2018 IEEE International Symposium on Performance Analysis of Systems and Software

A Workload Characterization of the SPEC CPU2017 Benchmark Suite

Ankur Limaye and Tosiron Adegbija
Department of Electrical and Computer Engineering
University of Arizona, Tucson, Arizona 85721
Email: {ankurlimaye, tosiron}@email.arizona.edu

***Abstract*—The Standard Performance Evaluation Corporation (SPEC) CPU benchmark suite is commonly used in computer architecture research and has evolved to keep up with system microarchitecture and compiler changes. The SPEC CPU2006 suite, which remained the state-of-the-art for 11 years, was retired in 2017, and is being replaced with the new SPEC CPU2017 suite. The new suite is expected to become mainstream for simulation-based design and optimization research for next-generation processors, memory subsystems, and compilers. In this paper, we extensively characterize the SPEC CPU2017 applications with respect to several metrics, such as instruction mix, execution performance, branch and cache behaviors. We compare the CPU2017 and the CPU2006 suites to explore the workload similarities and differences. We also present detailed analysis to enable researchers to intelligently choose a diverse subset of the CPU2017 suite that accurately represents the whole suite, in order to reduce simulation time.**

***Index Terms*—SPEC CPU2017, workload characterization, benchmarks, computer architecture, performance analysis, microprocessor optimization.**

I. Introduction

Benchmarking is one of the critical stages of the continuous development cycle of compilers and system microarchitectures. A set of representative applications (collectively referred to as 'benchmark suite') is executed on the computer system to either identify potential optimization opportunities in the system software or hardware, or to qualitatively determine the benefits of the optimizations performed. There are several benchmark suites available that focus on different application domains. For example, the EEMBC [1] and MiBench [2] benchmark suites focus on embedded system applications in different domains like automation, communication, and office software. The SPLASH [3] and PARSEC [4] benchmark suites, comprise of multi-threaded workloads and evaluate a processor's multi-threading capabilities.

One of the most widely used benchmark suites for general-purpose high-performance computing research is the Standard Performance Evaluation Corporation (SPEC) CPU [5] benchmark suite. The SPEC CPU2006 benchmark suite [6], for example, is cited in over 7,250 research articles and has over 43,000 published results on the SPEC website. The SPEC CPU2006 suite, which remained the state-of-the-art for almost 11 years, is being replaced with the new SPEC CPU2017 suite [7], released in June 2017[1]. The new suite will significantly influence design and optimization research for next-generation microprocessors, memory subsystems, and compilers.

Given a new benchmark suite like the CPU17, several practical questions must be answered to assist researchers in making decisions about the appropriateness of the suite for their research. First, it is essential to understand the suite's workload characteristics (e.g., memory characteristics, branches, instruction mix, etc.). It is also well-known that SPEC benchmarks typically take a long time to run—potentially, several days or weeks—when using simulators. As a result, it is common practice, especially in computer architecture research, to select a subset of the suite's applications for use [8], [9]. However, an arbitrary subset can lead to incorrect or inaccurate results and inferences [9], [10]. The subset must be carefully selected to maximize the variety of execution characteristics, and minimize redundancy in the applications. Furthermore, some of the benchmarks have multiple input data for each application, and the choice of application-input pairs is often arbitrary. The CPU17 can also have cost implications, especially for non-commercial use. Since the suite is not open-source, researchers who already have the CPU06 suite may be asking themselves if purchasing the CPU17 suite provides any significant academic/research benefits. Our goal in this work is to take a step towards addressing these questions regarding the new CPU17 suite.

This paper presents the first analysis—to the best of our knowledge—of the execution and performance characteristics of the applications in the CPU17 benchmark suite. We use hardware performance counter statistics from a modern computing system to analyze the suite with respect to several metrics, such as instructions per cycle (IPC), branch and cache behaviors, etc. We also briefly compare the CPU17 and CPU06 suites to derive insights into the overall similarities and differences between both suites. Using statistical analysis techniques—Principal Components Analysis (PCA) and Hierarchical Clustering—we present detailed analysis to enable researchers to identify the redundancies among the CPU17 applications, and select a subset that represents the whole suite. To enable researchers to reduce simulation time, we also suggest a representative subset of the CPU17 applications.

The rest of the paper is organized as follows: Section II presents a brief overview of the CPU17 benchmark suite. Section III overviews the experimental setup and evaluation methodology for the workload characterization, and the results

[1]Hereafter, we refer to the SPEC CPU2006 and SPEC CPU2017 benchmark suites as CPU06 and CPU17 suites respectively.

0-7695-6375-9/18/$31.00 ©2018 IEEE
DOI 10.1109/ISPASS.2018.00028

and analysis are presented in Section IV. For each characteristic discussed, we also present an overall comparison of the CPU17 suite with CPU06. Finally, Section V provides the discussion and methodology for generating a diverse, yet representative, subset of the CPU17 benchmark suite.

II. CPU17 Suite: A Brief Overview

The SPEC CPU benchmark suites have evolved since the first release of the CPU89 benchmark suite in order to keep up with changes in system hardware and software [11]. The CPU17 benchmark suite—released in June 2017—is the fifth generation of the SPEC CPU benchmark suites. The CPU17 suite features updated and improved applications, multi-threading options for a few applications using OpenMP, and an optional power consumption measurement metric [7]. The applications in the CPU17 suite have increased in size and complexity. As compared to the CPU06 suite, the CPU17 suite's C++, C, and Fortran lines of code have increased by approximately 2×, 2.5× and 1.5×, respectively.

The CPU17 suite has 43 applications, organized into four 'mini-suites': 'SPECrate 2017 Integer', 'SPECrate 2017 Floating Point', 'SPECspeed 2017 Integer' and 'SPECspeed 2017 Floating Point' suites[2]. The list of CPU17 benchmark suite applications can be found in the SPEC CPU17 Documentation [12]. The *rate_int* and *speed_int* suites consist of 10 applications each, while the *rate_fp* and *speed_fp* consist of 13 and 10 applications, respectively. Most of the applications have both *rate* and *speed* versions (denoted with '_r' and '_s', respectively), except for `508.namd_r`, `510.parest_r`, `511.povray_r` and `526.blender_r`, which only have the *rate* versions, and `628.pop2_s`, which only has the *speed* version.

