Measuring Program Similarity

Aashish Phansalkar [†], Ajay Joshi [†], Lieven Eeckhout [‡], and Lizy K. John [†]
{aashish, ajoshi, ljohn}@ece.utexas.edu, leeckhou@elis.ugent.be
[†]University of Texas, Austin [‡]Ghent University, Belgium

Abstract

Performance evaluation using only a subset of programs from a benchmark suite is commonplace in computer architecture research. This is especially true during early design space exploration when a variety of enhancements need to be evaluated to reach a good microprocessor architecture in a limited amount of time. When such a subset of benchmark programs is used for performance evaluation of architectural enhancements, it is essential that the subset is well distributed within the target workload space rather than clustered in specific areas. Past efforts for identifying subsets have primarily relied on using microarchitecture-dependent metrics of program performance, such as cycles per instruction and cache miss-rate. The shortcoming of this technique is that the results could be biased by the idiosyncrasies of the chosen configurations.

We believe that a technique based on measuring the inherent characteristics of a program will make the results applicable to any microarchitecture. The objective of this paper is to present a methodology to measure similarity of programs based on their inherent microarchitecture-independent characteristics. We apply our methodology to the SPEC CPU2000 benchmark suite and demonstrate that a subset of 8 programs can be used to effectively represent the entire suite. We validate the usefulness of this subset by using it to estimate the average IPC, speedup, and L1 data cache miss-rate of the entire suite. The average IPC of 8-way and 16-way issue superscalar processor configurations could be estimated with 3.9% and 4.4% error respectively. This methodology is applicable not only to find subsets from a benchmark suite, but also to identify programs for a benchmark suite from a list of potential candidates.

We also apply the microarchitecture-independent program characterization methodology to understand how the inherent characteristics of programs in four generations of SPEC CPU benchmark suites have evolved over the last decade. 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 more or less remained the same.

1. Introduction

During the early design space exploration phase of the microprocessor design process, a variety of enhancements and design options are evaluated by analyzing the performance model of the microprocessor. Simulation time is limited, and hence it is often required to use only a subset of the benchmark programs to evaluate the enhancements and design options. A poorly chosen set of benchmark programs may not accurately depict the true performance of the processor design. On one hand, selecting the wrong set of benchmarks could incorrectly estimate the performance of a particular enhancement; while on the other hand, simulating similar programs will increase simulation time without providing additional information. Therefore, a good workload should have programs that are well distributed within the target workload space without being clustered in specific areas. Understanding similarity between programs can help in selecting benchmark programs that are distinct, but are still representative of the target workload space. A typical approach to study similarity in programs is to measure program characteristics and then use statistical data analysis techniques to group programs with similar characteristics.

Programs can be characterized using implementation (machine) dependent metrics such as cycles per instruction (CPI), cache miss-rate, and branch prediction accuracy, or microarchitecture-independent metrics such as temporal locality, and parallelism. Techniques that have been previously proposed primarily concentrate on measuring microarchitecture-dependent characteristics of programs [7] [17]. This involves measuring program performance characteristics such as instruction and data cache miss-rate, branch prediction accuracy, CPI, and execution time across multiple microarchitecture configurations. The results obtained from these techniques could be biased by the idiosyncrasies of a particular microarchitecture if the program behavior is not observed across a carefully chosen range of microarchitecture configurations. Moreover, conclusions based on performance metrics such as execution time could categorize a program with unique characteristics as insignificant, only because it shows similar trends on the microarchitecture configurations used in the study. For instance, a prior study [7] ranked programs in the SPEC CPU 2000 benchmark suite using the SPEC peak performance rating. The program ranks were based on their uniqueness i.e. the programs that exhibit different speedups on most of the machines were given a higher rank as compared to other programs in the suite. In this scheme of ranking programs, *gcc* ranks very low, and seems to be less unique. However, our results show that the inherent characteristics of *gcc* are significantly different from other programs in the benchmark suite. This indicates that analysis based on microarchitecture-dependent metrics could undermine the importance of a program that is really unique.

We believe that by measuring the inherent characteristics of a program, it is possible to ensure that the results of such experiments will be applicable to any microarchitecture. The objective of this paper is to present a technique to measure similarity of programs based on their microarchitecture-independent characteristics, and demonstrate its application to find a representative subset of programs from the SPEC CPU 2000 benchmark suites. We also use the methodology presented in this paper to understand similarity in program characteristics across four generations of SPEC CPU benchmark suites.

In this study we classify two programs to be similar if they have similar inherent characteristics such as instruction locality, data locality, branch predictability, and instruction level parallelism (ILP). In order to remove the correlation between the measured metrics, and make it possible to visualize the program workspace, we use a multivariate statistical data analysis technique called principal component analysis (PCA) to reduce the dimensionality of the data while retaining most of the information. We then use the K-means clustering algorithm to group programs that have similar inherent characteristics.

Following are the contributions of this paper:

(i) The paper motivates and presents an approach that can be used to measure similarity between programs in a microarchitecture-independent manner.

(ii) The paper finds a subset of programs from the SPEC CPU 2000 benchmark suite. We demonstrate the usefulness of this subset by using it to estimate the average IPC of the entire suite for two different configurations of a microprocessor, and average L1 data cache miss-rate of the entire suite for 9 cache configurations.

(iii) The paper provides an insight into how characteristics of SPEC CPU benchmark suites have evolved since its inception in 1989.

The roadmap of this paper is as follows: In section 2 we describe a microarchitecture-independent methodology to characterize benchmarks. In section 3 we apply the presented methodology to find a subset of programs from the SPEC CPU 2000 benchmark suite and validate that these programs are indeed representative of the entire benchmark suite. Section 4 uses the presented methodology to provide a historical insight into how characteristics of SPEC CPU benchmark suites have changed over the last decade. In section 5 we describe the related work, and in section 6 summarize the key learning and contributions of this study.

2. Characterization Methodology

This section proposes our methodology to measure similarity between benchmark programs: the microarchitecture-independent metrics used to characterize the benchmarks, the statistical data analysis techniques, the benchmarks, and the tools.

2.1 Metrics

In this paper we use 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 that 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 our metrics cover a wide enough range of the program characteristics to make a meaningful comparison between the programs. Other program characteristics, such as value predictability, can also be added to the analysis if they are exploited by the microarchitecture, and hence determine program performance. 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.

Dynamic Basic Block Size: A basic block is a section of code with one entry and one exit point. We measure the dynamic basic block size as the average number of instructions between two consecutive branches in the dynamic instruction stream of the program. A larger basic block size is useful in exploiting instruction level parallelism (ILP).

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: This metric is defined as the ratio of taken branches to the total number of branches in the dynamic instruction stream of the program.

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

Dependency Distance: We use a distribution of dependency distances 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 (write) and the first consumption (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 dependency distances that are greater than 32 are likely to exhibit a higher ILP (provided control flow is not the limiting factor).

Data Temporal Locality: 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 quantified by computing the average distance (in terms of number of memory accesses) between two consecutive accesses to the same address, for every unique address in the program. The evaluation is performed in four distinct *window* sizes, analogous to cache block sizes. The *data_tlocality* metric is calculated for *window* sizes of 16, 64, 256 and 4096 bytes. The choice of the *window* sizes is based on the experiments conducted by Lafage et.al. [31]. Their experimental results show 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.

Data Spatial Locality: In order to measure spatial locality we computed the *data_tlocality* metric for four different *window* sizes: 16, 64, 256, and 4096 bytes. Spatial locality information is characterized by the ratio of the *data_tlocality* metric for *window* sizes mentioned above.

