The Dissertation Committee for Aashish Shreedhar Phansalkar certifies that this is the approved version of the following dissertation:

Measuring Program Similarity for Efficient Benchmarking and Performance Analysis of Computer Systems

Committee:

Lizy K. John, Supervisor

Joydeep Ghosh

Robert Flake

Nur A. Touba

Cheranellore Vasudevan

Measuring Program Similarity for Efficient Benchmarking and Performance Analysis of Computer Systems

by

Aashish Shreedhar Phansalkar, B.E.; M.S.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2007

Dedication

To my wife Sangita, my family and friends

Acknowledgements

First I would like to thank my advisor, Dr. Lizy K. John, for her advice, support, and guidance. She always motivated me, whether it was about research or staying focused during the course of my graduate school research. She was available to answer my questions, give feedback or comments even during weekends or spring break. I really appreciate her leadership and initiative during the project we did with SPEC. I am also thankful to her for the flexibility she gave me throughout the course of my study. I am grateful to my committee members (in alphabetical order), Dr. Robert Flake, Dr. Joydeep Ghosh, Dr. Nur Touba all from UT Austin and Dr. Cheranellore Vasudevan from IBM, Austin for their comments, constructive suggestions.

I am grateful to Dr. Lieven Eeckhout from Ghent University with whom I had multiple opportunities to collaborate during the course of my research. His comments and feedback on many technical issues were invaluable. A significant part of the research work in this dissertation has been in collaboration with him and his research group. I would like to thank Ajay Joshi from Laboratory of Computer Architecture(LCA) with whom I had a chance to work during the early part of the work and co-author publications. He was always ready to read the paper multiple times before submission and had detailed constructive comments and suggestions which certainly helped to improve the work and the manuscript. Later in the course, we had long discussions related to each other's research work over coffee-breaks and in the lobby. I am also thankful to Dr. Yue Luo who was my primary source of guidance regarding tools and benchmarks when I started my research. I also had an opportunity to work with him on a research paper. I would also like to thank Dr. Rob Bell Jr, Dr. Madhavi Valluri, Dr. Ravi Bhargava, Dr. Juan Rubio, Dr. Byeong Kil Lee, Dr. Shiwen Hu and Dr. Tao Li who passed on a lot of their research experiences when I started my research work. I would also like to thank my friends Moin and Elias who are members of other research groups in the department of ECE, for their time and interest in the discussions I had with them about my research. I would like to thank the current members of LCA, Dimitris, Ciji, Lloyd, Jian, Karthik, Deepak, Arun, Nidhi, Jeff, Rajeev and Sarah for their comments on my practice talks before the defense. I also enjoyed working with Dimitris on a small project related to my research. Thanks to the staff members Amy and Melanie who were very prompt and extremely helpful.

I am very thankful to my parents, my brother and his family, my parent-in-laws and family for their support and encouragement. My friends have also been of great help many times and have certainly made my journey in graduate school enjoyable.

Finally, I am very thankful to my dear wife Sangita who has been very patient all throughout my PhD studies literally living across the country for most of the time. This was certainly not easy and I will be eternally thankful for her love and encouragement. I could not have accomplished this without her cheerfulness and unswerving support.

Measuring Program Similarity for Efficient Benchmarking and Performance Analysis of Computer Systems

Publication No._____________

Aashish Shreedhar Phansalkar, PhD
The University of Texas at Austin, 2007

Supervisor: Lizy K. John

Computer benchmarking involves running a set of benchmark programs to measure performance of a computer system. Modern benchmarks are developed from real applications. Applications are becoming complex and hence modern benchmarks run for a very long time. These benchmarks are also used for performance evaluation in the early design phase of microprocessors. Due to the size of benchmarks and increase in complexity of microprocessor design, the effort required for performance evaluation has increased significantly. This dissertation proposes methodologies to reduce the effort of benchmarking and performance evaluation of computer systems.

Identifying a set of programs that can be used in the process of benchmarking can be very challenging. A solution to this problem can start by identifying similarity between programs to capture the diversity in their behavior before they can be considered for benchmarking. The aim of this methodology is to identify redundancy in the set of benchmarks and find a subset of representative benchmarks with the least possible loss of information. This dissertation proposes the use of program characteristics which capture

the performance behavior of programs and identifies representative benchmarks applicable over a wide range of system configurations. The use of benchmark subsetting has not been restricted to academic research. Recently, the SPEC CPU subcommittee used the information derived from measuring similarity based on program behavior characteristics between different benchmark candidates as one of the criteria for selecting the SPEC CPU2006 benchmarks.

The information of similarity between programs can also be used to predict performance of an application when it is difficult to port the application on different platforms. This is a common problem when a customer wants to buy the best computer system for his application. Performance of a customer's application on a particular system can be predicted using the performance scores of the standard benchmarks on that system and the similarity information between the application and the benchmarks. Similarity between programs is quantified by the distance between them in the space of the measured characteristics, and is appropriately used to predict performance of a new application using the performance scores of its neighbors in the workload space.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Performance analysis of processors and computer systems poses many challenges due to increase in complexity of processor design and rapid evolution of application software. As Bose [7] pointed out, understanding the target workloads is important not only for performance but also for power and reliability in the early design phase of modern microprocessor designs. Understanding target workload involves detailed characterization and comparison of different workloads and available benchmarks.

Performance evaluation of design trade-offs is an integral part of the design process. Very early in the design process computer architects model the designs in a very high level language like C, C++. The models built using the high level language which can model the behavior of a microprocessor at the granularity of a cycle are called cycle accurate simulators or performance simulators. Due to increase in complexity of modern processors, the cycle accurate simulators have become very slow. Earlier, simulators were driven by traces of programs but recently execution driven simulators have become quite common and programs can be directly run on the simulator. The simulator approach is very popular because it is flexible and modifications to a simulator to analyze different architectural ideas can be done with more ease compared to other techniques.

In computer benchmarking, performance is measured using a well defined set of test-programs. A certain set of well-defined and established rules are followed to compile and run these test-programs. In the past programs were specifically written for the purpose of benchmarking e.g. Whetstone and Dhrystone benchmarks [61]. Modern computer benchmarking involves use of application programs as benchmarks. Standard Performance Evaluation Corporation (SPEC) is an organization which was formed in 1988 by a group of companies that came together to define and develop standard benchmark suite called SPEC89, from compute-intensive workloads. SPEC has served a long way in developing and distributing technically credible, portable, real-world application-based benchmarks. In order to keep pace with the technological advancements, compiler improvements, and emerging workloads, in each generation of SPEC benchmarks, new programs are added, programs susceptible to unfair compiler optimizations are retired, program run times are increased, and memory access intensity of programs is increased. Recently, SPEC CPU2006, the fifth generation of compute-intensive benchmarks was released by SPEC. Performance analysts build new benchmark suites from the emerging applications to compare different computer systems and evaluate design trade-offs. The process of developing a benchmark suite begins by collecting applications from many different application areas. The development of SPEC CPU benchmark suites starts off with a collection of applications from software developers who are ready to share their source code. These applications are then thoroughly evaluated for many different features. The most important characteristic of such benchmarks is that they should be easily portable to many different platforms. Apart from the constraints related to portability and compilation of these applications, it is necessary to evaluate the benchmarks based on their performance characteristics. Comparing candidate benchmarks based on their performance characteristics is important to make sure that the benchmarks have diverse characteristics.

1.1 MOTIVATION

This section describes the motivations for this research and also provides background for each one.

1.1.1 Partial use of benchmark suite for simulation

Researchers and designers use benchmarks to evaluate design trade-offs in simulation based studies. Due to increase in complexity of cycle accurate simulators, simulation takes significantly long. The benchmarks are also becoming longer which leads to further increase in simulation time. As a work around, many times researchers use only a small set of randomly chosen benchmarks to evaluate their research ideas. Issues in running certain benchmarks, compiling and porting the benchmark to the simulation environment also force researchers to choose only a few benchmarks. Citron [10] did a survey on benchmark subsets used by the computer architecture research community in the recent top conference publications and showed that partial use of suites can cause misleading results. A quantitative approach to select benchmarks is proposed in this dissertation. The central idea behind the approach is to find a subset of benchmarks that have diverse characteristics and are spread around in the workload space. Previous research about choosing simulation points [51][52][62] by finding phases in the program is being used by researchers but phases from only a few benchmarks are used in the study. The technique proposed in this dissertation is orthogonal to the one about choosing phases for simulation.

1.1.2 Selecting programs to form a benchmark suite

The development of a benchmark suite is a rigorous process. For an organization like SPEC, the development process starts by openly requesting application developers to submit their applications as potential candidates for benchmarks. There are several candidate benchmarks to begin with and the SPEC CPU subcommittee faces a big challenge of selecting only a right set of benchmarks. Some of the necessary conditions for selection of a benchmark are related to portability, but that is not the focus of this dissertation. This dissertation's focus is about selecting benchmarks with a diverse set of performance characteristics to form the benchmark suite. This process gives an idea about how the benchmarks test the computer system for many different possible performance bottlenecks. A benchmark suite with diverse characteristics builds confidence amongst customers and designers.

1.1.3 Comparing workload space coverage of benchmark suites

General purpose processors have many target applications. Many different benchmark suites need to be evaluated while analyzing performance. Comparison of the workload space coverage of different benchmark suites is very important to understand the behavior of new and existing benchmark suites. Usually, different domains have their own standard benchmark suites e.g. the MediaBench [38] and MiBench [21] benchmark suites are a collection of media applications. From the point-of-view of a designer it is worthwhile to see how the behaviors of programs from different application domains compare in the workload space.

The comparison of new and existing benchmark suites is also necessary to evaluate the similarity between benchmarks across different benchmark suites. Once a new benchmark suite is released, its comparison with the older generations of the same suite is a useful exercise.

1.1.4 Performance prediction for customer's application

Customers wish to know the performance of their applications on a certain platform before purchasing it, but it is very difficult to run the user's application on all the possible platforms because it is usually expensive to port an application on another platform. Just looking at benchmark scores to rank the machines may mislead the customer because the ranking is not specific to the customer's application. This is a classical problem in benchmarking. In such a scenario the program similarity analysis can help identify benchmarks that are similar to the user's application and help to generate rankings for computer systems specific to the customer's application rather than all the benchmarks. This dissertation develops a methodology to perform such performance prediction.

1.1.3 Summary of motivations

The notion of measuring program similarity plays an important role in solving the problems described above. Measuring program similarity essentially involves characterizing workloads and comparing them based on the measured characteristics. The characteristics measured depend on the objective. This dissertation's focus is on the idea of measuring similarity between programs to improve the efficiency of the process of performance evaluation and benchmarking.

1.2 OBJECTIVES

The similarity information forms the basis for developing techniques to solve the described problems. Each of these specific objectives is described in detail in this section with measuring program similarity being the primary goal in each of these cases.

1.2.1 Finding a subset for simulation based performance evaluation

One objective of this research is to develop a methodology to identify a representative subset of a benchmark suite for use when time constraints prevent simulation of all benchmarks. In this dissertation finding a set of representative benchmarks is referred to as *Benchmark Subsetting*. A subset formed after benchmark subsetting should be able to accurately project performance on behalf of all the benchmarks in the suite. First step in benchmark subsetting is measuring similarity between benchmarks. This information of similarity is then used to find a subset of programs using clustering [30]. Principal Components Analysis (PCA) [17] is used as a preprocessing step. There are two main types of benchmark characteristics i.e. microarchitecture independent metrics and microarchitecture dependent metrics. In the early design phase it is difficult to measure microarchitecture dependent metrics because it essentially involves measuring them using a performance simulator. Usually microarchitecture dependent metrics are easy to measure on real system where performance monitoring counters can be used. But these characteristics are measured only on one particular configuration and hence the subset obtained by using these characteristics can be biased to a particular configuration. One of the objectives of this dissertation is to find a subset of benchmarks that is applicable to a wide range of configurations. If the subsetting analysis is done using microarchitecture independent metrics which are inherent to a program, the subset will be applicable to a wider range of systems, but will still be dependent on the compiler and Instruction Set Architecture (ISA). Validation of the subset formed, is very important part of the experiment. The subset should accurately project the performance of the whole benchmark suite.

1.2.2 Measuring program similarity for benchmark suite formulation

The process of building a benchmark suite starts with a large set of applications submitted by software developers. These applications are then thoroughly evaluated for portability. During this process the benchmarks keep changing almost every week and hence possibly their behavior. It is very difficult to measure microarchitecture independent metrics in such a short span of time. Also, the length of benchmarks is increasing drastically for the new generations which increases the time required for the analysis. Since the benchmarks are portable and will be used on many different platforms, it will be good if the analysis can take into account the change in behavior of programs caused by the use of different compilers. One objective of this research is to come up with a fast subsetting methodology where the resulting subset can be identified quickly. An approach utilizing microprocessor performance monitoring counters is developed. In order to reduce microarchitecture dependence the benchmarks are characterized on multiple real systems with different microarchitecture configurations, ISAs and compilers. This approach was used in practice to guide selection of benchmarks during the process of SPEC CPU2006 suite formulation.

1.2.3 Comparison of different benchmark suites by analyzing workload space coverage

One of the objectives of this research is to use the methodology of measuring similarity to compare benchmark suites. Mapping the benchmarks in the workload space can give a good idea of relative positions of benchmarks in the workload space or the area covered by whole benchmark suite. If a benchmark suite overlaps another benchmark suite in the workload space then the effort of performance evaluation can be reduced by finding a representative set common to both the suites. Another part of this objective is that the similarity analysis should be independent of the microarchitecture configuration to make the results more applicable to wide range of systems or microarchitecture configurations. An experiment comparing the SPEC CPU2000 and media benchmarks is described.

1.2.4 Performance prediction using program similarity

Another objective of this research is to develop a methodology to predict performance of a customer's application using it similarity with benchmarks. The methodology uses a repository of well characterized benchmarks with their performance scores. All the benchmarks from the repository are mapped into the workload space built using their characteristics. Same characteristics are measured for the new target application. These characteristics are then used to map the new applications in the same workload space of the well characterized benchmarks. The distance between the application and the benchmarks is the measure of their similarity. This information and the already known performance scores of the benchmarks are used to predict performance of the application. To improve the accuracy of this method it is important to choose the correct set of metrics. The different methods of selecting metrics or assigning weights to the metrics are also discussed.

1.5 THESIS STATEMENT

Program similarity information can be used to reduce the redundancy in existing benchmark suites which in turn reduces the effort of performance evaluation, to choose programs to form a benchmark suite, and to study the coverage of workload space of different benchmark suites. The similarity between a new application and the existing well characterized benchmarks can be used to predict performance of the new application.

