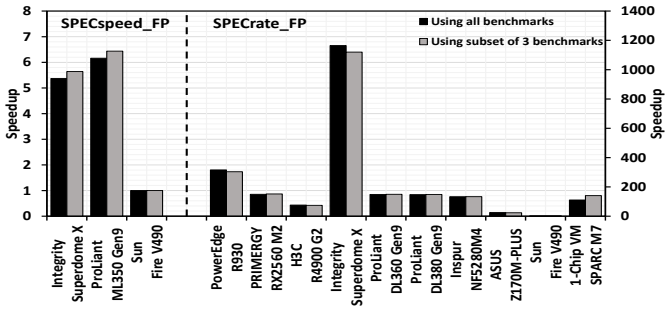


Fig. 6: Validation of SPECspeed FP and SPECrate FP subsets using performance scores of commercial systems from SPEC's web page.

on the output of the *specinvoke* tool. For this analysis, ten PCs are chosen covering 94% of variance using the Kaiser criterion. We can see that for all the benchmarks, different input sets have very similar characteristics. For example, the five different input sets of 502.*gcc_r* are clustered together in the dendrogram plot. This is in contrast to more pronounced variations between the various inputs for 403.*gcc* benchmark in the CPU2006 [14].

We perform similar analysis on the different input sets of the floating-point benchmarks. The 603.*bwaves_s* and

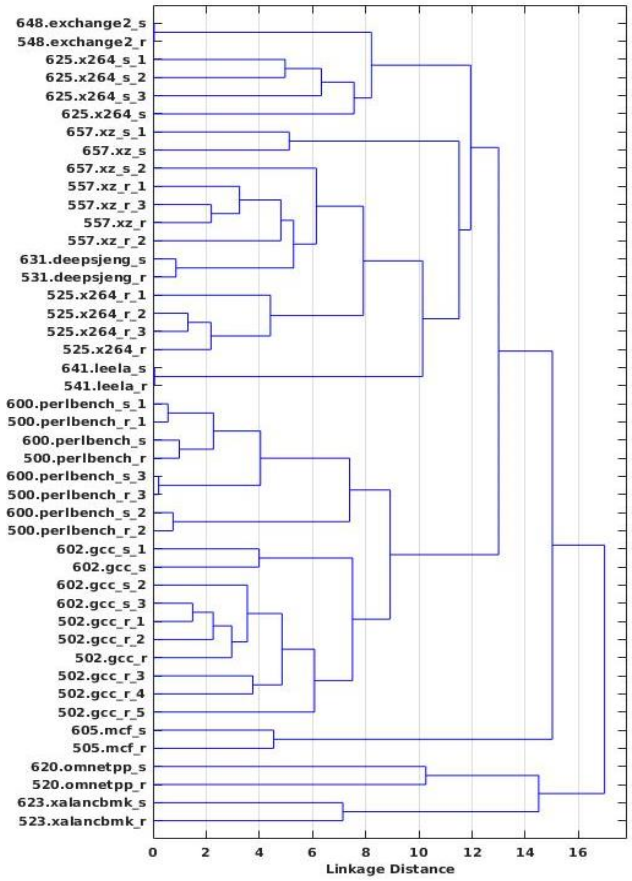


Fig. 7: Dendrogram showing similarity between program input sets of each SPEC 2017 INT benchmark.

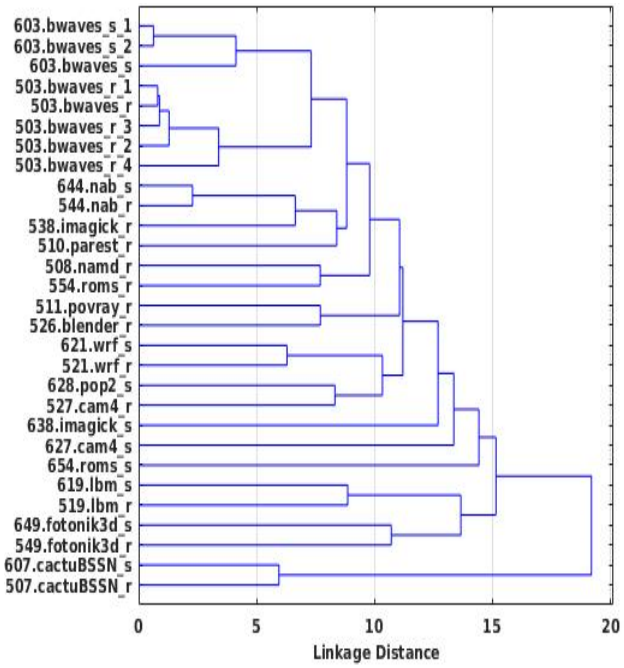


Fig. 8: Dendrogram showing similarity between input sets of each SPEC 2017 FP benchmark.

TABLE XI: List of representative input sets of CPU2017 benchmarks.

SPECrate INT benchmarks	SPECspeed INT benchmarks
500.perlbench_r - input set 1	600.perlbench_s - input set 1
502.gcc_r - input set 2	602.gcc_s - input set 1
525.x264_r - input set 3	625.x264_s - input set 3
557.xz_r - input set 1	657.xz_s - input set 1
SPECrate FP benchmarks	**SPECspeed FP benchmarks**
503.bwaves_r - input set 1	603.bwaves_s - input set 1

502.*bwaves_r* benchmarks are the only two floating-point benchmarks with multiple input sets. Figure 8 shows the similarity between different input sets of the FP programs for both rate and speed categories. Twelve PCs covering 94% of the variance are used for this analysis. To identify the most representative input set of each benchmark, we choose the input set that is closest to the aggregated benchmark run. The most representative input set of each benchmark is summarized in Table XI. This analysis can help researchers in selecting the most representative input set for each benchmark.

D. Are Rate and Speed Benchmarks Different?

So far, our analysis has considered the rate and speed benchmarks separately. With the exception of a few benchmarks (508.*namd_r*, 510.*parest_r*, 511.*povray_r*, 526.*blender_r* and 628.*pop2_s*), most benchmarks are included in both rate and speed categories. Based on the information provided on SPEC's web page, rate and speed benchmarks differ in terms of the workload sizes, compilation flags and run rules. For example, SPEC's web page suggests that the 603.*bwaves_s* benchmark has a memory usage of 11.2 GB versus the 0.8GB usage of the 503.*bwaves_r* benchmark. Similarly, the 605.*mcf_s* and 649.*fotonik3d_s* benchmarks also have significantly higher memory usage than their rate versions. Furthermore, the speed benchmarks have much higher dynamic

instruction counts and runtime than the rate benchmarks. However, do these differences translate into low-level micro-architectural performance variations?

In this section, we use PCA and hierarchical clustering analysis to compare performance characteristics of the rate and speed benchmarks. We will use the dendrogram plots in Figures 7 and 8 for performing this analysis. From the dendrogram plot for the INT benchmarks in Figure 7, we can observe that most benchmarks belonging to the rate and speed categories have very similar performance characteristics. Only three benchmarks (620.*omnetpp_s*, 623.*xalancbmk_s* and 625.*x*264_*s*) have higher linkage distances to their respective rate versions. On the other hand, for the FP benchmarks, many benchmarks have significant differences between the rate and speed versions. The most notable example is the 638.*imagick_s* benchmark, which has $\geq$30% higher misses in all cache levels than the 538.*imagick_r* benchmark, resulting in the largest linkage distance between the two. Also, the high memory usage of 603.*bwaves_s* makes its cache performance significantly different from its rate version. FP benchmarks such as 644.*nab_s*, 621.*wrf_s*, 607.*cactuBSSN_s* etc. have similar performance as their rate equivalents. It should be noted that we consider only single-core performance of the rate and speed benchmarks (we suppress all OPENMP directives in the speed benchmarks).

E. Benchmark Classification based on Branch and Memory Behavior

So far, we have looked at the aggregate performance characteristics of CPU2017 benchmarks based on all the metrics shown in Table VII. However, many times, researchers are interested in studying only particular aspects of program performance, e.g., the control-flow predictor performance, cache performance etc. In this section, we compare different CPU2017 benchmarks in terms of the branch characteristics, data cache and instruction cache performance. This similarity analysis can help to identify important programs of interest when performing branch predictor or cache related studies. We analyze all the CPU2017 benchmarks from the speed and rate categories without classifying them into integer and floating-point groups.

Figure 9 shows the scatter-plot based on the first two PCs of the branch characteristics, covering over 94% of the variance. PC2 is dominated by branch mispredictions per kilo instructions and PC1 is dominated by the fraction of branch instructions and fraction of taken branches. The 541.*leela_r*, 641.*leela_s*, 505.*mcf_r* and 605.*mcf_s* benchmarks have a higher fraction of difficult-to-predict branches, and thus suffer from the highest branch misprediction rates among the different CPU2017 programs. In the CPU2017 suite, 505.*mcf_r*, 605.*mcf_s*, 502.*gcc_r* and 602.*gcc_r* benchmarks have the highest fraction of taken branches. It is also interesting to observe that a majority of C++ benchmarks (e.g., 623.*xalancbmk_s*, 523.*xalancbmk_r*, 620.*omnetpp_s*, 520.*omnetpp_r*) have a higher fraction of taken branches. Also, most floating-point benchmarks are clustered together, while the integer programs show greater diversity in terms of control-flow behavior.

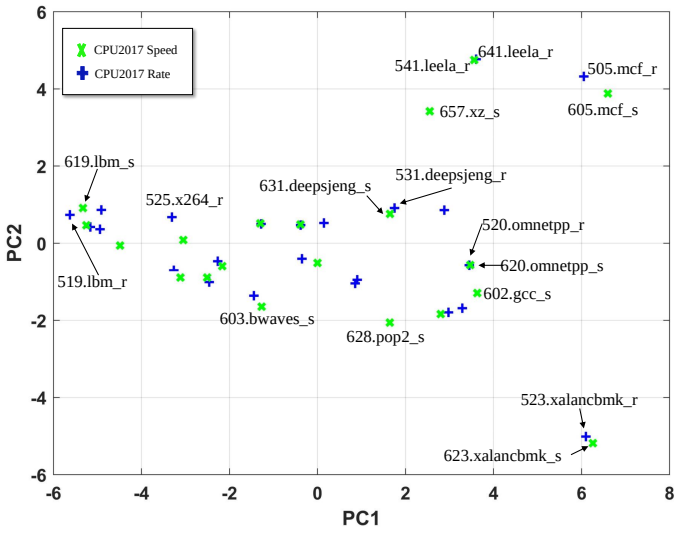


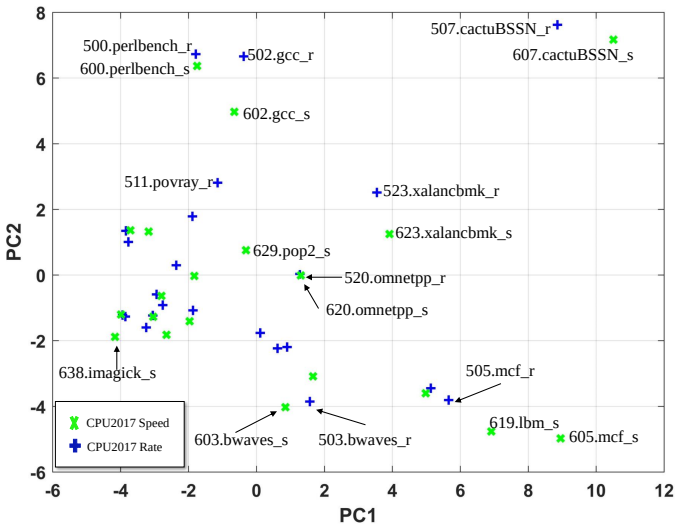
Fig. 9: Comparing CPU2017 benchmarks in the PC workload space based on branch performance metrics.

The PC1 values are dominated by high L1 and L2 data cache miss rates. Thus, benchmarks having higher PC1 values have poor data locality. The benchmarks that experience the highest data cache miss rates among the CPU2017 suite are 605.*mcf_s*, 505.*mcf_r*, 607.*cactuBSSN_s*, 507.*cactuBSSN_r*, 649.*fotonik3d_s* and 549.*fotonik3d_r*. Out of these benchmarks, the *cactuBSSN* and *fotonik3d* benchmarks have been recently introduced in the CPU2017 suite. The PC2 values are dominated by high data cache accesses. The 500.*perlbench_r*, 600.*perlbench_s* and 607.*cactuBSSN_s*, 507.*cactuBSSN_r* benchmarks from CPU2017 suite have a high number of data cache accesses. In the PC3-PC4 plot (see Figure 10), the PC4 values are dominated by instruction cache accesses and misses. SPEC CPU benchmarks have often been criticized as they do not have as much instruction cache activity and misses as some of the emerging big-data and cloud workloads [10], [15], [11]. In general, CPU2017 benchmarks also do not have very high instruction cache miss rates (instruction cache MPKI ranges between 0-11). Nonetheless, the 500.*perlbench_r*, 600.*perlbench_s*, 502.*gcc_r* and 602.*gcc_r* benchmarks have the highest instruction cache access and miss activity.

