FOUR GENERATIONS OF SPEC CPU BENCHMARKS: WHAT HAS CHANGED AND WHAT HAS NOT

Aashish Phansalkar[♣], Ajay Joshi[♣], Lieven Eeckhout[♠], and Lizy John[♣]

[♣]The University of Texas at Austin [♠]Ghent University, Belgium

Abstract

Standard Performance Evaluation Corporation (SPEC) CPU benchmark suite which was first released in 1989 as a collection of 10 computation-intensive benchmark programs (average size of 2.5 billion dynamic instructions per program), is now in its fourth generation and has grown to 26 programs (average size of 230 billion dynamic instructions per program). In order to keep pace with the architectural enhancements, technological advancements, software improvements, and emerging workloads, new programs were added, programs susceptible to compiler attacks were retired, program run times were increased, and memory activity of programs was increased in every generation of the benchmark suite. The objective of this paper is to understand how the inherent characteristics of SPEC benchmark programs have evolved over the last 1.5 decades – which aspects have changed and which have not. We measured and analyzed a collection of microarchitecture-independent metrics related to the instruction mix, data locality, branch predictability, and parallelism to understand the changes in generic workload characteristics with the evolution of benchmark suites. Surprisingly, we find that other than a dramatic increase in the dynamic instruction count and increasingly poor temporal data locality, the inherent program characteristics have pretty much remained unchanged. We also observe that SPEC CPU2000 benchmark suite is more diverse than its ancestors, but still has a over 50% redundancy in programs. Based on our key findings and learnings from this study: (i) we make recommendations to SPEC that will be useful in selecting programs for future benchmark suites, (ii) speculate about the trend of future SPEC CPU benchmark workloads, and (iii) provide a scientific methodology for selecting representative workloads should the cost of simulating the entire benchmark be prohibitively high.

1. Introduction

The Standard Performance Evaluation Corporation (SPEC), since its formation in 1988, has served a long way in developing and distributing technically credible, portable, real-world application-based benchmarks for computer designers, architects, and consumers. While often criticized for inadequacies and vulnerabilities[10][13], SPEC has strived to create credible and objective benchmarks. In order to keep pace with the architectural enhancements, technological advancements, software improvements, and emerging workloads, new programs were added, programs susceptible to compiler attacks were retired, program run times were increased, and memory activity of programs was increased in every generation of the benchmark suite. The initial CPU benchmark suite which was first released in 1989 as a collection 10 computation-intensive benchmark programs (average size of 2.5 billion dynamic instruction per program), is now in its fourth generation and has grown to 26 programs (average size of 230 billion dynamic instructions per program).

Designing, understanding, and validating benchmarks is as serious an issue as designing the computers themselves. In order to reach good computer designs in shorter time, it is important to have benchmarks that cover the program space well. It is desired that these benchmarks are uniformly distributed over the program space (rather than clustered in specific areas). Ofcourse, it is best to have fewer benchmarks in the suite if they meet the requirement of covering the program space.

As the SPEC benchmarks evolved through years, some aspects of the benchmarks have changed, and some aspects have not. The static instruction count of the programs in their binaries has not significantly grown from SPEC CPU89 to SPEC CPU2000. Have the basic program control flow aspects really changed as the benchmark suites have evolved? There was a conscious effort to increase run-times and to avoid getting contained in the growing data caches; what is the impact when the static count has not changed? How diverse are the different programs in the different suites? Do the 26 programs in SPEC CPU2000 model a more vast region of program space compared to what the 10 programs from SPEC89 suite did? Some programs appear in multiple suites. How did these program evolve between suites? Four generations of CPU benchmarks and 1.5 decades since the release of the first suite, our objective in this paper is to understand the trends in the evolution of the SPEC benchmarks and their implications.

Computer architects and computer performance analysts have some insight into the changes in the benchmark suites during the last 15 years, however, no past research has tried to study all the suites with a common perspective. Most of the insights from the past are from what performance analysts and designers might have done on specific microarchitectural components. The instruction set architecture (ISA) and compilers might have been different and the results might have been closely tied to specific microarchitectural elements in the evaluation. We perform this study using a collection of microarchitecture-independent metrics related to the instruction mix, data locality, branch predictability, and instruction level parallelism (ILP), to characterize the generic behavior of the benchmark programs. The same compiler is used to compile the four suites. The data is analyzed to understand the changes in workload characteristics with the evolution of benchmark suites, the workload space that is covered by each generation of the benchmark suites, and the redundancy in each benchmark suite. The contributions of this work are multifold: understand current benchmarks, provide useful insight for selecting programs for the next generation of benchmarks, and help in interpreting micro-architecture level performance measurements

This paper is organized as follows. We first detail our methodology in section 2 after which we present our results in section 3. Section 4 then discusses the implications of these results in detail. We present related work in section 5 and finally conclude in section 6.

2. Methodology

This section presents our methodology: the microarchitecture-independent metrics, the statistical data analysis techniques, the tools, and the benchmarks that are used in this paper.

2.1 Metrics

In this research work we selected microarchitecture-independent metrics to characterize the behavior of the instruction and data stream of every benchmark program. Microarchitecture-independent metrics allow for a comparison between programs by understanding the inherent characteristics of a program isolated from features of particular microarchitectural components. As such, we use a gamut of microarchitecture-independent metrics which we feel affect overall program performance. We provide an intuitive reasoning to illustrate how the measured metrics can affect the manifested performance. The metrics measured in this study are a subset of all the microarchitecture-independent characteristics that can be potentially measured, but we believe that they cover a wide enough range of the program characteristics to make a meaningful comparison between the programs.

We have identified the following microarchitecture-independent metrics:

Instruction Mix: Instruction mix of a program measures the relative frequency of various operations performed by a program. We measured the percentage of computation, data memory accesses (load and store), and branch instructions in the dynamic instruction stream of a program. This information can be used to understand the control flow of the program and/or to calculate the ratio of computation to memory accesses which gives us an idea of whether the program is computation bound or memory bound.

Basic Block Size: A basic block is a section of code with one entry and one exit point. We measure the basic block size which quantifies the average number of instructions between two consecutive branches in the dynamic instruction stream of the program. Programs with a larger basic block size will take a relatively smaller performance hit due to branch misprediction rate, as compared to programs with smaller basic block sizes.

Branch Direction: Backward branches are typically more likely to be taken than forward branches. This metric computes the percentage of forward branches out of the total branch instructions in the dynamic instruction stream of the program. Obviously, hundred minus this percentage is the percentage of backward branches.

Taken Branches: We measured the number of taken branches as a fraction of the total number of branches in the dynamic instruction stream.

Forward-taken Branches: We also measure the fraction of taken forward branches in the dynamic instruction stream.

Dependency Distance: We use a distribution of the dependency distances in a program as a measure of the inherent ILP in the program. Dependency distance is defined as the total number of instructions in the dynamic instruction stream between the production (a write) and the consumption (a read) of a register instance [3][22]. While techniques such as value prediction reduce the impact of these dependencies on ILP, information on the dependency distance is very useful in understanding ILP inherent to a program. The dependency distance is classified into six categories: percentage of total dependencies that have a distance of 1, and the percentage of total dependencies that have a distance of up to 2, 4, 8, 16, 32, and greater than 32. Programs that have a higher percentage of true dependencies greater than 32 are likely to exhibit a higher ILP than a program that has a higher percentage of true dependencies with a dependency distance less than 32 (provided control flow is not the limiting factor).

Data Temporal Locality: Temporal locality of a program's data stream is a measure of how soon recently accessed data items tend to be accessed in the near future. Several locality metrics have been proposed in the past [4][5][11][18][21][30][31], however, many of them are computation and memory intensive. We picked the average memory reuse distance metric from [31] since it is more computationally feasible than other metrics. In this metric, locality is evaluated by counting the number of memory accesses between two accesses to the same address, for every unique address in the program. The evaluation is performed in restricted *window* sizes analogous to cache block sizes. The data temporal locality (*tlocality*) metric is defined as the weighted (based on the number of times each unique data address is accessed) average memory reuse distance. The *tlocality* metric is calculated for *window* sizes of 16, 64, 256 and 4096.