1.6 CONTRIBUTIONS

This dissertation makes contributions towards measuring program similarity to improve the efficiency of performance evaluation and benchmarking of computer systems. The contributions are useful for users, including but not limited to microprocessor designers, performance engineers, architects and benchmarking engineers. It proposes and validates the use of microarchitecture independent metrics and microarchitecture dependent metrics in measuring program similarity. It further uses the similarity analysis to form a representative subset of benchmarks and predict performance of new applications. The following paragraphs summarize these contributions individually:

In the process of selecting benchmarks to form a suite or during the early design phase, benchmark subsetting helps to reduce redundancy and effort required for performance evaluation. Previous research either uses only microarchitecture dependent metrics [16][60][20] or a mix of simple program characteristics and microarchitecture dependent metrics [18][19] to measure similarity between programs. A subset of benchmarks derived using these metrics can be biased to the configuration or systems that are used to measure the metrics. This dissertation contributes to the process of measuring program similarity by using a set of microarchitecture independent metrics which are inherent characteristics of program and avoid having a biased subset. Moreover, if the analysis based on microarchitecture independent metrics is not possible, the goal of finding an unbiased subset can be achieved by using microarchitecture dependent metrics from several systems with varying features. The benchmark characteristics are pre-processed using a statistical analysis technique called Principal Components Analysis (PCA) and clustering. Subsetting using benchmark similarity was used as one of the criteria for choosing the SPEC CPU2006 benchmarks when the benchmark suite was being developed. This work was done in collaboration with the SPEC CPU subcommittee.

The design space of a general purpose processor is quite diverse which means that the target applications are from different domains. Many benchmark suites are also developed for a specific application domain. It is worthwhile for a designer to compare their workload space and evaluate different suites together. When a new benchmark suite is developed it is very important to study its coverage in the workload space with other benchmark suites. If the benchmarks are spread around in the workload space the suite shows diverse behavior and hence can test different features of the processor. As another contribution, this dissertation proposes the use of microarchitecture independent characteristics to study the coverage and compare different benchmark suites. The results of comparing the previous four generations of SPEC CPU benchmark suites have been discussed. Media benchmarks and SPEC CPU2000 benchmarks are also compared to study their individual coverage. Although these benchmarks are from different domains, they both have scientific and engineering applications.

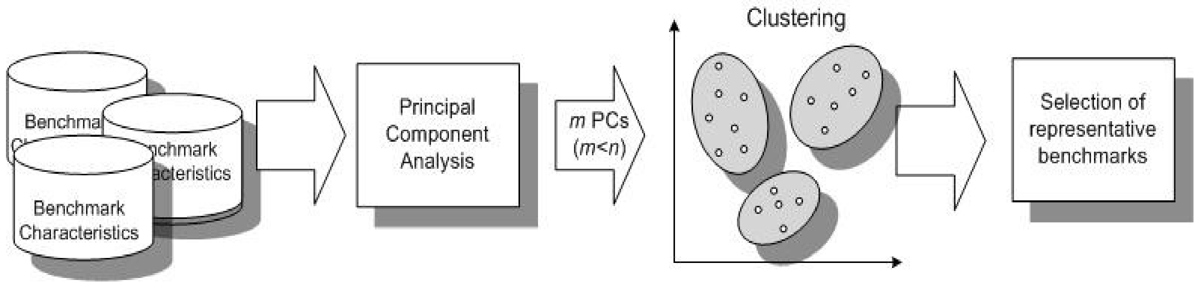
A computer system user's application is his best benchmark. But many times the standard benchmark suites used to compare the performance scores of different computer systems do not have the user's application in the suite and it is usually difficult to port an application on different platforms. This dissertation makes a contribution by proposing a methodology for predicting performance of a user's application based on its similarity with the available benchmarks and the performance numbers of these benchmarks on different computer systems.

Chapter 2: Benchmarking Subsetting Using Program Similarity

Analysis of program behavior has become important for guiding the process of design and performance evaluation of microprocessors. Characterization of benchmarks to study their position in the workload space can help to compare different features of benchmarks. A workload space is an *n* dimensional space formed using characteristics of benchmarks where *n* is the number of characteristics. Each point in the workload space represents a set of characteristics with certain values. Once the benchmarks are mapped in the workload space, a set of benchmarks with diverse characteristics can be obtained for efficient simulation based studies. The process of benchmark subsetting involves measuring similarity between benchmarks and finding a smaller set of benchmarks to represent the whole suite. In this chapter a methodology to obtain a representative subset of a benchmark suite is presented. In the experiments performed to demonstrate this methodology, the SPEC CPU2000 benchmark suite and the Mediabench [38] and MiBench [21] suites are used and subsets for each of the suites are presented. Standard Performance Evaluation Corporation (SPEC) is a benchmark consortium formed in 1988 by representatives from different computer vendors. Since then SPEC has released five CPU benchmark suites including the latest CPU2006 suite. The SPEC CPU2000 is a set of compute intensive scientific programs which stress the processor, memory system and also tests the compiler. The Mediabench and MiBench suites are benchmarks developed by academic researchers and are a set of real programs from media application domain. The methodology proposed in this chapter is applicable for other benchmark suites as well, as long as relevant characteristics of the programs are measured e.g. if the benchmarks do not stress the CPU then characteristics related to CPU may not be of interest while forming a subset.

Figure 2.1 shows the framework used for subsetting. The first and the most important step in benchmark subsetting is characterization of benchmark programs to measure program similarity. Two benchmarks are considered to be similar if they have similar program characteristics. The second step preprocesses the data to remove correlated metrics and reduce dimensionality of the data. PCA is a multivariate statistical analysis technique which is used in this dissertation to preprocess the benchmark characteristics. The transformed characteristics called Principal Components (PCs) are then used for clustering. Different types of clustering algorithms are considered with each one of them described in detail later in this chapter. The result of clustering is then used to choose one benchmark from each cluster to form a subset of representative benchmarks. In the remaining part of this chapter each of the blocks shown in Figure 2.1 are discussed in detail. This is followed by two experiments used to demonstrate the application of this methodology on two different benchmark suites from different domains.

Figure 2.1: Framework for benchmark subsetting


Benchmark Characteristics
Principal Component Analysis
m PCs (m<n)
Clustering
Selection of representative benchmarks

2.1 PROGRAM CHARACTERIZATION METHODOLOGY

In this dissertation the metrics used to characterize benchmarks can be broadly classified as *Microarchitecture Independent Metrics* and *Microarchitecture Dependent*

Metrics. One of the main contributions of this dissertation is the use of microarchitecture independent metrics to measure program similarity. These metrics are inherent to the program. On the other hand microarchitecture dependent metrics can be measured using performance monitoring counters [14] on a real system or by using a cycle accurate simulator. The microarchitecture dependent metrics are specific to the microarchitecture configuration on which they are measured. Each of the two different types of metrics is described in detail in the following sub-sections.

2.1.1 Microarchitecture Independent Metrics

A wide range of microarchitecture independent metrics that affect overall program performance have been used. An intuitive reasoning to illustrate how the measured metrics can affect the manifested performance is also discussed with the description of each of the metrics below. The metrics measured in this study cover a wide enough range of the program characteristics to make a meaningful comparison between the programs. In this dissertation microarchitecture independent metrics are broadly classified into the following categories:

- Instruction Mix
- Branch predictability behavior
- Metrics to measure Instruction Level Parallelism (ILP)
- Metrics to measure Data Locality
- Metrics to measure Instruction Locality

Instruction Mix

Instruction mix of a program measures the relative frequency of various operations performed by a program. The mix mainly has 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 an idea of whether the program is computation bound or memory bound.

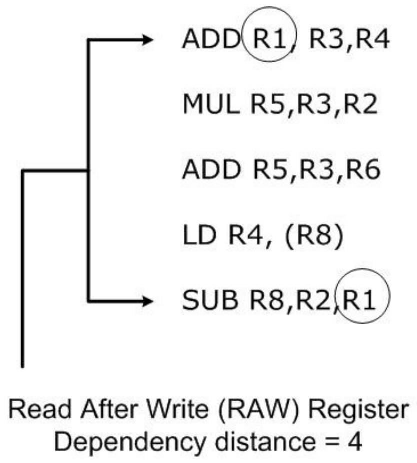
Branch Predictability Behavior

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.

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

Fraction of forward-taken branches: This metric measures the fraction of forward taken branches in the dynamic instruction stream of the program.

Figure 2.2: Illustration of measuring RAW register dependency distance



Metrics to measure Instruction level Parallelism (ILP)

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

Dependency Distance: A distribution of dependency distances is used to measure 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 consumption (read) of a register instance [15][45]. Figure 2.2 shows an illustration of how the dependency distance is measured. In Figure 2.2, there is a Read After Write (RAW) dependence on register R1 and the distance is equal to four. 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 measured dependency distance is represented by a distribution of six buckets: 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). Higher ILP can help issue multiple instructions at a given time (provided the computer system can issue multiple instructions) which helps to improve instruction throughput of a program. Programs with high instruction throughput will see better performance.

Metrics to measure Data Locality

Data Temporal Locality: Several locality metrics have been proposed in the past [11][12] [31][36] [55][56][58], however, many of them are computation and memory intensive. A modified average memory reuse distance metric from [36] is used 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. In [36] the distance between reuse is measured in terms of number of instructions. The evaluation is performed in four distinct block sizes, analogous to cache block sizes. This metric is calculated for block sizes of 16, 64, 256 and 4096 bytes. The choice of the block sizes is based on the experiments conducted by Lafage et.al. [36]. Their experimental results show that the above set of block sizes was sufficient to characterize the locality of the data reference stream with respect to a wide range of data cache configurations.

Example for measuring memory reuse distance:

Consider the following data memory address stream (address, access #): 0x2004 (#1), 0x2022 (#2), 0x300c (#3), 0x2108 (#4), 0x3204(#5), 0x200a (#6), 0x2048 (#7), 0x3108(#8), 0x3002(#9), 0x320c (#10), 0x2040(#11), 0x202f (#12). For a memory line of 16 bytes, the memory lines to which these addresses maps is calculated by masking the least significant 4 bits in the address. Therefore, the address in the data stream, 0x2004 will map to memory line 0x200, etc. The sequence of memory lines accessed by this address stream is: 0x200 (#1), 0x202 (#2), 0x300 (#3), 0x210 (#4), 0x320(#5), 0x200(#6), 0x204(#7),0x310(#8), 0x300(#9), 0x320(#10), 0x204(#11), 0x202(#12)

Addresses for reference #1 and #6 are different, but they map to the same memory line, 0x200, and therefore form a reuse pair (#1, #6). The list of all the reuse pairs in the example address stream is (#1, #6), (#2, #12), (#3, #9), (#5, #10), (#7, #11). For reuse pair (#1, #6), the reuse distance is the number of memory lines accessed between the reference #1 and #6, which is equal to 4.

Data Spatial Locality: Spatial locality information for data accesses is characterized by the ratio of the data temporal locality metric for higher block sizes to that of block size 16 mentioned above. The intuition here is that program with higher data spatial locality will see much shorter reuse distances on higher block sizes. Thus the ratio will get smaller. Hence it will help to find similarity in spatial locality between programs

Metrics to measure Instruction Locality

Instruction Temporal Locality: The instruction temporal locality 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 also calculated for *block* sizes of 16, 64, 256, 4096 bytes.

Instruction Spatial Locality: Spatial locality of the instruction stream is characterized by the ratio of instruction temporal locality metrics for higher block sizes to that of block size 16.

2.1.1 Microarchitecture Dependent Metrics

These metrics are specific to a certain microarchitecture on which they are measured. This makes the subset of benchmarks found using these characteristics applicable to a microarchitecture that is similar to the one that is used to measure the metrics. The metrics that can be measured are dependent on the capability of the real system and the performance monitoring counter infrastructure provided by the processor. Tools like PAPI [14] can be used to measure metrics such as cache miss-rates and branch misprediction rates for Intel processors. Apart from the simple counts, there are many complex events that can be measured. Because different tools are already available to measure microarchitecture independent metrics this dissertation does not describe the detail procedure to measure them. Readers can refer to the documents available for the tools like PAPI [14] to see how it can be used to do hardware performance counter measurements.

2.1.2 Advantages of microarchitecture independent metrics

Measuring the inherent characteristics of a program makes the results of the analysis applicable to wide range of microarchitecture configurations. This property of the microarchitecture independent metrics is advantageous because measuring data on different configurations and microarchitecture may not be possible due to lack of availability of different systems. On the other hand a drawback of microarchitecture independent metrics is that it takes longer to measure these metrics as compared to the microarchitecture dependent metrics, but still much faster than a cycle accurate simulator. Microarchitecture dependent metrics are used in subsetting the SPEC CPU2006 benchmark suite. But they are measured on different state of the art computer systems with different ISAs, compilers and microarchitecture configurations to account for the wide applicability of results. It is difficult to have such a wide variety of computer systems available for a study. The data was obtained from different computer system manufacturers that are a part of the SPEC CPU subcommittee. Without their help the experiment of subsetting using microarchitecture dependent metrics would not have been possible.

2.2 PRINCIPAL COMPONENTS ANALYSIS AS A PREPROCESSING STEP

PCA [17] 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 this dissertation, 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 X1, X2,...., Xp into p principal components (PC) Z1,Z2,…,Zp such that:

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

This transformation has the property Var [Z1] > Var [Z2] >…> Var [Zp] which means that Z1 contains most of the information and Zp the least. Given this property of decreasing variance of the principal components, the components with the lower values of variance can be removed from the analysis. This reduces the dimensionality of the data set while controlling the amount of information that is lost. In other words, q principal components (q << p) are retained that explain at least 75% to 90 % of the total information. Further, cluster analysis uses these q PCs as a set of variables.