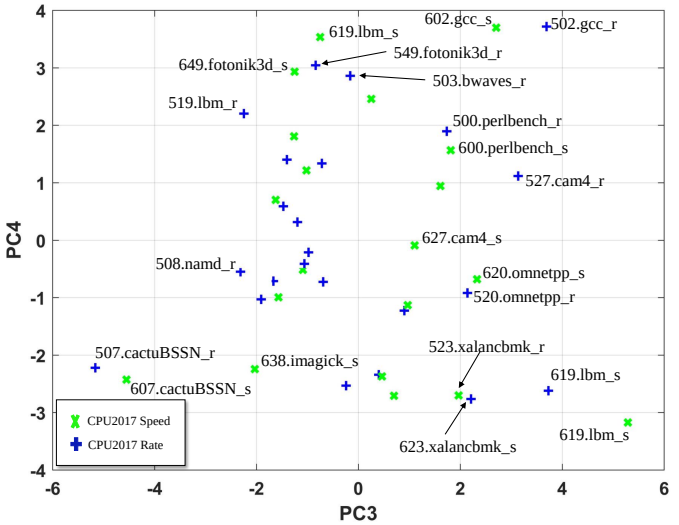
Although this analysis helps in identifying benchmarks that exercise a certain performance metric, care should be exercised when selecting benchmarks for any particular study so that the chosen benchmarks cover the entire workload space. Selecting outlier benchmarks will only emphasize the best-case or worst-case performance behavior, which may lead to misleading conclusions.

F. Difference Between Benchmarks from Same Application Area

In this section, we classify the CPU2017 benchmarks based on their application domain (see Table XII) and seek to find (dis)similarities between different benchmarks belonging to the same category. The benchmarks that are marked in bold in the table have distinct performance behaviors and should be used to cover the performance spectrum for their



(a) PC1 vs PC2



(b) PC3 vs PC4

Fig. 10: CPU2017 (rate and speed) benchmarks in the PC workload space using data and instruction cache characteristics

respective application domain. For those benchmarks which have similar performance behavior in the rate and speed mode, we mark only the rate versions in the table (as they are short-running). For example, in the compiler/interpreter application domain, 502.*gcc_r* and 500.*perlbench_r* have distinct performance characteristics, but are similar to their respective speed equivalents. Thus, running the 502.*gcc_r* and 500.*perlbench_r* benchmarks can represent the performance spectrum of that application domain. As we discussed before, many CPU2017 benchmarks exhibit different behaviors in the rate and speed versions. For example, for the fluid dynamics and climatology domains, both speed and rate versions of the *bwaves*, *roms*, *lbm* benchmarks should be used to achieve comprehensive domain coverage.

TABLE XII: Classification of benchmarks based on application domains.

INT Benchmarks	
App domain	SPEC 2017
Compiler	**502.gcc_r**, 602.gcc_s **500.perlbench_r**, 600.perlbench_s
Compression	**525.x264_r**,**557.xz_r**, **625.x264_s**, 657.xz_s
AI	**531.deepsjeng_r**, 631.deepsjeng_s, **541.leela_r**, 641.leela_s, **548.exchange2_r**, 648.exchange2_s
Combinatorial optimization	**505.mcf_r**, 605.mcf_s
DE Simulation	**520.omnetpp_r**, **620.omnetpp_s**
Doc Processing	**523.xalancbmk_r**, **623.xalancbmk_s**
FP Benchmarks	
App domain	SPEC 2017
Physics	**507.cactuBSSN_r**, **549.fotonik3d_r**, 607.cactuBSSN_s, **649.fotonik3d_s**
Fluid dynamics	**519.lbm_r**, **503.bwaves_r**. **619.lbm_s**, **603.bwaves_s**
Molecular dynamics	**508.namd_r**,**544.nab_r**, 644.nab_s
Visualization	**511.povray_r**,526.blender_r, **538.imagick_r**,**638.imagick_s**
Biomedical	**510.parest_r**
Climatology	**521.wrf_r**, 527.cam4_r, 628.pop2_s, **554.roms_r** 621.wrf_s, 627.cam4_s, **654.roms_s**

V. Balance in the SPEC CPU2017 Benchmark Suites

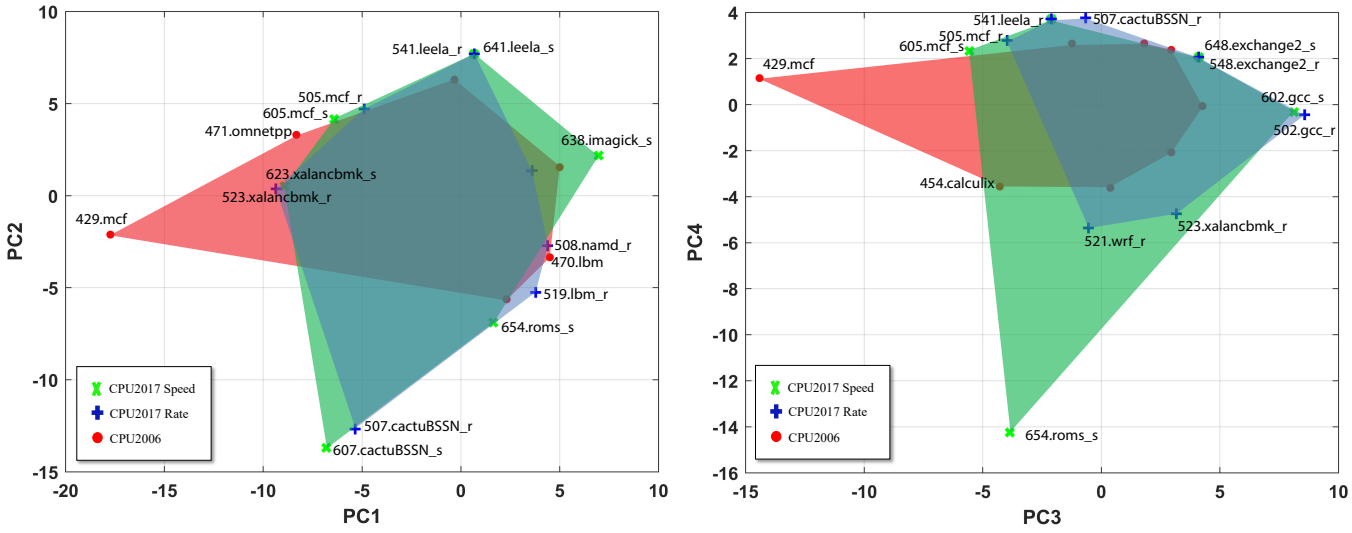
This section compares the CPU2017 benchmarks with the CPU2006 benchmarks and with popular workloads from other domains, such as graph analytics, EDA and data-serving applications. Finally, we also analyze the sensitivity of the CPU2017 benchmarks to different micro-architectural performance characteristics.

A. Comparing Performance Spectrum of CPU2017 & CPU2006 Suites

The CPU2017 suite has revamped many of the benchmarks in the SPEC CPU2006 suite or replaced them with larger/more complex workloads in order to allow stress-testing of powerful modern-day processors and their successors. However, it is not known whether these workloads have different performance demands or whether they stress machines differently compared to CPU2006 benchmarks. Have the new CPU2017 benchmarks managed to expand the workload design-space beyond the CPU2006 benchmarks? Did removing or replacing any CPU2006 benchmarks cause a loss in coverage of the performance spectrum?

Figure 11 shows the scatter plot comparing the CPU2006 and CPU2017 benchmarks based on the top four PCs (covering 80% of the variance), using the performance metrics shown in Table VII. In terms of the PC1-PC2 spectrum, CPU2017 only slightly expands the coverage area; however, more than 25% of the CPU2017 benchmarks fall outside the space covered by the CPU2006 programs. In terms of PC3-PC4 spectrum, the 2017 benchmarks cover twice as much area as the 2006 benchmarks. From these results, we can conclude that the CPU2017 benchmarks are spread farther in the workload space as compared to the CPU2006 benchmarks in terms of performance characteristics, thereby expanding the envelope of the workload design space. The newly added benchmarks, such as 507.*cactuBSSN_r*, 654.*roms_s*, 638.*imagick_s*, 641.*leela_s*, etc., contribute significantly to this increased diversity.

It is also interesting to note that with the exception of a few CPU2017 programs (e.g., 520.*omnetpp_r* and 503.*bwaves_r*),



(a) PC1 vs PC2

(b) PC3 vs PC4

Fig. 11: CPU2017 (rate and speed) and CPU2006 benchmarks in the PC workload space.

which have been retained from the CPU2006 suite, most benchmarks have quite different overall performance characteristics as compared to their predecessors. This implies that the benchmarks have been changed to not only have a higher instruction count and bigger data footprint, but they have also undergone changes in control-flow, instruction and data locality behavior. As an exception, the 429.*mcf* benchmark from the CPU2006 suite, a highly popular benchmark to evaluate cache and memory behavior, exerts the data caches (all cache-levels) more than the *mcf* benchmarks from the CPU2017 suite (the 505.*mcf_r* and 605.*mcf_s* programs).

B. Comparison of Application Domains

Comparing the application domains of the CPU2017 (see Table XII) and CPU2006 benchmarks, we can see that many new application domains have been introduced or greatly expanded in the CPU2017 suite. For example, the artificial intelligence domain has been expanded in the CPU2017 suite to include three new benchmarks. Similarly, 510.*parest_r* benchmark is added to represent the biomedical category. On the other hand, many application domains from the CPU2006 suite have been omitted as well: speech recognition (483.*sphinx*3), linear programming (450.*soplex*), quantum chemistry (e.g., 416.*gamess*, 465.*tonto*), etc.

Loss of an application domain does not necessarily imply a loss in the performance spectrum. Any two benchmarks from different application domains may have similar behavior if they stress similar micro-architectural structures. Similarly, two benchmarks from the same application domain can have very different performance characteristics. Using PCA and hierarchical clustering (see cluster plots in Figure 11), we analyzed every benchmark of the CPU2006 suite, which have been removed from the CPU2017 suite and identify those CPU2006 benchmarks whose performance characteristics are not covered by the CPU2017 benchmarks. Interestingly, we find that only three benchmarks (429.*mcf*, 445.*gobmk* and 473.*astar*) are not covered. The workload space of the remaining removed benchmarks is covered by the CPU2017 benchmarks.

C. Comparing Power Consumption

Next, we compare the power characteristics of the CPU2017 and CPU2006 benchmarks. Power is measured by using RAPL counters available on three different Intel-based micro-architectures (Skylake, Ivybridge, and Broadwell). Figure 12 shows the scatter-plot based on first two PCs (covering more than 84% of the variance). PC1 is dominated by the power spent in DRAM memory and PC2 is dominated by the power spent in the processor cores. Overall, we observe that the CPU2017 benchmarks have much higher coverage space as compared to the CPU2006 benchmarks. It should be noted that many newly added benchmarks (e.g., 648.*exchange*2_*s*, 548.*exchange*2_*r*, 641.*leela_s*, 554.*roms_r*, 557.*xz_r*, and 538.*imagick_r*) contribute to this broader coverage. In general, CPU2006 benchmarks exhibit greater diversity in the PC1 spectrum as compared to the PC2 spectrum. On the other hand, over 20 benchmarks from the CPU2017 suite have significant variations in terms of core power consumption. To the best of our knowledge, CPU2017 benchmarks are more computationally-intensive. This results in the higher diversity in the core power consumption. Therefore, we can conclude that CPU2017 benchmarks can be more useful than CPU2006 benchmarks for power/energy efficiency related studies.

D. Case Study on EDA Applications

Applications from the Electronic Design Automation (EDA) domain were included in early SPEC CPU benchmark suites (e.g., CPU2000). However, EDA benchmarks were removed from the CPU2006 suite. Nonetheless, it has been shown by prior research that CPU2006 suite contains several benchmarks that show similar behavior as the EDA benchmarks [14], which makes the CPU2006 suite balanced even without the EDA applications. No EDA application is included in the CPU2017 suite either. Do the CPU2017 benchmarks cover the

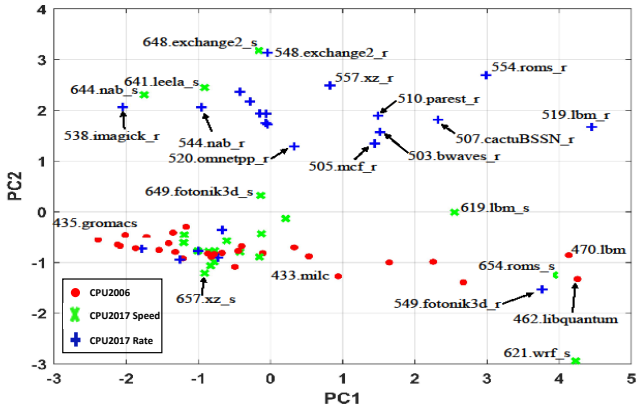


Fig. 12: CPU2017 (rate and speed) benchmarks in the PC workload space using power characteristics.