Data Spatial Locality: Cache memories exploit spatial locality through the use of cache lines. In order to measure spatial locality we computed the above mentioned *tlocality* metric, or the weighted average memory reuse distance, for four different *window* sizes: 16, 64, 256, and 4096. Spatial locality information is characterized by the difference between the *tlocality* metric for the various line sizes. The choice of the *window* sizes is based on the experiments conducted by Lafage et. al.[31]. Their experimental results showed that the above set of *window* sizes was sufficient to characterize the locality of the data reference stream with respect to a wide range of data cache configurations.

2.2 Statistical Data Analysis

Obviously, the amount of data in the analysis is huge. There are many variables (18 microarchitecture-independent characteristics) and many cases (60 benchmarks). It is humanly impossible to simultaneously look at all the data and draw meaningful conclusions from them. We thus use multivariate statistical data analysis techniques, namely *Principal Component Analysis* and *Cluster Analysis*, to compare and discriminate programs based on the measured characteristics, and understand the distribution of programs in the workload space. *Cluster Analysis* is used to group *n* cases in an experiment (benchmark programs) based on the measurements of the *p* principal components. The goal is to cluster programs that have the same intrinsic program characteristics.

Principal Components Analysis: Principal components analysis (PCA) [6] is a classic multivariate statistical data analysis technique that is used to reduce the dimensionality of the data set while retaining most of the original information. It builds on the assumption that many variables (in our case, microarchitecture-independent program characteristics) are correlated. PCA computes new variables, called principal components, which are linear combinations of the original variables, such that all the principal components are uncorrelated. PCA transforms *p* variables $X_1, X_2, ..., X_p$ into *p* principal components $Z_1, Z_2, \ldots, Z_p$ such that:

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

This transformation has the property *Var* $[Z_1] >$ Var $[Z_2] > \ldots >$ Var $[Z_p]$ which means that Z_1 contains the most information and Z_p the least. Given this property of decreasing variance of the principal components, we can remove the components with the lower values of variance from the analysis. This reduces the dimensionality of the data set while controlling the amount of information that is lost. In other words, we retain *q* principal components ($q << p$) that explain at least 80% to 90 % of the total information; in this paper *q* varies between 2 and 4. By examining the most important principal components, which are linear combinations of the original program characteristics, meaningful interpretations can be given to these principal components in terms of the original program characteristics.

Cluster Analysis: There exist two flavours of clustering techniques, *linkage clustering* and *K-means clustering* [1]. In *linkage clustering*, cases (or benchmarks) are grouped iteratively in a multi-dimensional space (the PCA space in this paper) until all cases are in one single group (cluster). A *dendrogram* is used to graphically present the cluster analysis by showing the linkage distance (Euclidean distance) between each of the clusters. The dendrogram is a tree representation of the clusters; elements of a cluster and their merging can be visualized in this representation. The distance from a node to the leaves indicates the distance between the merging clusters. Based on the results of the dendrogram it is up to the user to decide how many clusters to consider. *K-means clustering* on the other hand, tries to group all cases into exactly K clusters. Obviously, not all values for K fit the data set well. As such, we will explore various values of K in order to find the optimal clustering for the given data set.

2.3 Tools

Compilers: The programs from the four SPEC CPU benchmark suites were compiled on a Compaq Alpha AXP-2116 processor using the Compaq/DEC C, C++, and the FORTRAN compiler. The programs were statically built under OSF/1 V5.6 operating system using full compiler optimization.

***SCOPE*:** The various workload characteristics were measured using a custom-grown analyzer called *SCOPE. SCOPE* was created by modifying the *sim-safe* functional simulator from the *SimpleScalar* 3.0 [29] tool set. *SCOPE* analyzes the dynamic instruction stream and generates statistics related to instruction mix, data locality, branch predictability, basic-block size, ILP etc. Essentially, the front-end of *sim-safe* is interfaced with home-grown analyzers to obtain various locality and parallelism metrics. We also used ATOM to measure the static instruction count of the benchmark programs.

Statistical data analysis: We use STATISTICA version 6.1 for performing PCA as well as for linkage clustering. For *K-means clustering*, we use the *SimPoint* software [32].

2.4 Benchmarks

The different benchmark programs and their dynamic instruction counts are shown in Tables 1-4. Due to the differences in libraries, data type definitions, pointer size conventions, and known compilation issues on 64-bit

Table 1: SPEC CPU 89**.**

Program	Input	INT/FP	Dynamic Instrn Count
espresso	bca.in	INT	0.5 billion
li	li-input.lsp	INT	7 billion
eqntott	*	INT	*
gcc	*	INT	*
spice2g6	*	FP	*
doduc	doducin	FP	1.03 billion
fpppp	natoms	FP	1.17 billion
matrix300	-	FP	1.9 billion
nasa7	-	FP	6.2 billion
tomcatv	-	FP	1 billion

Table 2: SPEC CPU 92.

Program	Input	INT/FP	Dynamic Instrn Count
espresso	bca.in	INT	0.5 billion
li	li-input.lsp	INT	6.8 billion
eqntott	*	INT	*
compress	in	INT	0.1 billion
sc	*	INT	*
gcc	*	INT	*
spice2g6	*	FP	*
doduc	doducin	FP	1.03 billion
mdljdp2	input.file	FP	2.55 billion
mdljsp2	input.file	FP	3.05 billion
wave5	-	FP	3.53 billion
hydro2d	hydro2d.in	FP	44 billion
Swm256	swm256.in	FP	10.2 billion
alvinn	In_pats.txt	FP	4.69 billion
ora	params	FP	4.72 billion
ear	*	FP	*
su2cor	su2cor.in	FP	4.65 billion
fpppp	natoms	FP	116 billion
nasa7	-	FP	6.23 billion
tomcatv	-	FP	0.9 billion

Table 3: SPEC CPU 95.

Program	Input	INT/FP	Dynamic Instrn Count
go	null.in	INT	18.2 billion
li	*.lsp	INT	75.6 billion
m88ksim	ctl.in	INT	520.4 billion
compress	bigtest.in	INT	69.3 billion
ijpeg	penguin.ppm	INT	41.4 billion
gcc	expr.i	INT	1.1 billion
perl	perl.in	INT	16.8 billion
vortex	*	INT	*
wave5	wave5.in	FP	30 billion
hydro2d	hydro2d.in	FP	44 billion
swim	swim.in	FP	30.1 billion
applu	applu.in	FP	43.7 billion
mgrid	mgrid.in	FP	56.4 billion
Turb3d	turb3d.in	FP	91.9
Su2cor	su2cor.in	FP	33 billion
fpppp	natmos.in	FP	116 billion
apsi	apsi.in	FP	28.9 billion
tomcatv	tomcatv.in	FP	26.3 billion

Table 4: SPEC CPU 2000.

Program	Input	INT /FP	Dynamic Instrn Count
Gzip	input.graphic	INT	103.7 billion
vpr	route	INT	84.06 billion
gcc	166.i	INT	46.9 billion
mcf	inp.in	INT	61.8 billion
crafty	crafty.in	INT	191.8 billion
parser		INT	546.7 billion
eon	chair.control.cook chair.camera chair.surfaces chair.cook.ppm	INT	80.6 billion
perlbmk	*	INT	*
vortex	lendian1.raw	INT	118.9 billion
gap	ref.in	*INT*	*269.0 billion*
bzip2	input.graphic	INT	128.7 billion
twolf	ref	INT	346.4 billion
swim	swim.in	FP	225.8 billion
wupwise	wupwise.in	FP	349.6 billion
mgrid	mgrid.in	FP	419.1 billion
mesa	mesa.in	FP	141.86 billion
galgel	gagel.in	FP	409.3 billion
art	C756hel.in	FP	45.0 billion
equake	inp.in	FP	131.5 billion
ammp	ammp.in	FP	326.5 billion
lucas	lucas2.in	FP	142.4 billion
fma3d	fma3d.in	FP	268.3 billion
apsi	apsi.in	FP	347.9 billion
applu	applu.in	FP	223.8 billion
facerec	*	FP	*
sixtrack	*	FP	*

machines, we were unable to compile some programs (mostly from old suites - SPEC CPU 89 and SPEC CPU 92). The instruction counts of these programs are therefore missing from the tables.

3. Results

In this section, we present our results. Before presenting overall behavioral characteristics, we first focus on particular benchmark characteristics: dynamic instruction count, branch characteristics, data stream locality and instruction-level parallelism. The raw data is presented in Appendix A while the paper focuses on insights and observations. The raw data shows that no single characteristic has changed as dramatically as the dynamic instruction count.