The *speed_fp* suite applications and `657.xz_s` have an optional multi-threading capability using OpenMP. The performance characteristics and differences between the four benchmark suites are discussed in detail in Section IV. Similarly to the CPU06 suite, all the CPU17 applications can be executed with three different input sizes: *test*, *train*, and *ref*. The *test* is the smallest input set and has the least execution time, while the *ref* is the largest input size and has the longest execution time. Most of the CPU17 suite applications have a single input in each input size, except for ten applications: `500.perlbench_r`, `502.gcc_r`, `503.bwaves_r`, `525.x264_r`, `557.xz_r`, `600.perlbench_s`, `602.gcc_s`, `603.bwaves_s`, `625.x264_s`, and `657.xz_s`. Thus, even though there are 43 applications in the CPU17 suite, there are 69, 61, and 64 distinct application-input pairs for the *test*, *train*, and *ref* input sizes, respectively. The analysis performed herein includes the characterization of all 194 application-input pairs using the experimental setup and methodology discussed in Section III.

[2]Hereafter, we will refer to the 'SPECrate 2017 Integer', 'SPECrate 2017 Floating Point', 'SPECspeed 2017 Integer' and 'SPECspeed 2017 Floating Point' mini-suites as *rate_int*, *rate_fp*, *speed_int* and *speed_fp*, respectively.

TABLE I: Experimental system configurations

Processors	Intel Xeon E5-2650L v3 - Dual socket x86_64 Haswell architecture 12 cores–can execute 24 threads (per processor)
Memory	64 GB DDR4
L1 I Cache	8-way set associative 32 kB (per core)
L1 D Cache	8-way set associative 32 kB (per core)
L2 Cache	8-way set associative 256 kB (per core)
L3 Cache	30 MB shared by all cores (per processor)
OS	Red Hat Enterprise Linux server v7.4 Linux kernel: 3.10.0-514.26.2.el7.x86_64 gcc: 4.8.5

III. Experimental Setup and Methodology

We characterized the CPU17 applications by running them on a real system. We then collected execution data from the system's hardware performance counters for analysis. Table I depicts some of the system configuration details of our experimental setup. We used an Intel Xeon E5-2650L processor featuring the Haswell architecture. We disabled the Intel Turbo Boost on all the cores to prevent any dynamic changes in the operating frequency. Each core features separate 32 KB level one (L1) instruction and data caches with 8-way set associativity, and unified 256 KB L2 caches with 8-way set associativity. All the cache levels have a line size of 64 bytes. The system also features a 30 MB L3 cache shared by all the cores, and a main memory of 64 GB. We ran all our experiments on the Red Hat Enterprise Linux server v7.4 operating system. We chose to use hardware performance counters on a real system due to the much faster execution speed, as compared to using architecture simulators (e.g., GEM5 [13]). In addition, executing the applications on a real system provides a more accurate sense of actual application characteristics.

We used the Linux `perf` utility to instrument the hardware performance counters. The exhaustive list of the hardware counters available for the specific microprocessor can be generated by using the `perf list` command. We used 15 counters that we deemed to be relevant for our analysis. To enable reproducibility of the results presented herein, we mention the specific counter flags used for different performance characteristics discussed in Section IV.

We installed the CPU17 suite according to the documentation provided on the SPEC website [12] using the `install.sh` file with `linux-x86_64` toolset. All the applications were built for the *base* tuning, without any additional optimization compiler flags. We set the number of copies to 1 for the *rate* applications; for the *speed* applications we set the number of threads to 4 (the default values in the configuration file). We executed one application-input pair at a time and collected the hardware performance counter values for each execution. Although we were able to install all the CPU17 applications successfully, we encountered errors for five application-input pairs while collecting the `perf` output data. The applications that reported errors were `627.cam4_s` for all the three input sizes, and `500.perlbench_r`'s and

TABLE II: CPU17 benchmarks' average performance characteristics

Suite	Input Size	Instruction Count (in billions)	IPC	Execution Time (s)
rate_int	*test*	76.922	1.716	18.250
	train	230.553	1.765	75.660
	ref	1751.516	1.724	573.627
rate_fp	*test*	47.431	1.692	15.445
	train	357.233	1.651	114.034
	ref	2291.092	1.635	795.579
speed_int	*test*	77.078	1.698	18.396
	train	232.961	1.739	77.438
	ref	2265.182	1.635	670.742
speed_fp	*test*	58.825	0.681	4.510
	train	477.316	0.710	37.366
	ref	21880.115	0.706	670.972

`600.perlbench_s`' `test.pl` *test* input. We also similarly installed the CPU06 suite, to compare the results with the CPU17 suite.

IV. Workload Characterization Results

Table II presents an overview of the CPU17 suite applications' execution characteristics, including the average instruction count, IPC, and execution time across all the applications. For the applications with multiple inputs, we have reported the average values of hardware counters across all the inputs. We make three general observations based on these average characteristics:

1) The average instruction count and execution time increase significantly as the input size increases from *test* to *ref* across all the applications.
2) For *int* and *fp* applications, the *speed* versions have higher average instruction count compared to the *rate* versions. Interestingly, the IPC values for the *int* applications remain relatively same (within 1.060% for *test*, 1.495% for *train*, and 5.443% for *ref* input sets) across all the benchmarks despite the increased instruction count and execution times. However, the IPC values for the *fp* applications reduce drastically (by 56.820% – 59.752%) for the *speed* compared to the *rate* versions. We attribute this to the multi-threaded execution, and higher L2 and L3 cache miss rates of the *speed_fp* applications.
3) There are also differences between the *int* and *fp* applications within the *rate* and *speed* mini-suites. Compared to the *int* applications, the *fp* applications show higher average instruction count, higher execution time, and lower IPC values, especially for the *ref* input size.

In what follows, we focus on the *ref* input size—for brevity—to discuss detailed performance characteristics. For the comparative analysis, we have considered the IPC, microarchitecture-independent characteristics like instruction mix and memory footprint, and microarchitecture-dependent characteristics like the cache and branch behaviors.

TABLE III: IPC Comparison of CPU17 and CPU06 suites

Suite	Average	Std. Dev.
CPU06 *int*	1.762	0.707
CPU17 *int*	1.679	0.640
CPU06 *fp*	1.815	0.706
CPU17 *fp*	1.255	0.636
CPU06 *all*	1.784	0.707
CPU17 *all*	1.457	0.672