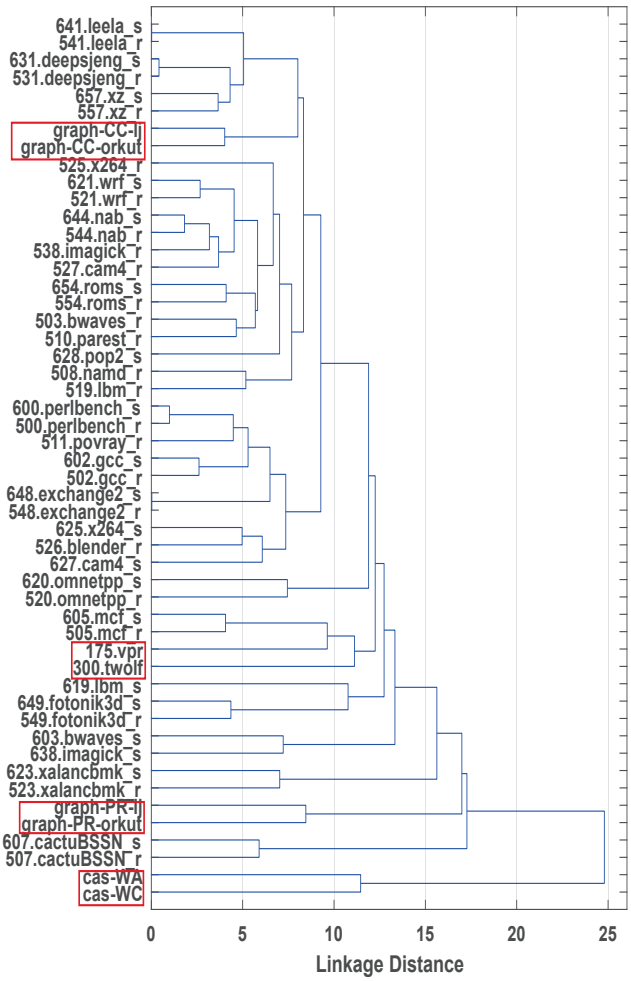


Fig. 13: Similarity among CPU2017, EDA, graph analytics, and database applications.

performance spectrum of the EDA applications? To answer this, we select two benchmarks from the CPU2000 suite: 175.*vpr* and 300.*twolf*. Figure 13 shows the dendrogram plot comparing the CPU2017 benchmarks, EDA benchmarks and several graph analytics and database applications (which we will discuss next). From the figure, we can clearly see that the EDA benchmarks are close to many CPU2017 applications (especially 505.*mcf_r* and 605.*mcf_s*). Therefore, although the EDA application domain is still not included in new CPU2017 suite, the hardware behavior of the EDA applications are well covered.

E. Case Study on Database Applications

The big-data revolution has created an unprecedented demand for efficient data management solutions. While the traditional data management systems were primarily driven by relational database management systems based on the structured query language (SQL), recent years have seen a rise in the popularity of NoSQL databases. Several prior research studies have compared the CPU2006 benchmarks with the database applications and have concluded that their performance characteristics are highly different [15], [16], [10]. In this section, we compare the performance of the CPU2017 benchmarks with a popular NoSQL database, Cassandra [17] running the Yahoo! Cloud Serving Benchmark (YCSB) [18] benchmarks. Figure 13 shows that the database applications (*cas*−*WA* and *cas*−*WC*) also have very different characteristics than the CPU2017 benchmarks. Deep diving into their performance characteristics, we can see that the difference between the two application classes is primarily caused by their instruction cache and instruction TLB performance.

F. Case Study on Graph Applications

Graph processing workloads [19], [20], [21], [22], [23] have recently gained attention from both system and architecture researchers. Many architects have proposed various hardware accelerators [24], [25] to solve the problem of random memory access from hardware side, as it is one of the major bottlenecks for most graph workloads. To test the balance of SPEC 2017 benchmarks, we compare two popular graph analytics workloads with two real-world graphs. Figure 13 shows that pagerank (*pr*) has distinct program characteristics with both graph inputs, having high linkage distance due to high L1 TLB activity caused by random data requests [26], [27]. However, Connected Components (*cc*) has very similar hardware performance behavior to SPEC benchmarks, such as the speed and rate versions of *leela*, *deepsjeng* and *xz*. This shows that the newly added benchmarks improve the balance of the suite. Therefore, missing graph applications in CPU2017 suite have not significantly impacted the overall balance of the CPU2017 suite.

G. Sensitivity of CPU2017 Programs to Performance Characteristics

In this section, we present a classification of different CPU2017 programs based on their sensitivity to branch predictors, data cache and TLB configurations across four different machines. To measure the sensitivity of a program to different branch predictor, cache and TLB configurations, we ranked the different CPU2017 programs based on these characteristics on every machine. The difference in ranks of the same benchmark across all machines is used as an indicator of the sensitivity of the benchmark for a specific characteristic.

Table XIII shows the classification of different CPU2017

TABLE XIII: Sensitivity to branch misprediction rate, L1 D-cache miss rate and TLB miss rate. Benchmarks with low sensitivity are not listed.

Branch Prediction	
High	603.bwaves_s, 503.bwaves_r
Medium	544.nab_r, 521.wrf_r, 511.povray_r, 527.cam4_r, 648.exchange2_s, 623.xalancbmk_s, 621.wrf_s, 602.gcc_s, 627.cam4_s, 628.pop2_s
L1 D-cache	
High	549.fotonik3d_r, 649.fotonik3d_s
Medium	548.exchange2_r, 505.mcf_r, 519.lbm_r, 648.exchange2_s, 627.cam4_s, 607.cactuBSSN_s & 628.pop2_s,
L1 D TLB	
High	503.bwaves_r, 507.cactuBSSN_r, 557.xz_r, 511.povray_r, 657.xz_s, 649.fotonik3d_s, 607.cactuBSSN_s
Medium	526.blender_r, 544.nab_r, 508.namd_r, 549.fotonik3d_r, 500.perlbench_r, 521.wrf_r, 541.leela_r, 527.cam4_r, 531.deepsjeng_r 631.deepsjeng_s, 621.wrf_s, 641.leela_s, 600.perlbench_s, 603.bwaves_s,

programs based on their sensitivity to branch predictor, L1 data cache and TLB configurations. For every characteristic, benchmarks are categorized into low, medium and highly sensitive categories. The most important observations are as follows: both 503.*bwaves_r* and 603.*bwaves_s* show a lot of variation in terms of branch performance. In terms of data cache performance, 549.$fotonik3d\_r$ and 649.$fotonik3d\_s$ show significant performance variability across different machines. In terms of the data TLB performance, the 503.*bwaves_r*, 507.*cactuBSSN_r*, 557.*xz_r*, 511.*povray_r*, 649.$fotonik3d\_s$ and 607.*cactuBSSN_s* benchmarks experience the greatest variability. One should note that having the highest sensitivity to a parameter does not imply that the benchmark has the worst/best behavior in terms of that parameter. For example, 541.*leela_s*, 641.*leela_r*, 657.*xz_s* and 605.mcf_s benchmarks have low sensitivity to branch predictors, because they perform similarly poor across the different machines. In fact, they suffer from the highest misprediction rates across all the systems.

VI. Large Scale Phase Analysis

A. Phase-level Variability

In this section, we explore the phase behavior of the SPEC CPU2017 speed benchmarks, and the same experiments are conducted on the SPEC CPU2006 benchmarks as well for comparison. The prior generations of SPEC CPU benchmarks have exhibited large scale phases [28], [29]. We investigate whether the large scale phases in the SPEC CPU 2017 benchmarks have many fluctuations or are they largely stable.

The analysis is based on statistics of performance counter sampling. We periodically (e.g., 100 milliseconds) read performance event counts (e.g., instructions, cycles) then record them. Later using all the samples from the execution of a benchmark, we calculate three statistical measures: mean, standard deviation and coefficient variation (COV). Mean is the average of a set of samples, and standard deviation quantifies the amount of dispersion of the samples. COV is also known as relative standard deviation, because it is the ratio of the standard deviation to the mean. To some extent, COV removes the influence caused by the differences existing in means, and tries to make the comparison in a fairer manner. (For metrics that is always no less than zero, like IPC and MPKI, standard deviation is somehow proportional to mean.) Programs with large COV indicate more fluctuations.

TABLE XIV: Variability in SPEC CPU2017 speed benchmarks

	IPC			L1 D$ MPKI		
Benchmarks	mean	stdev[2]	COV[3]	mean	stdev	COV
600.perlbench_s	2.90	0.12	0.04	1.43	0.74	0.52
602.gcc_s	1.81	0.34	0.19	12.16	6.09	0.50
605.mcf_s	0.70	0.22	0.31	91.79	67.60	0.73
620.omnetpp_s	1.07	0.13	0.12	29.67	2.69	0.09
623.xalancbmk_s	1.58	0.45	0.28	44.84	6.60	0.15
625.x264_s	2.75	0.06	0.02	0.98	0.63	0.64
631.deepsjeng_s	1.82	0.23	0.13	5.92	23.11	3.90
641.leela_s	1.25	0.04	0.03	4.01	1.04	0.26
648.exchange2_s	2.41	0.03	0.01	0.01	0.00	0.00
657.xz_s	1.50	0.45	0.30	12.48	40.41	3.24
Average of INT	1.78	0.21	0.14	20.33	14.89	1.00
603.bwaves_s	3.11	0.93	0.30	9.17	15.10	1.65
607.cactuBSSN_s	1.27	0.27	0.21	108.50	52.31	0.48
619.lbm_s	0.86	0.15	0.17	80.92	14.67	0.18
621.wrf_s	1.29	0.16	0.12	12.76	11.04	0.87
627.cam4_s	1.81	0.09	0.05	11.73	3.03	0.26
628.pop2_s	1.97	0.08	0.04	26.03	6.05	0.23
638.imagick_s	0.85	0.20	0.24	10.83	3.40	0.31
644.nab_s	1.47	0.15	0.10	9.80	3.09	0.32
649.fotonik3d_s	1.41	0.72	0.51	42.38	25.13	0.59
654.roms_s	1.73	0.50	0.29	32.67	20.85	0.64
Average of FP	1.58	0.33	0.20	34.48	15.47	0.55

Table XIV lists statistics on IPC of SPEC CPU2017 speed applications. Additionally, L1 data cache MPKI is studied as well, because Table II shows that among all the important characteristics having influence on performance, L1 data cache MPKI has the the greatest variation between benchmarks. We choose speed version here, because it only runs single copy of the program as what SPEC CPU2006 does. Hence, we can get rid of such a concern that rate version has more variation because of the interference between multiple copies. For those programs having more than one input, the representative one (see the Table XI) is used. According to our experiments, we observe that some of the new programs, for instance exchange2, have a stable behavior in terms of IPC and L1 data cache MPKI, while some benchmarks, like 605.mcf_s and 603.*bwaves_s*, have more performance variations. Such phase-level analysis reveals a great diversity on the variation level among programs. This phenomenon is observed in SPEC CPU2006 as well.

Benchmarks such as 605.mcf_s, 623.*xalancbmk_s*, 657.*xz_s* show high variations, 600.*perlbench_s*, 641.*leela_s*, 648.*exchange*2_*s* show very low variations, and others are medium. Among the FP programs, 603.*bwaves_s*, 607.*cactusBSSN_s*, 649.$fotonik3d\_s$, and 654.*roms_s* show high fluctuations, while 627.*cam*4_*s*, 628.*pop*2_*s* are very stable.

To facilitate the comparison of CPU2017 with CPU 2006, Table XVI shows the statistics for SPEC CPU2006 benchmarks. In general, the variability of the two suites are comparable. The CPU2017 suite includes some programs (631.$deepsjeng\_s$ and 657.*xz_s*) with very high COV in L1

[2]stdev stands for standard deviation.

[3]COV stands for Coefficient Of Variation.

TABLE XV: Level of Phase Variations in the SPEC CPU 2017 Integer and FP Programs (based on IPC). The omitted ones have medium level of variability.