Instruction Temporal Locality: The instruction temporal locality metric is quantified by computing the average distance (in terms of number of instructions) between two consecutive accesses to the same static instruction (*instrn_tlocality*), for every unique static instruction in the program that is executed at least twice. The instruction temporal locality (*instrn_tlocality*) is calculated for window sizes of 16, 64, 256, and 4096 bytes.

Instruction Spatial Locality: Spatial locality of the instruction stream is characterized by the ratio of the *instrn_tlocality* metric for the *window* sizes mentioned above.

2.2 Statistical Data Analysis

Obviously, the amount of data in the analysis is huge. There are many variables (29 microarchitecture-independent characteristics) and many cases (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:

$$Z_i = \sum_{j=0}^{p} a_{ij} X_j$$

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 ($q \ll p$) that explain at least 75% 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: We use *K-means* clustering for our analysis [1]. *K-means clustering* 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 Benchmarks

The different benchmark programs used in this study and their dynamic instruction counts are shown in *Table 1*.

Due to the differences in libraries, data type definitions, pointer size conventions, and known compilation issues on 64-bit 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. 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. Although our results are microarchitecture-independent, they are dependent on the instruction set architecture (ISA) and the compiler. However, we feel that with CISC ISAs or RISC style micro-ops, our results will not change significantly.

2.4 Tools

***SCOPE*:** The workload characteristics were measured using a custom-grown analyser called *SCOPE*. *SCOPE* was developed by modifying the *sim-safe* functional simulator from the *SimpleScalar* 3.0 [29] tool set. *SCOPE* analyses the dynamic instruction stream and generates statistics related to instruction mix, data locality, branch predictability, basic-block size, and ILP. Essentially, the front-end of *sim-safe* is interfaced with homegrown analyzers to obtain various locality and parallelism metrics.

Statistical data analysis: We use STATISTICA version 6.1 for performing PCA. For K-means clustering we use the *SimPoint* software [32]. However, unlike *SimPoint* we do not use random projection before applying K-means clustering; instead, we use the transformed PCA space as the projected space.

3. Subsetting SPEC CPU2000 benchmark suite

Benchmark subsetting involves measuring the characteristics of benchmark programs and grouping programs with similar characteristics such as temporal locality, spatial locality, and branch predictability. A representative program from each group can then be selected for simulation, without losing significant information. In this section we apply the microarchitecture-independent technique to measure benchmark similarity presented in this paper, to the problem of finding a representative subsets of programs from the SPEC CPU 2000 benchmark suite. We measured the microarchitecture-independent characteristics mentioned in section 2 for the SPEC CPU 2000 benchmark programs from the SPEC CPU 2000 benchmark suite. We measured the microarchitecture-independent characteristics mentioned in section 2 for the SPEC CPU2000 benchmark programs and computed two subsets of programs, the first based on similarity in all the important program characteristics described in section

Program	Input	INT/ FP	Dynamic Instruction Count
SPEC CPU89			
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
SPEC CPU92			
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
SPEC CPU95			
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
SPEC CPU2000			
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	ref	INT	546.7 billion
eon	cook	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	*

Table 1: Programs from SPEC CPU benchmark suites used in the study

2, and the second based on similarity in data locality characteristics. We reduce the dimensionality of the data using the PCA technique described earlier in the paper. We then use K-means clustering algorithm, provided in the *SimPoint* software, to group programs based on similarity in the measured characteristics. The *SimPoint* software identifies the optimal number of clusters, K, by computing the minimal number of clusters for which the Bayesian Information Criterion (BIC) is optimal. The BIC is a measure of the goodness of fit of a clustering to a data set. In the following sections we describe two experiments to find a of programs in SPEC CPU 2000 benchmark suite, and validate that they are indeed representative of the entire benchmark suite.

3.1 Subsetting using overall program characteristics

We measured all the microarchitecture-independent program characteristics mentioned in section 2 for SPEC CPU 2000 programs (raw data is presented in *Appendix A*). Using the PCA and K-means clustering technique described above, we obtain 8 clusters as a good fit for the measured data set. *Table 2* shows the 8 clusters and their members. The programs marked in bold are closest to the center of their respective cluster and are hence chosen to be the representatives of that particular group. For clusters with just two programs, any program can be chosen as a representative. Citron [2] presented a survey on the use of SPEC CPU2000 benchmark programs in papers from four recent ISCA conferences. He observed that some programs are more popular than the others among computer architecture researchers.

The programs in the SPEC CPU2000 integer benchmark suite in their decreasing order of popularity are: *gzip, gcc, parser, vpr, mcf, vortex, twolf, bzip2, crafty, perlbmk, gap,* and *eon.* For the floating-point CPU2000 benchmarks, the list in decreasing order of popularity is: *art, equake, ammp, mesa, applu, swim, lucas, apsi, mgrid, wupwise, galgel, sixtrack, facerec* and *fma3d*. The clusters we obtained in *Table 2* suggest that the most popular programs in the listing provided by Citron [2] are not a truly representative subset of the benchmark suite (based on their inherent-characteristics). For example, subsetting SPEC CPU 2000 integer programs using *gzip*, *gcc*, *parser*, *vpr*, *mcf*, *vortex, twolf* and *bzip2* will result in three uncovered clusters, namely 1, 3 and 7. We also observe that there is a lot of similarity in the characteristics of the popular programs listed above. The three popular benchmarks *parser, twolf*, and *vortex* in the subset belong to the same cluster, Cluster 6, and hence do not provide any additional information. The results from *Table 2* suggest that using *applu, gzip, equake, fma3d, mcf, twolf, mesa,* and *gcc* as a representative subset of the SPEC CPU 2000 benchmark suite would be a better practice.

We observe that *gcc* is in a separate cluster by itself, and hence has characteristics that are significantly different from other programs in the benchmark suite. However, in the ranking scheme used in a prior study [7], *gcc* ranks very low and does not seem to be a very unique program. Their study uses microarchitecture-dependent metric, SPEC peak performance rating, and hence a program, such as *gcc*, that shows similar speedup on most of the machines will be ranked lower. This example shows that results based on analysis using microarchitecture-independent metrics can identify redundancy more effectively.

Cluster 1	*applu, mgrid*
Cluster 2	*gzip, bzip2*
Cluster 3	*equake, crafty*
Cluster 4	***fma3d***, *ammp, apsi, galgel, swim, vpr, wupwise*
Cluster 5	***mcf***
Cluster 6	***twolf*** *, lucas, parser, vortex*
Cluster 7	***mesa***, *art, eon*
Cluster 8	***gcc***

Table 2: Optimum number of clusters for SPEC CPU2000 benchmarks when measuring similarity based on locality, branch predictability and ILP program characteristics.

3.2 Subsetting using data locality characteristics

In this analysis we find a subset of the SPEC CPU2000 benchmark suite by only considering the 7 characteristics of SPEC CPU2000 programs that are closely related to the temporal and spatial data locality of a program viz. *data_tlocality* for *window* sizes of 16, 64, 256, and 4096 bytes, and the ratios of each of the *data_tlocality* metric for *window* sizes of 64, 256, and 4096 bytes, to the *data_tlocality* metric for *window* size of 16 bytes. The first four metrics measure temporal data locality of the program, whereas the remaining three characterize the spatial data locality of the program. We use the same methodology for data reduction and clustering as mentioned above. *Table 3* shows the groups of programs that have similar data locality characteristics.

3.3. Validating benchmark subsets

It is important to know whether the subsets we created are meaningful and are indeed representative of the SPEC CPU 2000 benchmark suite. We used the subsets to estimate the average IPC and L1 data cache miss-rate of the entire benchmark suite. We then compared our results with those obtained by using the entire benchmark suite.