A. Instructions per Cycle (IPC)

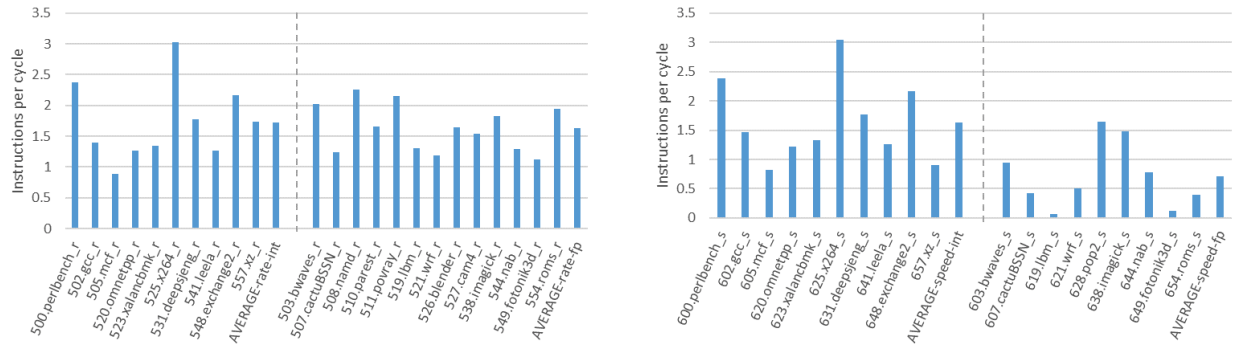
The IPC is one of the most critical performance characteristics of the microprocessor. The IPC value is influenced by both the underlying microarchitecture of the processor (e.g., pipeline depth, in-order vs. out-of-order execution paradigm, branch predictor accuracy, cache configuration, etc.) and the application's characteristics (e.g., instruction mix, memory access patterns, types of branch instructions, etc.). We used the `inst_retired.any` and `cpu_clk_unhalted.ref_tsc` flags to calculate the IPC values.

Fig. 1a and Fig. 1b show the IPC values for the *rate* and *speed* applications, respectively (dotted vertical lines are used to separate *int* and *fp* applications in the figures). As observed from Table II and Fig. 1, the *rate_int* and *speed_int* applications exhibit relatively similar IPC values, except for `557.xz_r` (1.741) and `657.xz_s` (0.903). The *speed_fp* applications show significantly lower IPC values compared to their *rate_fp* counterparts. We attribute these low IPC values to the applications' high L2 and L3 cache miss rates, high instruction count and high memory footprints (details in Section IV-C and Section IV-D). Among the *int* applications, `525.x264_r` (3.024) and `625.x264_s` (3.038) show the highest IPC values, while `505.mcf_r` (0.886) and `657.xz_s` (0.903) show the lowest IPC values for *rate* and *speed* mini-suites, respectively. Similarly, among the *fp* applications, `508.namd_r` (2.265) and `628.pop2_s` (1.642) have the highest IPC values, and `549.fotonik3d_r` (1.117) and `619.lbm_s` (0.062) have the lowest IPC values for *rate* and *speed*, respectively.

Table III compares the IPC values for the CPU06 and CPU17 suites. In general, we observed that the CPU17 applications exhibited lower IPC values than the CPU06 applications by 18.330% on average across all the applications, and by 4.711% and 30.854% for the *int* and *fp* applications, respectively.

B. Instruction Mix Analysis

The instruction mix is a microarchitecture-independent application characteristic that shows the applications' diversity based on the types of instructions executed. The instruction mix is also crucial in identifying how each instruction type contributes to application execution and the impact of specific system components on the overall performance. For example, a high number of memory references may result in slower execution if the memory hierarchy is not rightly-provisioned;



(a) (b)

Fig. 1: Instructions per cycle: (a) *rate*, and (b) *speed* mini-suites

a high number of branch instructions may also slow down execution if the processor's branch predictor is inefficient. We have considered the memory load and store operations, and branch instructions in our analysis.

1) Memory references: To access the number of memory load and store micro-operations, we used the `mem_uops_retired.all_loads` and `mem_uops_retired.all_stores` flags respectively. We compared the memory operations with the retired micro-operations, accessed using the `uops_retired.all` flag. Figures 2a and 2b depict the breakdown of the load and store micro-operations executed by the different *rate* and *speed* applications. The CPU17 applications have 33.993% memory micro-operations on average. All *int* applications have a similar percentage of memory micro-operations for the *rate* (34.750%) and *speed* (34.712%) versions.

However, the *rate_fp* and *speed_fp* mini-suites show a 7.265% difference in the memory micro-operations, with *speed_fp* and *rate_fp* having 29.029% and 36.294% memory micro-operations, respectively. `654.roms_s` has the lowest percentage of memory micro-operations, with only 11.504% loads and 0.895% stores. The applications with the highest percentage of memory micro-operations are: `507.cactuBSSN_r` and `607.cactuBSSN_s` at 48.375% and 41.146%, respectively, for the *rate* and *speed* suites. Among *int* applications, `523.xalancbmk_r` (29.151%) and `605.mcf_s` (29.581%), and among *speed* applications, `507.cactuBSSN_r` (39.786%) and `607.cactuBSSN_s` (33.536%) have the highest percentage load micro-operations. `548.exchange2r_r` and `648.exchange2r_s` have the highest percentage store micro-operations—15.911% and 15.910%, respectively—among the *int* applications, while among the *fp* applications, `519.lbm_r` (13.076%) and `619.lbm_s` (13.480%) have the highest percentage of store micro-operations.

2) Branch instructions: Fig. 3a and 3b show the percentage of branch instructions for all the *rate* and *speed* CPU17 applications. We used the `br_inst_exec.all_branches` flag to obtain the number of branch instructions executed.

On average, branch instructions make up 14.743% of the CPU17 applications' instructions, where 78.662% of these branch instructions are conditional branches. The *fp* applications have 7.621% fewer branch instructions than the *int* applications. `505.mcf_r` and `605.mcf_s` have the highest percentage of branch instructions at 31.277% and 32.939%, respectively, while `519.lbm_r` and `619.lbm_s` have the lowest at 1.198% and 3.646%, respectively.

TABLE IV: Instruction mix comparison of CPU17 and CPU06 suites

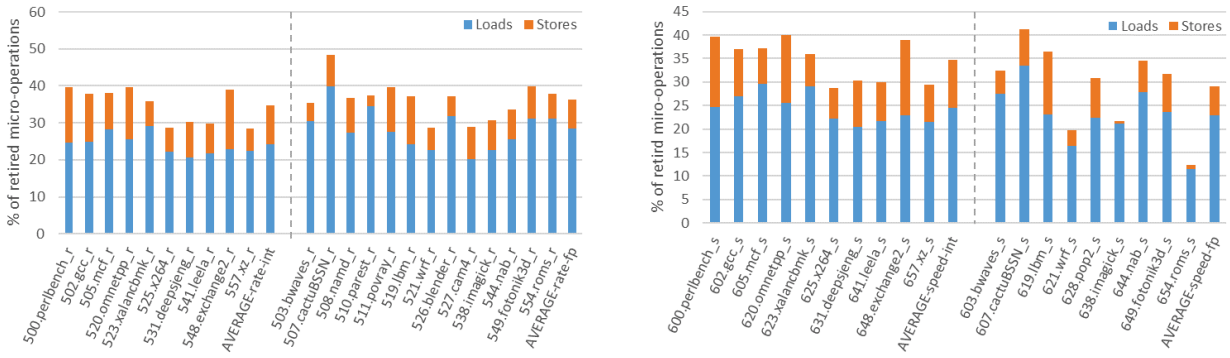
Suite	% Loads		% Stores		% Branches	
	Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.
CPU06 *int*	26.234	4.032	10.311	3.534	19.055	6.526
CPU17 *int*	24.390	2.882	10.341	3.444	18.735	7.168
CPU06 *fp*	23.683	4.625	7.176	3.342	10.805	7.165
CPU17 *fp*	26.187	6.190	7.136	3.346	11.114	6.475
CPU06 *all*	24.739	4.566	8.473	3.755	14.219	8.014
CPU17 *all*	25.331	4.983	8.662	3.751	14.743	7.804