High (INT)	605.mcf_s, 623.xalancbmk_s, 657.xz_s
Low (INT)	600.perlbench_s, 641.leela_s, 648.exchange2_s
High (FP)	603.bwaves_s, 607.cactusBSSN_s, 649.fotonik3d_s, 654.roms_s
Low (FP)	627.cam4_s, 628.pop2_s

TABLE XVI: Variability in SPEC CPU2006 benchmarks

	IPC			L1 D$ MPKI		
Benchmarks	mean	stdev	COV	mean	stdev	COV
400.perlbench	2.69	0.18	0.07	4.20	1.47	0.35
401.bzip2	1.93	0.38	0.20	8.35	2.38	0.29
403.gcc	1.48	0.80	0.54	50.88	47.95	0.94
429.mcf	0.39	0.18	0.46	154.16	62.39	0.40
445.gobmk	1.38	0.12	0.09	4.96	1.77	0.36
456.hmmer	2.66	0.00	0.00	4.53	0.01	0.00
458.sjeng	1.66	0.07	0.42	2.36	0.60	0.25
462.libquantum	2.47	0.15	0.06	23.71	3.30	0.14
464.h264ref	2.50	0.16	0.06	13.26	4.01	0.30
471.omnetpp	0.86	0.13	0.15	35.95	4.94	0.14
473.astar	1.16	0.74	0.64	28.95	21.29	0.73
483.xalancbmk	1.91	0.61	0.32	27.28	4.53	0.17
Average of INT	1.75	0.29	0.22	29.88	12.89	0.34
410.bwaves	2.03	0.62	0.31	24.74	10.26	0.41
416.gamess	3.38	0.18	0.05	4.82	1.42	0.29
433.milc	0.83	0.34	0.41	26.10	5.27	0.20
434.zeusmp	1.74	0.43	0.25	24.78	11.63	0.47
435.gromacs	2.53	0.29	0.11	11.06	4.23	0.38
436.cactusADM	1.76	0.30	0.17	7.97	0.21	0.03
437.leslie3d	2.21	0.04	0.02	35.81	0.30	0.01
444.namd	2.15	0.16	0.07	10.46	1.95	0.19
447.dealII	2.34	0.91	0.39	15.84	23.04	1.45
450.soplex	1.16	0.08	0.07	53.08	9.27	0.17
453.povray	2.48	0.08	0.07	20.23	1.88	0.09
454.calculix	2.76	0.45	0.16	4.06	7.48	1.84
459.GemsFDTD	1.53	0.18	0.12	33.38	6.03	0.18
465.tonto	2.22	0.45	0.20	8.27	6.39	0.77
470.lbm	1.34	0.04	0.03	51.23	0.94	0.02
481.wrf	2.40	0.57	0.24	11.00	3.28	0.30
482.sphinx3	2.25	0.12	0.05	17.89	2.08	0.12
Average of FP	2.07	0.31	0.16	21.22	5.63	0.41

data cache MPKI compared to the COV in CPU2006 L1 MPKI. The program 456.*hmmer*, has COVs less than 0.01 on both IPC and L1 data cache MPKI. While programs like 473.*astar*, have a COV 2× larger compared to the average.

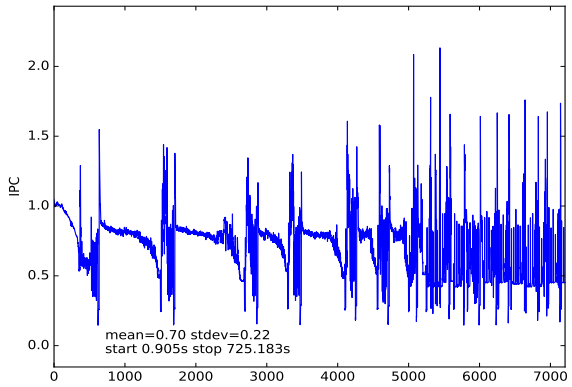


Fig. 14: IPC time varying graph of 605.mcf_s

Here we present two example time varying graphs to illustrate the phase level behaviors, but graphs for other benchmarks could be found in the Appendix A.Figure 14 shows that the average IPC (in a 100ms interval) of 605.*mcf_s*. It varies between 0.15 and 2.15. Obviously, there are several periodical patterns, for example, the curve from about interval 2700 to interval 3300 looks like the curve from interval 3300 to interval 4100. Compared with the integer benchmarks, a few of the floating point benchmarks appear to have frequent fluctuations between a few values. In order to show the phase behavior of those benchmarks, we use scatter plots (as in Figure 15 , where samples are drawn as separated dots, instead of connecting points with lines (with connected dots, there is a large band making it difficult to know what the actual data is) . For example, in the Figure 15 which illustrates the IPC of 649.*fotonik3d_s*, we could see there is a "line" formed by relatively denser samples around 2.0. But the "line" is not continuous and straight, for instance, before and after 5000 seconds, there are two short terms, when the "line" gets broken and promoted a bit more higher. Similarly, there are two more "lines" around 2.5 and 0.8. But there are few samples left apart from those three dense "lines". Graphs for the other 18 benchmarks in the SPEC CPU2017 Speed suite are in Appendix A for readers' further reference.

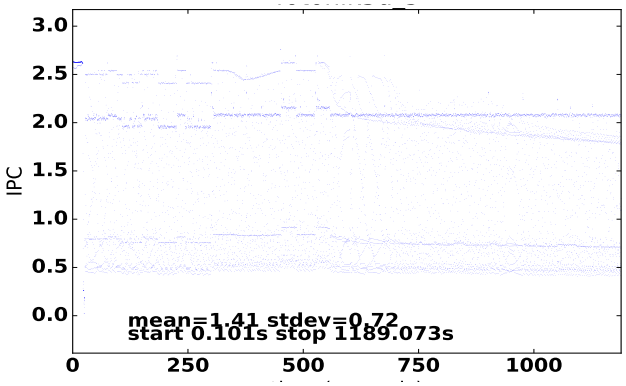


Fig. 15: IPC time varying graph of 649.fotonik3d_s

In conclusion, we can find some periodically repeating patterns and discrete performance levels from the time varying graphs of SPEC CPU2017 benchmarks. Even averaged over 100ms intervals, there are recognizable differences between phases.

Similar to what we did on SPEC CPU2017 speed benchmarks, we analyze the variation and phase-level behavior of SPEC CPU2006 for comparison. The time varying graphs for 12 integer and 17 floating point benchmarks in SPEC CPU2006 suite can be found in the Appendix A. Compared with SPEC CPU2017 benchmarks, more benchmarks in SPEC CPU2006 appear to have more obvious and interesting phase-level behaviors. An interesting one is 473.*astar*, whose time varying graph illustrate 4 very long phases in contrast to frequently fluctuating phases in 454.*calculix*. Others like 401.*bzip*2, 416.*gamess*, 447.*dealII* and 465.*tonto* show periodically repeating patterns through the whole execution.

In summary, phase-level behaviors observed in SPEC CPU2006 suite are similar to what is observed in SPEC CPU2017 suite.

B. Simulation Points

Prior research [30] has shown that a small number of large scale phases can capture the bulk of information in the earlier SPEC CPU benchmark suites. This section presents generated SimPoints of the SPEC CPU2017 benchmarks and compare the SimPoints with prior SPEC CPU suites. Before presenting the data for SPEC CPU2017, we will give a brief overview of how the SimPoints are generated.

The SimPoint generator first slices the program into many equal sized program chunks. For each region, SimPoint measures its Basic Block Vector (BBV), which is a count of how many times a single basic block is executed within a region. The multi-dimensional BBV is compressed into an approximately 16-dimensional vector through a linear transformation. Using this reduced BBV, SimPoint then attempts to find the optimal clustering of the programs regions using K-means clustering with K varying up to a certain maximum value. Once clustered, a single region is chosen from each cluster as a representative SimPoint. More information on SimPoints can be found in the original paper by Sherwood et al. [30]. SimPoint in effect compresses programs into representative regions that can be used to accurately model the overall program behavior with a significantly reduced simulation time.

We generated the SimPoints for SPEC CPU2017 using standard configurations. The actual benchmark binaries are compiled using the default flags outlined on SPEC's website. Once the binaries are compiled we ran the SimPoint generator with the following settings. We set the region size to 100 million instructions and allow a maximum of 32 clusters. It would be interesting to find out whether the new benchmarks contain more distinct phases and will need more clusters to represent the behavior.

We have posted the SimPoint results for SPEC CPU2017 online at our public GitHub repository [1].

Most of the benchmarks show phases with very diverse behavior. Hence it is more appropriate to identify multiple simulation points to capture the variety of behaviors. We ran the SimPoint generator to identify multiple simulation points. The starting instruction count of the SimPoints can be seen in Table XVII. The percentage weights for each SimPoint is also indicated. The SimPoints are presented in descending order of weights. These SimPoints can be simulated in a simulator that supports fast forwarding or by using pinplay replay tools. Tables XVIII and XIX present the number of SimPoints for SPEC CPU2017 and SPEC CPU 2006. The SimPoints are available for download upload from the github repository [31].

Comparing CPU2017 with CPU2006 we can see the SimPoint counts have had minor changes. An interesting observation is that despite the approximately 10× increase in runtimes in CPU 2017, the number of simulation points (distinct phases) have largely stayed the same. The number of SimPoints in the integer benchmarks range from 8 (620.*omnetpp_s*) to 29 (641.*leela_s*), but the average number of SimPoints for the integer benchmarks increased slightly from 16.4 to 19. This is a much smaller increase compared to the 10× increase in the dynamic instruction count. This means that the programs in SPEC CPU2017 have a similar number of phases as SPEC CPU2006 but re-execute those phases repeatedly. This can also be seen in the time varying graphs in Appendix A.

Although the dynamic instruction counts of the floating point benchmarks are large, Table XIX shows that 90% program coverage for floating point on average requires only about 5 simulation points. This is less than half the number needed for the integer benchmarks. There is also less variation in the floating point benchmarks.

[1]GitHub Repo URL: https://github.com/UT-LCA/Scalability-Phase-Simpoint-of-SPEC-CPU2017/releases/tag/v1.0

VII. Related Work

Vandierendonck and Bosschere [32] analyzed the CPU2000 benchmarks and identified a smaller benchmark subset that can accurately predict the performance of the entire suite. Similarly, Giladi and Ahituv [33] found that reducing the SPEC89 suite into 6 programs does not affect the SPEC rating. Phansalkar et al. [14] analyzed the redundancy and benchmark balance in CPU2006 suite. Eeckhout et al. [34], [35] leverage PCA and clustering analysis to select representative program inputs for processor design space exploration. Sherwood et al. [29] proposed to use basic block distribution to find representative simulation points for SPEC CPU 2000 benchmarks. Nair et al. [36] leverage this method to generate simpoints for SPEC CPU 2006 benchmark suite. Moreover, Eeckhout et al. [37] studies the (dis)similarity among these benchmarks to reduce the simulation time for entire suite.

Che et al. [38] compared GPU benchmarks from the Rodinia suite to contemporary CMP benchmarks. Sharkawi et al. [39] performed performance projection of HPC applications using the SPEC CFP06 suite. Woodlee [40] compared the SPEC CPU06 suite with SPEC OMP01 suite to study the transferability between them. Goswami et al. [41] and Ryoo et al. [42] performed comprehensive analysis to explore GPGPU workloads, analyzed their performance spectrum and studied the similarity among different GPGPU benchmark suites. Several research studies [43], [15], [10], [16] characterized big-data benchmarks and found that these benchmarks cannot fully represent real world big data workloads. Wu et al. [44], [45] proposed benchmark suites for emerging mobile platform and perform comprehensive studies in terms of energy, thermal and performance.

VIII. Conclusion

In this paper, we studied the similarities and redundancies among the CPU2017 benchmarks using performance counter based characterization on several state-of-the-art machines. Our analysis shows that using a subset of 3 programs can accurately predict the performance of SPECrate INT, SPECspeed INT, SPECrate FP, and SPECspeed FP sub-suites with ≥93% accuracy. Moreover, we evaluated the representativeness of different input sets of CPU2017 benchmarks, and identified the most representative inputs. We also observed that rate and speed versions of most benchmarks (except *imagick*, $fotonik3d$ etc.) have very similar performance characteristics.