2.2 CLUSTERING TECHNIQUES FOR BENCHMARK SUBSETTING

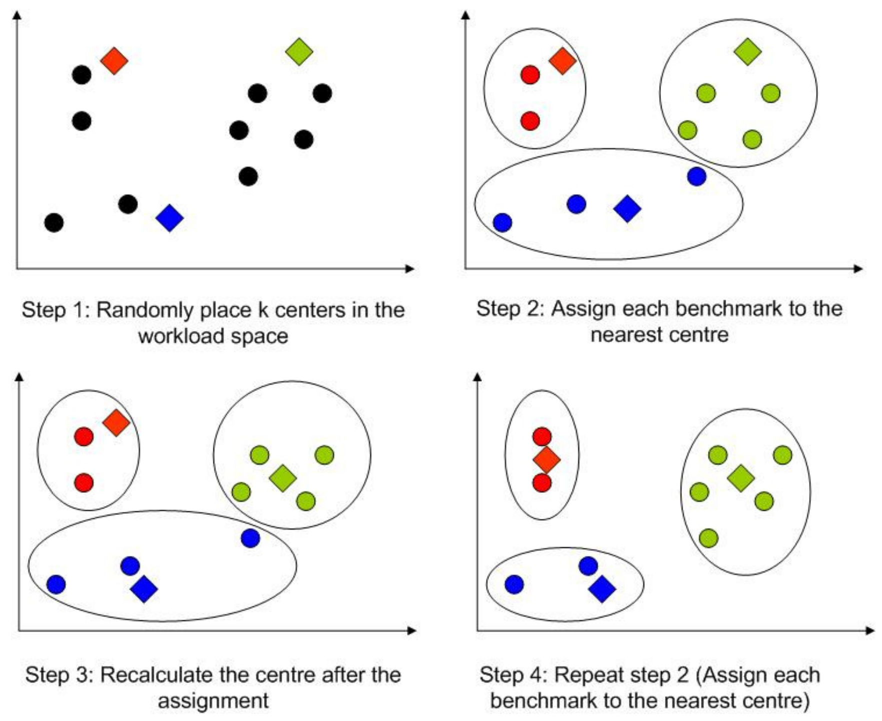
The two main clustering techniques that are used in this dissertation are K-means clustering and Hierarchical clustering [30]. Each of the clustering techniques is described in detail.

K-means clustering groups all cases into exactly *k* distinct clusters which show maximum difference in their characteristics (workload characteristics in this dissertation). Figure 2.3 shows the step-by-step process for k-means clustering. The steps are as follows:

1. Randomly place *k* centers in the space built using the data-points (benchmarks in this case)
2. Assign each benchmark to the nearest centre
3. Based on the assignment of the benchmarks, re-calculate the position of the centers.

4. Reassign the benchmarks to the nearest center. In Figure 2.3 a benchmark which was assigned to the lowermost cluster got assigned to the cluster on top right. In this way the benchmarks get assigned to the clusters and swap clusters. These steps are repeated usually till none of the benchmarks change from one cluster to another.

Figure 2.3: Illustration of the step-by-step process of K-means clustering


Step 1: Randomly place k centers in the workload space
Step 2: Assign each benchmark to the nearest centre
Step 3: Recalculate the centre after the assignment
Step 4: Repeat step 2 (Assign each benchmark to the nearest centre)

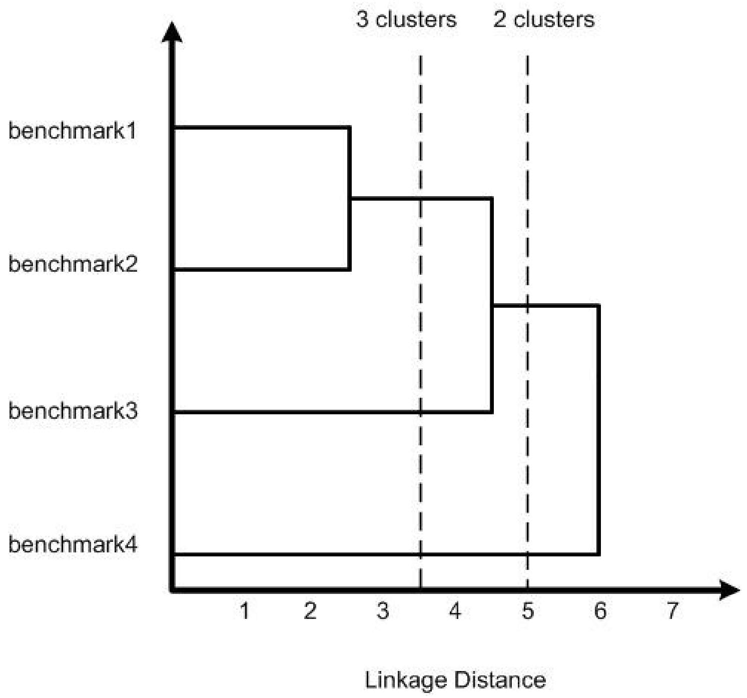
Only a certain value of *k* fits the data set well. As such, various values of *k* are explored in order to find the optimal clustering for the given data set. Also, it is a well-known fact that result of K-means clusters depends a lot on the initial placement of cluster centers. So, K-means clustering is done for hundred different random seeds to find the best initial placement of centers. To find the best values of *k,* Bayesian Information Criterion (BIC) as shown by Sherwood et.al [52] is used. The BIC is a measure of the goodness of fit of a clustering to a data set. The value of *k* that shows the highest BIC score is selected.

Hierarchical clustering is a bottom to top approach. It starts off with each data point being a cluster of its own. In the next iteration two data-points that are closest to each other are combined into a single cluster. The complete linkage distance measurement defines distance between two clusters as the distance between the farthest data points in those two clusters. Hierarchical clustering is a technique for finding relatively homogeneous clusters of items based on their measured characteristics. Given a set of N programs to be clustered and an N x N similarity matrix containing the distance between the programs using the measured workload characteristics, the hierarchical technique starts with each case (benchmark) in a separate cluster and then combines the clusters sequentially, merging the clusters at each step until all cases merge to form one cluster. When there are N cases, this involves N-1 clustering steps, or fusions. The algorithm used for hierarchical clustering can be described in steps as follows:

1. Each program is assigned to its own cluster, such that if there are N programs, there are N clusters, each containing just one program. The distances (similarities) between the clusters equal the distances (similarities) between the pro-grams they contain. Complete linkage distance measurement as described above is used.
2. Find the closest (most similar) pair of clusters and merge them into a single cluster.
3. Compute distances (similarities) between the new cluster and each of the old clusters.
4. Repeat steps 2 and 3 until all items are clustered into a single cluster of size N.

This hierarchical clustering process can be represented by a plot in a tree format called dendrogram, where each step in the clustering process is illustrated by a joint in the tree. The numbered scale corresponds to the linkage distance obtained from the hierarchical cluster analysis. Figure 2.4 shows a simple illustration of a dendrogram. The plot connects two clusters at a point where the distance between two clusters is equal to the linkage distance shown on the horizontal axis. This technique does not provide an optimal number of clusters. It is up to the user to decide the number of clusters based on the linkage distance. Smaller linkage distance means the two data points are closer and hence similar to each other. In Figure 2.4 the two benchmarks 1 and 2 are close to each other than the other ones. These two benchmarks join at a linkage distance of 2.5. Benchmark 4 is the farthest one and it joins all the rest of the benchmarks at linkage distance of 6. If a user needs 3 out of 4 benchmarks then he should choose one of the benchmarks from a cluster of 1 and 2 but definitely choose both 3 and 4 as can be seen by drawing a dotted line. Similarly 2 benchmarks can be chosen out of 4.

Figure 2.4: Illustration of a dendrogram



2.3 SUBSETTING SPEC CPU2000 BENCHMARK SUITE USING MICROARCHITECTURE INDEPENDENT METRICS

SPEC CPU benchmarks are a set of computation intensive programs which stress memory and CPU. Table 2.1 shows a list of all the SPEC CPU2000 benchmarks that are used in this experiment.

Table 2.1: SPEC CPU2000 benchmarks.

SPEC CPU2000			
Name	**Input**	**INT /FP**	**Instruction Count**
Gzip	input.graphic	INT	103.7 billion
Vpr	Route	INT	84.06 billion
Gcc	166.i	INT	46.9 billion
Mcf	inp.in	INT	61.8 billion
Crafty	crafty.in	INT	191.8 billion
Parser	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	*

* Programs that did not run due to issues with system calls

The benchmarks are compiled using Compaq Alpha AXP-2116 processor using the Compaq/DEC C, C++, and the FORTRAN compiler. The benchmarks are statically built under OSF/1 V5.6 operating system using full compiler optimization.

Microarchitecture independent metrics that are described earlier in this chapter are used for subsetting. All the microarchitecture independent metrics are measured using a modified version of Simplescalar simulator [2]. Two subsets of the SPEC CPU2000 benchmarks are generated, the first one using all the microarchitecture independent metrics described previously, and the second based only on similarity in data locality characteristics. The dimensionality of the data is reduced using the PCA technique described earlier in the paper. In this experiment K-means clustering algorithm is applied using the tool provided in the SimPoint kit [52], 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) score is optimal.

Table 2.2: Clusters based on all the microarchitecture independent metrics.

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***

2.3.1 Subsetting using all the microarchitecture independent metrics

Using K-means clustering technique described above, the BIC shows 8 clusters as a good fit for the measured data set. Table 2.2 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 [10] 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 in Table 2.2 suggests that the most popular programs in the listing provided by Citron [10] 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. Also, 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 using both programs adds redundancy. Clusters in Table 2.2 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. The benchmark, *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 [60], 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.

2.3.2 Subsetting using only the data locality characteristics

In this part of the analysis a subset of the SPEC CPU2000 benchmark suite is formed only considering the 7 characteristics of SPEC CPU2000 programs that are closely related to the temporal and spatial data locality of a program for block sizes of 16, 64, 256, and 4096 bytes, and the ratios of each of the temporal data locality metric for window sizes of 64, 256, and 4096 bytes, to that for block 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. Same methodology i.e. PCA for data reduction and removing correlation amongst variables and cluster analysis for grouping similar programs is used to form the subset. Table 2.3 shows the groups of programs that have similar data locality characteristics. The programs marked in bold are the programs that lie closest to the center of their cluster and hence are the representatives of their own cluster. For clusters that contain two programs, any one program can be the representative since both the programs are equidistant from the center. The benchmark, *mcf* which stresses memory the most and is known to show very high cache miss-rates falls in its own cluster which in a way validates that the characteristics are able to capture the data access behavior well.

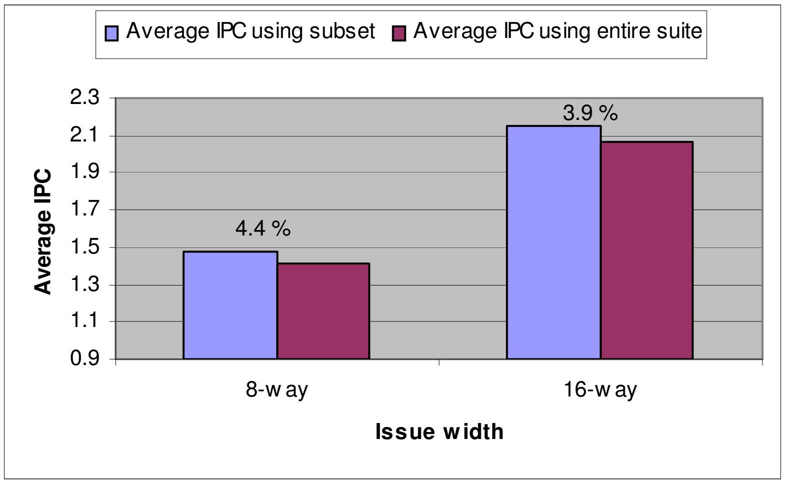
Table 2.3: Clusters based on only the data locality characteristics.

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*

2.3.3 Validation of SPEC CPU2000 benchmark subsets

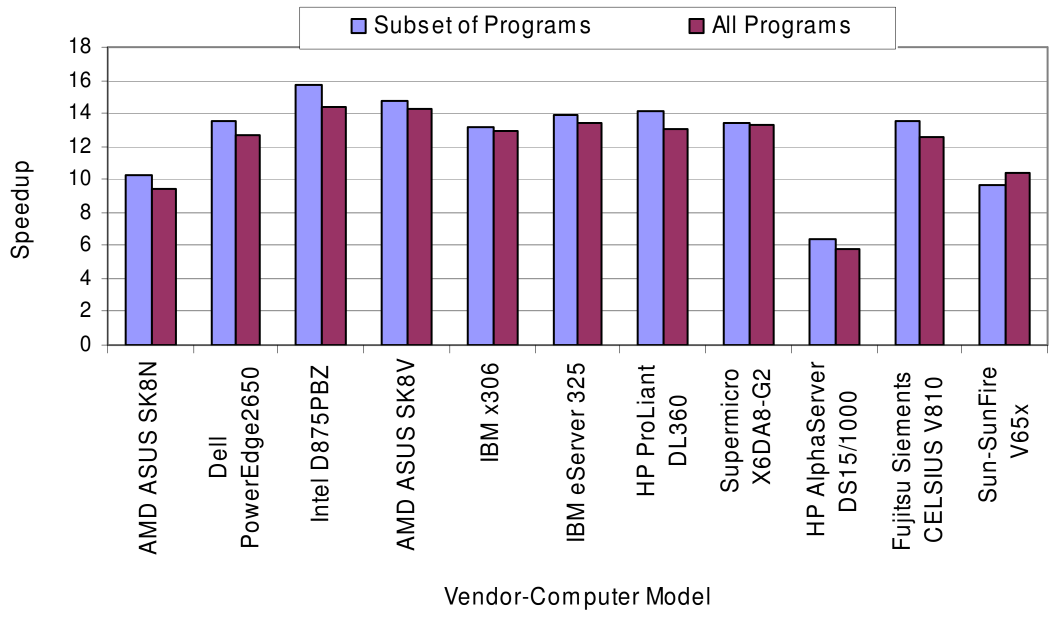
It is important to know whether the subsets that are formed are meaningful and are indeed representative of the SPEC CPU 2000 benchmark suite. To validate the subsets the average IPC, speedup and cache miss-rate of the subset is compared the average values of the respective performance numbers for the whole benchmark suite. This will increase the confidence in using these subsets for experiments in computer architecture studies.

Figure 2.5: Validation of a subset of SPEC CPU2000 benchmarks using average IPC



Using the subset based on overall program characteristics average IPC of the entire suite for two different microarchitecture configurations with issue widths of 8 and 16 is calculated. Figure 2.5 shows the average IPC of the entire benchmark suite calculated using the program subset, and also using the entire benchmark suite. It takes very long to obtain the IPC numbers for whole benchmarks hence the IPC numbers on 8-way and 16-way issue widths for every program in the SPEC CPU2000 benchmarks are taken from Wenisch et. al. [62]. The processor configuration used to measure IPC were: 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 [62]. In Table 2.2 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 2.2, the weight for fma3d (cluster 4) is 7. The error in average IPC for both configurations is less than 5% (shown in Figure 2.5). Since the IPC of the entire suite can be estimated with reasonable accuracy using the subsets, it can be concluded that the subset is a good representative of the whole suite.