Cluster 1	***gzip***
Cluster 2	***mcf***
Cluster 3	***ammp***, *applu, crafty, art, eon, mgrid, parser, twolf, vortex, vpr*
Cluster 4	***equake***
Cluster 5	***bzip2***
Cluster 6	*mesa, gcc*
Cluster 7	***fma3d***, *swim, apsi*
Cluster 8	*galgel, lucas*
Cluster 9	***wupwise***

Table 3: Optimum number of clusters for SPEC PU2000 benchmarks based on similarity in data locality characteristics

3.3.1 Computing IPC

Using the subset based on overall program haracteristics we calculated the average IPC of the entire suite for two different microarchitectures with issue widths of 8 and 16. *Figure 1* shows the average IPC of the entire benchmark suite calculated using the program subset, and also using every program in the benchmark suite.

We obtained the performance data of IPC on 8-way and 16-way issue widths for every program in the SPEC CPU2000 benchmarks from Wenisch et. al. [33]. The following are the microarchitecture details: 8-way (RUU-128, LSQ-64, Memory System - 32 KB 2-way L1 I/D, 2 ports, 8 MHSR, 1M 4-way L2, 16-entry store buffer, ITLB-4-way 128 entries, DTLB-4-way 256 entries – 200 cycle miss penalty, L1/L2/memory latency – 1/12/100 cycles, Functional Units 4 I-ALU, 2 I-MUL/DIV, 2 FP-ALU, 1 FP-MUL/DIV, and branch predictor – combined 2K tables 7 cycle misprediction penalty – 1 prediction/cycle), and 16-way (RUU-256, LSQ-128, Memory System – 64 KB 2-way L1 I/D, 4 ports, 16 MHSR, 2M 8-way L2, 32-entry store buffer, ITLB- 4-way 128 entries, DTLB-4-way 256 entries, 200 cycle miss penalty, L1/L2/memory latency – 2/16/100 cycles, Functional Units 16 I-ALU, 8 I-MUL/DIV, 8 FP-ALU, 4 FP-MUL/DIV, and branch predictor – combined 8K tables 10 cycle misprediction penalty – 2 predictions/cycle),

From *Table 2* we observe that each cluster has a different number of programs, and hence the weight assigned to each representative program should depend on the number of programs that it represents (i.e. the number of programs in its cluster). For example, from *Table 2*, the weight for *fma3d* (cluster 4) is 7. The error in average IPC computed using the subset of programs for both, 8-way and 16-way issue widths, is less than 5%. It also shows percentage error on top of the bar graphs for each of the configurations. If the IPC of the entire suite can be estimated with reasonable accuracy using the subsets, we feel that it is a good validation for the usefulness of the subset.

3.3.2 Computing data cache miss-rate.

Figure 2 shows average L1 data cache miss-rate of the benchmark suite estimated using the subset of programs obtained in section 3.2 along with the average miss-rate using the entire benchmark suite.

We obtained the miss-rates for 9 different L1 data cache configurations from Cantin et. al. [34]. As mentioned in the earlier section, the weight for each representative program is assigned as the number of programs it represents (i.e. the number of programs in its cluster). From these results we can conclude that the program subset derived in section 4.2 is indeed representative of the data locality characteristics of programs in SPEC CPU 2000 benchmark suite.

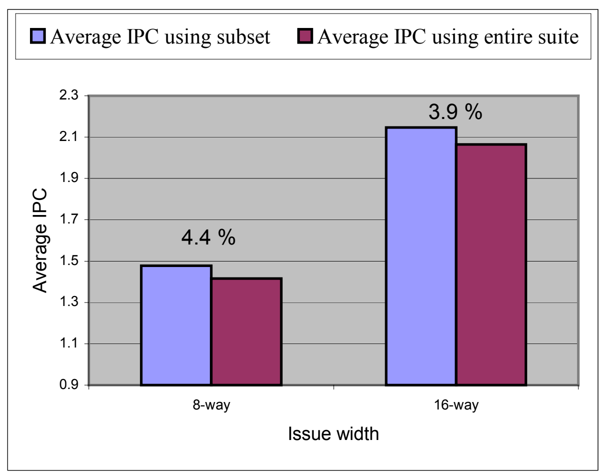

Figure 1: Estimated average IPC of benchmark suite using subset versus True average IPC of benchmark suite

We also used the subset based on overall characteristics (obtained in section 3.1) to estimate the average cache miss-rate of the entire suite; the results are also shown in *Figure 2.* Although the accuracy of the average cache miss-rate calculated using the subset based on overall characteristics is not as high as that of the subset based on locality characteristics, it is reasonably good. It is interesting to note that in 5 of the 9 cases, the clusters based on overall characteristics performed better in estimating the average miss-rate of the entire suite, than the clusters based on locality characteristics.

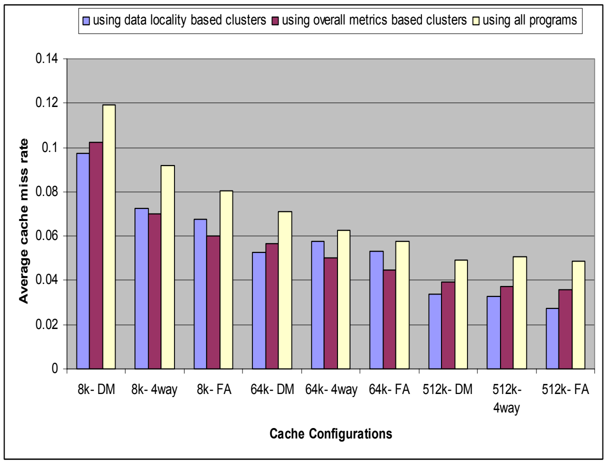

Figure 2: Average miss-rate of entire suite estimated using the subset based on locality characteristics, and the subset based on overall characteristics.

3.3.3 Computing execution speed-up

Figure 3 shows the estimated average (geometric mean) speedup of the entire suite using the subset based on overall program characteristics, and the true speedup of the entire suite for computers from various manufacturers. The speedup numbers were directly obtained from the results published by SPEC [38]. As described in Section 3.3.1, each representative program in the subset was assigned a weight corresponding to the number of programs that it represents (i.e. the number of programs in its cluster).

The maximum error in the speedup estimated using the subset is 9.1%. If the speedup of the entire suite can be estimated with reasonable accuracy using the subsets, we feel that it is a good validation for the usefulness of the subset.

3.3.4 Sensitivity to number of clusters

The number of representative programs to be chosen from a benchmark suite depends on the level of accuracy desired. Theoretically, as we increase the number of representative programs, the accuracy should increase i.e. the average miss-rate of the suite calculated using the subset will be closer to that calculated using the entire suite. In this section we show that the average miss-rate of the benchmark suite can be calculated with an increasing level of accuracy if we partition the programs into higher number of clusters i.e. more programs are chosen to represent the benchmark suite. The optimum number of clusters for subset using data locality characteristics is 9 according to the *SimPoint* algorithm. *Figure 4* shows the estimated miss-rate of the benchmark suite using a subset of 5, 9, and 15 programs that were clustered based on the locality characteristics. We observe that as we increase the number of representative programs (clusters), the estimated miss-rate using the subset moves closer to the true average miss-rate using the entire suite. The number of clusters can therefore be chosen depending on the desired level of accuracy. This can be achieved by simply specifying the number of representative programs, K, in the K-means algorithm.

4. Similarity across four generations of SPEC CPU benchmark suites

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 tweaks were retired, program run times were increased, and memory activity of programs was increased in every generation of the benchmark suite.