To evaluate the balance in the CPU2017 suite, we compared the application domain coverage, the performance and power spectrum of CPU2017 benchmarks to the CPU2006 benchmarks. We observed that the included CPU2017 pro-

TABLE XVII: Multiple SimPoints of SPEC CPU2017

Benchmarks	Start Instructions(100 million)/Weight(%)
600.perlbench_s	(214681/28.85), (924424/12.60), (1277200/8.97), (1327590/7.96), (58779/7.77), (54633/5.43), (521547/5.39), (1233410/3.88), (752064/2.42), (701679/2.40), (789265/2.19), (1258340/2.06), (236975/1.78), (48882/1.71), (482810/1.44), (467463/0.77), (478321/0.68), (29698/0.68), (199580/0.61), (607466/0.55), (236842/0.54), (1339730/0.53), (1044350/0.30), (790578/0.30), (1141890/0.23)
602.gcc_s	(23077/48.65), (35817/23.62), (21069/12.00), (2812/5.28), (29282/3.60), (1353/1.36), (35911/1.21), (34118/0.87), (24369/0.84), (33794/0.82), (33733/0.69), (36933/0.44), (27933/0.36), (27185/0.17), (910/0.09)
605.mcf_s	(5943/21.24), (16201/12.54), (6315/10.82), (14440/10.75), (587/7.69), (6563/6.97), (13680/6.04), (7430/5.14), (14107/3.40), (12030/3.36), (12099/2.41), (9255/2.25), (7606/2.12), (14669/2.02), (7843/0.93), (15653/0.89), (12189/0.50), (17489/0.42), (9375/0.27), (15180/0.19), (1765/0.07)
620.omnetpp_s	(6053/32.56), (6737/30.32), (2966/15.25), (623/9.00), (1387/6.55), (9548/5.76), (10990/0.51), (10994/0.06)
625.x264_s	(49503/16.54), (22073/14.94), (3910/12.74), (42644/12.32), (10435/12.03), (46339/10.83), (26034/5.69), (41393/4.04), (11557/4.00), (24352/3.17), (4650/1.17), (11998/1.08), (53156/0.96), (31753/0.47)
631.deepsjeng_s	(17909/8.84), (21851/7.87), (84/7.02), (9700/7.00), (20151/6.98), (6204/6.37), (637/5.85), (7853/5.74), (20454/5.69), (15965/5.60), (14515/5.28), (9851/5.05), (3602/4.01), (12151/3.94), (5259/3.73), (11812/2.92), (11347/2.76), (3900/2.53), (15357/2.39), (0/0.46)
641.leela_s	(13198/8.63), (10150/7.15), (9272/6.85), (9870/6.75), (10635/6.57), (6871/5.64), (12252/5.49), (19208/4.29), (7341/4.17), (1830/3.90), (17685/3.67), (6004/3.59), (4934/3.45), (17328/3.37), (4952/3.10), (923/2.66), (21528/2.52), (1463/2.50), (22193/2.40), (1105/2.03), (820/1.91), (12296/1.84), (13929/1.60), (15764/1.47), (1193/1.36), (6041/1.15), (21296/0.98), (3514/0.96), (4/0.04)
648.exchange2_s	(41189/9.81), (46142/9.62), (28949/9.02), (32593/7.77), (46603/7.71), (11758/7.30), (50970/7.10), (27060/6.76), (3146/6.75), (66198/4.31), (38948/3.81), (17591/3.70), (31131/3.13), (20553/3.04), (9923/2.98), (56692/2.95), (424/1.68), (5824/1.63), (10190/0.95)
657.xz_s	(36567/13.00), (16895/9.39), (1400/8.92), (39442/8.76), (38731/6.97), (31742/6.59), (35941/6.33), (6913/5.80), (25357/5.25), (28962/4.09), (35764/3.86), (22587/3.21), (10870/3.02), (45973/2.94), (29995/2.88), (39598/2.72), (45675/2.26), (40484/2.04), (38264/1.88), (18/0.07), (44672/0.02)
603.bwaves_s	(59291/71.46), (248007/14.61), (287430/5.89), (313019/3.23), (315053/2.21), (95707/1.67), (318138/0.34), (263763/0.31), (303136/0.28)
619.lbm_s	(7134/69.69), (10742/14.92), (43825/4.43), (11740/4.36),(6240/3.65), (9400/2.38),(8491/0.56), (0/0.01)
638.imagick_s	(551856/43.83), (394592/27.24), (467091/8.12), (28460/4.97), (91452/4.24), (4209/2.76), (41027/2.50), (515828/2.45), (72623/1.27), (656007/1.21), (651005/0.78), (10504/0.54), (102890/0.04), (272722/0.04), (567401/0.04)
644.nab_s	(9237/34.24), (52046/22.75), (56855/17.91), (58776/9.76), (122599/7.45), (133757/6.91),(48950/0.26), (120710/0.25), (130263/0.16), (104620/0.14), (76009/0.09), (84205/0.08)
649.fotonik3d_s	(19088/43.91), (2468/29.02), (56395/6.90), (1608/4.72), (11626/2.65), (53620/2.21), (17985/2.12), (33097/1.95), (35644/1.20), (38568/0.91), (31846/0.83), (44869/0.83), (3563/0.80), (46262/0.64), (33943/0.49), (43842/0.41), (36949/0.41)

TABLE XVIII: SimPoints of SPEC CPU2006 benchmarks.

Benchmarks	Simulation Points	90 percentile Points	Instructions (billions)
SPEC Integer			
400.perlbench-splitmail	21	12	756.9
401.bzip2-combined	17	13	371.92
403.gcc-scilab	17	9	68.57
429.mcf	14	9	464.98
445.gobmk-trevord.tst	18	13	359.52
456.hmmer-retro.hmm	17	15	2472.91
458.sjeng	16	12	2654.13
462.libquantum	22	15	4534.27
464.h264ref-sss encoder main	20	14	3289.98
471.omnetpp	9	6	787.08
473.astar-rivers.cfg	8	6	961.44
483.xalancbmk	18	13	1401.34
Int Average	16.4	11.4	1510
SPECspeed Floating-point			
410.bwaves	22	10	2780.95
416.games-triazolium	15	11	3717.7
433.milc	23	18	1649.57
434.zeusmp	26	19	2273.56
435.gromacs	20	19	2267
436.cactusADM	21	3	3115.92
437.leslie3d	22	20	4745.74
444.namd	26	18	3293.89
447.dealII	21	14	2809.95
450.soplex-ref.mps	21	17	414.17
453.povray	20	15	1287.36
454.calculix	10	7	8499.78
459.gemsFDTD	20	12	308.88
465.tonto	20	15	3002.2
470.lbm	21	12	1567.55
482.sphinx	20	16	3135.75
FP Average	20.5	14.1	2804

TABLE XIX: SimPoints of SPEC CPU2017 speed benchmarks.

Benchmarks	Simulation Points	90 percentile Points	Instructions (billions)
SPECspeed Integer			
600.perlbench_s	25	13	2696
602.gcc_s	15	5	7226
605.mcf_s	21	11	1775
620.omnetpp_s	8	5	1102
625.x264_s	14	9	12546
631.deepsjeng_s	20	16	2250
641.leela_s	29	21	2245
648.exchange2_s	19	15	6643
657.xz_s	21	15	8264
Int Average	19.1	12.22	4607
SPECspeed Floating-point			
603.bwaves_s	9	3	66395
619.lbm_s	8	4	4416
638.imagick_s	15	6	66788
644.nab_s	12	5	13489
649.fotonik3d_s	17	7	4280
FP Average	12.2	5	24204

grams expand the workload coverage area in terms of both performance and power, especially due to the addition of new benchmarks. Furthermore, an analysis from the perspective of program characteristics shows that the CPU2017 programs offer characteristics broader than the EDA programs' space, some overlap with graph analytics, but do not cover the characteristics of the Cassandra workloads. Since addition of the multithreaded programs is one of the major changes in the CPU2017 suite compared to prior suites, we analyze the scalability of the multithreaded programs. We grouped the multithreaded programs into groups with high, medium, and low scalability. We also study the large scale phases in the programs and identify simulation points for simulation-based computer architecture research.

IX. Acknowledgements

The authors express their gratitude to John Henning for providing technical support/data and to Jeff W. Reilly for facilitating the clustering analysis. The authors would also like to thank Texas Advanced Computing Center (TACC) at UT Austin for providing compute resources. The authors of this work are supported partially by National Science Foundation (NSF) under grant numbers 1725743 and 1745813. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other sponsors.

References

[1] "SPEC CPU2017." https://www.spec.org/cpu2017.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, Aug. 2011.

[3] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marssx86: A full system simulator for x86 cpus," 2011.

[4] C. Jiang, Z. Yu, L. Eeckhout, H. Jin, X. Liao, and C. Xu, "Two-level hybrid sampled simulation of multithreaded applications," *ACM Trans. Archit. Code Optim.*, vol. 12, pp. 39:1–39:25, Nov. 2015.

[5] T. E. Carlson, W. Heirmant, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC*, 2011.

[6] C. Jiang, Z. Yu, H. Jin, C. Xu, L. Eeckhout, W. Heirman, T. E. Carlson, and X. Liao, "Pcantorsim: Accelerating parallel architecture simulation through fractal-based sampling," *ACM Trans. Archit. Code Optim.*, vol. 10, pp. 49:1–49:24, Dec. 2013.

[7] "SPEC CPU2006." https://www.spec.org/cpu2006.

[8] "Linux perf tool." https://perf.wiki.kernel.org/index.php/Main_Page.

[9] A. Phansalkar, A. Joshi, L. Eeckhout, and L. John, "Measuring program similarity: Experiments with spec cpu benchmark suites," *ISPASS*, pp. 10–20, 2005.

[10] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *ASPLOS*, (New York, NY, USA), pp. 37–48, ACM, 2012.

[11] W. Gao, Y. Zhu, Z. Jia, C. Luo, L. Wang, Z. Li, J. Zhan, Y. Qi, Y. He, S. Gong, X. Li, S. Zhang, and B. Qiu, "Bigdatabench: a big data benchmark suite from web search engines," *CoRR*, vol. abs/1307.0320, 2013.

[12] A. Yasin, "A top-down method for performance analysis and counters architecture," *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, vol. 00, pp. 35–44, 2014.

[13] G. Dunteman, *Principal Component Analysis*. Sage Publications, 1989.

[14] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," in *ISCA*, 2007.

[15] R. Panda, C. Erb, M. Lebeane, J. Ryoo, and L. K. John, "Performance characterization of modern databases on out-of-order cpus," in *IEEE SBAC-PAD*, 2015.

[16] R. Panda and L. K. John, "Data analytics workloads: Characterization and similarity analysis.," in *IPCCC*, pp. 1–9, IEEE, 2014.

[17] "Cassandra." wiki.apache.org/cassandra/FrontPage.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, pp. 143–154, 2010.

[19] M. LeBeane, S. Song, R. Panda, J. H. Ryoo, and L. K. John, "Data partitioning strategies for graph workloads on heterogeneous clusters," in *SC*, 2015.

[20] S. Song, M. Li, X. Zheng, M. LeBeane, J. H. Ryoo, R. Panda, A. Gerstlauer, and L. K. John, "Proxy-guided load balancing of graph processing workloads on heterogeneous clusters," in *ICPP*, 2016.

[21] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *OSDI*, 2012.

[22] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng, "Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 125–137, USENIX Association, 2017.

[23] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng, "Exploring the hidden dimension in graph processing," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (Savannah, GA), pp. 285–300, USENIX Association, 2016.

[24] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *HPCA*, 2017.

[25] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[26] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 469–480, ACM, 2017.

[27] Y. Marathe, N. Gulur, J. H. Ryoo, S. Song, and L. K. John, "Csalt: Context switch aware large tlb," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), pp. 449–462, ACM, 2017.

[28] A. A. Nair and L. K. John, "Simulation points for spec cpu 2006," in *IEEE International Conference on Computer Design (ICCD)*, 2008.

[29] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *PACT*, pp. 3–14, 2001.

[30] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, (New York, NY, USA), pp. 45–57, ACM, 2002.

[31] "LCA github repository for Experiments with SPEC CPU 2017: Similarity, Balance, Phase Behavior and SimPoints." https://github.com/UT-LCA/Scalability-Phase-Simpoint-of-SPEC-CPU2017/releases/tag/v1.0.

[32] H. Vandierendonck and K. De Bosschere, "Many benchmarks stress the same bottlenecks," in *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, vol. 2, pp. 57–64, 2004.

[33] R. Giladi and N. Ahitav, "Spec as a performance evaluation measure," *Computer*, vol. 28, pp. 33–42, Aug 1995.

[34] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere, "Workload design: Selecting representative program-input pairs," *PACT*, vol. 0, p. 83, 2002.

[35] M. B. Breughe and L. Eeckhout, "Selecting representative benchmark inputs for exploring microprocessor design spaces," *ACM Trans. Archit. Code Optim.*, vol. 10, pp. 37:1–37:24, Dec. 2013.

[36] A. A. Nair and L. K. John, "Simulation points for spec cpu 2006," in *2008 IEEE International Conference on Computer Design*, pp. 397–403, 2008.

[37] L. Eeckhout, J. Sampson, and B. Calder, "Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation," in *IEEE IISWC*, pp. 2–12, 2005.

[38] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *IISWC*, 2010.

[39] S. Sharkawi, D. DeSota, R. Panda, R. Indukuru, S. Stevens, V. Taylor, and X. Wu, "Performance projection of hpc applications using spec cfp2006 benchmarks," in *IPDPS*, 2009.

[40] E. Ould-Ahmed-Vall, K. A. Doshi, C. Yount, and J. Woodlee, "Characterization of spec cpu2006 and spec omp2001: Regression models and their transferability," in *ISPASS*, 2008.