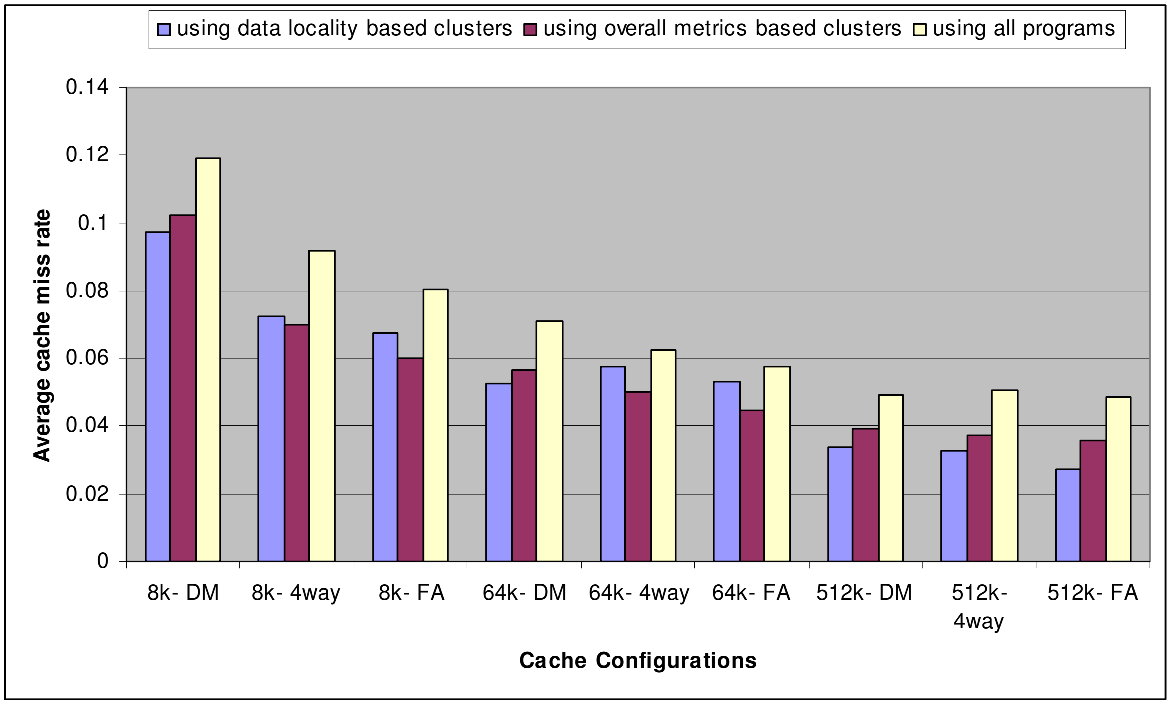
Figure 2.6: Validation of a subset of SPEC CPU2000 benchmarks using average speedup



Another validation experiment is done to demonstrate the usefulness of SPEC CPU2000 subsets and estimate the speedup on eleven different machines. Figure 2.6 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 are directly obtained from the results published by SPEC [57]. Similar to the previous validation experiment, a weight corresponding to the number of programs that the program represents (i.e. the number of programs in its cluster) is assigned to all the benchmarks in the subset. The maximum error in the speedup estimated using the subset is 9.1%. This supports the statement that the subset formed in Table 2.2 represents the benchmark suite very well.

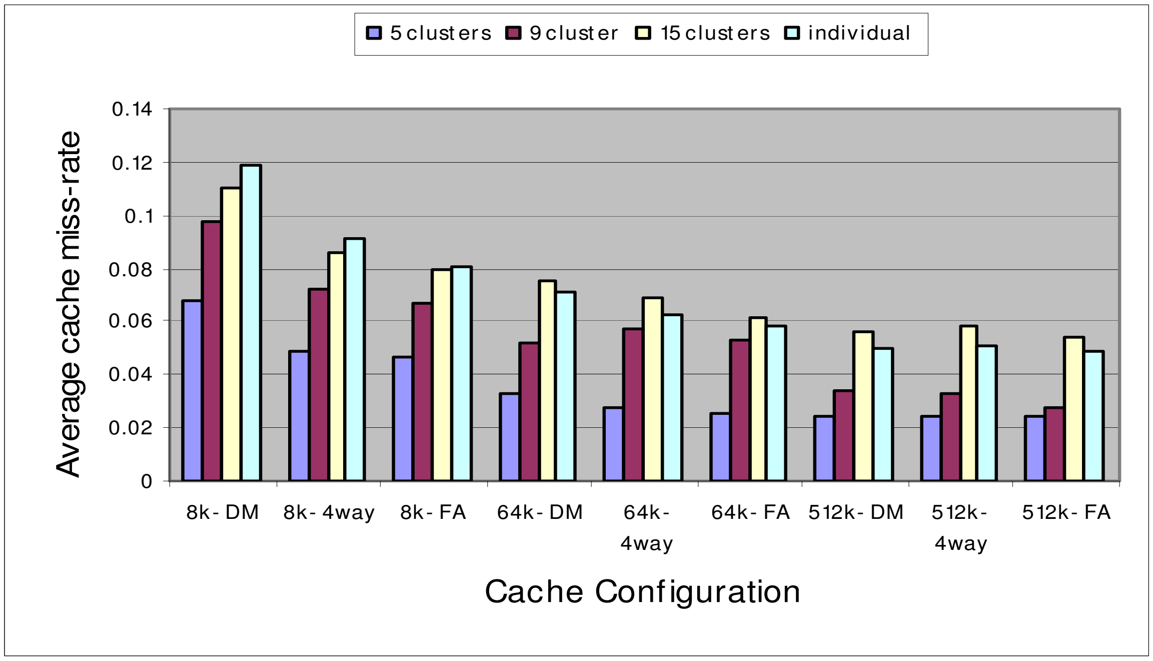
Figure 2.7 shows average L1 data cache miss-rate of the benchmark suite estimated using the subset of programs obtained from Table 2.3 along with the average miss-rate using the entire benchmark suite.

Figure 2.7: Validation of a subset of SPEC CPU2000 benchmarks using average cache miss-rate



Cache miss-rates for 9 different L1 data cache configurations are used from Cantin et.al. [9] to validate the subsets. The subset should be able to estimate average cache miss-rates accurately. From these results it can be inferred that the program subset derived in Table 2.3 is indeed representative of the data locality characteristics of programs in SPEC CPU 2000 benchmark suite.

Figure 2.8: Sensitivity to number of clusters on estimation of average cache miss-rate



The number of representative programs to be chosen from a benchmark suite depends on the level of accuracy desired. Theoretically, as the number of representative programs increases, 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. The average miss-rate of the benchmark suite can be calculated with an increasing level of accuracy if the programs are partitioned 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 BIC criterion. Figure 2.8 shows the estimated miss-rate of the benchmark suite using a subset of 5, 9, and 15 programs that are clustered based on only the locality characteristics and compared to the average cache miss-rate using all the benchmarks. As there is increase in 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.

2.4 SUBSETTING MEDIA BENCHMARK SUITE USING MICROARCHITECTURE INDEPENDENT METRICS

In the previous section, results of subsetting SPEC CPU2000 benchmarks are discussed based on the microarchitecture independent metrics. SPEC CPU benchmarks are computation intensive scientific applications. To demonstrate that the methodology is also applicable to a different type of benchmark suite, subsetting is applied to media benchmarks. Media benchmarks are mostly used to evaluate performance of embedded processors or systems. Table 2.4 shows a list of media benchmarks that are used in the subsetting experiment. Mediabench and MiBench are two different benchmark suites formed using applications from similar domain. The first column shows the names of the benchmarks and the second column shows the area of the application. Looking at the last column in Table 2.4 which is the instruction count of each benchmark and comparing it with the last column of Table 2.1, it is obvious that media benchmarks are much shorter than the SPEC CPU2000 benchmarks. All the microarchitecture independent metrics that are discussed earlier in this chapter are measured for all the benchmarks shown in Table 2.1. After measuring the characteristics, PCA and clustering is applied to find a subset of media benchmarks. These subsets are then validated using IPC and cache miss-rates.

Table 2.4: List of media benchmarks used in the subsetting experiment

MiBench		
Application	**Type**	**Dynamic Instruction Count**
Basicmath	Automotive	1.52 billion
Bitcount	Automotive	688.3 million
Qsort	Automotive	513.8 million
susan –input1	Automotive	327.33 million
susan –input2	Automotive	76.06 million
susan –input3	Automotive	31.06 million
Cjpeg	Consumer	1.18 billion
Djpeg	Consumer	26.86 million
Typeset	Consumer	0.48 million
Dijkstra	Network	257.78 million
Patricia	Network	399.30 million
Ghostscript	Office	872.97 million
Rsynth	Office	878.83 million
Stringsearch	Office	3.45 million
Sha	Security	107.79 million
crc32	Telecomm	692.20 million
Fft	Telecomm	238.89 million
Invfft	Telecomm	218.26 million
Gsm	Telecomm	2.10 billion
Mediabench		
Application	**Type**	**Dynamic Instruction Count**
Adpcm	Compression	7.09 million
Adpcm	Decompression	8.86 million
Epic	Compression	58.37 million
Epic	Decompression	10.25 million
g.721	Encoder	381.84 million
g.721	Decoder	399.82 million
Ghostscript	-	877.77 million
Jpeg	Compression	18.65 million
jpeg	Decompression	4.75 million
Mesa	3D graphics	127.95 million
Mpeg2	Decoder	161.62 million
Mpeg2	Encoder	1.55 billion
Rasta	-	24.86 million

Table 2.5: An optimal subset of media benchmarks using k-means clustering

Cluster 1	mediabench_unepic, mediabench_ghostscript, **mediabench_cjpeg**, mibench_ consumer_cjpeg, mibench_office_ghostscript, mibench_office_rsynth
Cluster 2	mediabench_mesa, **mediabench_rasta**, mibench_automotive_qsort, mibench_network_dijkstra, mibench_network_patricia, mibench_office_stringsearch, mibench_security_sha, mibench_telecomm_CRC32
Cluster 3	mediabench_epic, mediabench_g721_decoder, mediabench_g721_encoder, mediabench_djpeg, mediabench_mpeg2_decoder, mediabench_mpeg2_encoder, mibench_automotive_basicmath, mibench_automotive_susan2, mibench_automotive_susan3, mibench_consumer_djpeg, mibench_consumer_typeset, mibench_ telecomm_FFT, **mibench_telecomm_invFFT**, mibench_telecomm_gsm
Cluster 4	**mediabench_adpcm_decoder**, mediabench_adpcm_encoder, mibench_automotive_susan1
Cluster 5	mibench_automotive_bitcount

2.4.1 Subsetting media benchmarks using all the microarchitecture independent metrics

Table 2.5 shows the list of program-input pairs that fall into five optimal clusters after performing k-means clustering. The program-input pairs marked in bold are closest to the centre of the cluster and are hence the representatives of their own cluster. Although MiBench and Mediabench are two different suites, they still have some similar programs e.g. cjpeg and djpeg. Cjpeg compresses a ppm image into jpeg and djpeg decompresses a jpeg representation into a ppm image. Although the cjpeg programs have a different image as a workload in the MiBench and Mediabench suite they show similar behavior and hence fall in the same cluster (Cluster 1). Similarly, djpeg, from both MiBench and Mediabench, lies in the same cluster (Cluster 3). This shows that the input set did not affect the program behavior of djpeg benchmark, but in case of a MiBench benchmark susan, It has three input sets and input set 1 is different from 2 and 3. In case of Mediabench programs g.721 and adpcm, their encoder and decoder show similar program behavior and hence lie in the same cluster (Clusters 3, 4).

Table 2.6: An optimal subset of media benchmarks using only data locality characteristics.

Cluster 1	mibench_automotive_susan1
Cluster 2	mibench_automotive_susan3
Cluster 3	mediabench_epic, mediabench_cjpeg, mediabench_djpeg, mediabench_mpeg2_decode, **mibench_ consumer_djpeg,** mediabench_mpeg2_encoder, mibench_consumer_cjpeg, mibench_consumer_typeset, mibench_telecomm_fft, mibench_telecomm_invfft
Cluster 4	mediabench_adpcm_decoder**,** mediabench_adpcm_encoder
Cluster 5	mediabench_mesa, mediabench_rasta,mibench_ automotive_susan2, mibench_network_dijkstra, mibench_office_stringsearch, **mibench_security_sha_large**
Cluster 6	automotive_basicmath_large**,** network_patricia_large
Cluster 7	mediabench_unepic, mediabench_g721_decoder, ediabench_g721_encoder, automotive_bitcount, **mibench_automotive_qsort,** mibench_office_rsynth, mibench_telecomm_crc32, mibench_telecomm_gsm
Cluster 8	mediabench_ghostscript, mibench_office_ghostscript

2.4.2 Subsetting media benchmarks using only the data locality characteristics

Table 2.6 shows clusters of media programs based on the four temporal data locality and three spatial data locality characteristics described before. The temporal data locality characteristics are measured for block sizes of 16, 64, 256, and 4096. PCA and K-means clustering method is applied to all the media programs but just using the seven data locality characteristics to obtain 8 optimal clusters. The programs marked in bold are closest to the center of the cluster and hence are chosen as representative program-input pairs. Any program can be chosen as the representative for clusters that have two programs.

2.3.3 Validation of media benchmark subsets

Using the subset of media benchmarks in Table 2.5 the average IPC of the entire suite is estimated for two different superscalar microarchitecture configurations with issue widths of 2 and 4. The IPC of each media program is measured on the following two configurations using sim-outorder simulator in Simplescalar tool-kit. The details of these configurations are 2-way issue, RUU/LSQ 32/16, Memory System 8KB 2-way L1 I/D, 256K 4-way L2, ITLB/DTLB 4-way 16 entries/ 4-way 32 entries 30 cycle misses, L1/L2/mem latency of 1/6/36 cycles, Functional Units 2 I-ALU, 1 I-MUL/DIV, 2FP-ALU, 2 FP-MUL/DIV, branch predictor Combined 2k tables 4 cycle misprediction penalty. Another configuration used, is a 4 way issue with RUU/LSQ 64/32, Memory System 16KB 2-way L1 I/D, 512K 4-way L2, ITLB/DTLB 4-way 16 entries/ 4-way 32 entries 30 cycle misses, L1/L2/mem latency 2/8/36 cycles, Functional Units 4 I-ALU, 2 I-MUL/DIV, 4 FP-ALU, 2 FP-MUL/DIV, Branch Predictor Combined 2k tables 4 cycle misprediction penalty. Weighted average IPC for the subset is calculated.