In this section, we use our collection of microarchitecture-independent metrics, described in section 2, to characterize the generic behavior of the benchmark programs as the evolved over the last decade. The same compiler is used to compile the four suites. The data is analyzed using PCA and cluster analysis to understand the changes in workload.

4.1 Instruction Locality

We perform PCA on the raw data measured for the instruction locality metric, which yields two principal components explaining 68.4 % and 28.6 % of variance. *Figure 5* shows the benchmarks in PC space.

PC1 represents the instruction temporal locality of benchmarks. Benchmarks with higher value of PC1 show poor temporal locality for instruction stream. Benchmarks with higher value of PC2 will benefit more from increase in block size. *Figure 6* shows that all SPEC CPU

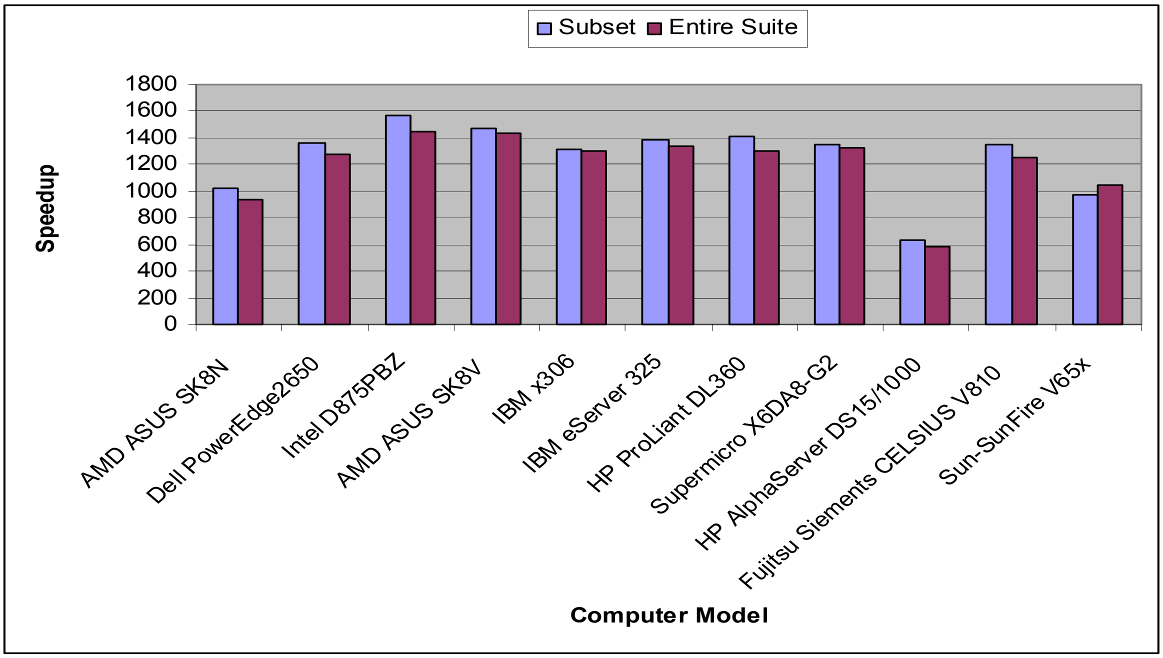

Figure 3: Average speedup of entire suite estimated using subset versus true speedup of entire suite

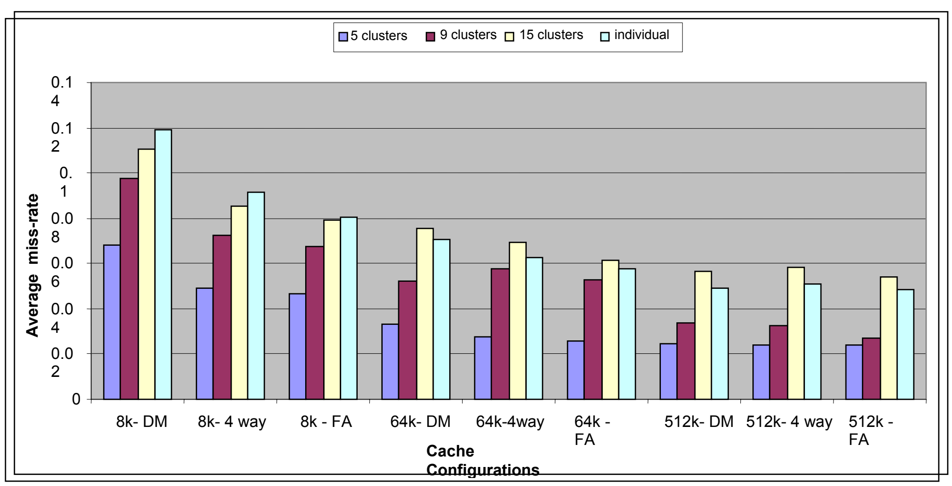

Figure 4: Sensitivity of estimated average L1 data cache miss-rate of benchmark suite to number of clusters

generations overlap. The biggest exception is *gcc* in SPECint2000 and SPECint95 (the two dark points on the plot on extreme right). *gcc* in SPECint2000 and SPECint95 suite exhibits poor instruction temporal locality – as shown by the *instrn_tlocality* (*Appendix A*) metric. *gcc* also shows very low values for PC2 due to poor spatial locality. Except *gcc*, almost all programs in the 4 different generations of SPEC CPU benchmark suite show similar instruction locality.

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.

4.2 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 6* plots the various SPEC CPU benchmarks in this PCA space.

We observe that the integer benchmarks are clustered in an area. 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 are very well structured, and 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 of SPEC CPU programs did not significantly change over the past 1.5 decades. Indeed, all SPEC CPU suites overlap in this graph.

4.3 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 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 7*.

We observe that the integer benchmarks typically have a high value along *PC1*, which indicates that these benchmarks have more short dependencies. The floating 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

4.4 Data 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 ratios between them is a measure for spatial locality. We perform PCA analyses of raw data. *Figure 8* shows a plot of the benchmarks in this PCA space. 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. 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.

In *Figure 8* 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. 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