3.1 Dynamic and static instruction count

Due to the increasing microprocessor performance, SPEC also has to increase the dynamic instruction count of their CPU benchmark suites. The dynamic instruction count has grown 100 times on the average from SPEC CPU89 to SPEC CPU2000. This is to enable performance measurement during a sufficiently long time window. We observe that although the average dynamic instruction count of the benchmark programs has increased by a factor of x100, the static count has remained more or less constant. This suggests that the dynamic instruction count of the SPEC CPU benchmark programs could have simply been scaled – more iterations through the same instructions. This could be a plausible reason for the observation that instruction locality of programs has more or less remained the same across the four generations of benchmark suites.

3.2 Branch characteristics

For studying the branch behavior we have included the following metrics: the percentage branches in the dynamic instruction stream, the average basic block size, the percentage forward branches, the percentage taken branches, and the percentage forward-taken branches. From PCA analysis, we retain 2 principal components explaining 62% and 19% of the total variance, respectively. Figure 1 plots the various SPEC CPU benchmarks in this PCA space. The integer benchmarks are observed to be much clustered. We also observe that the floating-point benchmarks typically have a positive value along the first principal component (PC1), whereas the integer benchmarks have a negative value along PC1. The reason is that floating-point benchmarks typically have fewer branches, and thus have a larger basic block size; floating-point benchmarks also typically have a smaller percentage of forward branches, and fewer percentage forward-taken branches. In other words, floating-point benchmarks tend to spend most of their time in loops. The two outliers in the top corner of this graph are SPEC2000's *mgrid* and *applu* programs due to their extremely large basic block sizes, 273 and 318, respectively. The two outliers on the right are SPEC92 and SPEC95 *swim* due to its large percentage taken branches and small percentage forward branches.

We conclude from this graph that branch characteristics did not significantly change over the past 1.5 decades. Indeed, all SPEC CPU suites overlap in this graph.

3.3 Data stream locality

For studying the temporal and spatial locality behavior of the data stream we used the locality metrics as proposed by Lafage et. al. [31] for four different *window* sizes: 16, 64, 256, and 4096. Recall that the metrics by themselves quantify temporal locality whereas the differences between them is a measure for spatial locality. Since PCA is a linear transformation, PCA will be able to extract the spatial locality from the raw data. From the PCA analyses of raw data, we concluded that several SPEC CPU2000 and CPU95 benchmark programs: *bzip2*, *gzip*, *mcf*, *vortex*, *vpr*, *gcc*, *crafty*, *applu*, *mgrid*, *wupwise,* and *apsi* from CPU2000, and *gcc*, *turbo3d*, *applu,* and *mgrid* from CPU95 exhibit a temporal locality that is significantly worse than the other benchmarks. Concerning spatial locality, most of these benchmarks exhibit a spatial locality that is relatively higher than that of the remaining benchmarks, i.e. increasing *window* sizes improves performance of these programs more than they do for the other benchmarks. The only exceptions are *gzip* and *bzip2* which exhibit poor spatial locality. Obviously, we expected temporal locality of the data stream to get worse for newer generations of SPEC CPU given one of the objectives of SPEC which is to increase the working set size along the data stream for subsequent SPEC CPU suite generations.

On the remaining benchmarks we perform PCA which yields us two principal components explaining 57.2% and 38.6% of the total variance, respectively. The first principal component basically measures temporal locality, i.e. a more positive value along PC1 indicates poorer temporal locality. The second principal component basically measures spatial locality. Benchmarks with a high value along PC2 will thus benefit more from an increased line size.

Figure 2 plots the benchmarks in this PCA space. This graph shows that for these benchmarks, all SPEC CPU generations overlap. This indicates that although SPEC's objective is to worsen the data stream locality behavior of subsequent CPU suites, several benchmarks in recent suites exhibit a locality behavior that is similar to older versions of SPEC CPU. Moreover, several CPU95 and CPU2000 benchmarks show a temporal locality behavior that is better than most CPU89 and CPU92 benchmarks.

3.4 Instruction-level parallelism

In order to study the instruction-level parallelism (ILP) of the SPEC CPU suites we used the dependency metrics as well as the basic block size. Both metrics are closely related to the intrinsic ILP available in an application. Long dependency distances and large basic block sizes generally imply a high ILP. Basic block related and dependency related limitations can be overcome by branch prediction and value prediction respectively. However, both these metrics can be used to indicate the ILP or to motivate the use of better branch and value predictors. The first two principal components explain 96% of the total variance. The PCA space is plotted in Figure 3. We observe that the integer benchmarks typically have a high value along *PC1* which indicates that these benchmarks have more short dependencies. The floating-point benchmarks typically have larger dependency distances. We observe no real trend in this graph. The intrinsic ILP did not change over the past 1.5 decades - except for the fact that several floating-point SPEC89 and SPEC92 benchmarks (and no SPEC CPU95 or SPEC CPU2000 benchmarks) exhibit relatively short dependencies compared to other floating-point benchmarks; these overlap with integer benchmarks in the range $-0.1 < PC1 < 0.6$.

3.5 Overall characteristics

When considering all our metrics together in a single analysis, we retain four principal components explaining 82.8% of the total variance. The first principal component (47.6%) basically measures ILP, the second PC (14%) measures temporal data stream locality, the third PC (11.2%) measures the memory intensiveness, and the fourth PC (10%) quantifies branch behavior. We thus conclude that the larger variability among the SPEC CPU benchmarks is due to (in decreasing order) ILP, temporal data stream locality, memory intensiveness, and branch behavior.

Figure 4 shows a projection of the PCA space on its first two principal components; Figure 5 does the same for the third and fourth principal components. We can make several interesting observations from these graphs. First, we observe several outliers in Figure 4, namely *bzip2*, *gzip*, *mcf*, *wupise*, *apsi* and *turbo3d*. This is due to poor temporal locality in the data stream. Second, Figure 5 shows that there is more diversity in the floating-point benchmarks than in the integer benchmarks. This was also apparent from the previous subsections when discussing the individual metrics.

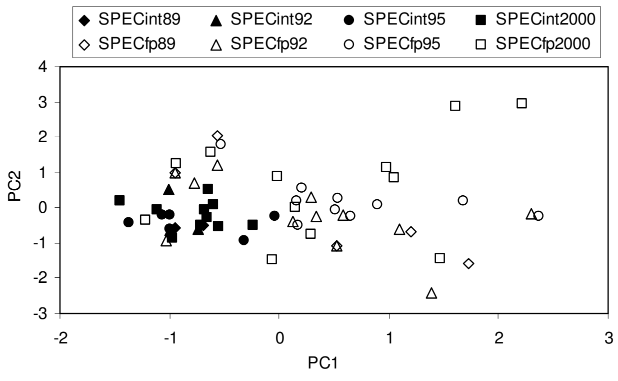

Figure 1: PCA space built up from the branch characteristics.

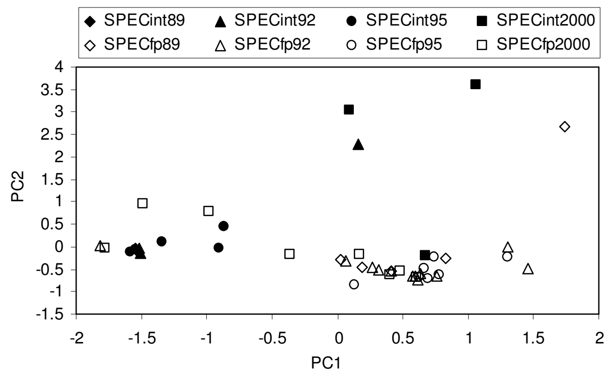

Figure 2: Data stream locality for the SPEC CPU benchmarks after excluding the SPEC95 and SPEC2000 benchmarks with poor locality.

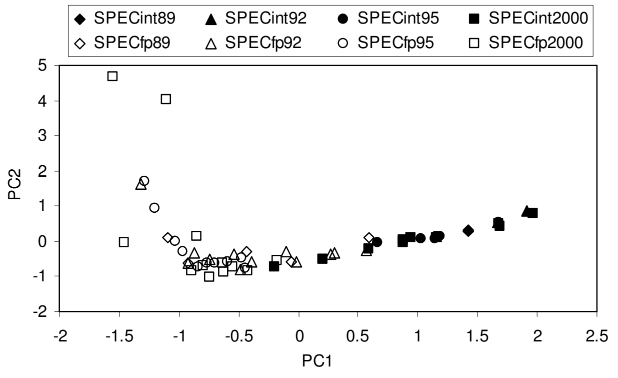

Figure 3: Parallelism in SPEC CPU.