Figure 2.9: Validation of subset of media benchmarks using IPC

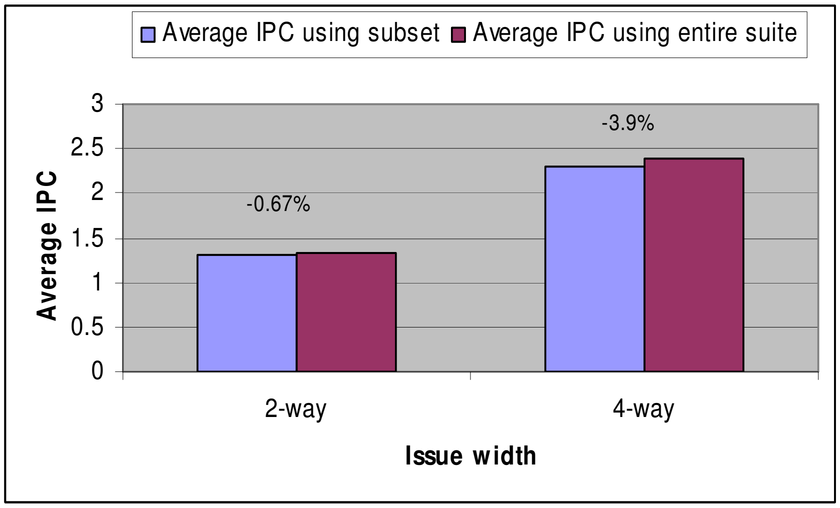
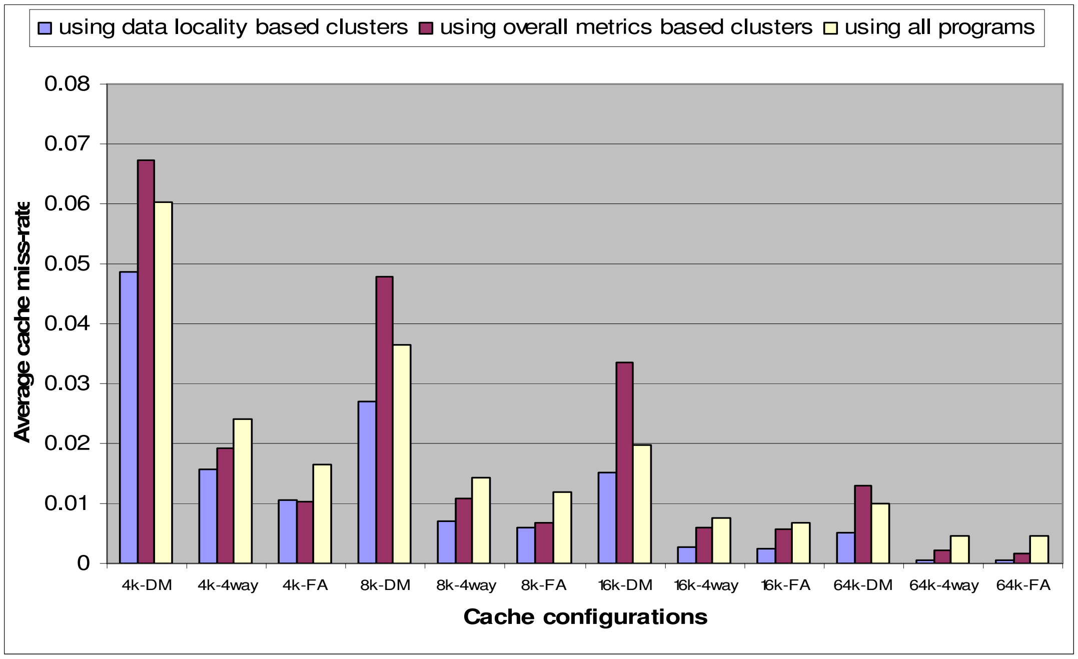


Figure 2.9 shows the plot comparing the weighted average IPC of the subset with the true average of the suite. The weight for each representative is equal to number of programs in its cluster. The error in estimating IPC for a 2-way configuration is -0.67% and for issue width of 4 is -3.9%.

Figure 2.10 shows the average data cache miss-rate of the entire set of media programs using the subset that is obtained using only data locality characteristics. Four different cache configurations are used to validate the subset. The cache configurations chosen for this analysis are: 4k, 8k, 16k and 64k size, and each of these sizes with a direct mapped, 4-way set associative, and full associative. All cache configurations have 64 bytes block size and an LRU replacement policy. Smaller cache size configurations are chosen as compared to the ones used for validation of SPEC CPU2000 benchmarks because, for higher sizes, cache miss-rates for media programs are very close to zero.

Figure 2.10: Validation of subset of media benchmarks using cache miss-rates



2.4 SUMMARY

Partial use of benchmark suites for evaluating design trade-offs is very common. Many times the benchmarks are randomly picked without a careful analysis of all the benchmarks. An educated choice of benchmarks is a necessary part of good performance evaluation practice. The process of choosing benchmark based on their characteristics is called subsetting. The first and the most important step of subsetting is characterization of benchmarks. Two types of characteristics i.e. Microarchitecture independent metrics and Microarchitecture dependent metrics are described. Also the advantage of the first one over the second approach is discussed. Other than the characterization of benchmarks, the process of subsetting also involves a data preprocessing step called PCA followed by clustering. The subsets formed are validated by checking if the average performance metric (IPC and speedup) of the complete benchmark suite can be projected by just the benchmarks in the subset. Many times the study is specific to a certain part of the system e.g. memory hierarchy, branch predictor. For these studies, it is good to analyze the characteristics separately and find a subset for each of these characteristics. For each benchmark suite a subset based on memory access behavior has been shown. The quality of such a subset is judged by comparing the average cache miss-rates of the complete suite and the subset.

This chapter demonstrates the use of microarchitecture independent metrics to form a subset of the SPEC CPU2000 benchmarks. Another experiment with similar analysis is done for the media benchmarks suites (Mediabench and MiBench). The subsets formed showed that the average error of IPC projection for the SPEC CPU2000 and media benchmark suites is less than 5%. The average error in projection of speedup for the SPEC CPU2000 benchmarks is less than 10%.

Based on the results and validation experiments, if the time required to simulate the entire SPEC CPU2000 benchmark suite is prohibitively high, the following subset of representative programs found using the subsetting based on microarchitecture independent metrics can be used for simulation based studies: applu, equake, fma3d, gcc, gzip, mcf, mesa, and twolf. A similar list for media benchmarks is as follows:

mibench_automotive_susan1, mibench_automotive_susan3, mibench_consumer_djpeg, mediabench_adpcm_decoder, mibench_automotive_basicmath_large, mibench_automotive_qsort, mediabench_ghostscript, mibench_security_sha_large.

Chapter 3: Comparing Benchmark Suites by Analyzing Workload Space Coverage

Ideally, a benchmark suite with good coverage should have benchmarks in all the areas of workload space. Usually, modern computer applications or the emerging applications that are used to form a benchmark suite are not evaluated rigorously to analyze their coverage in the workload space. Their position in the workload space shows which features of the design or computer system the benchmarks will stress and hence important for a good quantitative analysis. If there are multiple benchmark suites made from target applications of a design, it is important to compare them before using all of them e.g. in case of general purpose processors, where the target application list can be long comparing and analyzing different suites is essential. Comparison of benchmark suites can be done by mapping them together in the workload space together and comparing their coverage. If the dimensionality of the workload space is as small as two or three it will be easy to do visual inspection. But many times that is not the case and hence technique like clustering which is described earlier in this dissertation can be used. Each characteristic can be used separately to form different workload spaces of lower dimensionality to compare different suites. With the release of new generation of benchmark suites, its predecessor is retired. It is worthwhile effort to see if the new benchmark suite is really different from the older one. This might also suggest some trends in the evolution of applications since modern benchmarks are real applications. Many times the programs get carried forward from one suite to the other with some change in the source code and input. It will be interesting to see how that program's behavior changed over time.

Two experiments are described in this chapter. First experiment uses microarchitecture independent metrics measured for all the four generations of SPEC

CPU benchmarks including CPU2000 and three of its predecessors to study the coverage of each of the suites. The second experiment compares SPEC CPU2000 benchmarks with the Mediabench and MiBench suites. PCA and clustering is used in the experiments to compare multiple suites. PCA will reduce dimensionality which will make it easy to even visually identify the coverage of each benchmark suite. Clustering in case of higher dimensional workload space will split the workload space filled by the existing benchmarks and point out the outliers which increase the coverage.

3.1 SIMILARITY ANALYSIS ACROSS FOUR GENERATIONS OF SPEC CPU BENCHMARK SUITES

SPEC 89 was the first SPEC CPU benchmark suite. SPEC CPU2000 which was fourth in succession with the second in 1992 and third in 1995. In order to keep pace with the architectural enhancements, technological advancements, software improvements, and emerging workloads, new programs were added, programs susceptible to easy compiler optimizations were retired, program run times were increased, and memory activity of programs was increased in every generation of the benchmark suite. The objective of this paper is to understand how the inherent characteristics of SPEC benchmark programs have evolved over the first four generations. CPU2000 was being used for the longest time until recently before the CPU2006 was released. This experiment analyzes CPU benchmarks from the first four generations.

A collection of microarchitecture independent metrics, described earlier are used to characterize the generic behavior of four generations of SPEC CPU benchmark programs. The data is analyzed using PCA and cluster analysis to understand the changes in the CPU workloads over time. First, k-means clustering is used to find an optimal number of clusters for all the four generations of SPEC CPU benchmarks. In the subsequent sections, each important characteristic is analyzed separately for all the generations. In order to visualize the workload space the scores for the first two PCs for sixty programs on a two dimensional graph, and also plot a dendrogram showing the similarity between the programs.

3.1.1 Analysis using all microarchitecture independent metrics

In order to understand the (dis)similarity between programs across the four generations of SPEC CPU benchmark suites, cluster analysis is done to all the 60 benchmarks in the PC space. Clustering all the 60 benchmarks yields 12 optimal clusters, which are shown in Table 3.1. The benchmarks in bold are the cluster representatives and are closest to the centre of the cluster. For clusters that have exactly two programs none of the benchmarks is marked bold because any one can be the representative of that cluster. The benchmark names are appended with the suite they are from e.g. gcc (95) means the gcc benchmark from SPEC CPU95. There are benchmarks with the same name in different benchmark suites e.g. gcc(95) and gcc(2000).

Table 3.1: Optimal clusters for the four generations of SPEC CPU benchmark suites

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(2000), 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)**, fppp(92), eon(2000), vpr(2000), fppp(89), fma3d(2000), mesa(2000), ammp(2000)
Cluster 12	bzip2(2000**)**, gzip(2000)

A quick look at Table 3.1 gives us several interesting insights. First, out of all the 60 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. Because of this, the gcc programs from the 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) and gzip (2000) form one cluster (cluster 12). Perl(95), li(89) and li(95) are all interpreter type applications and all three of them lie in the same cluster. Compress and swim are the only two programs which have their benchmarks from different suites in different clusters. All other benchmarks which are a part of different generations fall in the same cluster. 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 CPU2000 benchmark suite is more diverse than its ancestors and shows a larger coverage in the workload space.

3.1.2 Analysis based on microarchitecture independent instruction locality metrics

PCA is done for only the microarchitecture independent instruction locality metrics and two principal components explaining 68.4 % and 28.6 % of the total variance are retained. Figure 3.1 shows the benchmarks in the PC space. In order to visualize the relative positions of the benchmarks in the workload space a tree, or dendrogram is also plotted using hierarchical clustering. Figure 3.2 shows the dendrogram obtained from applying hierarchical clustering to the data set in the PCA space. The horizontal scale of the dendrogram lists the benchmarks, and the horizontal scale corresponds to the linkage distance obtained from the hierarchical clustering analysis.

Figure 3.1: PC space for four generation of SPEC CPU benchmarks using instruction locality characteristics

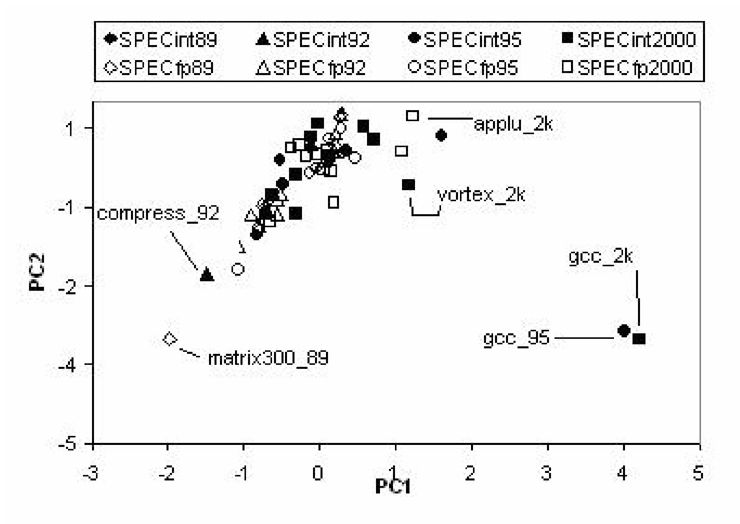
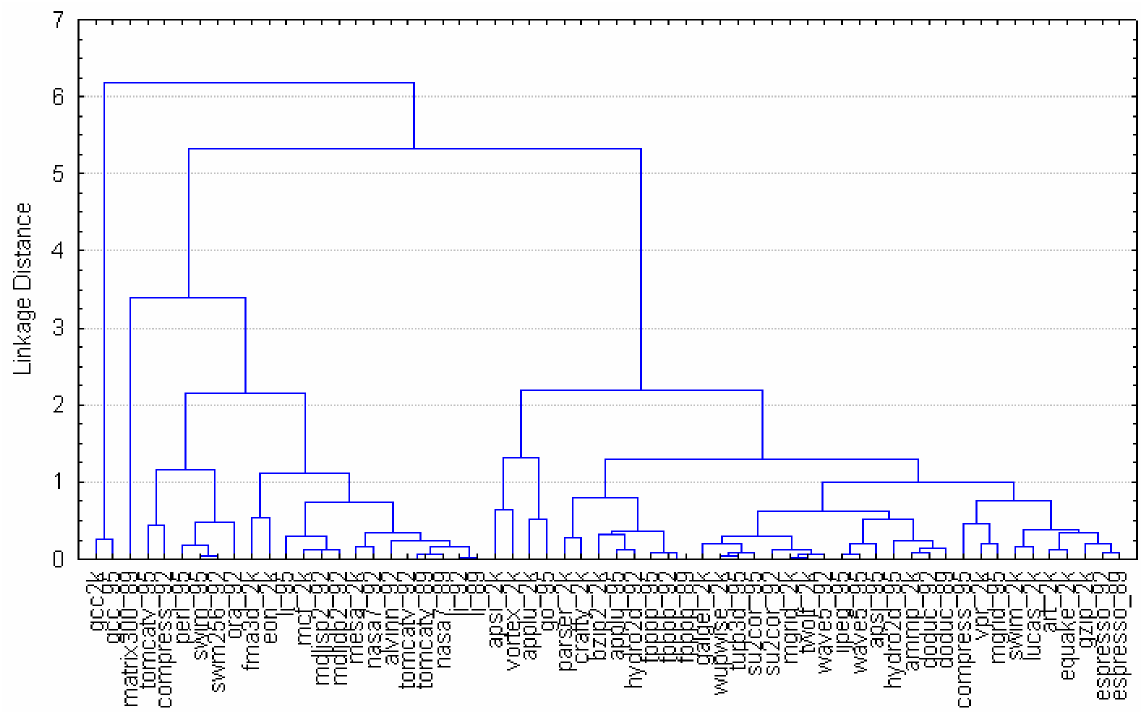


Figure 3.2: Dendrogram for four generation of SPEC CPU benchmarks using instruction locality characteristics



The shorter the linkage distance the closer, i.e., more similar, the benchmarks are to each other in the workload space. For example, in Figure 5, the gcc (2000) and gcc

(95) benchmarks combine into a cluster at a linkage distance of 0.2, and the cluster containing the two gcc benchmarks combines into a cluster containing all the other programs at a linkage distance of 6.2. This means that the gcc benchmarks from SPEC CPU95 and SPEC CPU2000 benchmark suites are more similar to each other than with all the other programs.