[41] N. Goswami, R. Shankar, M. Joshi, and T. Li, "Exploring gpgpu workloads: Characterization methodology, analysis and microarchitecture evaluation implications," in *IISWC*, 2010.

[42] J. H. Ryoo, S. J. Quirem, M. Lebeane, R. Panda, S. Song, and L. K. John, "Gpgpu benchmark suites: How well do they sample the performance spectrum?," in *ICPP*, pp. 320–329, 2015.

[43] W. Xiong, Z. Yu, Z. Bei, J. Zhao, F. Zhang, Y. Zou, X. Bai, Y. Li, and C. Xu, "A characterization of big data benchmarks," in *2013 IEEE International Conference on Big Data*, 2013.

[44] D. Pandiyan, S. Y. Lee, and C. J. Wu, "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite - mobilebench," in *IISWC*, pp. 133–142, 2013.

[45] C. Gao, A. Gutierrez, M. Rajan, R. G. Dreslinski, T. Mudge, and C. J. Wu, "A study of mobile device utilization," in *ISPASS*, pp. 225–234, 2015.

APPENDIX

In the Appendix, we want to include more insights and results gained from experiments about the SPEC CPU2017. This section contains the benchmarks' scalability analysis, phase-level behaviors, and simulation points (SimPoints) we generated for public uses.