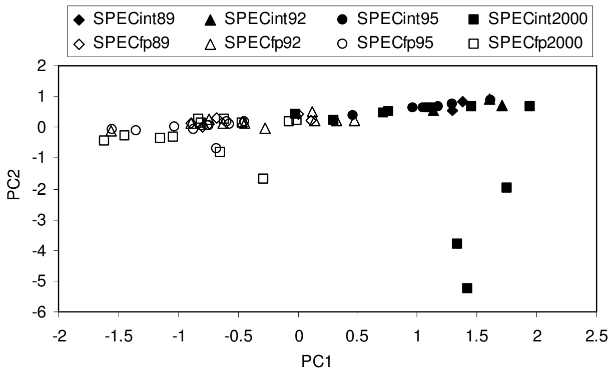

Figure 4: The PCA space built by all the characteristics: PC1 Vs PC2.

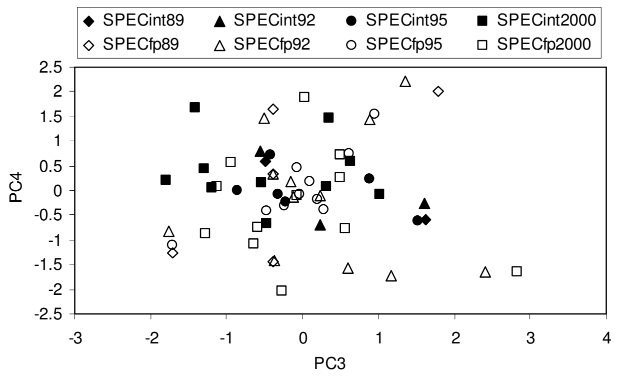

Figure 5: The PCA space built up by all the characteristics: PC3 versus PC4.

4. Discussion

This section discusses the implications of the results presented in the previous section.

4.1 Selecting representative benchmarks

The research done in the computer architecture community typically uses SPEC CPU benchmarks and simulation [14]. An important consequence of this practice is that it is difficult to compare results published in 1999 using SPEC CPU95 versus results published in 2001 using SPEC CPU2000. The reason is that the benchmarks and their inputs in the various suites change over time, as pointed out earlier in this paper. One solution is to better understand how (dis)similar the various benchmarks are from different CPU suites. In order to detect cross CPU suite (dis)similarity we perform a cluster analysis in the PCA space. We use K-means clustering using the *SimPoint* software [29]. The *SimPoint* software identifies the best 'K' so that the clustering in K clusters fits the data well according to the Bayesian Information Criterion (BIC). The BIC is a measure of the goodness of fit of a clustering to a data set. Unlike *SimPoint* we do not use random projection before applying K-means clustering; we use the transformed PCA space instead as the projected space. *SimPoint* identifies the best 'K' by trying a number of K's and selecting the minimal K for which the BIC is near optimal (within 90% of the best BIC). Using the *SimPoint* software, we obtain 10 clusters as a good fit for the given data set.

These 10 clusters are shown in Table 5. The benchmarks in bold are the benchmarks closest to the centroid of the cluster and can thus be considered the representatives for that cluster. An analysis of Table 5 gives us several interesting insights. First, these results confirm our previous statement that the integer benchmarks show less diversity over the various CPU suites than the floating point benchmarks. This is reflected here by the fact that nearly all integer benchmarks reside in 5 of the 10 clusters; the floating point benchmarks reside in the remaining 6 clusters. There are two clusters containing the outliers, *mcf* and *bzip2/gzip*, respectively. All other integer benchmarks are in clusters 2 and 5, except for *ijpeg* which is in cluster 3. Second, this clustering in conjunction with the results from the previous section can give us meaningful interpretations to observed (dis)similarities between the different CPU suites. There are three clusters containing benchmarks with a poor temporal data stream locality, namely clusters 4 (*mgrid* and *applu*), 6 (*gzip* and *bzip2*) and 8 (*mcf*). The integer cluster 2 differs from cluster 5 due to its relatively low percentage memory operations. Concerning the floating-point clusters, cluster 1 seems to have the highest value along PC4 followed by cluster 7 and cluster 9. In other words, the benchmarks in cluster 1 generally have more taken branches and less forward branches than the benchmarks from clusters 7 and 9. They thus spend most of their time in tight loops without conditional branches inside the loop; the floating-point benchmarks in cluster 9 on the other hand, tend to have more conditional branches inside loops. The two remaining floating-point clusters 3 and 10 also have more conditional branches inside loops. The only difference however is that cluster 3 has a relatively lower percentage of memory operations as compared to cluster 10.

4.2 Subsetting benchmark suites

Citron [2] presented a survey on the use of SPEC CPU2000 benchmarks by researchers in the computer architecture community. He observed that some benchmarks are more popular than others. For the integer CPU2000 benchmarks, the list in decreasing order of popularity is: *gzip, gcc, parser, vpr, mcf, vortex, twolf, bzip2, crafty, perlbmk, gap* and *eon.* For the floating-point CPU2000 benchmarks, the list is *art, equake, ammp, mesa, applu, swim, lucas, apsi, mgrid, wupwise, galgel, sixtrack, facerec* and *fma3d.* Table 5 suggests that these subsets might not be well chosen for two reasons: (i) some parts of the workload space might be uncovered by the subset and (ii) there might be significant redundancy within the subset. For example, subsetting CINT2000 using *gzip*, *gcc*, *parser*, *vpr*, *mcf* and *vortex* will result in two uncovered clusters, namely 2 and 10 in Table 5; at the same time, this subset contains significant redundancy since four benchmarks in the subset come from the same cluster. The results from Table 5 suggest that subsetting CINT2000 using *twolf*, *gcc*, *gzip*, *mcf* and *eon* would be better practice.

Another observation made by Citron is that several researchers dealing with data cache performance do not or insufficiently consider floating-point benchmarks in their analyses. To address this issue we have computed a representative clustering on the data stream locality metrics from section 3.4. The results are given in Table 6 and suggest indeed that not using or only using a random subset of CPU2000 floating-point benchmarks might not be representative for studying data stream behavior. Several clusters in this analysis do not contain CPU2000 integer benchmarks; as such, not using representatives from those clusters might lead to a distorted result when doing memory hierarchy design optimizations. Similarly, only using CPU2000 floating-point benchmarks and no CPU2000 integer benchmarks might also be unrepresentative.

4.3 Similarity of common benchmarks among different suites

Several benchmarks appear in various versions of the SPEC CPU suites. The results from the previous section on cross CPU suite (dis)similarity will allow us to identify which benchmarks did change significantly over different CPU suite generations and which did not. To further support this observation, we also provide a dendrogram as obtained through linkage clustering. This is shown in Figure 6. In this dendrogram, benchmarks that are connected through small linkage distances are similar, whereas benchmarks connected through large linkage distances are dissimilar. The benchmarks that did not significantly change over time are *nasa7* (89/92), *swim* (92/95), *espresso* (89/92), *li* (89/92/95), *gcc* (95/2000), *tomcatv* (89/92/95), *hydro2d* (92/95), *wave5* (92/95), *doduc* (89/92) and *fpppp* (89/92/95). It is surprising to see gcc included in this list because it is known that gcc (2000) performs aggressive inlining and other optimizations and is significantly different from gcc (95) [10]. In Table 6 which provides clustering results for data memory access behavior, gcc2k is very uniquely positioned compared to gcc(95), however, in the clustering based on overall characteristics, gcc does not exhibit significant changes. Since architectural changes sometimes affect only one specific aspect of the program, it is important to use a clustering based on individual features as opposed to overall characteristics. For a number of other benchmarks that did change significantly over time we can point to behavioral differences, for example *swim* (the CPU2000 version has more conditional branches inside loops than its ancestors), *applu* (the CPU2000 version has a significantly worse temporal locality in its data stream than its ancestors), and *compress* (the CPU95 version has a higher branch taken rate and a smaller number of forward branches than the CPU92 version.