PC1 represents the instruction temporal locality and PC2 represents the instruction spatial locality of the benchmarks, i.e., the benchmarks with a higher value along PC1 show poor temporal locality for the instruction stream, and the benchmarks with a higher value along PC2 show good spatial locality in the instruction stream. Figures 3.1 and Figure 3.2 show that most of the benchmarks from all the SPEC CPU generations overlap. The biggest exception is gcc in SPECint2000 and SPECint95 (the two dark points on the plot on the extreme right). The gcc benchmark from the SPECint2000 and SPECint95 suites exhibits poor instruction temporal locality. It also shows very low values for PC2 due to poor spatial locality. The floating point program matrix300 from SPEC CPU89 suite and compress from SPEC CPU92 show very good temporal and spatial locality. The benchmark program applu from SPEC CPU2000 shows a very high value for PC2 and would therefore benefit a lot from an increase in block size. The fppp benchmarks from SPEC CPU89, SPEC CPU92, SPEC CPU95 suites, and the bzip2 and gzip benchmarks from the SPEC2000 suite show similar instruction locality.

Although the average dynamic instruction count of the benchmark programs has increased by a factor of x100, the static instruction count has remained more or less constant. This suggests that the dynamic instruction count of the SPEC CPU benchmark programs have been scaled drastically without significant increase in the static size of the benchmark code.

3.1.3 Analysis based on microarchitecture independent branch metrics

Figure 3.3: PC space for four generation of SPEC CPU benchmarks using branch metrics

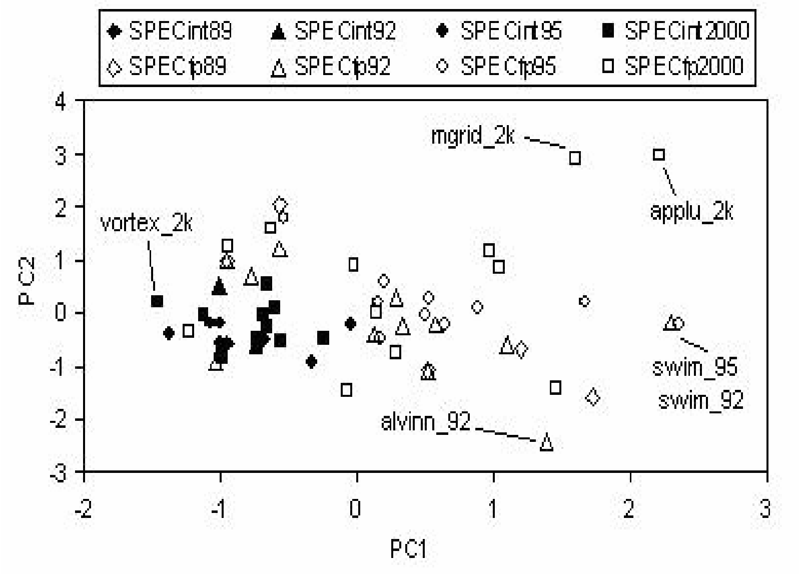
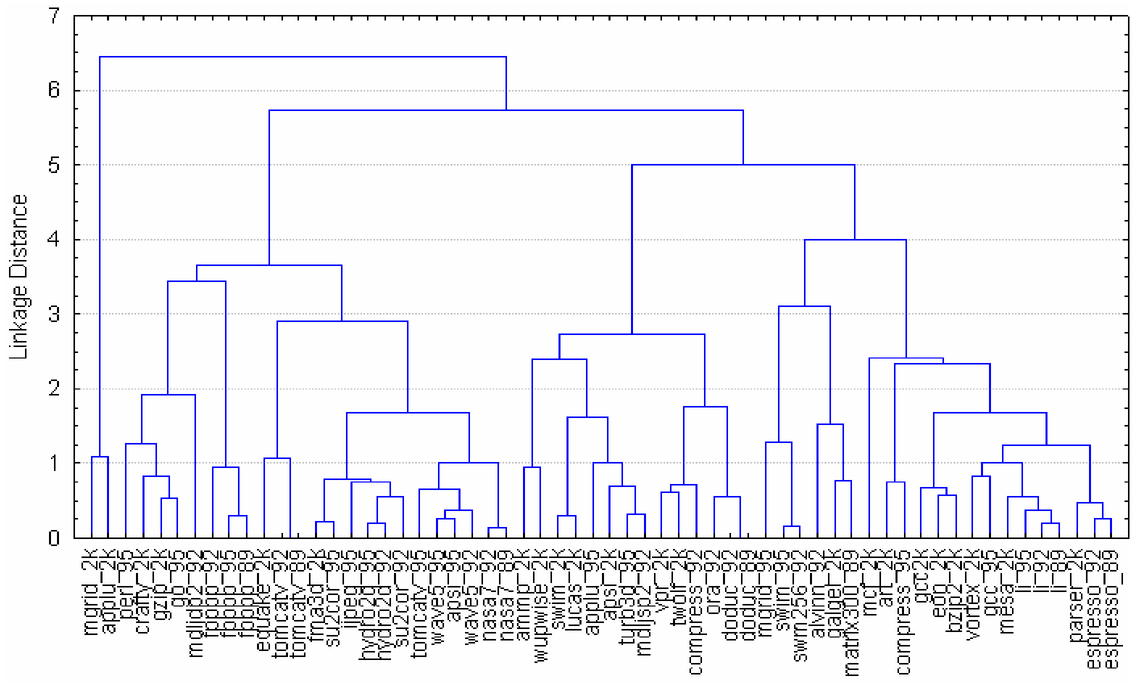


Figure 3.4: Dendrogram for four generation of SPEC CPU benchmarks using branch metrics



For studying the branch behavior the following characteristics are included in the analysis: the percentage of 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, 2 principal components are retained which explain 62% and 19% of the total variance, respectively. Figure 3.3 plots the various SPEC CPU benchmarks in this PCA space and Figure 3.4 is a dendrogram showing the linkage distance between the programs based on the branch characteristics.

From Figure 3.3 it can be seen that the integer benchmarks are clustered in an area. The floating-point benchmarks show 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; also, floating-point benchmarks typically are very well structured, and have a smaller percentage of forward branches, and fewer forward-taken branches. In other words floating point benchmarks tend to spend most of their time in loops. The two prominent outliers in the top right corner of this graph are SPEC 2000's mgrid and applu programs due to their extremely large average basic block sizes, 273 and 318 instructions, respectively. The two outliers on the right are swim benchmarks from SPEC92 and SPEC95 suites, due to their large percentage taken branches and small percentage forward branches. On the extreme left of the PCA space is vortex from SPEC2000 which shows a very low average basic block size. Due to a significant overlap seen in the plot it can be concluded that the branch characteristics of the SPEC CPU programs did not significantly change over the past four generations of SPEC CPU programs. Figure 3.4 also suggests that the branch behavior of programs doduc, espresso, fppp, hydro2d, li, and tomcatv whose branch characteristics have not changed across generations of SPEC CPU benchmark suites.

3.1.4 Analysis based on microarchitecture independent metrics for instruction level parallelism (ILP)

Figure 3.5: PC space for four generation of SPEC CPU benchmarks using ILP metrics

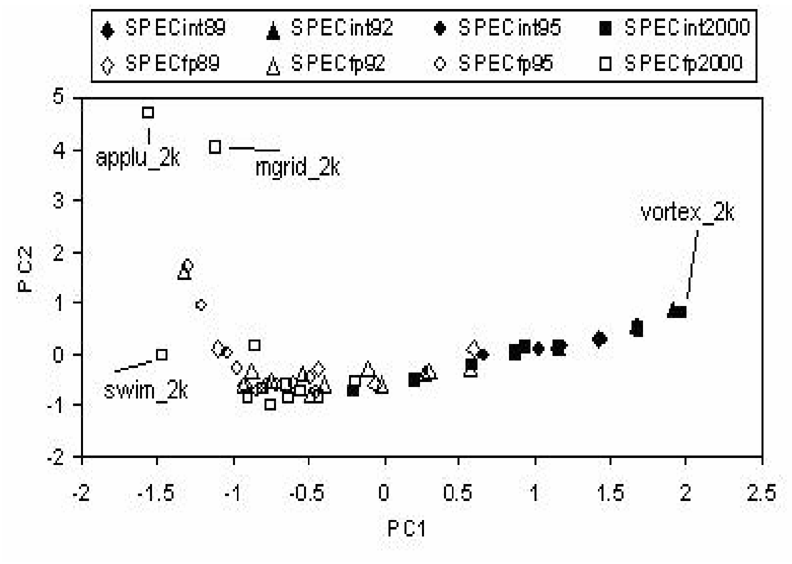
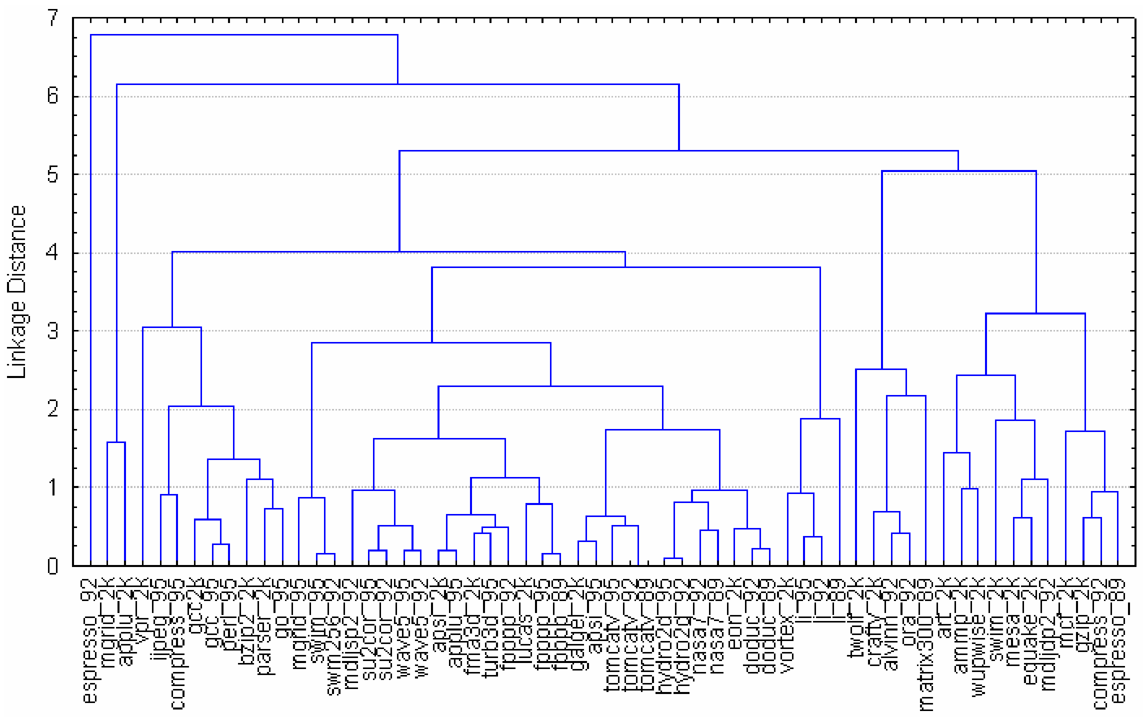


Figure 3.6: Dendrogram for four generation of SPEC CPU benchmarks using ILP metrics



In order to study the instruction-level parallelism (ILP) of the SPEC CPU suites the inter-instruction register dependency characteristic are used. This characteristic is closely related to the intrinsic ILP available in an application. Long dependency distances generally imply a high ILP. The first two principal components explain 96% of the total variance. The PCA space is plotted in Figure 3.5, and Figure 3.6 shows the dendrogram with the linkage distance between the programs based on their ILP characteristics.

The integer benchmarks typically have a high value along PC1, which indicates that these benchmarks have a higher percentage of short dependency distances. The floating-point benchmarks typically have larger dependency distances. The intrinsic ILP did not change over the 4 benchmark suites except for the fact that several floating-point programs from SPEC CPU89 and SPEC CPU92 suites (and no SPEC CPU95 or SPEC CPU2000 benchmarks) exhibit relatively short dependencies compared to other floating-point benchmarks; these overlap with integer benchmarks in the range $-0.1 < PC1 < 0.6$. In the top left corner in Figure 3.5 there are two outliers, mgrid and applu, that are quite far from a lot of other programs and show large dependency distances, which implies better ILP. The program swim from the SPEC CPU2000 suite also shows large dependency distances. The majority of the programs on the right side of the PCA space are integer programs with vortex from SPEC 2000 being the one with the largest number of short dependency distances. In Figure 3.6 shows that a lot of floating point programs across various generations, e.g., fppp, tomcatv, nasa7, li, and doduc, form a tight cluster. Hence it can be concluded that there is a lot of similarity between the ILP characteristics exhibited by the floating point programs across all four generations of the SPEC CPU suites.