A. Supplemental Graphs and Tables

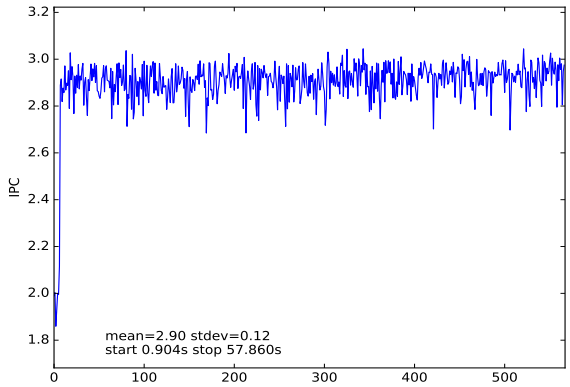


Fig. 16: IPC time varying graph of 600.perlbench_s

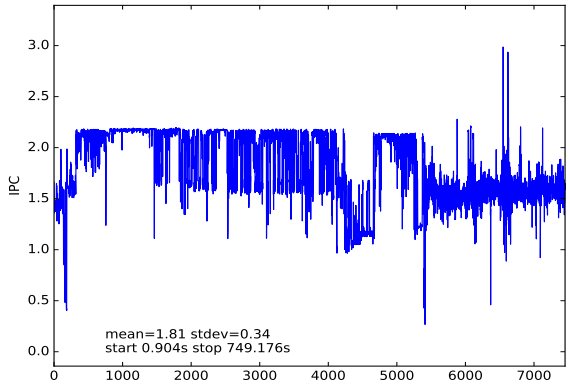


Fig. 17: IPC time varying graph of 602.gcc_s

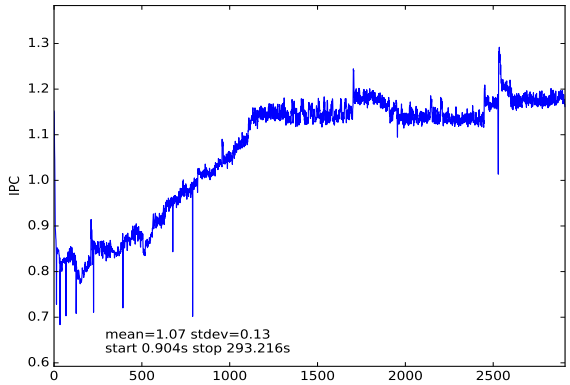


Fig. 18: IPC time varying graph of 620.omnetpp_s

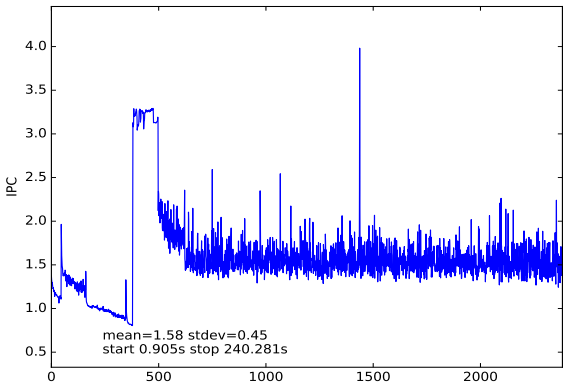


Fig. 19: IPC time varying graph of 623.xalancbmk_s

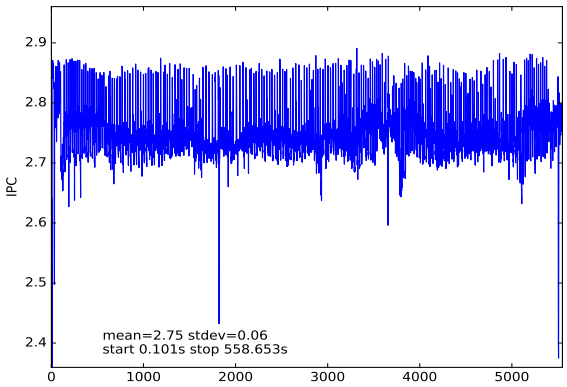


Fig. 20: IPC time varying graph of 625.x264_s

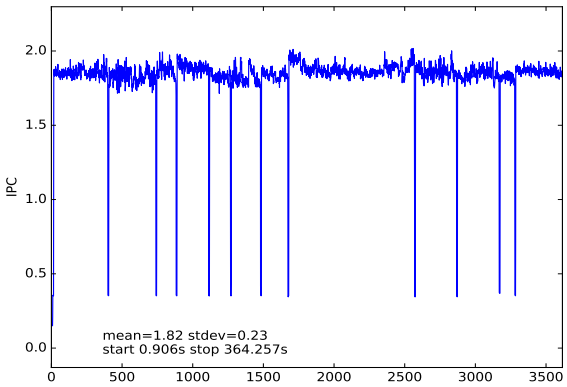


Fig. 21: IPC time varying graph of 631.deepsjeng_s

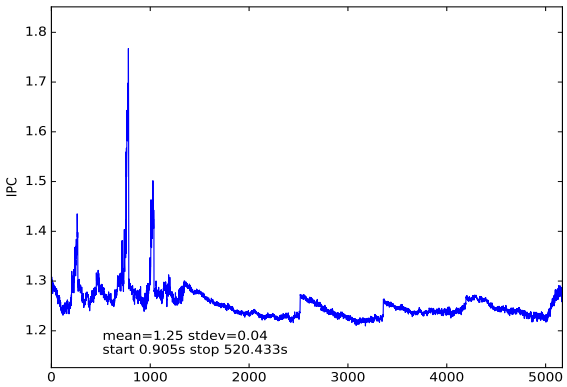


Fig. 22: IPC time varying graph of 641.leela_s

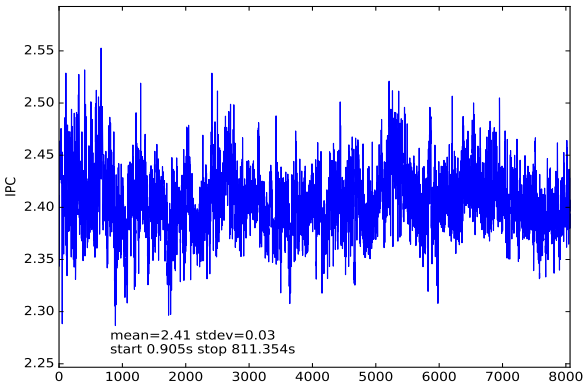


Fig. 23: IPC time varying graph of 648.exchange2_s

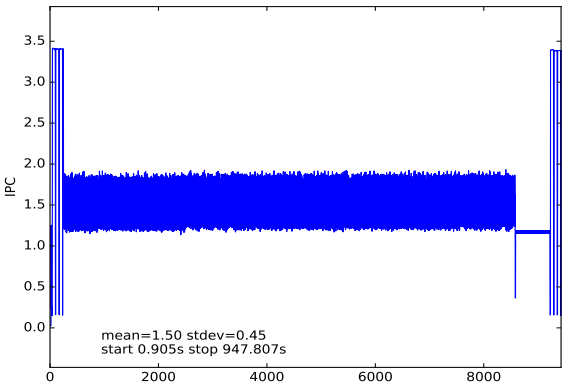


Fig. 24: IPC time varying graph of 657.xz_s

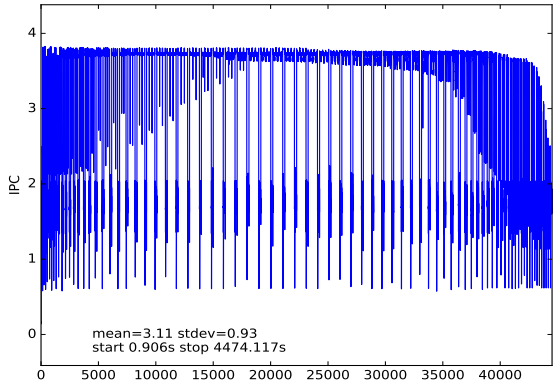


Fig. 25: IPC time varying graph of 603.bwaves_s

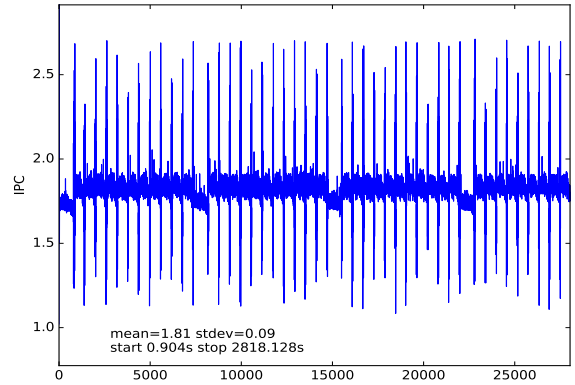


Fig. 26: IPC time varying graph of 627.cam4_s

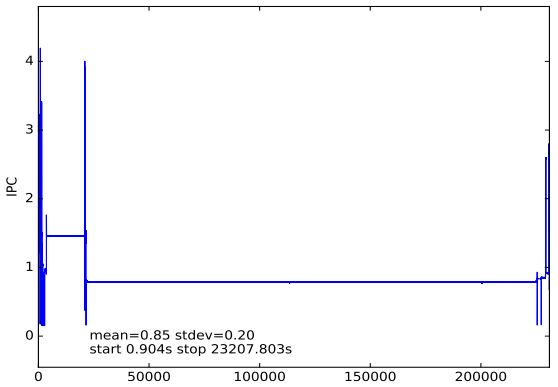


Fig. 27: IPC time varying graph of 638.imagick_s

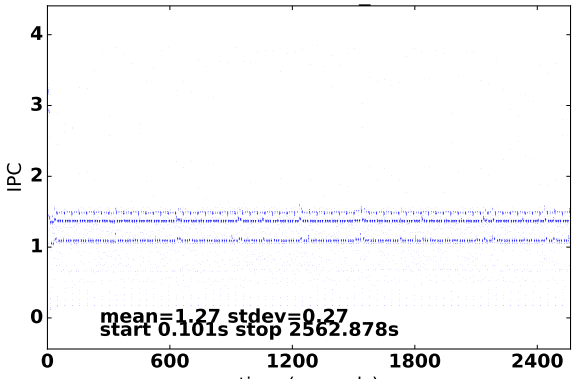


Fig. 28: IPC time varying graph of 607.cactuBSSN_s

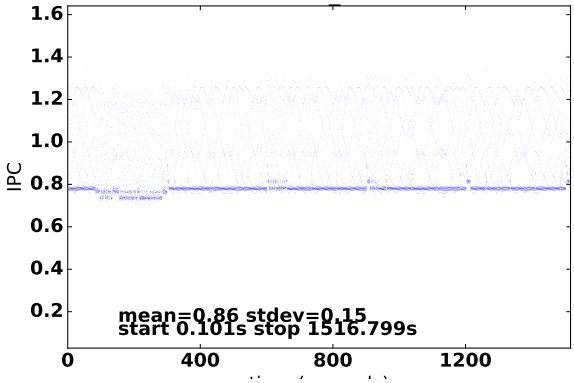


Fig. 29: IPC time varying graph of 619.lbm_s

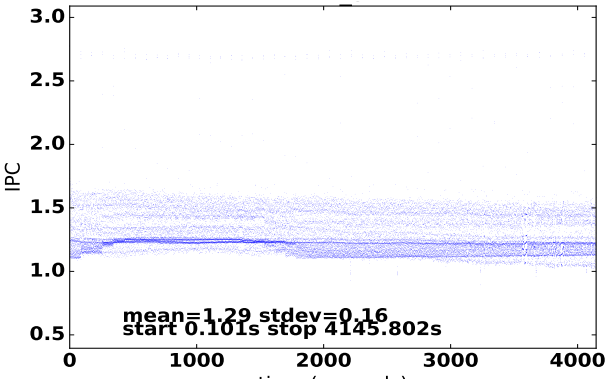


Fig. 30: IPC time varying graph of 621.wrf_s

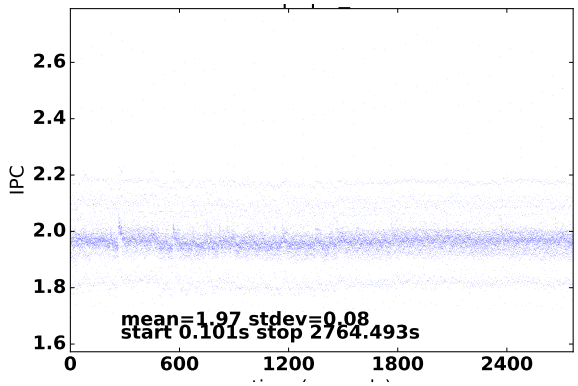


Fig. 31: IPC time varying graph of 628.pop2_s

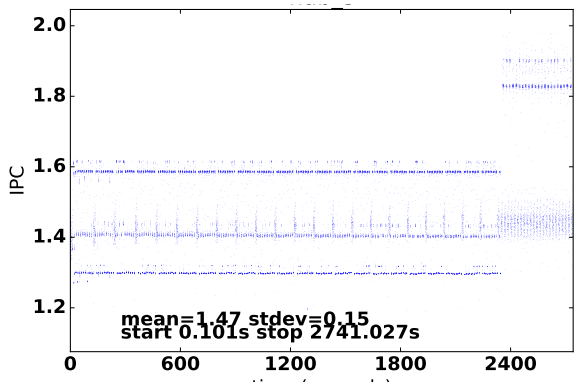


Fig. 32: IPC time varying graph of 644.nab_s

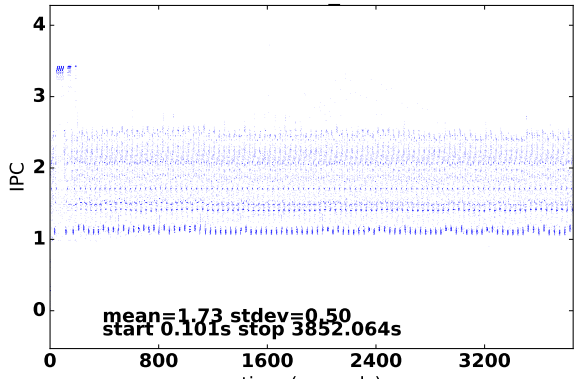


Fig. 33: IPC time varying graph of 654.roms_s

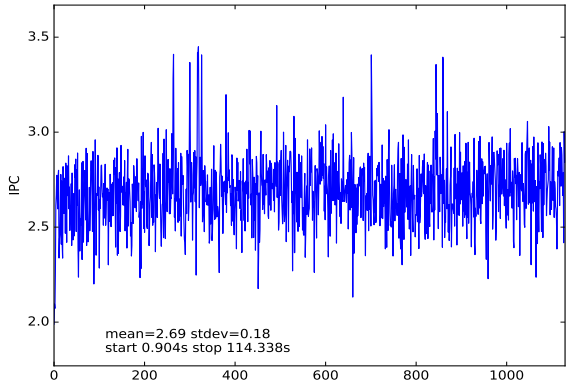


Fig. 34: IPC time varying graph of 400.perlbench

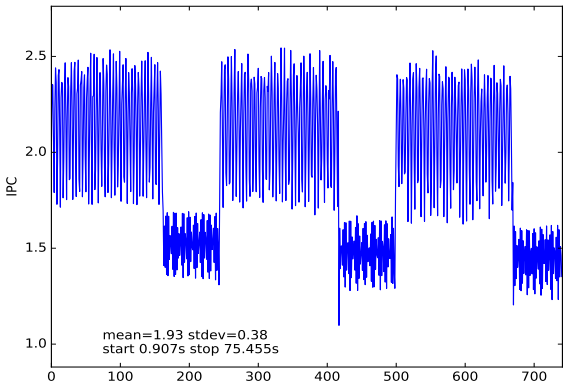


Fig. 35: IPC time varying graph of 401.bzip2

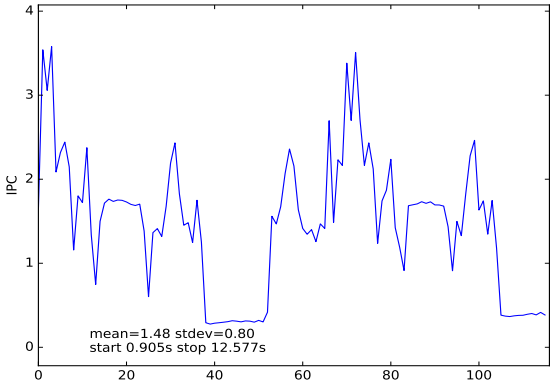


Fig. 36: IPC time varying graph of 403.gcc

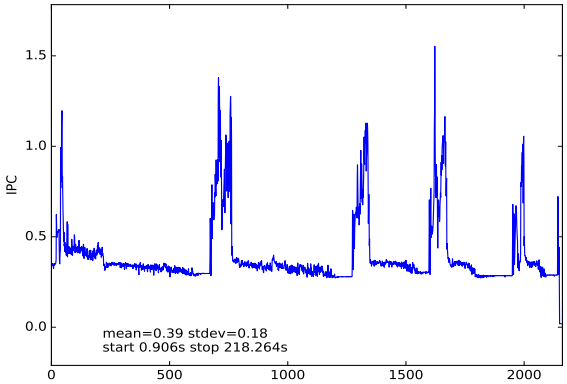


Fig. 37: IPC time varying graph of 429.mcf

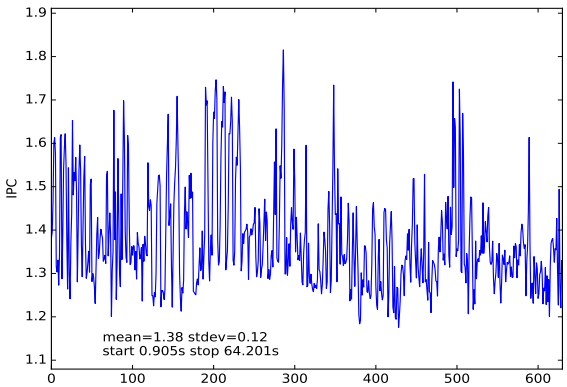


Fig. 38: IPC time varying graph of 445.gobmk