4.4 Redundancy

As pointed out by previous research [7] [9], there is a lot of redundancy in benchmark suites. That is, benchmarks might exhibit similar behavioral characteristics questioning the need to include all those redundant benchmarks in the suite. Redundancy in benchmark suites is especially a problem for simulation purposes. Simulating benchmarks with similar behavioral characteristics will add to the overall simulation time without providing any additional insight. The purpose of this section is to quantify how redundancy has changed over the past 1.5 decades in SPEC CPU. To this end, we define the *redundancy of a CPU suite* as *1 – (number of similarity clusters / number of benchmarks in the suite).* As such, a benchmark suite with as much clusters as benchmarks, will have a redundancy of zero. A benchmark suite on the other hand with several benchmarks in the same cluster will have a redundancy greater than zero. Obviously, smaller is better. We use the clustering from Table 5 to calculate the redundancy of the various CPU suites: 14.3% for CPU89, 53.3% for CPU92, 56.2% for CPU95 and 54.5% for CPU2000. We thus conclude that CPU89 has the lowest redundancy and that from then on the redundancy remained more or less constant in subsequent generations of SPEC CPU.

4.5 Should experimentation using older suites be condemned?

Often researchers and reviewers get upset with the use of benchmarks from SPEC CPU95 or older suites. It is however interesting to observe that in the clustering based on overall characteristics (Table 5) and data memory access characteristics (Table 6), several programs from the older suites appear clustered with newer programs. While vendors should use newest suites for the *SPECmark* numbers, an occasional use of an older, shorter benchmark by a researcher to reduce simulation time with complex cycle-accurate simulators is not that sinful, especially if the relative position of the used benchmark in the benchmark space is known. We hope that data in Tables 5 and 6 will be very useful to the research community for such analysis.

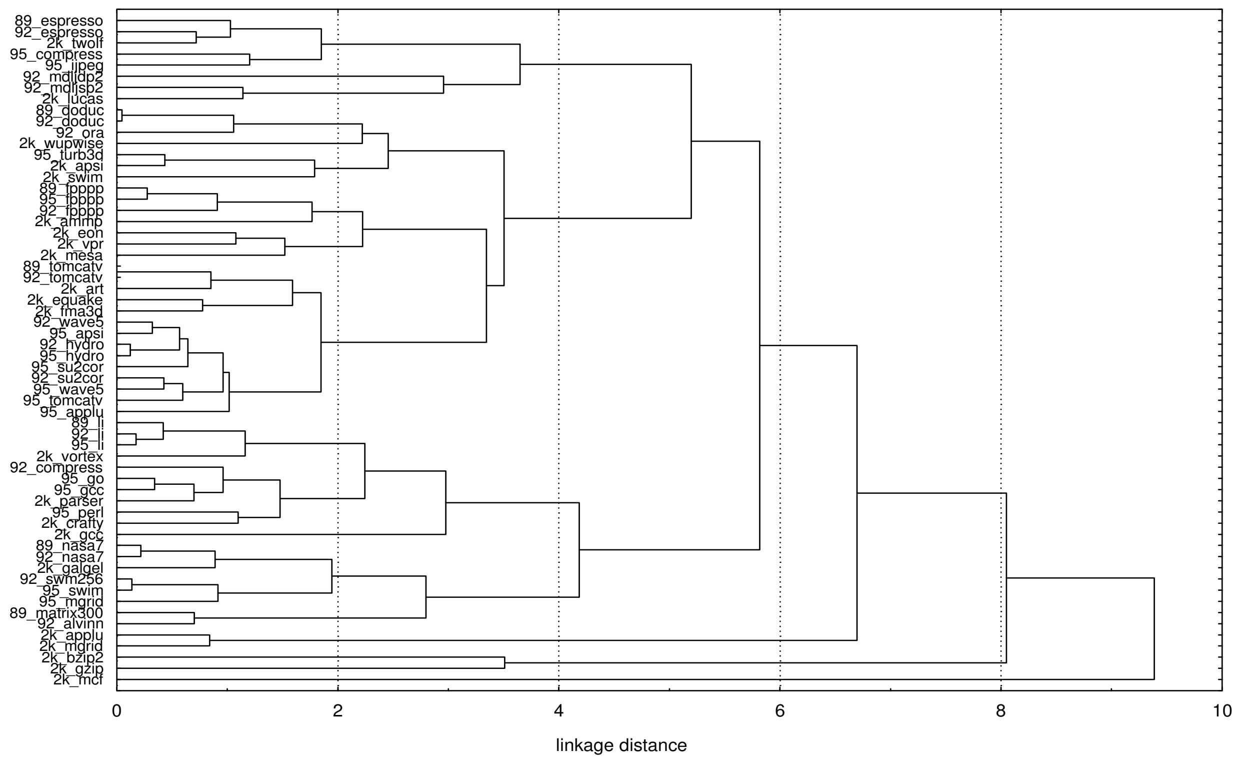

Figure 6: Dendrogram obtained through complete linkage clustering in the PCA space**.**

4.6 Speculations about future SPEC CPU suites

We believe that the results from this paper are also useful for the designers of future computer systems. Indeed, designing a new microprocessor is extremely time-consuming taking up to seven years [26]. As a result of that, a future computer system will be designed using yesterday's benchmarks. This might lead to a suboptimal design if the designers do not anticipate future program characteristics. The results from this paper suggest that the temporal locality of future benchmarks will continue to get worse.

cluster 1	nasa7 (89)	**cluster 6**	***gzip (2000)***
	matrix300 (89)		bzip2 (2000)
	alvinn (92)		
	swm256 (92)	**cluster 7**	tomcatv (89)
	nasa7 (92)		wave5 (92)
	swim (95)		tomcatv (92)
	mgrid (95)		***su2cor (92)***
	galgel (2000)		hydro2d (92)
			tomcatv (95)
cluster 2	***espresso (89)***		su2cor (95)
	espresso (92)		hydro2d (95)
	compress (95)		applu (95)
	twolf (2000)		apsi (95)
			wave5 (95)
cluster 3	***mdljsp2 (92)***		equake (2000)
	mdljdp2 (92)		fma3d (2000)
	ijpeg (95)		art (2000)
	lucas (2000)		
		cluster 8	***mcf (2000)***
cluster 4	***mgrid (2000)***		
	applu (2000)	**cluster 9**	***doduc (89)***
			doduc (92)
cluster 5	li (89)		ora (92)
	li (92)		turbo3d (95)
	compress (92)		apsi (2000)
	go (95)		swim (2000)
	li (95)		wupwise (2000)
	perl (95)		ammp (2000)
	gcc (95)		
	crafty (2000)	**cluster 10**	fpppp (89)
	gcc (2000)		***fpppp (92)***
	parser (2000)		fpppp (95)
	vortex (2000)		eon (2000)
	vpr (2000)		mesa (2000)

Table 5: Clustering the SPEC CPU benchmarks using the overall characteristics.

Indeed, from our detailed analysis of the temporal data stream locality we observed that several CPU95 and CPU2000 benchmarks exhibit poor temporal locality compared to CPU89 and CPU92. As such, we anticipate that this will continue to be the fact in future benchmarks and we thus recommend computer designers to design well performing memory subsystems in future microprocessors to deal with the increasingly poor temporal locality of computer applications.

cluster 1	***bzip2 (2000)***	**cluster 11**	doduc (89)
			fpppp (89)
cluster 2	***gcc (2000)***		doduc (92)
			mdljdp2 (92)
cluster 3	***gzip (2000)***		wave5 (92)
			ora (92)
cluster 4	***mcf (2000)***		mdljsp2 (92)
			swm256 (92)
cluster 5	***wupwise (2000)***		su2cor (92)
			hydro2d (92)
cluster 6	***applu (2000)***		nasa7 (92)
	vortex (2000)		fpppp (92)
	vpr (2000)		tomcatv (95)
	mgrid (2000)		swim (95)
			su2cor (95)
cluster 7	***mgrid (95)***		***hydro2d (95)***
	applu (95)		apsi (95)
	crafty (2000)		fpppp (95)
			wave5 (95)
cluster 8	li (89)		eon (2000)
	li (92)		galgel (2000)
	li (95)		lucas (2000)
	ijpeg (95)		swim (2000)
	compress (95)		
	go (95)	**cluster 12**	espresso (89)
	ammp (2000)		alvinn (92)
			espresso (92)
cluster 9	***turb3d (95)***		***perl (95)***
	apsi (2000)		equake (2000)
			art (2000)
cluster 10	nasa7 (89)		
	matrix300 (89)	**cluster 13**	***parser (2000)***
	tomcatv (89)		twolf (2000)
	tomcatv (92)		gcc (95)
	fma3d (2000)		compress (92)
	mesa (2000)		

Table 6: Clustering the SPEC CPU benchmarks based on data stream locality metrics.