Table IV compares the instruction mix of the CPU06 and CPU17 suites. Both the CPU06 and CPU17 show similar trends for percentage stores and branch instructions: the *int* applications have higher values than the *fp* applications. For percentage loads, however, the trend reverses; *fp* applications have higher values than the *int* applications. It is interesting to note that despite the CPU17 suite's 3.830× increase in the instruction count, the instruction mix—loads, stores, and branches—for all CPU06 and CPU17 applications remain within 2.5% of each other.

C. Memory Footprint

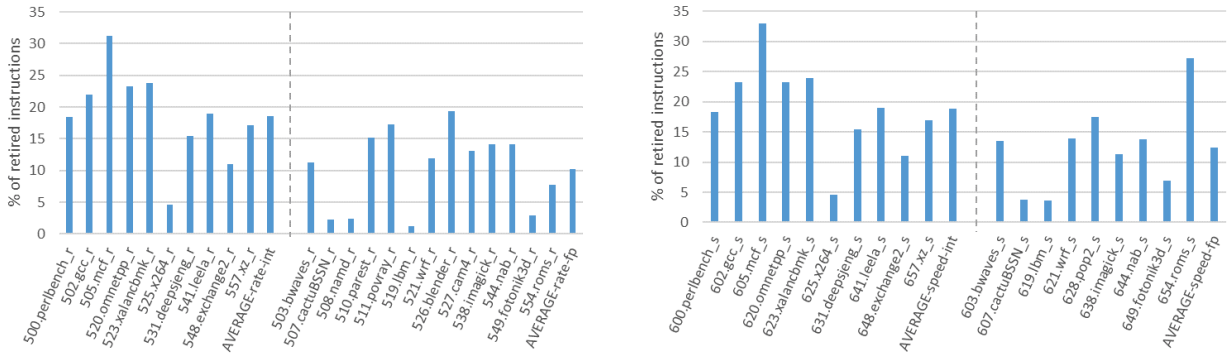
The Virtual Set Size (VSZ) and Resident Set Size (RSS) provide insights into an application's memory usage and working set size [14], [15]. The RSS measures the physical memory used, while the VSZ is the address space reserved by the operating system for the executing application. There are no hardware performance counters to measure the VSZ and RSS directly; thus, we used the `ps -o vsz,rss` Linux command to output the VSZ and RSS values at 1s intervals, and report the maximum values.

Fig. 4 depicts the VSZ and the RSS for the CPU17 applications. Fig. 4a and 4b depict the memory footprints of the *rate* and *speed* CPU17 applications with respect to the VSZ and the RSS. The *speed* applications have 8.276× and 9.764× higher



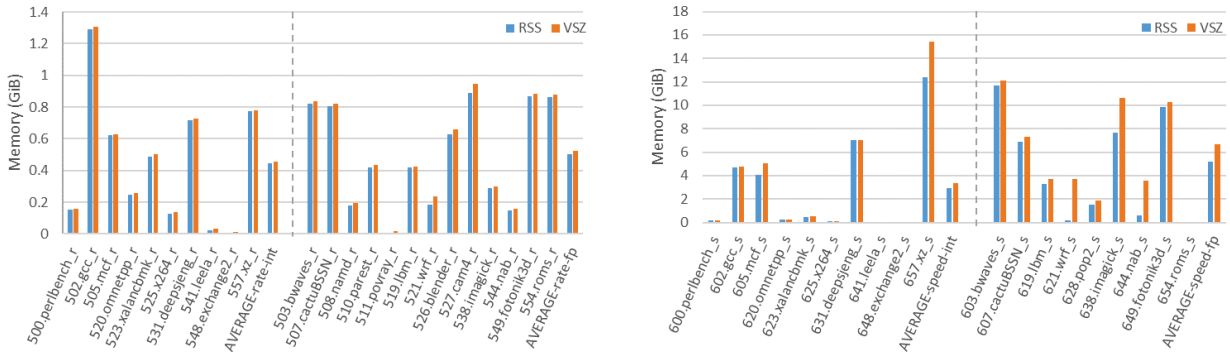
(a) (b)

Fig. 2: Breakdown of memory micro-operations for: (a) *rate*, and (b) *speed* mini-suites



(a) (b)

Fig. 3: Branch characteristics for: (a) *rate*, and (b) *speed* mini-suites



(a) (b)

Fig. 4: Memory footprint for: (a) *rate*, and (b) *speed* mini-suites

RSS and VSZ, respectively, than the *rate* applications because the *speed* applications have larger input sizes [12]. Among all the applications, `657.xz_s` has the maximum RSS and VSZ values of 12.385 GiB and 15.422 GiB, respectively. On the other hand, `548.exchange2_r` has the lowest RSS and VSZ values of 1.148 MiB and 15.160 MiB, respectively. The RSS and VSZ also show high negative correlation values of -0.465 and -0.510 with the IPC.

Table V compares the RSS and VSZ for the CPU17 with the CPU06 suite. As compared to the CPU06 applications, CPU17 applications' average RSS increased by 4.307× and 6.276× for the *int* and *fp* applications, respectively, and by 5.314× for all applications. Similarly, CPU17 applications' VSZ increased by 4.759× and 5.817× for the *int* and *fp* applications, and by 5.285× for all applications.

D. Cache Behavior

We analyzed the cache access behavior of the CPU17 applications with respect to the L1, L2, and L3 cache miss rates. The cache miss rates have a high impact on the applications' overall performance, due

TABLE V: RSS and VSZ comparison of CPU17 and CPU06 mini-suites

Suite	RSS (GiB)		VSZ (GiB)	
	Average	Std. Dev.	Average	Std. Dev.
CPU06 *int*	0.391	0.454	0.399	0.453
CPU17 *int*	1.684	3.073	1.899	3.658
CPU06 *fp*	0.366	0.342	0.491	0.400
CPU17 *fp*	2.297	3.434	2.856	3.755
CPU06 *all*	0.376	0.393	0.452	0.426
CPU17 *all*	1.998	3.278	2.389	3.739

to the delays caused by cache misses. We used the `mem_load_uops_retired` with `l1_hit`, `l1_miss`, `l2_hit`, `l2_miss`, `l3_hit` and `l3_miss` extensions (for example: `mem_load_uops_retired.l1_hit`) to obtain the number of cache hits and misses, with which we computed the miss rates for all the caches.

