Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites

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

Abstract

It is essential that a subset of benchmark programs used to evaluate an architectural enhancement, 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.

The objective of this paper is to present a methodology to measure similarity of programs based on their inherent microarchitecture-independent characteristics which will make the results applicable to any microarchitecture. 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 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.

Studying the four generations of SPEC CPU benchmark suites, 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. 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

Microarchitecture-independent metrics allow for a comparison between programs by understanding the inherent characteristics of a program isolated from features of particular microarchitectural components. 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. The actual raw data for all the metrics below can be found in Appendix A in [33]. We have identified the following microarchitecture-independent metrics:

2.1.1 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.

2.1.2 Dynamic Basic Block Size: A basic block is a section of code with one entry and one exit point. We measure the 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).

2.1.3 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.

2.1.4 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.

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

2.1.6 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 seven categories: percentage of total dependencies that have a distance of 1, and the percentage of total dependencies that a distance of upto 2, 4, 8, 16, 32, and greater than 32. Programs that show greater percentage of long dependency distances show higher ILP (provided control flow is not a limiting factor).

2.1.7 Data Temporal Locality: Several locality metrics have been proposed in the past [4] [5] [11] [18] [21] [29], however, many of them are computation and memory intensive. We picked the average memory reuse distance metric from [29] 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. This 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. [29]. 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.

2.1.8 Data Spatial Locality: Spatial locality information for data accesses is characterized by the ratio of the temporal locality metric for *window* sizes mentioned above.

2.1.9 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, for every unique static instruction in the program that is executed at least twice. The instruction temporal locality is calculated for window sizes of 16, 64, 256, and 4096 bytes.

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

2.2 Statistical Data Analysis

Obviously, the amount of data to analyze in this experiment is huge. There are many variables (29 microarchitecture- independent characteristics) and many cases (program-input pairs). It is humanly impossible to simultaneously look at all the data and draw meaningful conclusions from them. Hence, we use multivariate statistical data analysis techniques, namely *Principal Component Analysis (PCA)* 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 similar intrinsic program characteristics.

2.2.1 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] >…> 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 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.

2.2.2 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 and a few from SPEC 2000). The instruction counts of these programs are therefore missing from Table 1 and have a * in the column. 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 may be dependent on the instruction set architecture (ISA) and the compiler.

Table 1: List of SPEC CPU benchmarks

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	*	INT	*
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	*

2.4 Tools

SCOPE: The workload characteristics were measured using a custom-developed analyzer called *SCOPE*. *SCOPE* was developed by modifying the *sim-safe* functional simulator from the *SimpleScalar* 3.0 [28] tool set. *SCOPE* analyzes 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 custom made analyzers to obtain various locality and parallelism metrics described in section 2.1.

Statistical data analysis: We use STATISTICA version 6.1 for performing PCA. For K-means clustering we use the *SimPoint* software [30]. 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 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.

Table 2: Clusters based on overall characteristics

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***

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 clusters 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. 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

Table 3: Clusters based on data locality characteristics

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***

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 popular programs *parser, twolf* and *vortex* are in the same cluster, Cluster 6 and hence do not add more information. Clusters in 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.

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 for *window* sizes of 16, 64, 256, and 4096 bytes, and the ratios of each of the data locality metric for *window* sizes of 64, 256, and 4096 bytes, to the data locality 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 PCA 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.

3.3.1 Computing IPC. Using the subset based on overall program characteristics we calculated the average IPC of the entire suite for two different microarchitecture configurations 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 IPC on 8-way and 16-way issue widths for every program in the SPEC CPU2000 benchmarks from Wenisch et. al. [31]. The configurations in brief are: 8-way machine (32KB 2 way L1 I/D cache, 1M 4-way L2, Functional Units 4 I-ALU, 2 I-MUL/DIV, 2 FP-ALU, 1 FP-MUL/DIV) and 16-way machine(64 KB 2-way L1 I/D, 2M 8-way L2, Functional Units 16 I-ALU, 8 I-MUL/DIV, 8 FP-ALU, 4 FP-MUL/DIV). The rest of the details about branch predictor and different penalties in cycles can be found in [31]. 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 for both configurations is less than 5%(shown in Figure 1). 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.

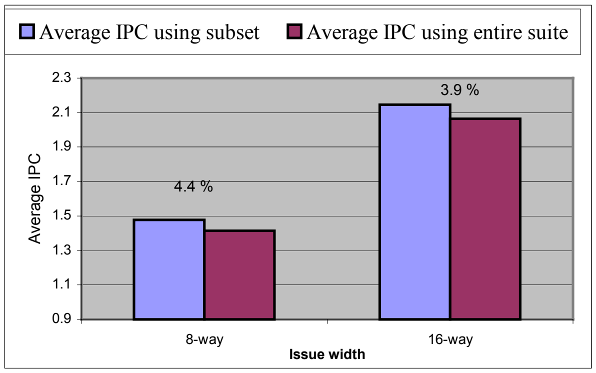

Figure 1: Estimated average IPC of benchmark suite using subset versus average IPC of entire suite

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.

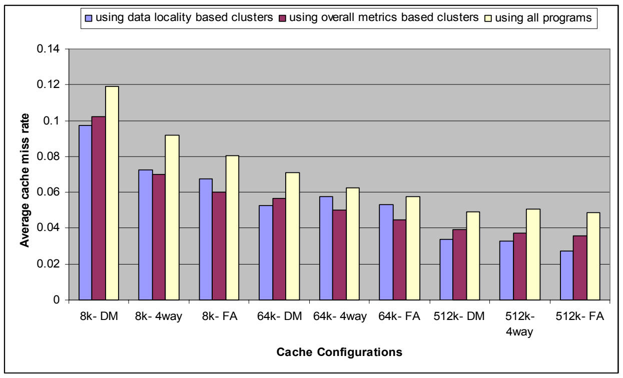

Figure 2: Average L1 D-cache miss-rate using the subset based on locality characteristics, and the subset based on overall characteristics

We obtained the miss-rates for 9 different L1 data cache configurations from Cantin et. al. [32]. 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. 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. 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.