4.7 Recommendations to SPEC

We believe that the results from this paper suggest several recommendations to SPEC for the design of future CPU suites. In general, the static instruction count of any commerical software application binary tends to increase with every generation as the software application evolves with increase in new features and functionality. However, we observe that the static instruction count in SPEC CPU benchmark binaries has not significantly increased during the last four generations. We therefore recommend that SPEC should select programs with higher static instruction count in binaries when designing next generation of benchmark suites. Another important recommendation we make to SPEC is to reduce the redundancy in the CPU suites. According to the results of the previous section, around 50% of the most recent CPU suites were redundant. This suggests that SPEC can build an equally well representative benchmark suite with 50% less benchmarks. We believe that the methodology as used in this paper could be used by SPEC in its search for representative, non-redundant next generation CPU benchmark suites. Another recommendation we make to SPEC is to broaden the scope of applications. Our results indicate that SPEC was successful in this respect when designing CPU2000; the CPU2000 benchmarks reside in all 10 clusters from Table 5, whereas previous CPU suites only resided in 6 or 7 clusters.

5. Related Work

Weicker [25] used characteristics such as statement distribution in programs, distribution of operand data types, and distribution of operations, to study the behavior of several stone age benchmarks. Saveedra et al. [24] characterized Fortran applications in terms of number of various fundamental operations, and predicted their execution time. They also develop a metric for program similarity that makes it possible to classify benchmarks with respect to a large set of characteristics. Source code level characterization has not gained popularity due to the difficulty in standardizing and comparing the characteristics across various programming languages. Moreover, nowadays, programmers rely on compilers to perform even basic optimizations, and hence source code level comparison may be unfair.

The majority of ongoing work in studying benchmark characteristics involves measuring microarchitecture level metrics such as cycles per instruction, cache miss rate, branch prediction rate etc. on various microarchitecture configurations that offer a different mixture of bottlenecks [12][15][16][17][27]. The variation in the microarchitecture metrics is then used to infer the generic program behavior. These inferred program characteristics, although seemingly microarchitecture-independent, may be biased by the idiosyncrasies of a particular configuration, and therefore may not be generally applicable.

Past attempts to understand benchmark redundancy used microarchitecture dependent metrics such as execution time or *SPECmark*. Vandierendonck et. al. [7] analyzed the SPEC CPU2000 benchmark suite peak results on 340 different machines representing eight architectures, and used PCA to identify the redundancy in the benchmark suite. Dujmovic and Dujmovic [9] developed a quantitative approach to evaluate benchmark suites. They used the execution time of a program on several machines and used this to calculate metrics that measure the size, completeness, and redundancy of the benchmark space. The shortcoming of these two approaches is that the inferences are based on the measured performance metrics due the interaction of program and machine behavior, and not due to the generic characteristics of the benchmarks. Ranking programs based on microarchitecture dependent metrics can be misleading for future designs because a benchmark might have looked redundant in the analysis merely because all existing architectures did equally well (or worse) on them, and not because that benchmark was uninteresting. The relatively lower rank of *gcc* in [7] and its better position in this work (Tables 5 and 6) is an example of such differences that become apparent only with microarchitecture-independent studies.

There has been some research on microarchitecture-independent locality and ILP metrics. For example, locality models researched in the past include working set models, least recently used stack models, independent reference models, temporal density functions, spatial density functions, memory reuse distance, locality space etc. [4][5][11][18][21][30][31]. Generic measures of parallelism were used by Noonburg et. al. [3] and Dubey et. al. [22] based on a profile of dependency distances in a program. Sherwood et. al. [32] proposed basic block distribution analysis for finding program phases which are representative of the entire program. Microarchitecture-independent metrics such as true computations versus address computations and overhead memory accesses versus true memory accesses have been proposed by several researchers [8][19]. This paper can benefit from more microarchitecture-independent metrics, but we believe that the metrics we have used cover a wide enough range of the program characteristics to make a meaningful comparison between the programs.

6. Conclusion

With the objective of understanding the SPEC CPU benchmarks since the inception of SPEC, we characterized 18 different microarchitecture-independent features of 60 SPEC CPU programs from SPEC89 to SPEC2000 suites. Analyzing the executables generated by compiling these programs on state of the art compilers with full optimization levels, we put the programs into a common perspective and examined the trends. Instruction mix, control flow, data locality, and parallelism characteristics were studied. No single characteristic has changed as dramatically as the dynamic instruction count. We observe that the dynamic instruction count of the programs has grown 100X on an average, and this trend will continue to meet the increasing processor performance. Surprisingly, the static instruction count in the binary has remained more or less constant and we feel that there should be an effort to increase it in the future generation of benchmark suites. Our analysis shows that the branch and ILP characteristics have not changed much over the last 1.5 decades, but the temporal data locality of programs has become increasingly poor, and we expect that the trend will continue. Although the diversity of newer generations of SPEC CPU benchmarks has increased, about half of the programs in SPEC CPU 2000 are redundant. While researchers in the past have picked subsets of suites based on convenience, we have presented results of clustering analysis based on several innate program characteristics and our results should be useful to select representative subsets (should experimentation with the whole suite be prohibitively expensive). We have also put program from four different suites into a common perspective, in case anyone wanted to compare results of particular programs from past suites with the newest programs.

Our recommendations to SPEC would be to continue broadening the diversity of programs in the future generation of benchmark suites while at the same time reduce the redundancy in programs, and increase the static instruction count in the program binaries. We also recommend that computer architects and researchers should concentrate on designing well performing memory hierarchies in anticipation of increasingly poor temporal data locality in future generation of SPEC CPU benchmark programs.

References

[1] A. Jain and R. Dubes, *Algorithms for Clustering Data*, Prentice Hall, 1988.

[2] D. Citron, “MisSPECulation: Partial and Misleading Use of SPEC CPU2000 in Computer Architecture Conferences”, *Proc. of International Symposium on Computer Architecture*, pp. 52-61, 2003.

[3] D. Noonburg and J. Shen, “A Framework for Statistical Modeling of Superscalar Processor Performance”, *Proc. of International Symposium on High Performance Computer Architecture*, 1997, pp. 298-309.

[4] E. Sorenson and J.Flanagan, “Cache Characterization Surfaces and Prediction of Workload Miss Rates”, *Proc. of International Workshop on Workload Characterization*, pp. 129-139, Dec 2001.

[5] E. Sorenson and J.Flanagan, “Evaluating Synthetic Trace Models Using Locality Surfaces”, *Proceedings of the Fifth IEEE Annual Workshop on Workload Characterization*, pp. 23-33, November 2002.

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

[7] H. Vandierendonck, K. Bosschere, “Many Benchmarks Stress the Same Bottlenecks”, *Proc. of the Workshop on Computer Architecture Evaluation using Commerical Workloads (CAECW-7)*, pp. 57-71, 2004.

[8] Hammerstrom, Davdison, “Information content of CPU memory referencing behavior”, *Proc. of International Symposium on Computer Architecture*, pp. 184-192, 1977.

[9] J. Dujmovic and I. Dujmovic, "Evolution and Evaluation of SPEC benchmarks", *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 3, pp. 2-9, 1998.

[10] J. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millenium", *IEEE Computer*, pp. 28-35, July 2000.

[11] J. Spirn and P. Denning, "Experiments with Program Locality", *The Fall Joint Conference*, pp. 611-621, 1972.

[12] J.Yi, D. Lilja, and D.Hawkins, "A Statistically Rigorous Approach for Improving Simulation Methodology", Proc. of International Conference on High-Performance Computer Architecture, pp. 281-291,2003.

[13] K. Dixit, "Overview of the SPEC benchmarks", *The Benchmark Handbook*, Ch. 9, Morgan Kaufmann Publishers, 1998.

[14] K. Skadron, M. Martonosi, D.August, M.Hill, D.Lilja, and V.Pai. "Challenges in Computer Architecture Evaluation." *IEEE Computer*, Aug. 2003.

[15] L. Barroso, K. Ghorachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads", *Proc. of the International Symposium on Computer Architecture*, pp. 3-14, 1998.

[16] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Designing computer architecture research workloads", *IEEE Computer*, 36(2), pp. 65-71, Feb 2003.