Fig. 5a and 5b depict the L1, L2 and L3 cache miss rates for the *rate* and *speed* applications. Our first observation was that the L2 cache miss rates were higher than the L3 miss rates for 34 CPU17 applications. We attribute this behavior to our system cache configuration; the 30 MB shared L3 cache is likely better provisioned for the CPU17 applications than the L2 cache.

The average L1, L2, and L3 cache miss rates were 3.424%, 32.515%, and 14.171%, respectively, for the entire CPU17 suite. The *speed* applications have higher cache miss rates than the *rate* applications across all the caches. Similarly, the *int* applications exhibit higher cache miss rates than the *fp* applications. For the L1 cache, `523.xalancbmk_r` and `605.mcf_s` have the highest cache miss rates of 12.174% and 14.138%, respectively, among the *rate_int* and *speed_int* mini-suites. Among the *fp* applications, `507.cactuBSSN_r` and `607.cactuBSSN_s` have the highest cache miss rates of 19.485% and 14.584%, respectively. `505.mcf_r` and `605.mcf_s` have the highest L2 cache miss rates of 65.721% and 77.824%, respectively, while `531.deepsjeng_r` and `631.deepsjeng_s` have the highest L3 cache miss rates of 67.516% and 68.579%, respectively, for the *rate_int* and *speed_int* mini-suites. Among the *rate_fp* and *speed_fp* applications, `549.fotonik3d_r` and `649.fotonik3d_s` have the highest L2 and L3 cache miss rates of 71.609% and 54.730%, and 66.291% and 41.369%, respectively. The L1, L2, and L3 cache load miss rates have negative correlation values of -0.282, -0.479, and -0.137, respectively, with the IPC values across all the applications.

Table VI compares the cache miss rates of the CPU06 and CPU17 suites. On average overall, the CPU17 applications' L2 cache miss rates decrease by 3.231% as compared to the CPU06 applications. The L1 and L3 cache miss rates, however, increase by 0.231% and 0.442% on average, respectively. We observed mixed trends in the L1 and L3 cache miss rates for CPU17 *int* and *fp* applications as compared to CPU06 *int* and *fp* applications; however, we observed that the CPU17 *int* and *fp* applications' L2 cache miss rates have decreased significantly by 2.240% and 8.775%, respectively, as compared to CPU06.

TABLE VI: Comparison of cache miss rates for CPU17 and CPU06 suites

Suite	L1 Miss Rate (%)		L2 Miss Rate (%)		L3 Miss Rate (%)	
	Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.
CPU06 *int*	4.129	6.390	40.854	19.760	12.152	15.044
CPU17 *int*	3.865	4.489	38.614	20.820	15.298	19.456
CPU06 *fp*	2.533	1.521	31.914	20.227	14.041	16.332
CPU17 *fp*	3.023	4.703	26.971	18.660	13.146	12.638
CPU06 *all*	3.193	4.344	35.746	20.511	13.259	15.839
CPU17 *all*	3.424	4.622	32.515	20.557	14.171	16.281

TABLE VII: Branch predictor accuracy comparison for CPU17 and CPU06 suites

Suite	Average (%)	Std. Dev.
CPU06 *int*	2.393	2.505
CPU17 *int*	3.310	2.441
CPU06 *fp*	1.971	1.653
CPU17 *fp*	1.188	1.202
CPU06 *all*	2.145	2.060
CPU17 *all*	2.198	2.172

E. Branch Predictor Accuracy

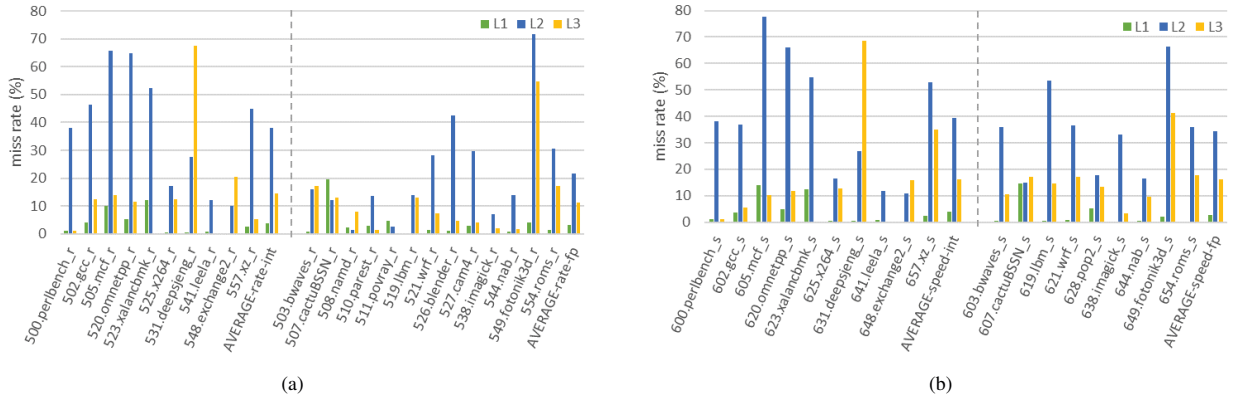
The branch predictor is a processor component that speculatively executes branch instructions to improve the instruction pipeline's efficiency. The branch prediction accuracy is critical for pipeline performance since a wrong speculation could necessitate a pipeline flush, thus wasting clock cycles. We used the `br_misp_exec.all_branches` flag to obtain the branch mispredicts.

Fig. 6a and 6b show the *rate* and *speed* applications' branch mispredict rates. Overall, the *int* applications have higher branch mispredict rates than the *fp* applications. The average mispredict rate for all the applications is 2.198%. `541.leela_r` and `641.leela_s` have the highest mispredict rates among all the applications at 8.656% and 8.636%, respectively—approximately 3.5× higher than the overall average. The *rate* and *speed* applications have very similar mispredict rates at 2.199% and 2.196%, respectively.

Table VII compares the branch mispredict rate between the CPU06 and CPU17 suites. The CPU06 suite shows a similar trend to CPU17: the mispredict rate for *fp* applications is lower than that of *int* applications. Overall, the average mispredict rate for CPU17 is marginally higher than CPU06 by 0.053%.