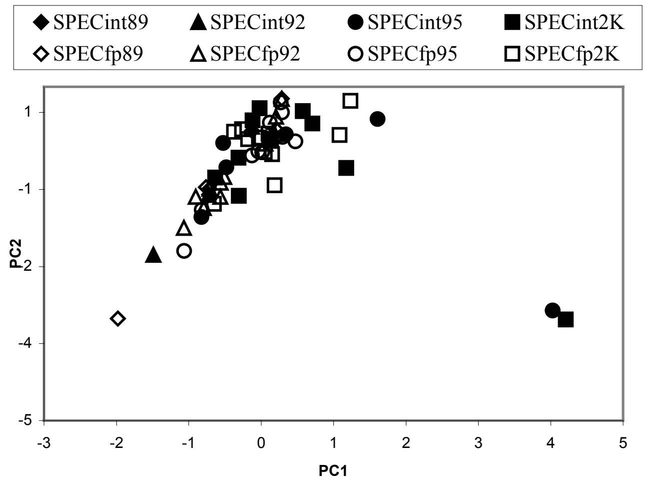

Figure 5: PCA space built from instruction locality characteristics

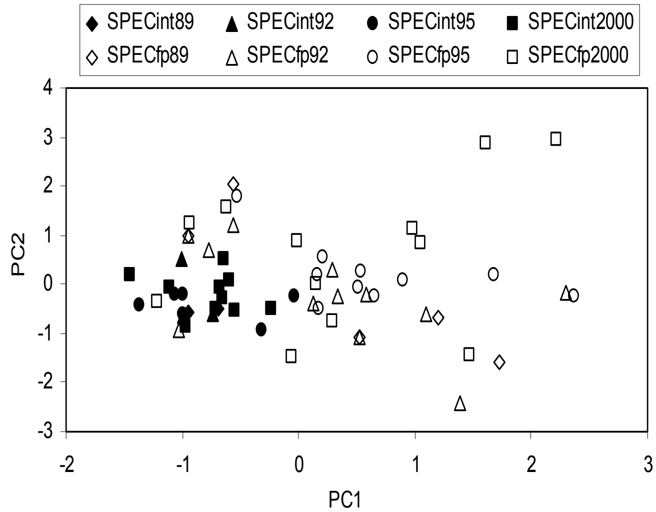

Figure 6: PCA characteristics built from branch characteristics

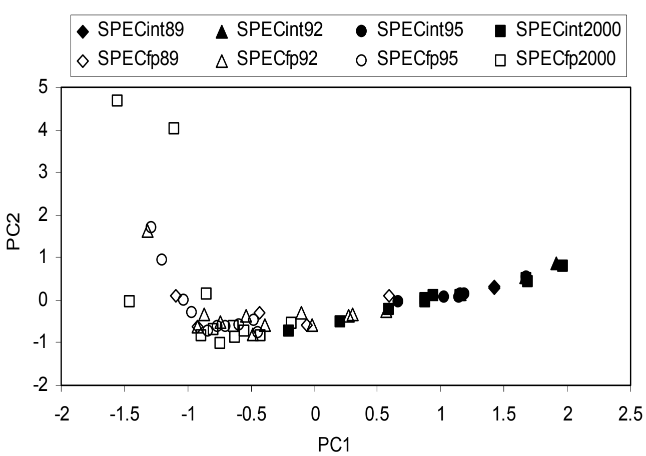

Figure 7: PCA space built from ILP characteristics

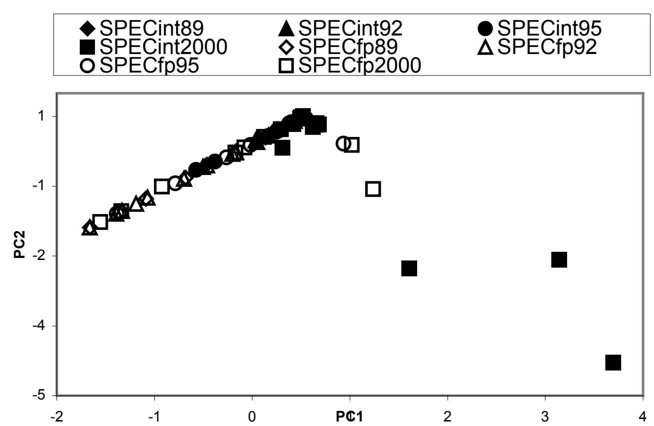

Figure 8: PCA space built from data locality characteristics

benchmarks show a temporal locality behavior that is better than most CPU89 and CPU92 benchmarks.

4.4 Overall Characteristics

In order to understand (dis) similarity across SPEC CPU benchmark suites we perform a cluster analysis in the PCA space as described in section 3. Clustering all 60 benchmarks yields 12 optimum clusters, which are shown in *Table 4*.

Cluster 1	gcc(95), gcc(2000)
Cluster 2	***mcf(2000)***
Cluster 3	***turbo3d (95)***, applu (95), apsi(95), swim(2000), mgrid(95), wupwise(2000)
Cluster 4	***hydro2d(95***), hydro2d(92), wave5(92), su2cor(92), succor(95), apsi(95), tomcatv(89), tomcatv(92), crafty(2000), art(2000), equake(2000), mdljdp2(92)
Cluster 5	***perl(95)***, li (89), li(95), compress(92), tomcatv(95), matrix300(89)
Cluster 6	***nasa7(92)***, nasa(89), swim(95), swim(92), galgel(2000), wave5(95), alvinn(92)
Cluster 7	applu(2000), mgrid(2000)
Cluster 8	***doduc(92***), doduc(89), ora(92)
Cluster 9	mdljsp2(92), lucas(2000)
Cluster 10	***parser(2000)***, twolf(2000), espresso(89), espresso(92), compress(95), go(95), ijpeg(95), vortex(2000)
Cluster 11	***fppp(95)***, fpppp(92), eon(2000), vpr(2000), fpppp(89), fma3d(2000), mesa(2000), ammp(2000)
Cluster 12	***bzip2(2000)***, gzip(2000)

Table 4: Optimum number of clusters for four generations of SPEC CPU benchmark programs using overall program

The benchmarks in bold are the benchmarks closest to the centroid of the cluster and can thus be considered the representatives for that cluster. For, clusters with 2 benchmarks either one can be picked as a representative since both are equidistant from the center of the cluster. A detailed analysis of *Table 4* gives us several interesting insights. First, out of all the benchmarks *gcc (2000)* and *gcc (95)* are together in a separate cluster. We observe that instruction locality for *gcc* is worse than any other program in all 4 generations of SPEC CPU suite; due to which *gcc* programs from SPEC CPU 95 and 2000 suites reside in their own separate cluster. Due to its peculiar data locality characteristics, *mcf (2000)* resides in a separate cluster (*cluster 2*), and *bzip2(2000), gzip(2000)* form one cluster (*cluster 12)*. SPEC CPU2000 programs exist in 10 out of 12 clusters, as opposed to SPEC CPU95 in 7 clusters, SPEC CPU92 in 6 clusters, and SPEC CPU89 in 5 clusters. This shows that SPEC CPU 2000 benchmarks are more diverse than their ancestors.

5. Related Work

The majority of ongoing work in studying benchmark characteristics involves measuring microarchitecture-dependent metrics e.g. cycles per instruction, cache miss rate, branch prediction accuracy etc., on various microarchitecture configurations that offer a different mixture of bottlenecks [12][15][16][17][27]. The variation in these metrics is then used to infer the generic program behavior. These inferred program characteristics may be biased by the idiosyncrasies of a particular configuration, and therefore may not be generally applicable. In this paper we measure program similarity based on the cause (microarchitecture-independent characteristics) rather than the effect (microarchitecture-dependent characteristics).

Past attempts to understand benchmark redundancy used microarchitecture-dependent metrics such as execution time or SPEC peak performance rating. 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 behaviour, 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 not unique. The relatively lower rank of *gcc* in [7] and its better position in this work (Tables 2 and 3) 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.

Several techniques have been proposed to reduce simulation time of programs [35][36][37]. But our techniques are relevant not only for identifying a subset from an existing suite, but also to select programs to include in a benchmark suite when there are several candidates.

6. Conclusion

In this paper we presented a methodology to measure similarity of programs based on their inherent microarchitecture-independent characteristics. We apply this technique to identify a small subset of nine programs in the SPEC CPU 2000 benchmark suite that are representative of the data locality exhibited by the suites, and a subset of eight programs that are representative of the overall characteristics (instruction locality, data locality, branch predictability, and ILP) of the programs in the entire suite. We validated this technique by demonstrating that the average data cache miss-rate and IPC of the entire suite could be estimated with a reasonable accuracy by just simulating the subset of programs. These results are applicable generally to any microarchitecture.

We also applied the microarchitecture-independent program characterization methodology to understand how the characteristics of the SPEC CPU programs have evolved since the inception of SPEC. We characterized 29 different microarchitecture-independent features of 60 SPEC CPU programs from SPEC89 to SPEC2000 suites. We find that no single characteristic has changed as dramatically as the dynamic instruction count. Our analysis shows that the branch and ILP characteristics have not changed much over the last decade, but the temporal data locality of programs has become increasingly poor. Our results indicate that although the diversity of newer generations of SPEC CPU benchmarks has increased, there still exists a lot of similarity between programs in the SPEC CPU2000 benchmark suite.