3.1.5 Analysis based on microarchitecture independent data locality metrics

Figure 3.7: PC space for four generation of SPEC CPU benchmarks using data locality metrics

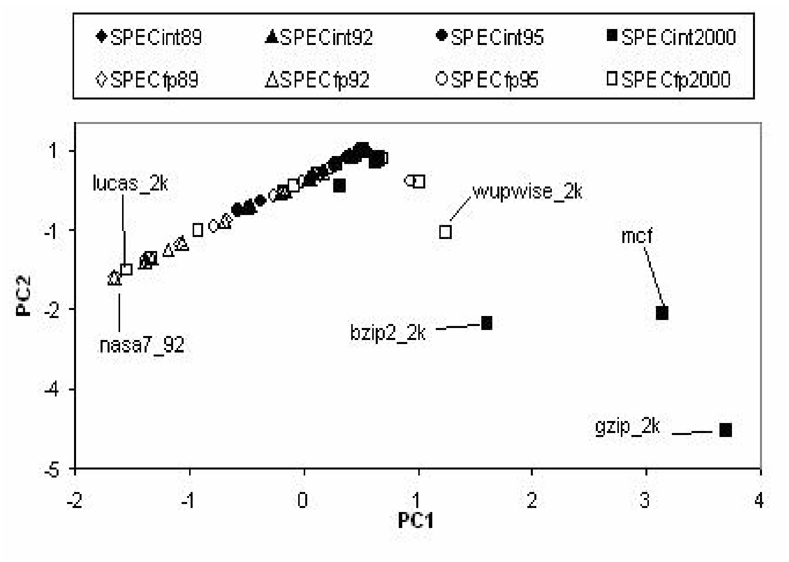
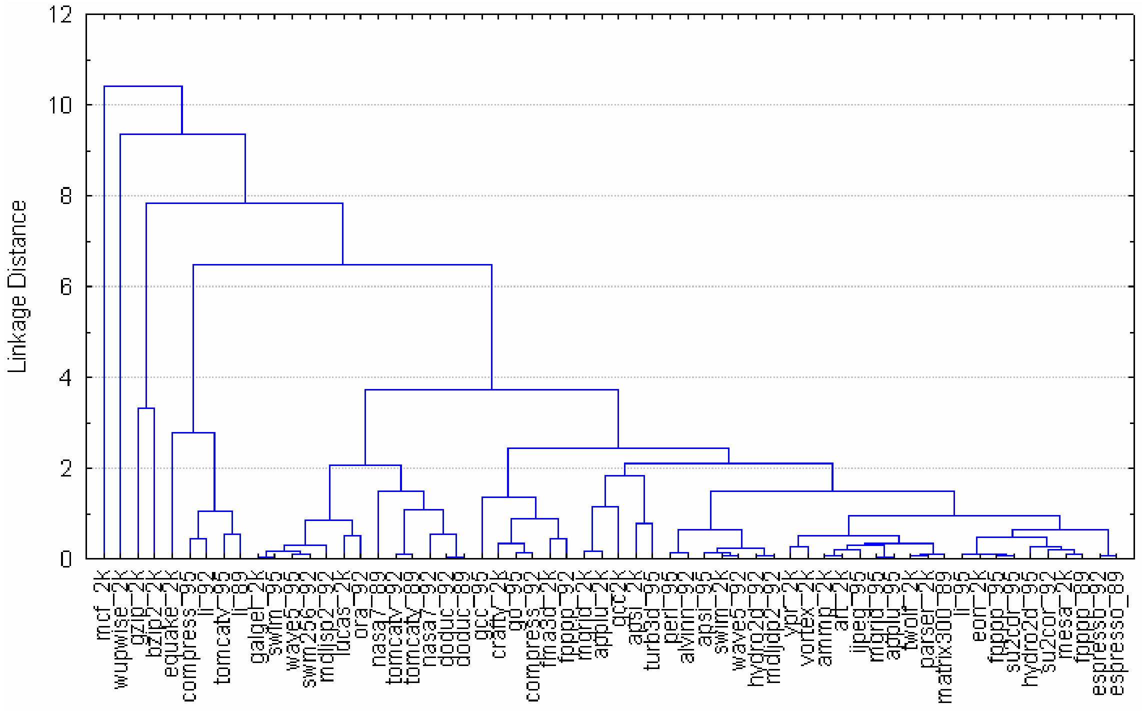


Figure 3.8: Dendrogram for four generation of SPEC CPU benchmarks using data locality metrics



For studying the temporal and spatial locality behavior of the data stream the locality characteristics based on reuse distance described before in this dissertation is used. Recall that the characteristics by themselves quantify temporal locality whereas the ratios between them are a measure for the spatial locality. Figure 3.7 shows a plot of the benchmarks in the PCA space built from these data locality characteristics, and Figure 3.8 shows the linkage distance between various programs on a dendrogram.

In Figure 3.7 the first principal component measures temporal locality, i.e., a more positive value along PC1 indicates poorer temporal locality. The second principal component measures spatial locality. Therefore, benchmarks with a high value along PC2 will thus benefit more from an increased cache line size. From Figure 3.7 it evident that several SPEC CPU2000 and CPU95 benchmark programs, namely bzip2, gzip, mcf, and wupwise, 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 the block sizes improves performance of these programs more than they do for the other benchmarks.

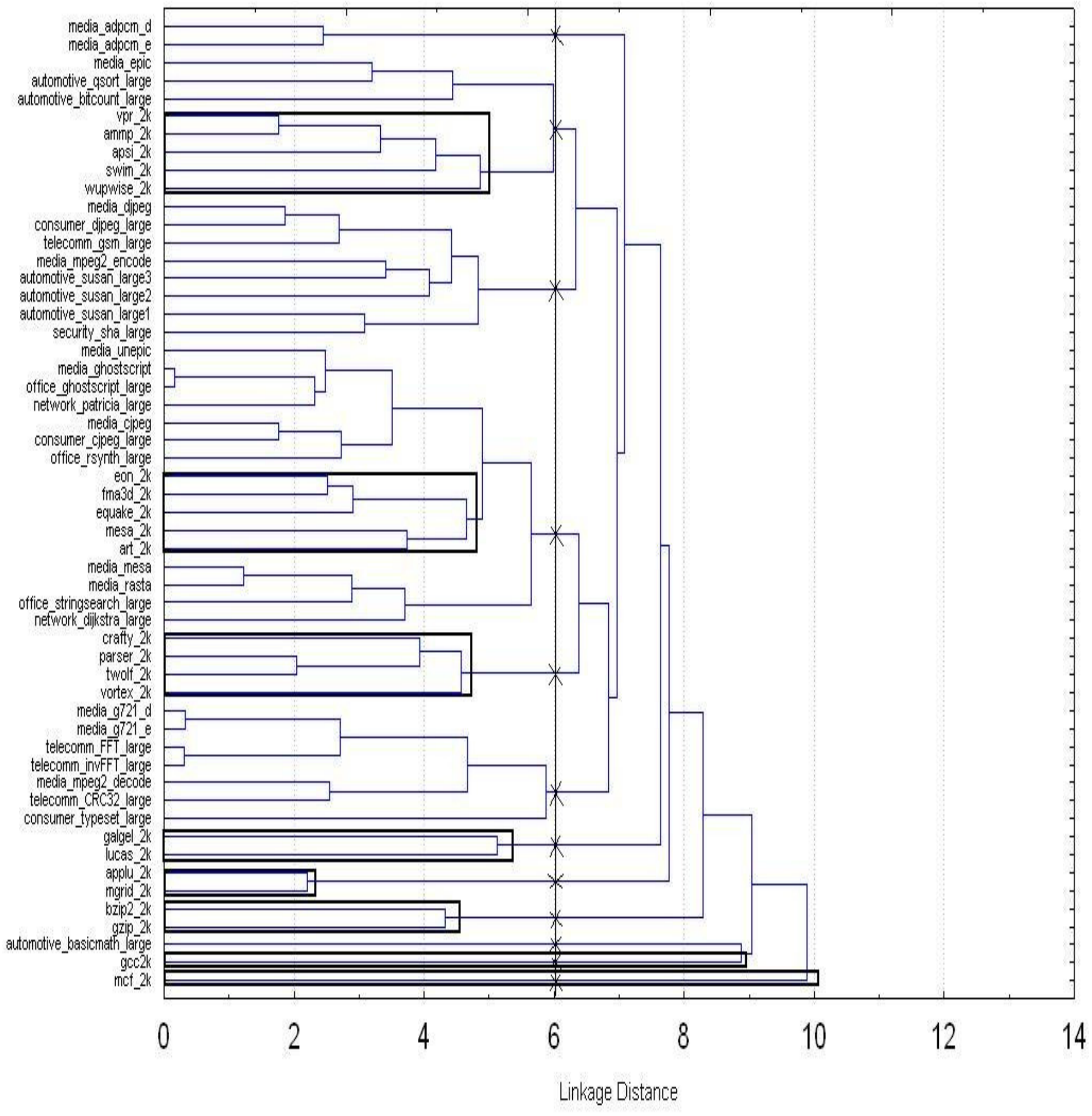
Programs like gzip, bzip2 and mcf show poor spatial locality. There are a lot of programs in all the four generations of SPEC CPU suites that overlap. This indicates that although the objective of SPEC is to worsen the data stream locality behavior of subsequent CPU suites, several benchmarks in recent suites exhibit a locality behavior that is similar to older suites of SPEC CPU benchmarks. Moreover, several CPU95 benchmarks like wave, perl, compress, apsi and CPU2000 benchmarks like equake, galgel, lucas and swim that show a temporal locality behavior that is better than some CPU89 and CPU92 benchmarks.

3.2 COMPARING SPEC CPU AND MEDIA BENCHMARKS

Using the microarchitecture independent metrics described earlier in this dissertation an experiment is done to compare the SPEC CPU2000 benchmarks with the media benchmarks. These two benchmark suites are developed from different application domains. SPEC CPU2000 benchmark suite is developed for evaluating workstation and server class computer systems. On the other hand media benchmarks are used to evaluate processors or systems developed for handheld and embedded processor systems. Media benchmarks are also computation intensive programs that essentially implement telecommunication algorithms. These programs are also scientific application. The difference between the SPEC CPU benchmarks and the media benchmarks is that media benchmarks may have real time input in many cases. Also a lot of compression algorithms where images are compressed and decompressed are used as benchmarks. It will be interesting to see how the scientific applications or benchmarks from a different application domain i.e. class of embedded systems compare with the workstation or server class benchmarks.

The methodology of experiment is similar to the one in the previous section except that all the microarchitecture independent characteristics for SPEC CPU2000 benchmarks and media benchmarks are put together. Media benchmarks used in the experiment are from the Mibench and Mediabench benchmark suites. PCA followed by hierarchical clustering is done to plot the data in a form of a dendrogram. A dendrogram can visually show the clusters of benchmarks but does not classify the benchmarks into fixed clusters. It shows the relative position of benchmarks with all other benchmarks. This experiment can be further extended to find a subset of programs from the mix of media and SPEC CPU2000 benchmarks. The dendrogram can also be used to select benchmarks for simulation.

Figure 3.9: Dendrogram to compare SPEC CPU2000 and media benchmarks using microarchitecture independent metrics



The dendrogram in Figure 3.9 shows all the programs on the vertical axis and the linkage distance on the horizontal axis. The program names can be explained as follows. All the programs with suffix '2k' are SPEC CPU2000 benchmarks. All the programs with a suffix 'media_' are from Mediabench benchmark suite and the rest of the programs are from MiBench suite with a suffix that shows the application domain of the program e.g. ‘automotive_’. Each line originating from a program name unites with another line at a certain linkage distance. The shorter the linkage distance, closer are the programs to each other. As shown in Figure 3.9, the program mcf from the SPEC CPU2000 suite is the farthest from all the rest of the programs. A box formed by solid black lines indicates the presence of the SPEC CPU2000 programs on the dendrogram.

It is very evident that the SPEC CPU2000 programs form small clusters that are well spread out in the workload space covered by the two suites. The SPEC CPU2000 programs vpr, ammp, apsi, swim, wupwise can form a closely bound cluster. Similarly eon, fma3d, equake, mesa, and art can form a tight cluster. The rest of the SPEC CPU 2000 programs that are shown at the bottom of the dendrogram are far away from each other and from the remaining programs in the workload space. The media programs also form small tight clusters except the basicmath program from the automotive application domain of MiBench suite. This program shows a very large linkage distance with other media and SPEC CPU2000 programs and also lies at the bottom of the dendrogram. Figure 3.9 also demonstrates how to pick a dozen programs from a set of media and SPEC CPU2000 programs. A vertical line is drawn at linkage distance of 6. The two benchmarks linked to the line on the left side of the first cross, starting from top, are media programs and can be considered to form a cluster. So, one program is picked from the cluster (media_adpcm_d and media_adpcm_e). At each intersection one program is picked. At the bottom of the dendrogram, there are three singleton clusters of automotive_basicmath_large, gcc_2k and mcf_2k. The second and fourth cluster from top, contain both media and SPEC CPU2000 programs. Either a media or a SPEC CPU2000 program can be chosen from each of the clusters. If all the twelve clusters are examined and one representative is chosen from each cluster, there are at least four media programs in the subset. If media programs are picked as a representative of both the second and fourth cluster then there are 6 out 12 media programs in a subset.

3.3 WORKLOAD SPACE ANALYSIS OF DACAPO BENCHMARKS

In the past researchers have developed benchmark suites for a particular application domain. Recently, the DaCapo benchmarks [6] were developed as a collection of applications which are more representative to the real java applications and their complexity. With the release of a new benchmark suite, it becomes inevitable to compare its characteristics and behavior with the existing java applications. As a part of the analysis the author of this dissertation worked with the developers of the DaCapo benchmark suite to compare the workload space of the new DaCapo benchmarks with the SPEC jvm98 benchmarks. There were different characteristics that were used to compare the two suites. The details of the analysis (PCA) and the characteristics are well described in [6]. This shows that the methodology of workload space analysis can be very useful to validate and compare the workload space coverage of the new benchmark suite.