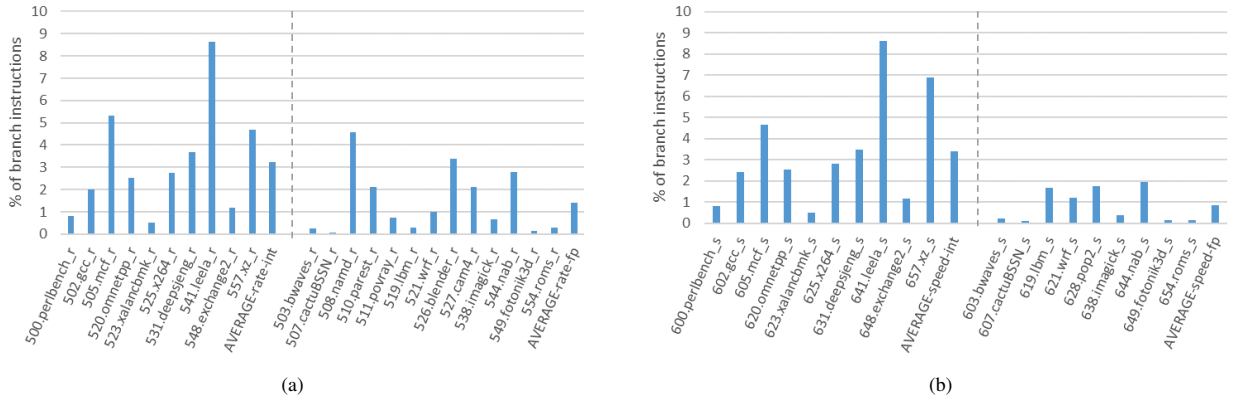
V. Redundancy in the CPU17 Suite

The CPU17 suite consists of 194 distinct application-input pairs and required about 10 hours and 53 minutes to completely run all the pairs our computer system. Microarchitecture research usually employs simulators, like GEM5, which are typically significantly slower. Simulating all the application-input pairs may be prohibitive, and perhaps, unnecessary. Thus, we performed statistical analysis on the



(a) (b)

Fig. 5: Cache miss rates for: (a) *rate*, and (b) *speed* mini-suites



(a) (b)

Fig. 6: Branch mispredict rates for: (a) *rate*, and (b) *speed* mini-suites

microarchitecture-independent application characteristics in order to measure the similarity between the application-input pairs and determine a diverse representative subset of application-input pairs. In this section, we discuss our methodology for generating a representative subset, and suggest a subset of CPU17 applications.

A. Principal Component Analysis

The CPU17 applications' characterization gives us a rich set of variables for the inter-application comparison. To analyze the similarity and redundancy between the 194 application-input pairs, we used 20 microarchitecture-independent application characteristics. Given the difficulty of analyzing 194 application-input pairs for 20 characteristics, we used Principal Component Analysis (PCA) [16], a multivariate statistical technique, to reduce the dimensionality of our data. Table VIII lists the characteristics used for the PCA.

The PCA generates a set of new variables, known as *principal components* (PCs), which are linear combinations of the original set of variables, such that all the principal components are uncorrelated to each other. The PCA transforms a set of p variables $\{X_1, X_2, ..., X_p\}$ into a set of p PCs $\{Z_1, Z_2, ..., Z_p\}$ such that:

$$Z_i = \sum_{j=1}^{p} a_{ij} X_j$$

TABLE VIII: List of PCA characteristics

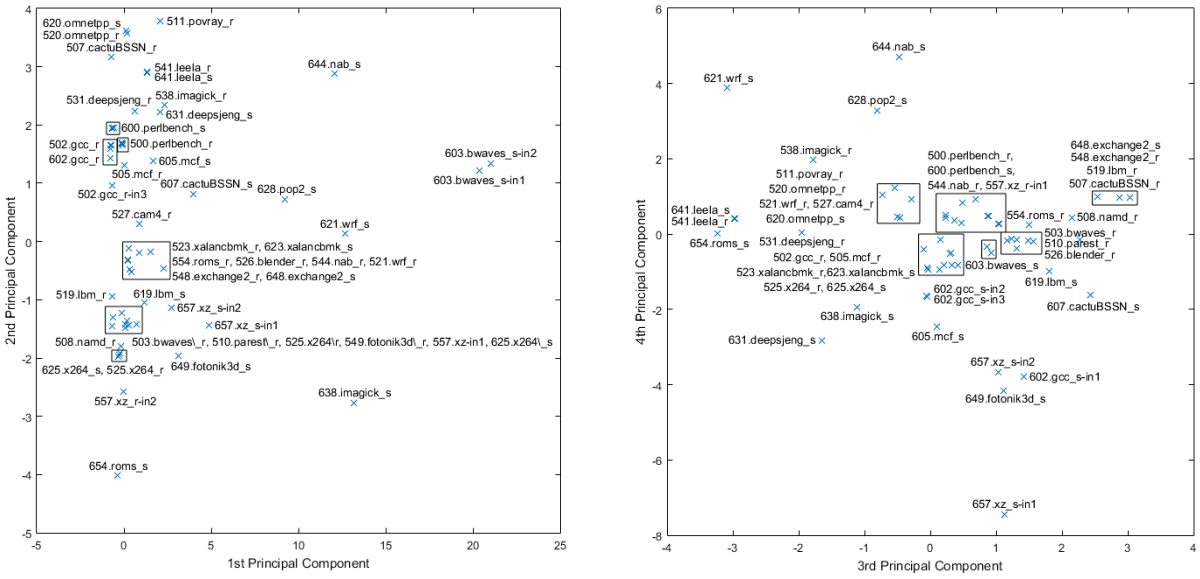
`inst_retired.any`;
`mem_uops_retired.all_loads`; `mem_uops_retired.all_stores`;
`load_uops(%)`; `store_uops(%)`; `total_mem_uops(%)`;
`br_inst_exec.all_branches`; `branch_inst(%)`;
`br_inst_exec.all_conditional`; `br_inst_exec.all_direct_jmp`;
`br_inst_exec.all_direct_near_call`;
`br_inst_exec.all_indirect_jump_non_call_ret`;
`br_inst_exec.all_indirect_near_return`;
`branch_conditional(%)`; `branch_direct_jump(%)`;
`branch_near_call(%)`; `branch_indirect_jump_non_call_ret(%)`;
`branch_indirect_near_return(%)`; `rss`; `vsz`

This transformation has the following useful properties:

(i)
$$\sum_{i=1}^{p} Var[X_i] = \sum_{i=1}^{p} Var[Z_i],$$
which means that the total variance in the data remains same and the transformation does not result in any information loss;

(ii)
$$Cov[Z_i, Z_j] = 0, \forall i \neq j,$$
which means there is no information overlap between the PCs, and the PCs are uncorrelated to each other; and

(iii)
$$Var[Z_1] > Var[Z_2] > ... > Var[Z_p],$$



(a) (b)

Fig. 7: Scatter-plot of CPU17 application-input pairs' PC values

which means that Z_1 has the most information (variance), and each subsequent Z_i has less information than previous Z_{i-1}. This property is useful in reducing the dimensionality of the data. The components with lower variance can be removed from the analysis without significant loss of information.