3.3.3 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 3 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

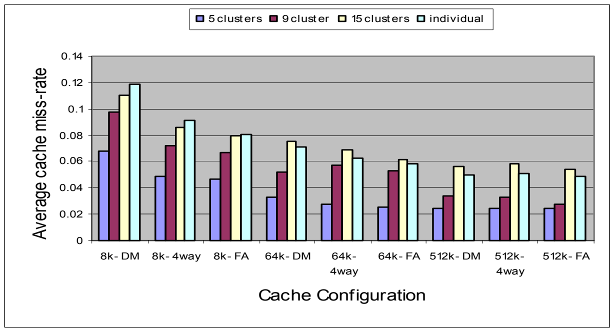

Figure 3: Sensitivity of average cache miss-rates to number of clusters

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 4 shows the benchmarks in PC space. PC1 represents instruction temporal locality of benchmarks.

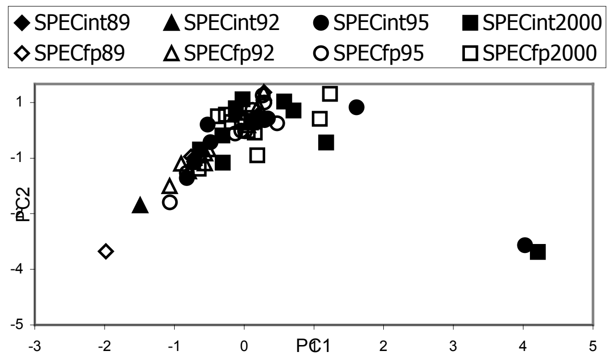

Figure 4: PCA space built from instruction locality characteristics

Benchmarks with higer 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 4 shows that all SPEC CPU 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 data in (*Appendix A [33]*). *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

We perform PCA analysis and retain 2 principal components explaining 62% and 19% of the total variance, respectively. Figure 5 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 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

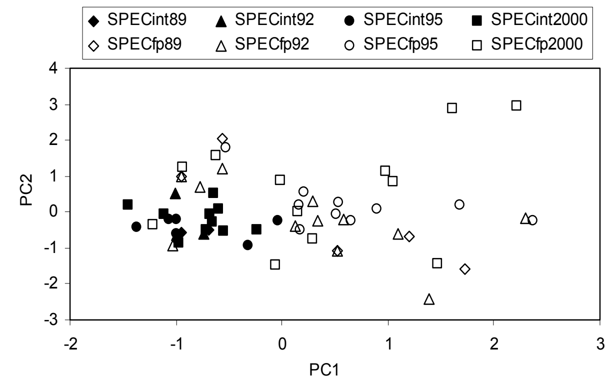

Figure 5: PCA space built from branch characteristics

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 Figure 5.

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. The first two principal components explain 96% of the total variance. The PCA space is plotted in Figure 6. 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 -0.1 <PC1 <0.6

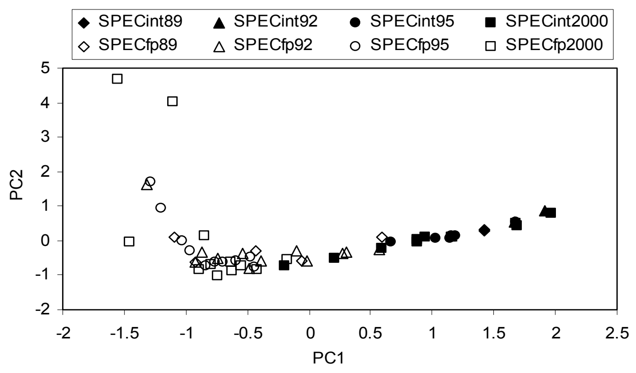

Figure 6: PCA space built from ILP characteristics

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. [29] 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 7 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 7 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, some CPU95 and CPU2000 benchmarks show a temporal locality behavior that is better than most CPU89 and CPU92 benchmarks.

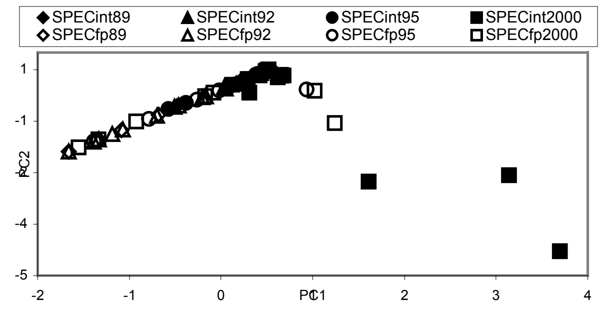

Figure 7: PCA space built from data locality characteristics

4.5 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. The benchmarks in bold are 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.

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

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)

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]. 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][29]. 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. 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].

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 L1 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.

7. Acknowledgement

This research is supported in part by NSF grants 0113105, 0429806, IBM and Intel corporations. Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research -- Flanders (Belgium) (F.W.O Vlaanderen).

8. 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 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*, 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 Intl Conf on High-Perf Comp Arch, ,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

[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 Perf Comp Arch*, 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] Standard Performance Evaluation Corporation, http://www.spec.org/benchmarks.html.

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

[29] 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.

[30] 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.

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

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

[33] A. Phansalkar, A. Joshi, L. Eeckhout, L. John "Measuring Program Similarity", Lab for Computer Architecture Technical Report 2005, TR-050127-01, University of Texas at Austin.