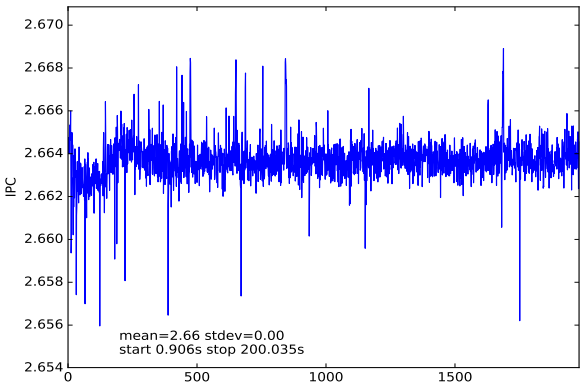


Fig. 39: IPC time varying graph of 456.hmmer

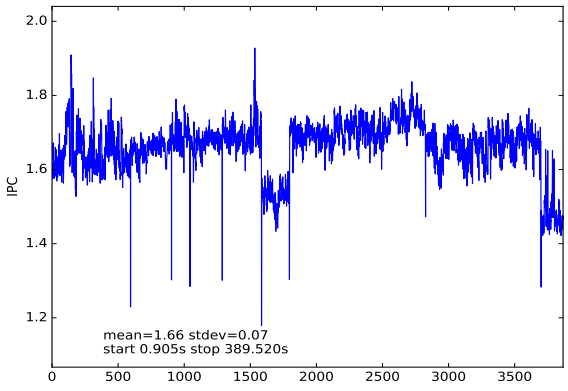


Fig. 40: IPC time varying graph of 458.sjeng

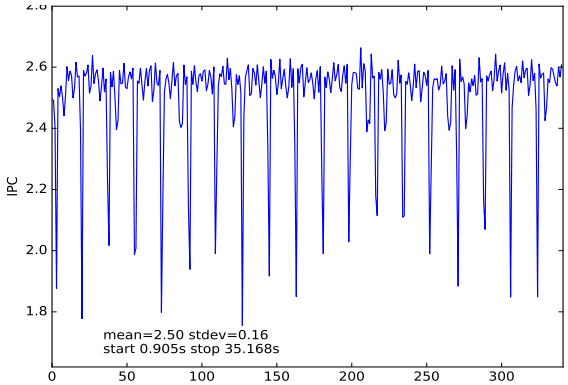


Fig. 41: IPC time varying graph of 464.h264ref

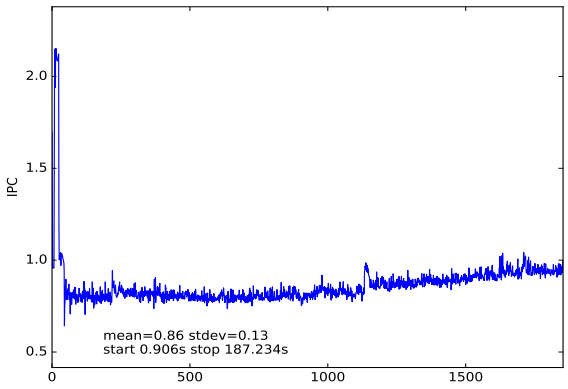


Fig. 42: IPC time varying graph of 471.omnetpp

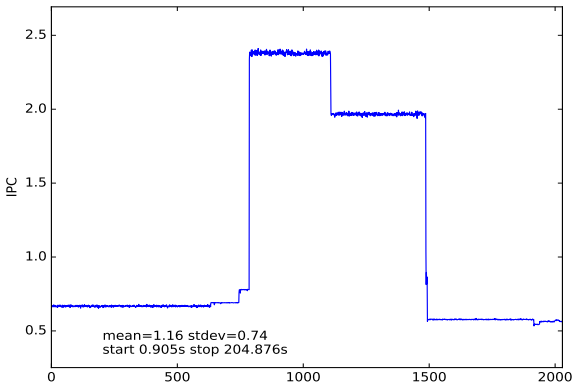


Fig. 43: IPC time varying graph of 473.astar

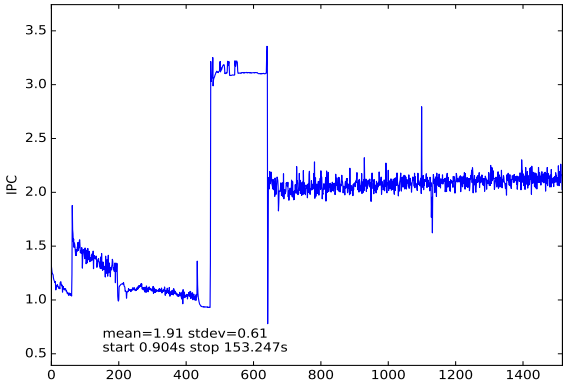


Fig. 44: IPC time varying graph of 483.xalancbmk

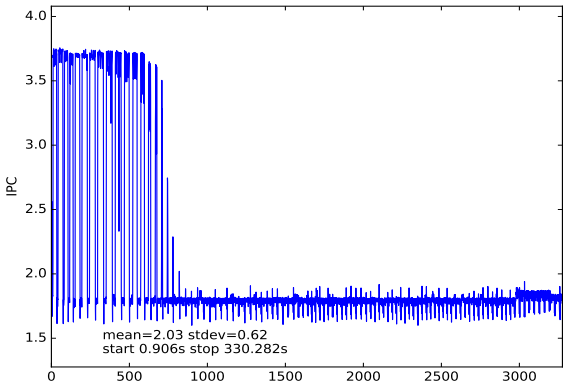


Fig. 45: IPC time varying graph of 410.bwaves

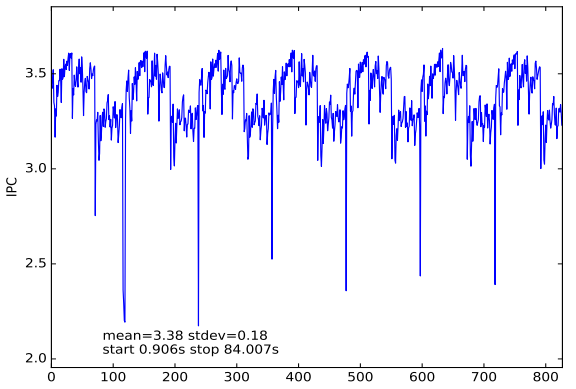


Fig. 46: IPC time varying graph of 416.gamess

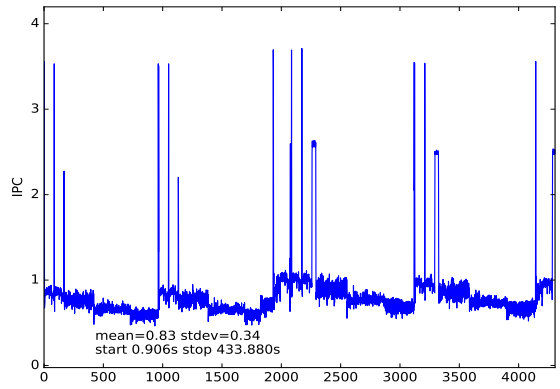


Fig. 47: IPC time varying graph of 433.milc

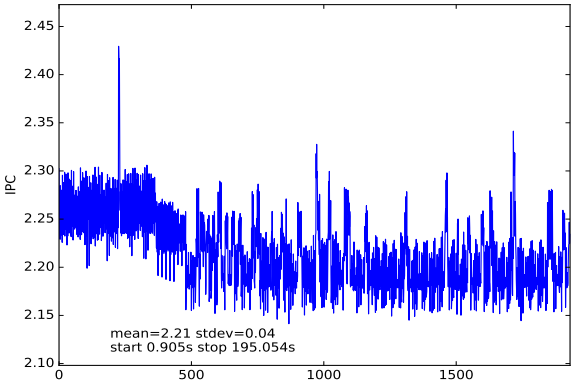


Fig. 48: IPC time varying graph of 437.leslie3d

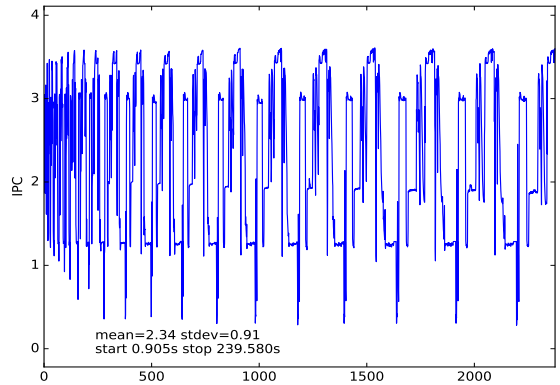


Fig. 49: IPC time varying graph of 447.dealII

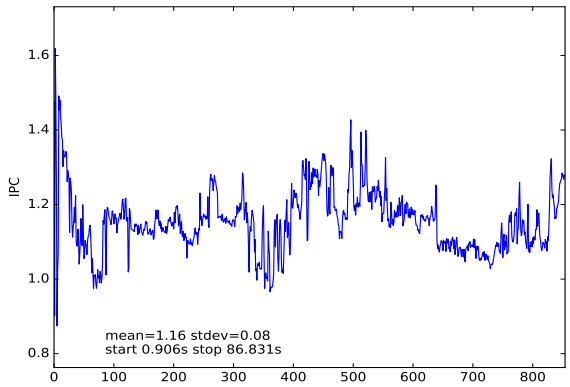


Fig. 50: IPC time varying graph of 450.soplex

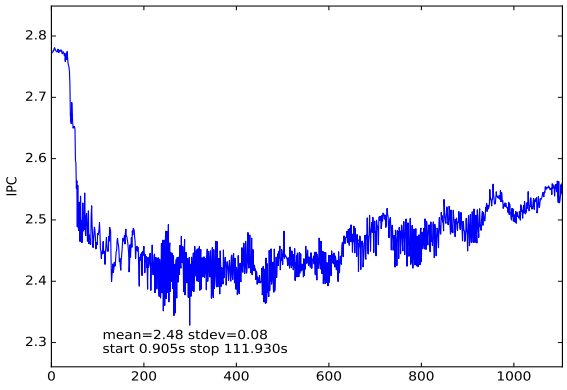


Fig. 51: IPC time varying graph of 453.povray

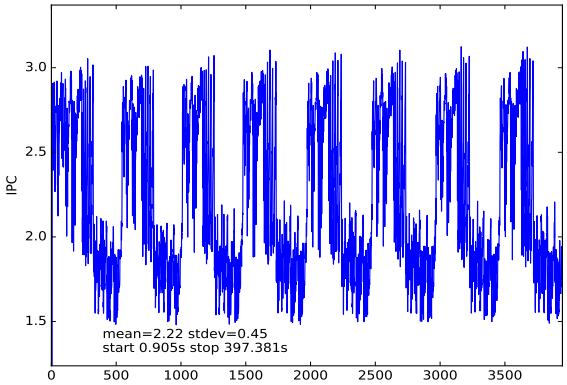


Fig. 52: IPC time varying graph of 465.tonto

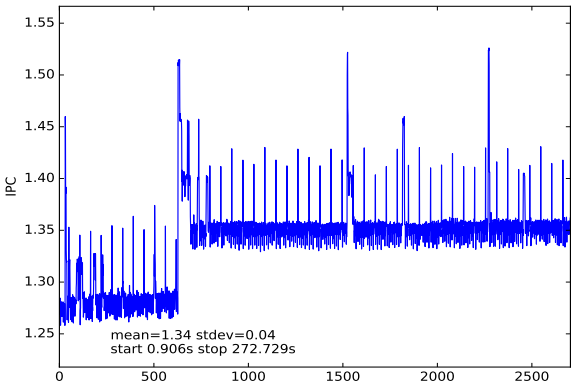


Fig. 53: IPC time varying graph of 470.lbm

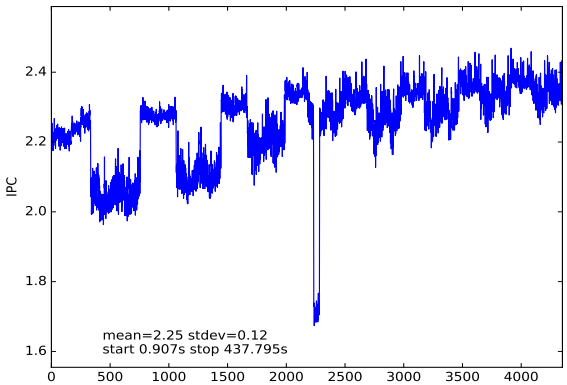


Fig. 54: IPC time varying graph of 482.sphinx3

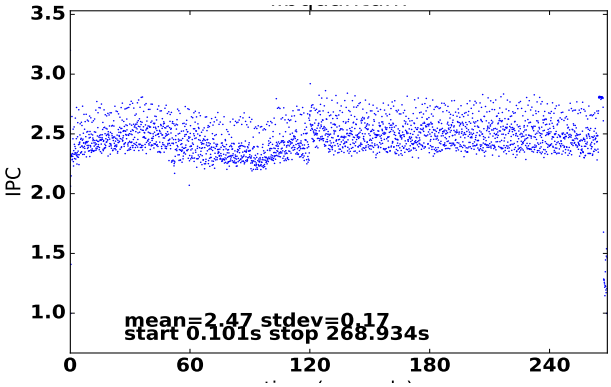


Fig. 55: IPC time varying graph of 462.libquantum

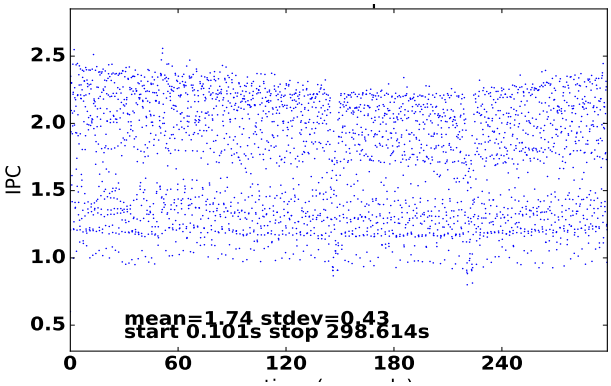


Fig. 56: IPC time varying graph of 434.zeusmp

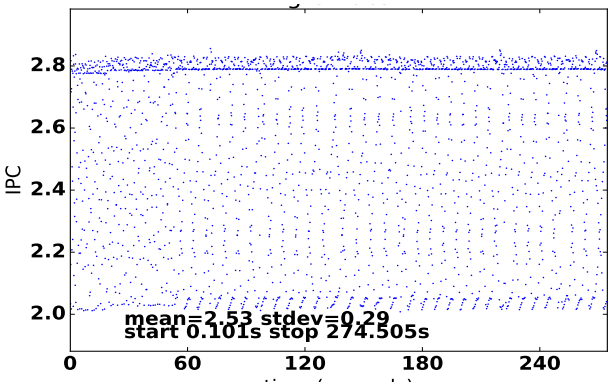


Fig. 57: IPC time varying graph of 435.gromacs

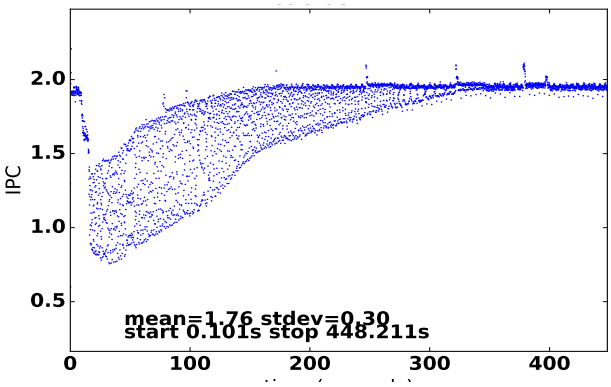


Fig. 58: IPC time varying graph of 436.cactusADM

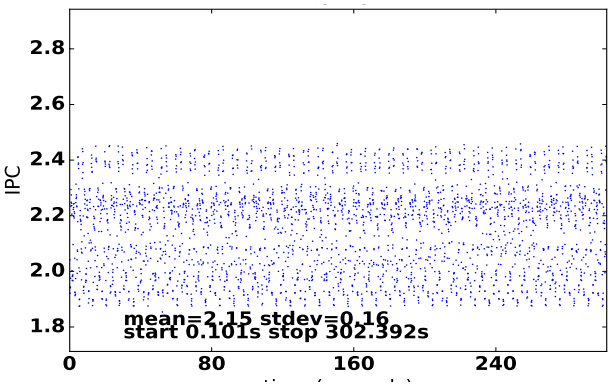


Fig. 59: IPC time varying graph of 444.namd

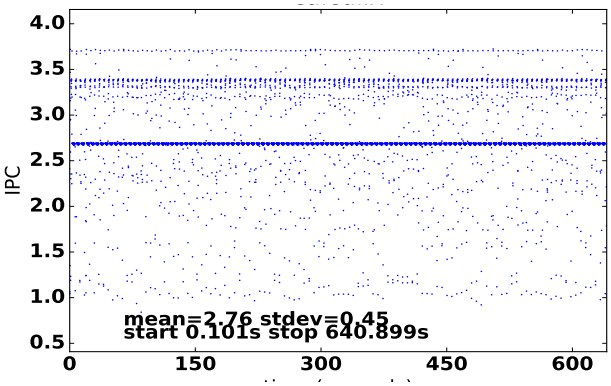


Fig. 60: IPC time varying graph of 454.calculix

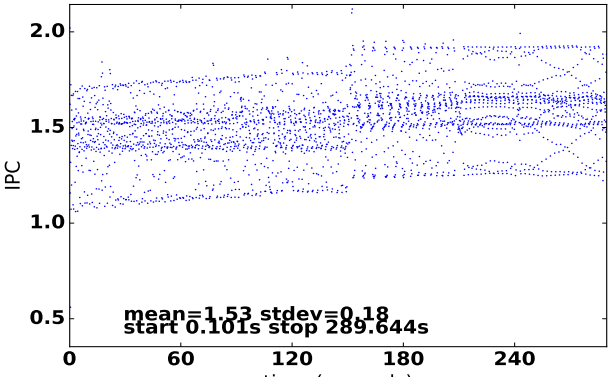


Fig. 61: IPC time varying graph of 459.GemsFDTD

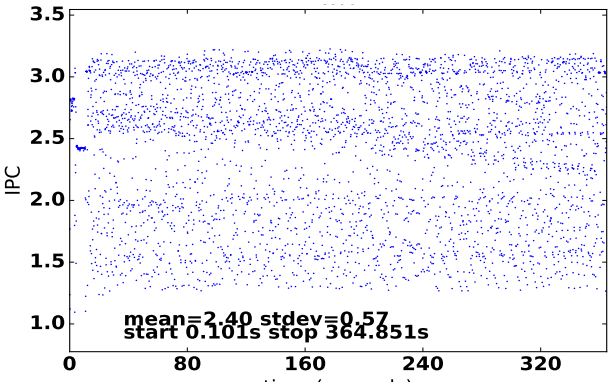


Fig. 62: IPC time varying graph of 481.wrf