We used `MATLAB` to perform the PCA on our data, and reduced our data dimensionality from [194 × 20] to [194 × 4] by choosing the first four PCs, accounting for 76.321% of the total variance.

Fig. 7a and 7b show the scatter plots of the first four PC values for all the application-input pairs with the *ref* input size. A few application-input pairs have PC values close together, and we have highlighted them with square boxes in Fig. 7. It is evident from Fig. 7 that PC1 has the maximum range (hence maximum variance), and the range of the PCs decrease from PC1 to PC4. The application-input pairs with similar PC values will show similar performance characteristics. For example, consider three sample application-input pairs: `603.bwaves_s` executed with two different *ref* inputs *in1* and *in2*, and `607.cactuBSSN_s`. As seen in Fig. 7, `603.bwaves_s`–*in1* and `603.bwaves_s`–*in2* are clustered together in both scatter plots, and `607.cactuBSSN_s` is isolated from `603.bwaves_s`' cluster.

To validate the clusters in Fig. 7, Table IX depicts the performance characteristics of the three sample application-input pairs. `603.bwaves_s`–*in1* and `603.bwaves_s`–*in2* exhibit similar characteristics, but both differ significantly from `607.cactuBSSN_s`' characteristics. Thus, the PCs can be used as a similarity metric; the application-input pairs with similar PC values can be clustered together.

We also performed *factor analysis* on the data to evaluate the impacts of the different application characteristics

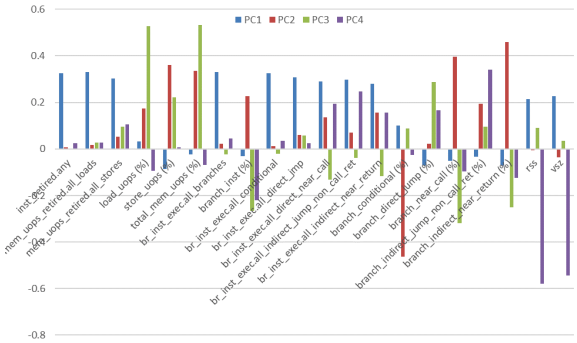


Fig. 8: Factor loadings for all CPU17 application-input pairs

on the different PCs. Factor analysis enables us to observe the PCs as a function of the application characteristics in order to determine the most important characteristics for the PCs. Fig. 8 depicts the factor loadings of all the CPU17 application-input pairs. We observed that the first component (PC1) is most positively dominated by number of instructions retired, load and store memory micro-operations, number of branch instructions and branch instruction types. The second component (PC2) is positively dominated by percent store micro-operations, percent total memory operations, percent near call and near return branch instructions, and negatively dominated by percent conditional branch instructions. The third component (PC3) is positively dominated by percent load micro-operations, percent total memory micro-operations, and negatively dominated by percent near call branch instructions. The fourth component (PC4) is negatively dominated by the memory footprint metrics: RSS and VSZ.

Based on the PC values, we performed hierarchical clustering, to derive representative subsets of the CPU17 suite.

TABLE IX: Validating PC clustering

Characteristic	603.bwaves_s		607.cactuBSSN_s
	in1	*in2*	
Instruction Count (in billions)	48788.718	50116.477	10616.666
% Loads	27.545	27.320	33.536
% Stores	4.982	5.015	7.610
% Branches	13.416	13.497	3.734
RSS (GiB)	11.677	11.750	6.885
VSZ (GiB)	12.078	12.145	7.287

B. Agglomerative Hierarchical clustering

Hierarchical clustering [17] is one of the most commonly used clustering techniques. We performed the hierarchical clustering as follows: first, we assumed that all the application-input pairs were in separate clusters. Thereafter, we iteratively merged the two clusters with the least linkage distance—we used the Euclidean distances between the PC co-ordinates—to form a new cluster. Thus, the number of clusters were decreased by one in every iteration.

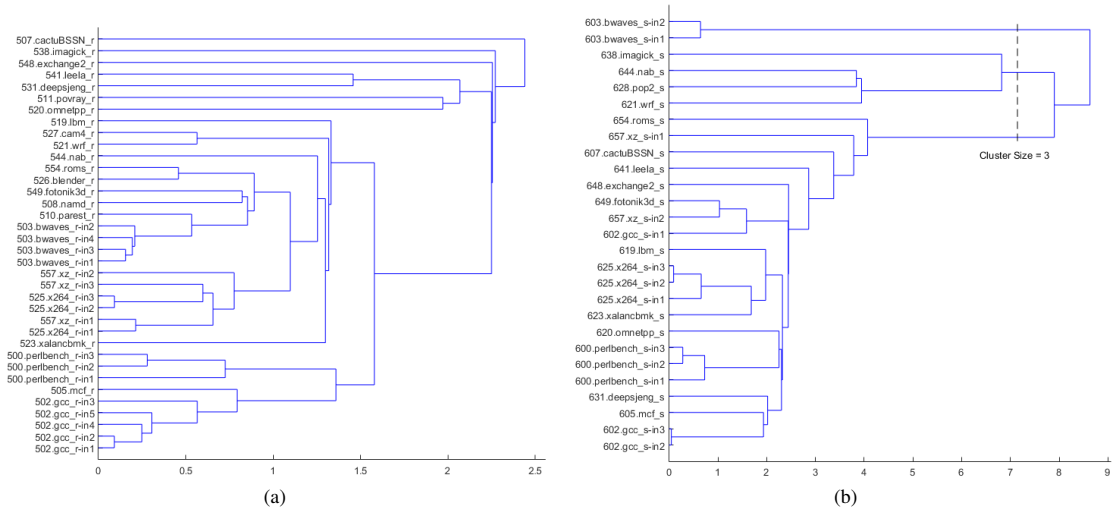
To illustrate the hierarchical clustering, Fig. 9a and 9b show the dendrogram for clustering of the *rate* and *speed* benchmark suites with the *ref* input size. The y-axis depicts the application-input pairs, and the x-axis depicts the Euclidean distance. As per the hierarchical clustering algorithm [17], two application-input pairs with the smallest distance are merged to form a new cluster in each step, until all the application-input pairs are included in the single cluster. For example, considering the *speed* applications in Fig. 9b, the application-input pairs of `602.gcc_s`-*in2* and -*in3* are clustered in the first iteration, whereas `638.imagick_s` is clustered in the last iteration. This visualization allows flexibility in the choice of application-input pairs for a variable number of clusters. Thus, given the clusters in Fig. 9b, if only three application-input pairs are to be selected, then the selected applications should be: one application from the set {`603.bwaves_s`-*in1*, `603.bwaves_s`-*in2*}, one application from the set {`638.imagick_s`, `644.nab_s`, `628.pop2_s`, and `621.wrf_s`}, and the third application can be chosen from the remaining pairs.