The methodology presented in this paper could be used to select representative programs for the characteristics of interest, should the cost of simulating the entire suite be prohibitively high. This technique could also be used during the benchmark design process to select only a fixed number of benchmark programs from a group of candidates.

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.

[33] T. Wenisch, R. Wunderlich, B. Falsafi, and J. Hoe, "Applying SMARTS to SPEC CPU2000", CALCM Technical Report 2003-1, Carnegie Mellon University, June 2003.

[34] J. Cantin, and M. Hill, “Cache Performance for SPEC CPU 2000 Benchmarks” http://www.cs.wisc.edu/multifacet/misc/spec2000 cache-data/

[35] T. Sherwood, E. Perelman, and B. Calder, “Basic block distribution analysis to find periodic behavior and simulation points in applications”, *Proc. of the International Conference on Parallel Architectures and Complication Techniques,* pp. 3-14, 2000.

[36] R. Wunderlich, R. Wenisch, B. Falfasi, and J. Hoe, “SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling”, *Proc. of International Symposium on Computer Architecture,* pp. 84-95, 2003.

[37] AJ KleinOswoski, D. Lilja, “MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research”, *Computer Architecture Letters,* pp. 10-13, 2002.

[38] The SPEC CPU2000 results published by SPEC http://www.spec.org/cpu2000/results/

Appendix A

Benchmark	%Memory	%Branches	Comp/Mem	BB Size	%Fwd	%taken	%Fwd-Taken	%Back-Taken	d-tlocality16	d-tlocality64	d-tlocality256	d-tlocality4096	d-tloc64/d-tloc16	d-tloc256/d-tloc16	d-tloc4096/d-tloc16
espresso_89	26.66	15.92	2.15	5.28	0.63	0.64	0.47	0.53	313.00	103.00	31.00	6.00	0.329073482	0.099041534	0.019169329
li_89	41.13	16.74	1.02	4.98	0.66	0.65	0.63	0.37	138.00	63.00	36.00	7.00	0.456521739	0.260869565	0.050724638
doduc_89	34.51	7.74	1.67	11.91	0.80	0.49	0.64	0.36	499.00	628.00	201.00	28.00	1.258517034	0.402805611	0.056112224
nasa7_89	46.24	2.47	1.11	39.56	0.26	0.84	0.14	0.86	338.00	593.00	182.00	25.00	1.75443787	0.538461538	0.073964497
matrix300_89	35.15	3.13	1.76	30.94	0.05	0.95	0.01	0.99	21312.00	1771.00	236.00	24.00	0.083098724	0.011073574	0.001126126
fpppp_89	43.36	1.29	1.28	76.73	0.82	0.51	0.72	0.28	2418.00	850.00	230.00	30.00	0.35153019	0.095119934	0.012406948
tomcatv_89	39.31	2.78	1.47	34.97	0.53	0.99	0.53	0.47	575.00	603.00	171.00	21.00	1.048695652	0.297391304	0.036521739
doduc_92	34.51	7.74	1.67	11.91	0.80	0.49	0.64	0.36	505.00	631.00	201.00	28.00	1.24950495	0.398019802	0.055445545
mdljdp2_92	24.72	12.65	2.53	6.91	0.86	0.84	0.83	0.17	1230.00	656.00	208.00	33.00	0.533333333	0.169105691	0.026829268
wave5_92	35.75	4.63	1.67	20.62	0.49	0.73	0.34	0.66	1020.00	576.00	184.00	27.00	0.564705882	0.180392157	0.026470588
tomcatv_92	39.31	2.78	1.47	34.97	0.53	0.99	0.53	0.47	575.00	605.00	172.00	22.00	1.052173913	0.299130435	0.03826087
ora_92	29.64	6.88	2.14	13.54	0.78	0.57	0.63	0.37	393.00	622.00	206.00	34.00	1.582697201	0.524173028	0.086513995
alvinn_92	36.48	10.32	1.46	8.69	0.04	0.98	0.02	0.98	54.00	33.00	15.00	2.00	0.611111111	0.277777778	0.037037037
mdljsp2_92	23.05	3.52	3.18	27.39	0.53	0.66	0.30	0.70	502.00	649.00	210.00	32.00	1.292828685	0.418326693	0.06374502
swm256_92	37.43	0.63	1.65	157.91	0.05	0.95	0.02	0.98	458.00	637.00	207.00	32.00	1.390829694	0.451965066	0.069868996
su2cor_92	38.84	2.81	1.50	34.64	0.46	0.78	0.32	0.68	2397.00	971.00	300.00	36.00	0.405089695	0.125156446	0.015018773
hydro2d_92	36.84	6.00	1.55	15.66	0.54	0.75	0.41	0.59	1294.00	672.00	217.00	35.00	0.519319938	0.167697063	0.027047913
nasa7_92	46.15	2.57	1.11	37.86	0.28	0.83	0.16	0.84	406.00	616.00	191.00	27.00	1.517241379	0.47044335	0.066502463
fpppp_92	44.96	2.05	1.18	47.82	0.79	0.61	0.75	0.25	3167.00	1161.00	273.00	30.00	0.36659299	0.086201452	0.009472687
espresso_92	27.85	17.10	1.98	4.85	0.63	0.64	0.47	0.53	309.00	106.00	37.00	6.00	0.343042071	0.1197411	0.019417476
li_92	42.53	17.65	0.94	4.67	0.67	0.65	0.63	0.37	139.00	61.00	34.00	8.00	0.438848921	0.244604317	0.057553957
compress_92	33.97	12.05	1.59	7.30	0.77	0.52	0.58	0.42	10178.00	1693.00	100.00	4.00	0.166339163	0.009825113	0.000393005