[17] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Quantifying the impact of input data sets on program behavior and its applications", Journal of Instruction Level Parallelism, vol 5, pp. 1-33, 2003.

[18] L. John, P. Vasudevan and J. Sabarinathan, "Workload Characterization: Motivation, Goals and methodology", In L. K. John and A. M. G. Maynard (Eds), Workload Characterization: Methodology and Case Studies, *IEEE Computer Society,* 1999.

[19] L. John, V. Reddy, P. Hulina, and L. Coraor, "Program Balance and its impact on High Performance RISC Architecture", *Proc. of the International Symposium on High Performance Computer Architecture*, pp.370-379, Jan 1995.

[20] N. Mirghafori, M. Jacoby, and D. Patterson, "Truth in SPEC Benchmarks", *Computer Architecture News* vol. 23,no. 5, pp. 34-42, Dec 1995.

[21] P. Denning, "The Working Set Model for Program Behavior", *Communications of the ACM*, vol 2, no. 5, pp. 323-333, 1968.

[22] P. Dubey, G. Adams, and M. Flynn, "Instruction Window Size Trade-Offs and Characterization of Program Parallelism", *IEEE Transactions on Computers*, vol. 43, no. 4, pp. 431-442, 1994.

[23] R. Giladi and N. Ahituv, " SPEC as a Performance Evaluation Measure", *IEEE Computer*, pp. 33-42, Aug 1995.

[24] R. Saveedra and A. Smith, "Analysis of benchmark characteristics and benchmark performance prediction", *Proc. of ACM Transactions on Computer Systems*, vol. 14, no.4, pp. 344-384, 1996.

[25] R. Weicker, "An Overview of Common Benchmarks", *IEEE Computer*, pp. 65-75, Dec 1990.

[26] S. Mukherjee, S. Adve, T. Austin, J. Emer, and P. Magnusson, "Performance Simulation Tools" , *IEEE Computer*, Feb 2002.

[27] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations", *Proc. of International Symposium on Computer Architecture*, pp. 24-36, June 1995.

[28] Standard Performance Evaluation Corporation, http://www.spec.org/benchmarks.html.

[29] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling", *IEEE Computer*, pp. 59-67, Feb 2002.

[30] T. Conte, and W. Hwu, "Benchmark Characterization for Experimental System Evaluation", *Proc. of Hawaii International Conference on System Science*, vol. I, Architecture Track, pp. 6-18, 1990.

[31] T. Lafage and A. Seznec, "Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A Preliminary Application to the Data Stream", *Workshop on Workload Characterization (WWC-2000)*, Sept 2000.

[32] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior", *Proc. of International Conference on Architecture Support for Programming Languages and Operating Systems*, pp. 45-57, 2002.

Appendix A – Measured value of Microarchitecture Independent Metrics for SPEC CPU benchmarks