C. Subsetting the CPU17 suite

Usually, the main motivation behind subsetting a benchmark suite is to reduce research simulation time, while representing the complete suite. To this end, we performed a hierarchical clustering of the CPU17 application-input pairs, and chose the application-input pair with the shortest execution time from each cluster. For instance, considering the example in Section V-B, with a cluster featuring `638.imagick_s`, `644.nab_s`, `628.pop2_s`, and `621.wrf_s`, the execution times for the application-input pairs were 486.279s, 332.640s, 1619.982s, and 762.382s, respectively. In this case, `644.nab_s` would be chosen to represent the cluster, since it has the shortest execution time.

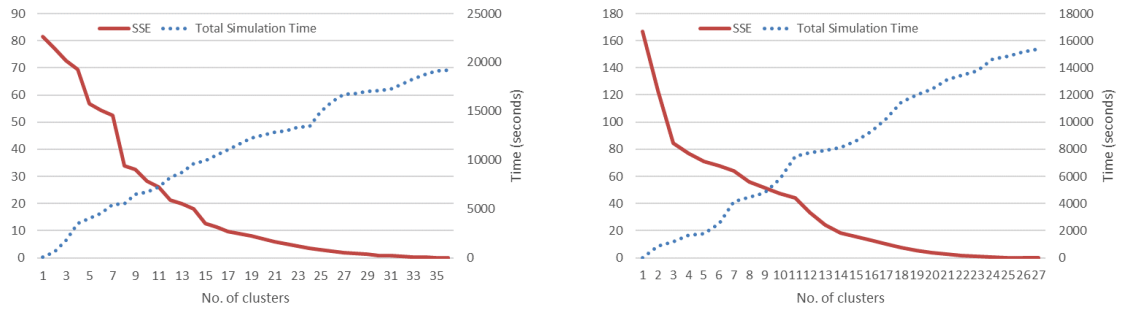
To determine the optimal subset of CPU17 applications, we used the sum of squared error (SSE) [17] to measure the quality of clustering. The SSE is the sum of Euclidean distances between the data points in a cluster and the centroid of the cluster. As the clusters are merged, the SSE value will increase with each iteration. In our work, we chose the number of clusters based on the Pareto-optimal solution for the SSE and execution time. Fig. 10a and 10b show the Pareto-optimal solution for the *rate* and *speed* applications. As seen in Fig. 10, the optimal subset for *rate* applications is 12, while for *speed* applications is 10.

Based on this analysis, Table X suggests a subset of 22 *rate* and *speed* CPU17 applications. The table also shows the total execution time of the applications and the savings as compared to running the full suite. Compared to running the full mini-suite of *rate* applications, using our suggested subset reduced the execution time by 57.116%; similarly, compared to the full mini-suite of *speed* applications, using our suggested subset reduced the execution time by 62.052%.



(a) (b)

Fig. 9: Dendrogram of: (a) *rate*, and (b) *speed* mini-suites



(a) (b)

Fig. 10: Finding Pareto-optimal cluster sizes for: (a) *rate*, and (b) *speed* mini-suites

TABLE X: Suggested subset of CPU17 benchmarks

Suite	Benchmarks	Execution Time	
		Time (s)	% Saving
rate	`500.perlbench_r`-***in3***, `502.gcc_r`-***in4***, `507.cactuBSSN_r`, `511.povray_r`, `519.lbm_r`, `520.omnetpp_r`, `525.x264_r`-***in1***, `527.cam4_r`, `531.deepsjeng_r`, `538.imagick_r`, `541.leela_r`, `548.exchange2_r`	8232.709	57.116
speed	`603.bwaves_s`-***in2***, `607.cactuBSSN_s`, `621.wrf_s`, `625.x264_s`-***in1***, `628.pop2_s`, `638.imagick_s`, `641.leela_s`, `644.nab_s`, `654.roms_s`, `657.xz_s`-***in1***	5885.485	62.052

VI. Conclusion

In this paper, we analyzed the new SPEC CPU2017 applications with respect to different characteristics, such as instructions per cycle, instruction mix, memory footprint, and cache and branch behaviors. We compared the characteristics of the CPU17 suite to the CPU06 suite to gain insights into the similarities and differences between the suites. We have deliberately omitted any conclusions about the value of the new CPU17 suite as compared to the CPU06 suite—we leave this judgment to the readers, and hope the analysis presented herein will enable the readers' decision-making.

We also presented a detailed methodology and results of redundancy analysis of the CPU17 applications. We have also suggested an approach to subsetting the CPU17 applications to obtain a representative set of CPU17 applications, which will aid in reducing the simulation time. However, given how much time simulations typically take, we note that the reduced simulation time achieved using the suggested approach may still be prohibitive. Thus, in future work, we will further analyze the CPU17 applications to explore their phase behavior in order to identify the applications' simulation phases, and evaluate the appropriateness of the CPU17 applications for phase-based optimization research.

References

[1] J. A. Poovey, M. Levy, S. Gal-On, and T. M. Conte, "A benchmark characterization of the EEMBC benchmark suite," 2009.

[2] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. 4th Annu. IEEE Int. Workshop Workload Characterization*, Dec 2001, pp. 3–14.

[3] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 24–36.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proc. 17th Int. Conf. Parallel Architectures and Compilation Techniques*, Oct 2008.

[5] "SPEC CPU Benchmarks," https://www.spec.org/benchmarks.html#cpu.

[6] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[7] "SPEC releases major new CPU benchmark suite," https://www.spec.org/cpu2017/press/release.html.

[8] D. Citron, "MisSPECulation: partial and misleading use of SPEC CPU2000 in computer architecture conferences," in *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2. ACM, 2003, pp. 52–61.

[9] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 412–423, 2007.

[10] A. A. Nair and L. K. John, "Simulation points for SPEC CPU 2006," in *Proc. IEEE Int. Conf. Computer Design (ICCD)*. IEEE, 2008, pp. 397–403.

[11] J. L. Henning, "SPEC CPU Suite Growth: An Historical Perspective," *SIGARCH Compututer Architecture News*, vol. 35, no. 1, pp. 65–68, Mar 2007.

[12] "SPEC CPU2017 Documentation," https://www.spec.org/cpu2017/Docs.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 Simulator," *Computer Architecture News*, vol. 40, no. 2, p. 1, 2012.

[14] J. L. Henning, "SPEC CPU2006 Memory Footprint," *SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 84–89, Mar 2007.

[15] D. Gove, "CPU2006 Working Set Size," *SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 90–96, Mar 2007.

[16] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere, "Workload design: selecting representative program-input pairs," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, 2002, pp. 83–94.

[17] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Pearson, 2006.