tomcatv_95	37.56	1.82	1.61	53.98	0.39	0.75	0.20	0.80	477.00	221.00	221.00	26.00	0.463312369	0.463312369	0.054507338
swim_95	37.40	0.62	1.66	160.73	0.03	0.97	0.01	0.99	461.00	643.00	210.00	33.00	1.394793926	0.455531453	0.071583514
su2cor_95	37.70	3.62	1.56	26.62	0.57	0.70	0.39	0.61	4175.00	910.00	291.00	33.00	0.217964072	0.069700599	0.007904192
hydro2d_95	36.55	5.82	1.58	16.20	0.54	0.78	0.41	0.59	1607.00	698.00	218.00	31.00	0.43434972	0.135656503	0.019290604
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	0.007660471	0.002202385	0.000340465
turb3d_95	37.88	3.30	1.55	29.28	0.49	0.60	0.35	0.65	1113236.00	124651.00	1078.00	38.00	0.111971765	0.000968348	3.41347E-05
apsi_95	35.71	3.31	1.71	29.23	0.43	0.72	0.31	0.69	1155.00	705.00	222.00	34.00	0.61038961	0.192207792	0.029437229
fpppp_95	43.86	1.40	1.25	70.37	0.80	0.54	0.72	0.28	3166.00	804.00	204.00	32.00	0.2539482	0.064434618	0.010107391
wave5_95	39.67	3.35	1.44	28.84	0.42	0.76	0.25	0.75	465.00	659.00	221.00	33.00	1.417204301	0.475268817	0.070967742
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	0.008527237	0.00263323	0.000344535
go_95	36.95	13.04	1.35	6.67	0.76	0.66	0.70	0.30	2856.00	548.00	69.00	9.00	0.191876751	0.024159664	0.003151261
li_95	41.36	18.05	0.98	4.54	0.65	0.64	0.62	0.38	1369.00	278.00	103.00	10.00	0.203067933	0.0752374	0.007304602
perl_95	40.80	16.72	1.04	4.98	0.85	0.67	0.79	0.21	153.00	81.00	42.00	5.00	0.529411765	0.274509804	0.032679739
gcc_95	37.92	14.91	1.24	5.70	0.75	0.62	0.66	0.34	7157.00	3412.00	730.00	5.00	0.476736063	0.101998044	0.000698617
compress_95	32.59	11.52	1.71	7.68	0.59	0.79	0.54	0.46	109.00	49.00	27.00	7.00	0.449541284	0.247706422	0.064220183
ijpeg_95	28.35	5.45	2.33	17.33	0.59	0.75	0.50	0.50	1700.00	195.00	34.00	9.00	0.114705882	0.02	0.005294118
bzip2_2k	39.50	12.29	1.22	8.14	0.63	0.70	0.56	0.44	337042.00	100375.00	69024.00	1875.00	0.297811549	0.204793468	0.005563105
crafty_2k	36.60	11.20	1.43	8.93	0.83	0.67	0.80	0.20	31962.00	7635.00	294.00	21.00	0.238877417	0.009198423	0.00065703
eon_2k	48.15	11.18	0.84	8.94	0.67	0.63	0.59	0.41	3622.00	707.00	229.00	28.00	0.195196024	0.063224738	0.007730536
gcc2k	53.26	10.68	0.68	9.36	0.58	0.71	0.43	0.57	26246.00	7112.00	2705.00	307.00	0.270974625	0.103063324	0.01169702
gzip_2k	32.17	10.44	1.78	9.58	0.72	0.70	0.62	0.38	3484076.00	296272.00	120821.00	2579.00	0.085036033	0.034678061	0.000740225
mcf_2k	37.27	21.10	1.12	4.74	0.63	0.64	0.53	0.47	6384474.00	801795.00	309.00	8.00	0.12558513	4.83987E-05	1.25304E-06
parser_2k	34.84	15.48	1.43	6.46	0.65	0.65	0.50	0.50	24700.00	1816.00	175.00	9.00	0.073522267	0.00708502	0.000364372
twolf_2k	32.28	12.08	1.72	8.28	0.62	0.57	0.48	0.52	21792.00	1240.00	102.00	6.00	0.056901615	0.004680617	0.00027533
vortex_2k	40.53	17.29	1.04	5.78	0.83	0.52	0.69	0.31	315137.00	27783.00	1419.00	60.00	0.088161657	0.004502804	0.000190393
vpr_2k	44.08	10.65	1.03	9.39	0.68	0.52	0.44	0.56	524568.00	15223.00	1829.00	4.00	0.02902007	0.003486679	7.62532E-06
applu_2k	38.17	0.31	1.61	317.61	0.26	0.69	0.04	0.96	557233.00	3638.00	218.00	34.00	0.006528687	0.000391219	6.10158E-05
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	0.065582827	0.000124542	1.54136E-05
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	0.595238095	0.261904762	0.095238095
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	0.539591837	0.164897959	0.015510204
galgel_2k	43.66	5.24	1.17	19.07	0.07	0.87	0.00	1.00	462.00	641.00	207.00	33.00	1.387445887	0.448051948	0.071428571
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	1.562827225	0.5	0.078534031
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	0.330590875	0.106207928	0.012715034
mgrid_2k	36.72	0.37	1.71	273.37	0.41	0.65	0.19	0.81	689344.00	1349.00	247.00	34.00	0.001956933	0.000358312	4.93223E-05
swim_2k	32.92	1.30	2.00	76.66	0.41	0.59	0.01	0.99	1163.00	622.00	201.00	30.00	0.534823732	0.172828891	0.025795357
wupwise_2k	30.78	9.76	1.93	10.24	0.67	0.37	0.56	0.44	768641.00	192694.00	48236.00	36.00	0.250694407	0.062754914	4.68359E-05
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	0.002474757	0.001286874	0.000692932