	Static Instruction Count						Branch Metrics				Data Locality Metrics				Dependecy Distance Metrics						
	From Binary	Instr executed at least once	%Memory	%Branches	Computation:Memory	Basic Block Size	%Fwd	%taken	%Fwd-T-of total br	%Back-T-of-total br	Tlocality16	Tlocality64	Tlocality25	Tlocality4096	1	Upto 2	Upto 4	Upto 8	Upto 16	Upto 32	> 32
espresso_89	106,416	20,913	26.66	15.92	2.15	5.28	0.63	0.64	0.47	0.53	313.00	103.00	31.00	6.00	28.25	40.94	54.92	65.36	76.79	83.64	16.36
li_89	70,164	10,074	41.13	16.74	1.02	4.98	0.66	0.65	0.63	0.37	138.00	63.00	36.00	7.00	27.71	39.01	48.88	62.15	77.33	88.73	11.27
doduc_89	221,660	26,006	34.51	7.74	1.67	11.91	0.80	0.49	0.64	0.36	499.00	628.00	201.00	28.00	7.38	13.95	24.38	36.87	50.31	64.44	35.56
nasa7_89	205,053	17,463	46.24	2.47	1.11	39.56	0.26	0.84	0.14	0.86	338.00	593.00	182.00	25.00	3.38	6.46	14.79	31.49	44.51	60.90	39.10
matrix300_89	194,397	9,308	35.15	3.13	1.76	30.94	0.05	0.95	0.01	0.99	21312.00	1771.00	236.00	24.00	9.41	16.95	32.05	60.26	73.30	77.13	22.87
fpppp_89	210,524	20,457	43.36	1.29	1.28	76.73	0.82	0.51	0.72	0.28	2418.00	850.00	230.00	30.00	1.11	2.39	5.09	16.61	32.24	45.80	54.20
tomcatv_89	194,517	9,375	39.31	2.78	1.47	34.97	0.53	0.99	0.53	0.47	575.00	603.00	171.00	21.00	2.71	3.67	6.47	15.32	33.71	49.89	50.11
doduc_92	221,660	26,007	34.51	7.74	1.67	11.91	0.80	0.49	0.64	0.36	505.00	631.00	201.00	28.00	7.37	14.16	24.97	37.40	50.98	67.23	35.56
mdljdp2_92	215,667	16,396	24.72	12.65	2.53	6.91	0.86	0.84	0.83	0.17	1230.00	656.00	208.00	33.00	18.94	22.90	35.34	42.83	55.07	63.31	36.69
wave5_92	238,629	22,934	35.75	4.63	1.67	20.62	0.49	0.73	0.34	0.66	1020.00	576.00	184.00	27.00	5.07	10.12	18.91	32.31	44.30	57.15	42.85
tomcatv_92	194,517	9,375	39.31	2.78	1.47	34.97	0.53	0.99	0.53	0.47	575.00	605.00	172.00	22.00	2.71	3.67	6.47	15.32	33.71	49.89	50.11
ora_92	198,304	10,303	29.64	6.88	2.14	13.54	0.78	0.57	0.63	0.37	393.00	622.00	206.00	34.00	7.61	20.17	35.77	45.72	55.98	69.15	30.85
alvinn_92	60,521	6,034	36.48	10.32	1.46	8.69	0.04	0.98	0.02	0.98	54.00	33.00	15.00	2.00	12.10	23.28	34.55	55.94	70.06	70.95	29.05
mdljsp2_92	215,695	16,532	23.05	3.52	3.18	27.39	0.53	0.66	0.30	0.70	502.00	649.00	210.00	32.00	7.70	14.52	27.17	38.03	48.23	61.88	38.12
swm256_92	200,721	12,673	37.43	0.63	1.65	157.91	0.05	0.95	0.02	0.98	458.00	637.00	207.00	32.00	1.22	2.33	5.16	12.07	27.98	42.69	57.31
su2cor_92	219,179	22,779	38.84	2.81	1.50	34.64	0.46	0.78	0.32	0.68	2397.00	971.00	300.00	36.00	2.71	5.27	12.26	23.77	39.04	51.08	48.92
hydro2d_92	216,278	24,337	36.84	6.00	1.55	15.66	0.54	0.75	0.41	0.59	1294.00	672.00	217.00	35.00	3.63	8.04	13.83	26.70	42.76	58.54	41.46
nasa7_92	213,052	21,133	46.15	2.57	1.11	37.86	0.28	0.83	0.16	0.84	406.00	616.00	191.00	27.00	3.67	5.77	12.76	29.99	42.64	57.51	42.49
fpppp_92	210,523	20,452	44.96	2.05	1.18	47.82	0.79	0.61	0.75	0.25	3167.00	1161.00	273.00	30.00	2.35	4.40	8.76	21.30	36.00	48.89	51.11
espresso_92	102,483	19,843	27.85	17.10	1.98	4.85	0.63	0.64	0.47	0.53	309.00	106.00	37.00	6.00	45.47	59.11	65.88	70.40	77.95	82.85	17.15
li_92	68,660	9,494	42.53	17.65	0.94	4.67	0.67	0.65	0.63	0.37	139.00	61.00	34.00	8.00	36.83	44.47	53.38	65.42	79.15	89.57	10.43
compress_92	46,080	3,418	33.97	12.05	1.59	7.30	0.77	0.52	0.58	0.42	10178.00	1693.00	100.00	4.00	21.53	36.54	51.02	61.76	71.85	80.82	19.18
tomcatv_95	203,525	13,269	37.56	1.82	1.61	53.98	0.39	0.75	0.20	0.80	477.00	221.00	221.00	26.00	1.68	3.18	5.35	17.06	34.31	49.44	50.56
swim_95	204,144	12,876	37.40	0.62	1.66	160.73	0.03	0.97	0.01	0.99	461.00	643.00	210.00	33.00	1.25	2.52	5.63	13.82	28.15	43.46	56.54
su2cor_95	218,311	25,086	37.70	3.62	1.56	26.62	0.57	0.70	0.39	0.61	4175.00	910.00	291.00	33.00	4.26	7.81	14.87	26.58	41.32	52.97	47.03
hydro2d_95	214,881	23,525	36.55	5.82	1.58	16.20	0.54	0.78	0.41	0.59	1607.00	698.00	218.00	31.00	3.99	9.20	14.93	27.30	43.10	59.09	40.91
applu_95	*	*	34.76	3.68	1.77	26.20	0.32	0.62	0.27	0.73	93989.00	720.00	207.00	32.00	1.94	5.98	9.52	21.53	36.45	47.82	52.18
turb3d_95	213,613	21,290	37.88	3.30	1.55	29.28	0.49	0.60	0.35	0.65	1113236.00	124651.00	1078.00	38.00	3.14	7.66	13.10	19.58	35.58	50.36	49.64
apsi_95	235,175	32,446	35.71	3.31	1.71	29.23	0.43	0.72	0.31	0.69	1155.00	705.00	222.00	34.00	3.24	6.97	11.70	21.32	37.20	53.88	46.12
fpppp_95	215,569	21,188	43.86	1.40	1.25	70.37	0.80	0.54	0.72	0.28	3166.00	804.00	204.00	32.00	1.30	2.79	5.70	17.66	33.50	47.00	53.00
wave5_95	241,194	26,677	39.67	3.35	1.44	28.84	0.42	0.76	0.25	0.75	465.00	659.00	221.00	33.00	4.54	8.53	18.59	30.60	42.04	55.51	44.49
mgrid_95	*	*	36.73	0.82	1.70	120.55	0.19	0.83	0.11	0.89	81269.00	693.00	214.00	28.00	0.46	2.16	5.03	16.00	33.23	43.60	56.40
go_95	129,840	68,562	36.95	13.04	1.35	6.67	0.76	0.66	0.70	0.30	2856.00	548.00	69.00	9.00	21.34	33.31	46.90	57.76	69.62	79.89	20.11
li_95	44,316	9,607	41.36	18.05	0.98	4.54	0.65	0.64	0.62	0.38	1369.00	278.00	103.00	10.00	37.60	45.49	54.28	66.53	78.39	88.75	11.25
perl_95	137,680	17,219	40.80	16.72	1.04	4.98	0.85	0.67	0.79	0.21	153.00	81.00	42.00	5.00	24.01	35.24	48.12	59.64	72.34	83.13	16.87
gcc_95	372,848	143,153	37.92	14.91	1.24	5.70	0.75	0.62	0.66	0.34	7157.00	3412.00	730.00	5.00	24.64	35.38	46.98	58.24	72.03	82.26	17.74
compress_95	59,983	1,556	32.59	11.52	1.71	7.68	0.59	0.79	0.54	0.46	109.00	49.00	27.00	7.00	18.01	29.98	45.74	62.28	76.04	86.06	13.94
ijpeg_95	108,404	20,199	28.35	5.45	2.33	17.33	0.59	0.75	0.50	0.50	1700.00	195.00	34.00	9.00	14.40	24.37	37.88	50.60	62.18	79.48	20.52
bzip2_2k	38,479	9,633	39.50	12.29	1.22	8.14	0.63	0.70	0.56	0.44	337042.00	100375.00	69024.00	1875.00	31.42	35.46	57.57	73.12	86.49	90.60	9.40
crafty_2k	116,296	34,073	36.60	11.20	1.43	8.93	0.83	0.67	0.80	0.20	31962.00	7635.00	294.00	21.00	13.80	24.51	38.62	52.66	64.37	72.75	27.25
eon_2k	189,016	43,503	48.15	11.18	0.84	8.94	0.67	0.63	0.59	0.41	3622.00	707.00	229.00	28.00	6.75	11.89	21.40	31.91	48.05	62.04	37.96
gcc2k	446,281	180,588	53.26	10.68	0.68	9.36	0.58	0.71	0.43	0.57	26246.00	7112.00	2705.00	307.00	22.81	29.63	44.87	51.53	68.92	75.86	24.14
gzip_2k	42,079	9,338	32.17	10.44	1.78	9.58	0.72	0.70	0.62	0.38	3484076.00	296272.00	120821.00	2579.00	22.12	33.67	43.96	61.23	69.05	74.19	25.81
mcf_2k	33,221	8,026	37.27	21.10	1.12	4.74	0.63	0.64	0.53	0.47	6384474.00	801795.00	309.00	8.00	19.47	34.29	46.45	58.32	68.91	72.19	27.81
parser_2k	65,607	25,345	34.84	15.48	1.43	6.46	0.65	0.65	0.50	0.50	24700.00	1816.00	175.00	9.00	20.47	32.34	49.97	61.18	74.00	83.41	16.59
twolf_2k	98,360	35,318	32.28	12.08	1.72	8.28	0.62	0.57	0.48	0.52	21792.00	1240.00	102.00	6.00	21.94	38.78	62.77	80.11	87.12	90.09	9.91
vortex_2k	163,748	69,692	40.53	17.29	1.04	5.78	0.83	0.52	0.69	0.31	315137.00	27783.00	1419.00	60.00	41.77	49.78	60.82	73.40	83.80	91.69	8.31
vpr_2k	73,791	29,901	44.08	10.65	1.03	9.39	0.68	0.52	0.44	0.56	524568.00	15223.00	1829.00	4.00	11.51	13.20	15.32	44.36	65.44	71.24	28.76
applu_2k	241,202	76,864	38.17	0.31	1.61	317.61	0.26	0.69	0.04	0.96	557233.00	3638.00	218.00	34.00	1.22	2.49	5.23	13.12	28.24	40.80	59.20
apsi_2k	*	*	37.22	3.60	1.59	27.80	0.55	0.55	0.39	0.61	1621949.00	106372.00	202.00	25.00	1.96	6.04	10.95	22.26	36.95	49.38	50.62
equake_2k	*	*	44.29	4.15	1.16	24.08	0.52	0.87	0.50	0.50	42.00	25.00	11.00	4.00	6.21	9.24	14.09	26.57	40.10	49.49	50.51
fma3d_2k	*	*	43.99	4.10	1.18	24.39	0.54	0.71	0.43	0.57	1225.00	661.00	202.00	19.00	1.69	3.21	7.63	20.22	34.74	48.42	51.58
galgel_2k	238,133	47,073	43.66	5.24	1.17	19.07	0.07	0.87	0.00	1.00	462.00	641.00	207.00	33.00	3.44	9.46	14.45	19.18	44.14	56.25	43.75
lucas_2k	*	*	22.13	1.43	3.45	69.91	0.36	0.62	0.02	0.98	382.00	597.00	191.00	30.00	4.07	5.99	12.18	21.90	36.33	47.99	52.01
mesa_2k	*	*	38.54	17.59	1.14	5.69	0.76	0.62	0.68	0.32	1337.00	442.00	142.00	17.00	7.97	15.33	20.86	28.23	37.82	52.71	47.29
mgrid_2k	178,553	18,349	36.72	0.37	1.71	273.37	0.41	0.65	0.19	0.81	689344.00	1349.00	247.00	34.00	1.77	3.65	9.51	28.76	40.90	48.61	51.39
swim_2k	181,467	20,283	32.92	1.30	2.00	76.66	0.41	0.59	0.01	0.99	1163.00	622.00	201.00	30.00	0.85	1.48	3.67	5.32	26.59	33.57	66.43
wupwise_2k	180,263	18,037	30.78	9.76	1.93	10.24	0.67	0.37	0.56	0.44	768641.00	192694.00	48236.00	36.00	0.74	5.25	17.95	27.46	37.66	47.08	52.92
art_2k	*	*	34.72	13.09	1.50	7.64	0.50	0.86	0.46	0.54	10102.00	25.00	13.00	7.00	7.28	12.24	16.49	28.90	36.68	45.75	54.25
ammp_2k	*	*	38.34	7.49	1.41	13.36	0.71	0.35	0.32	0.68	8928.00	196.00	79.00	9.00	9.07	16.23	27.00	37.49	46.27	56.03	43.97

** These are long running programs & their results are awaited*