Benchmark	Dep dist 1	Dep dist upto 2	Dep dist upto 4	Dep dist upto 8	Dep dist upto 16	Dep dist Upto 32	Dep dist > 32	i-tlocality16	i-tlocality64	i-tlocality256	i-tlocality4096	i-tloc64/d-tloc16	i-tloc256/d-tloc16	i-tloc4096/i-tloc16
espresso_89	28.24577373	40.94172679	54.92052202	65.36403516	76.78822241	83.64447094	16.35543529	1734	528	189	43	0.3045	0.1089	0.0250
li_89	27.70554918	39.00579288	48.87863995	62.15352425	77.3318135	88.73144154	11.26855363	1120	390	171	38	0.3486	0.1524	0.0340
doduc_89	7.375817486	13.95497374	24.38021798	36.8675402	50.31224938	64.44339106	35.5566	3408	1033	361	59	0.3031	0.1058	0.0173
nasa7_89	3.383647112	6.463081912	14.79243914	31.49066013	44.50820973	60.90195933	39.09808698	799	270	113	33	0.3376	0.1416	0.0415
matrix300_89	9.405020439	16.95375902	32.05268642	60.2594348	73.29750219	77.13009639	22.86995108	285	114	61	23	0.4003	0.2141	0.0812
fpppp_89	1.106479632	2.392775878	5.086533806	16.61190969	32.24427263	45.79965407	54.20043377	2999	849	275	44	0.2830	0.0917	0.0147
tomcatv_89	2.706408212	3.670326085	6.465692347	15.31506744	33.71480372	49.89402637	50.10597363	1012	356	153	31	0.3513	0.1507	0.0306
doduc_92	7.369569635	14.15684223	24.97240403	37.39501373	50.98217567	67.22516845	35.55650188	3439	1052	371	62	0.3059	0.1078	0.0180
mdljdp2_92	18.94120892	22.90179847	35.34422322	42.82562051	55.07027378	63.30777644	36.69225781	1385	481	195	41	0.3472	0.1410	0.0297
wave5_92	5.073910055	10.11623437	18.91032696	32.31444761	44.29766509	57.14925836	42.85074164	3032	935	343	61	0.3083	0.1132	0.0201
tomcatv_92	2.706408227	3.670325532	6.465691352	15.31506742	33.71480603	49.89402789	50.10600477	1012	356	153	31	0.3513	0.1507	0.0306
ora_92	7.611049877	20.17374465	35.76855713	45.72093359	55.9790114	69.15288504	30.84706633	749	279	122	34	0.3722	0.1630	0.0456
alvinn_92	12.10487609	23.2799133	34.54941864	55.93953617	70.06306546	70.94966673	29.05032609	588	208	88	23	0.3538	0.1495	0.0396
mdljsp2_92	7.70318037	14.52271482	27.17382032	38.03077007	48.22985935	61.88326164	38.11663836	1436	487	204	41	0.3390	0.1417	0.0286
swm256_92	1.215104135	2.331453237	5.155123269	12.07271202	27.98370066	42.68698114	57.31305055	1160	415	186	41	0.3576	0.1600	0.0354
su2cor_92	2.707182576	5.2692738	12.26371027	23.77122429	39.03594555	51.08441523	48.91557738	2977	926	342	63	0.3111	0.1149	0.0210
hydro2d_92	3.625165567	8.041499866	13.82861965	26.6983678	42.76260452	58.53971562	41.46024029	3000	879	300	54	0.2932	0.1000	0.0180
nasa7_92	3.665554989	5.774864472	12.76471127	29.98851954	42.64204269	57.51028818	42.48965873	1650	582	238	54	0.3528	0.1444	0.0329
fpppp_92	2.353216428	4.395173019	8.763440367	21.30104599	36.00287021	48.88848792	51.11147059	2998	850	275	44	0.2835	0.0918	0.0147
espresso_92	45.46639969	59.10839734	65.88239603	70.39990711	77.95290561	82.85030436	17.14969564	1646	501	174	40	0.3042	0.1060	0.0244
li_92	36.83269998	44.4652	53.37849999	65.42240004	79.14800002	89.57470001	10.42529999	1097	384	169	36	0.3497	0.1544	0.0329
compress_92	21.53149994	36.54360012	51.0211996	61.76150138	71.85390101	80.82090069	19.17919931	230	89	42	13	0.3874	0.1818	0.0556
tomcatv_95	1.677198842	3.181968312	5.345249983	17.06161871	34.31065691	49.43717595	50.56281349	1177	481	192	48	0.4089	0.1631	0.0412
swim_95	1.249567916	2.517823836	5.633384004	13.81630156	28.14650772	43.4559487	56.5440513	1129	416	177	38	0.3689	0.1569	0.0337
su2cor_95	4.260704086	7.808068596	14.87282368	26.57982199	41.32320668	52.9695105	47.0304895	2742	874	328	60	0.3189	0.1197	0.0217
hydro2d_95	3.991078996	9.204048648	14.93412558	27.295558	43.09640308	59.09473274	40.90532653	2698	808	273	54	0.2996	0.1013	0.0198
applu_95	1.938616155	5.977797339	9.522697376	21.5284511	36.44813854	47.82060164	52.17944372	3401	993	326	54	0.2919	0.0960	0.0159
turb3d_95	3.139914098	7.663209211	13.09510382	19.57900012	35.58454211	50.36408283	49.63575365	3073	999	357	61	0.3251	0.1161	0.0199
apsi_95	3.242025616	6.967425931	11.69774338	21.32114886	37.19709838	53.8835877	46.11650956	5187	1578	544	84	0.3043	0.1048	0.0162
fpppp_95	1.295660679	2.792299382	5.70101484	17.66442606	33.49827417	47.00494774	52.99505226	3024	863	284	46	0.2852	0.0939	0.0152
wave5_95	4.537476439	8.525762061	18.58577889	30.59781513	42.03833648	55.50583645	44.49425396	4085	1253	431	71	0.3067	0.1056	0.0173
mgrid_95	0.460169624	2.158106568	5.033334146	15.99715995	33.22709534	43.60114704	56.3990168	2466	803	301	55	0.3257	0.1222	0.0222
go_95	21.34404836	33.31246921	46.90191379	57.76185101	69.61896859	79.88843464	20.11156536	14014	3731	1029	92	0.2662	0.0734	0.0066
li_95	37.6025004	45.48547065	54.28051869	66.52520181	78.39445223	88.75450729	11.24539357	1318	443	176	37	0.3362	0.1334	0.0281
perl_95	24.0069258	35.23719605	48.12493458	59.63755984	72.34400846	83.12929771	16.87070229	1238	455	196	46	0.3671	0.1585	0.0374
gcc_95	24.63591849	35.38411936	46.97917412	58.24071352	72.02735631	82.25718934	17.74261084	33010	10179	3328	314	0.3084	0.1008	0.0095
compress_95	18.01075483	29.97885601	45.74111093	62.27942492	76.04136832	86.05779167	13.94220833	568	177	75	16	0.3109	0.1323	0.0282
ijpeg_95	14.40154365	24.36649296	37.88002103	50.59509179	62.17651747	79.48397176	20.51602169	4702	1499	477	55	0.3189	0.1014	0.0117
bzip2_2k	31.42387946	35.45925911	57.57294627	73.12391239	86.49331199	90.60139239	9.39863619	1691	502	172	29	0.2969	0.1018	0.0171
crafty_2k	13.79524488	24.51466878	38.61956916	52.65633508	64.36610601	72.74521463	27.25478537	6578	1925	619	73	0.2926	0.0942	0.0110
eon_2k	6.752331646	11.89105831	21.40168993	31.90866985	48.05191045	62.03596511	37.96404513	2372	782	334	88	0.3298	0.1406	0.0373
gcc2k	22.8095904	29.62519249	44.86550944	51.53146429	68.92398594	75.85878245	24.1411316	34199	10534	3416	337	0.3080	0.0999	0.0099
gzip_2k	22.12449237	33.66510763	43.96245593	61.23192373	69.04922034	74.19398644	25.80593305	1576	481	171	30	0.3050	0.1087	0.0193
mcf_2k	19.47193895	34.29095164	46.45298806	58.31641232	68.91389536	72.19232364	27.80767638	1055	370	155	26	0.3509	0.1470	0.0244
parser_2k	20.47269985	32.33808499	49.96636229	61.17772112	73.99880756	83.4104212	16.5894788	5507	1623	486	55	0.2947	0.0883	0.0100
twolf_2k	21.93760764	38.78265983	62.77025319	80.11326516	87.12390455	90.08897142	9.911028577	3387	1054	381	59	0.3114	0.1124	0.0174
vortex_2k	41.76606805	49.78237994	60.82341848	73.39741011	83.80272073	91.68711536	8.312851312	10170	3094	1025	134	0.3042	0.1008	0.0132
vpr_2k	11.50525103	13.20424629	15.31769834	44.35755264	65.44147953	71.23628346	28.76371654	1758	588	220	41	0.3346	0.1249	0.0233
applu_2k	1.223290027	2.494915219	5.226286336	13.1195204	28.24459492	40.79978446	59.20025531	10096	2663	744	79	0.2638	0.0737	0.0078
apsi_2k	1.960622173	6.036014625	10.94763116	22.26281403	36.95494988	49.38400803	50.6160258	9097	2619	804	108	0.2880	0.0883	0.0119
equake_2k	6.208077631	9.237213429	14.08927488	26.57030031	40.09568695	49.48966089	50.51033911	1189	366	138	27	0.3078	0.1164	0.0228
fma3d_2k	1.693835268	3.207988252	7.630875627	20.22057241	34.74243131	48.41702869	51.5828813	4628	1542	614	96	0.3331	0.1328	0.0207
galgel_2k	3.441307704	9.461174499	14.44773136	19.18091711	44.13934288	56.25292779	43.74697879	3691	1170	428	76	0.3171	0.1158	0.0206
lucas_2k	4.068021738	5.988135561	12.18175264	21.902093	36.32800032	47.99066416	52.00933584	1797	556	216	42	0.3094	0.1202	0.0233
mesa_2k	7.96936592	15.32562283	20.86178676	28.22666687	37.81823553	52.71027307	47.28973736	1512	531	233	53	0.3515	0.1543	0.0349
mgrid_2k	1.774994663	3.645573126	9.506618728	28.76230969	40.90246919	48.60636797	51.39363203	3238	1010	365	57	0.3120	0.1129	0.0175
swim_2k	0.853347871	1.482842604	3.671285424	5.322639619	26.58898752	33.56610623	66.43384721	2456	772	275	49	0.3145	0.1120	0.0201
wupwise_2k	0.743313618	5.248973448	17.9474739	27.45660453	37.66482233	47.07516796	52.92483204	3233	1022	396	63	0.3160	0.1224	0.0195
art_2k	7.283331723	12.23540119	16.49315851	28.89786636	36.68032618	45.752205	54.247795	802	245	95	22	0.3060	0.1187	0.0269
ammp_2k	9.072037666	16.22804392	26.99815399	37.48912743	46.27302047	56.03337482	43.96666965	3063	964	324	53	0.3148	0.1059	0.0172