3.4 SUMMARY

For general purpose processors the design space is quite large which means that the target applications come from many different domains. This means that different benchmark suites should be compared and a representative set of benchmarks from multiple suites should be used for performance analysis. The focus of this chapter is to show how the program similarity analysis can be used to compare different benchmark suites and evaluate their coverage of workload space.

Benchmark suites like the SPEC CPU evolve over time. An experiment is designed to study the evolution of the first four generations of the SPEC CPU benchmark suites. Microarchitecture independent metrics are used to measure the similarity between programs and compare the different suites. The experiment looks at each of the different characteristics separately and studies how the four suites compare for each one. All the benchmarks from the four suites are plotted on a scatter plot and a dendrogram.

The SPEC CPU2000 and media benchmarks are also compared using a dendrogram for the all the microarchitecture independent characteristics together. This experiment demonstrates how benchmarks from different domains can be compared and a subset of benchmarks from both the suites can be obtained for performance evaluation.

Chapter 4: Fast subsetting using performance monitoring counters

The approach used for measuring program similarity in this chapter is different from the one used in Chapter 2. In this chapter microarchitecture dependent metrics are used to measure program similarity. As mentioned earlier the disadvantage of using microarchitecture dependent metrics is that the results are not applicable to a wide range of microarchitecture configurations. But it takes longer to measure microarchitecture independent metrics as compared to microarchitecture dependent metrics. The approach of using microarchitecture dependent metrics was also used as one of the criteria in selecting benchmarks to form the SPEC CPU2006 suite [23]. The time window for measurement and analysis was very small since the changes are made to the benchmarks very often. To satisfy both the needs described above, the benchmarks are characterized using microarchitecture dependent metrics on five different machines with four different ISAs. The five machines have different microarchitecture and use different compilers. The microarchitecture independent approach is better because it captures inherent behavior of programs. The microarchitecture dependent approach may not capture the possible behavior of programs on systems dissimilar to the ones used in the experiment.

Another objective of this chapter is to demonstrate different analysis techniques that can be used to study a new benchmark suite. The rest of the chapter is aligned as follows. Section 4.1 gives a brief introduction to the SPEC CPU2006 benchmarks. Section 4.2 describes the results of some basic analysis of instruction locality at subroutine level. Section 4.3 describes the subsetting experiment and section 4.4 includes a discussion on the balance of CPU2006 suite.

4.1 INTRODUCTION TO SPEC CPU2006 BENCHMARKS

SPEC, since its formation in 1988, has served a long way in developing and distributing technically credible, portable, real-world application-based benchmarks for computer vendors, designers, architects, and consumers. The SPEC CPU benchmark suite, which was first released in 1989 as a collection of ten computation-intensive benchmark programs, is now in its fifth generation and has grown to 29 programs. In order to keep pace with the technological advancements, compiler improvements, and emerging workloads, in each generation of SPEC benchmarks, new programs are added, programs susceptible to easy compiler optimizations are retired, program run times are increased, and memory access intensity of programs is increased [13][25]. The SPEC CPU2006 suite, like its predecessors is divided into two parts: the integer benchmarks called CINT2006 and the floating point benchmarks called CFP2006 benchmarks. The integer group consists of 12 programs and the floating point group consists of 17 programs which stress the CPU, memory and effectiveness of the compiler. Table 4.1 and Table 4.2 show the dynamic instruction count and instruction mix for integer and floating point benchmarks respectively.

Table 4.1: Dynamic instruction count and I-mix for integer benchmarks in CPU2006

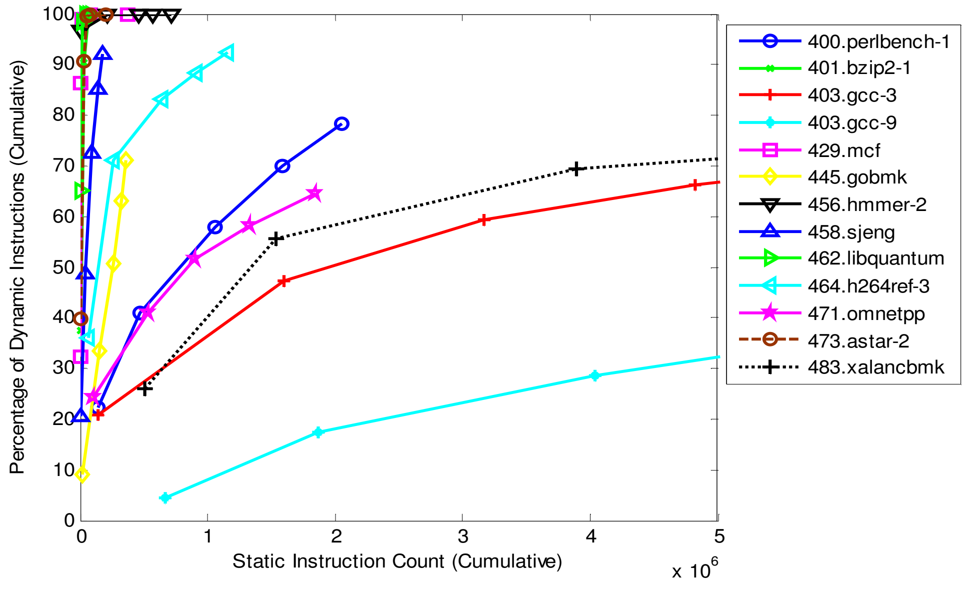
Name – Language	Instruction Count (billions)	%Branches	%Loads	%Stores
400.perlbench –C	2,378	20.96%	27.99%	16.45%
401.bzip2 – C	2,472	15.97%	36.93%	12.98%
403.gcc – C	1,064	21.96%	26.52%	16.01%
429.mcf –C	327	21.17%	37.99%	10.55%
445.gobmk –C	1,603	19.51%	29.72%	15.25%
456.hmmer –C	3,363	7.08%	47.36%	17.68%
458.sjeng –C	2,383	21.38%	27.60%	14.61%
462.libquantum-C	3,555	14.80%	33.57%	10.72%
464.h264ref- C	3,731	7.24%	41.76%	13.14%
471.omnetpp- C++	687	20.33%	34.71%	20.18%
473.astar- C++	1,200	15.57%	40.34%	13.75%
483.xalancbmk- C++	1,184	25.84%	33.96%	10.31%

Table 4.2: Dynamic instruction count and I-mix for floating point benchmarks in CPU2006

Name – Language	Instruction Count (billions)	%Branches	%Loads	%Stores
410.bwaves – Fortran	1,178	0.68%	56.14%	8.08%
416.gamess – Fortran	5,189	7.45%	45.87%	12.98%
433.milc – C	937	1.51%	40.15%	11.79%
434.zeusmp - C, Fortran	1,566	4.05%	36.22%	11.98%
435.gromacs- C, Fortran	1,958	3.14%	37.35%	17.31%
436.cactusADM- C, Fortran	1,376	0.22%	52.62%	13.49%
437.leslie3d – Fortran	1,213	3.06%	52.30%	9.83%
444.namd - C++	2,483	4.28%	35.43%	8.83%
447.dealII - C++	2,323	15.99%	42.57%	13.41%
450.soplex - C++	703	16.07%	39.05%	7.74%
453.povray - C++	940	13.23%	35.44%	16.11%
454.calculix - C, Fortran	3,041	4.11%	40.14%	9.95%
459.GemsFDTD - Fortran	1,420	2.40%	54.16%	9.67%
465.tonto – Fortran	2,932	4.79%	44.76%	12.84%
470.lbm – C	1,500	0.79%	38.16%	11.53%
481.wrf – C, Fortran	1,684	5.19%	49.70%	9.42%
482.sphinx3 – C	2,472	9.95%	35.07%	5.58%

A description of each program has been neatly described by Henning [26] and a summary of changes observed in CPU2006 from the CPU2000 suite has been concisely presented by McGhan [42]. The instruction mix and dynamic characteristics are measured using performance counters on a Pentium D system running SUSE Linux 10.1 and the benchmarks are compiled using Intel C/C++, Fortran compiler V9.1. The basic measurements are collected using the PAPI [14] tool set. The dynamic instruction count of 24 out of the 29 programs is in the trillions which indicate the overall increase in the length of the programs. The instruction mix points to several interesting observations: the percentage of branches in the dynamic instruction mix is close to the typical 20% in most of the integer programs; however, two programs, hmmer and h264ref have only 7% branches. One out of every 4 instructions is a branch in xalancbmk, which is one of the C++ programs in the integer suite. The other two C++ programs (omnetapp and astar) show typical branch frequencies of 20% and 15% respectively. Amongst FP programs, deall, soplex and povray have approximately 15% branches whereas most of the other FP programs have less than 5% branches. There are a few programs where the number of instructions per branch is higher than 100 (bwaves, lbm, cactusADM). The large dynamic basic block size in these programs will allow parallelism to be exploited by machines without being interrupted by branches.

Figure 4.1: Code reuse in SPEC CPU2006 integer programs by profiling top 20 hot functions

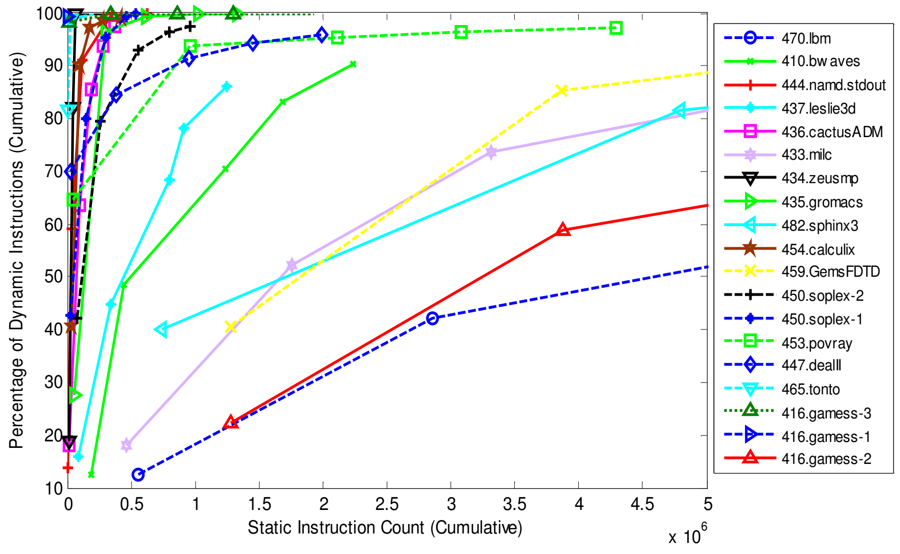


4.2 INSTRUCTION LOCALITY BASED ON SUBROUTINE PROFILING

Programs exhibit locality in instruction access. Subroutine profiling is done in order to understand the code locality in the CPU2006 programs using the PIN dynamic instrumentation tool [39]. PIN can identify hot subroutines based on subroutine call frequency. It can also count the number of dynamic/static instructions in them. Figure 4.1 and Figure 4.2 show the locality plots for integer and floating point workloads respectively. Not all benchmark input sets are plotted, but at least one for each benchmark can be seen. The cumulative percentage of dynamic instructions executed by a program is shown on Y-axis and the count of static instructions is shown on the X-axis.

Many plots climb up very steeply and hence show a very high ratio for the dynamic to static instruction count i.e. high reuse and very good instruction locality. The charts are based on the hottest 20 subroutines, which cover 80% or more of the dynamic instructions in most of the programs. Benchmark 456.hmmer shows a very high reuse of code in the hottest subroutine. More than 95% of the instructions come from the hottest subroutine which has 11080 static instructions.

Figure 4.2: Code reuse in SPEC CPU2006 floating-point programs by profiling top 20 hot functions



Similarly 436.CactusADM and 470.lbm show a very high code–reuse and hence good instruction locality. From Figure 4.1 403.gcc and 471.omnetpp show comparatively very low percentage of dynamic instruction executed in the top 20 hot functions as they climb up slowly. Even 483.xalancbmk shows a slower climb in Figure 4.1 and shows poor code reuse compared to the other workloads. This is a coarse metric of locality since all static instructions from an entire subroutine are counted on the x-axis. SPEC's effort to create applications with large foot print and low locality can be seen in some programs where 5 million static instructions only cover less than 50% of the dynamic instructions.

4.3 SUBSETTING FOR SPEC CPU2006 BENCHMARKS

SPEC CPU benchmark suites contain real world applications chosen from a diverse set of application areas. Preserving the original algorithms and realistic data input sets render a great sense of realism for the suite. However, run times have been very high. Since clock frequencies and cache sizes of machines increase every year, SPEC has increased the benchmark run times significantly to ensure that the benchmarks run for a reasonable amount of time to make meaningful measurements by vendors. However for architectural simulation studies, simulating every benchmark with every provided input set results in enormous amounts of simulation time and limits design space exploration. If same amount of information can be obtained from a smaller subset, it is certainly worthwhile, for researchers/designers in early design tradeoff analysis stages.

The new SPEC CPU2006 benchmark suite is analyzed to find a subset of representative benchmarks. The approach can be described as follows: Programs are characterized using performance counters on five different state-of-the-art machines with 4 different ISAs (IBM Power, Sun UltraSPARC, Itanium and x86) with varying microarchitecture, varying degrees of out of ordering, varying amounts of caches and differing cache hierarchy structures. Since the different characteristics are measured on different machines, each of them forms a characteristic of a program. If there are n machines and m metrics for each machine, each program has n x m characteristics. This dataset is then pre-processed using PCA and then clustered using these n x m characteristics of all programs. It is likely that some of these characteristics are correlated (for instance, consider that 2 machines have very similar microarchitecture features). This correlation will be removed by the PCA process. A concern is that one may accidentally include a characteristic with a large variance, but small impact on performance. In order to avoid that, a correlation analysis was performed between CPI and the characteristic, and characteristics with more correlation to performance were chosen. Table 4.3 shows the list of characteristics that are measured for each program on five different machines. Note that the important characteristics that affect performance for the integer and floating-point programs are different.

Table 4.3: List of characteristics measured for SPEC CPU2006 benchmarks

Integer benchmarks	**Floating point benchmarks**
Integer operations per instruction	Floating point operations per instruction
L1 instruction cache misses per instruction	Memory references per instruction
Number of branches per instruction	L2 data cache misses per instruction
Number of mispredicted branches per instruction	L2 data cache misses per L2 accesses
L2 data cache misses per instruction	Data TLB misses per instruction
Instruction TLB misses per instruction	L1 data cache misses per instruction

Figure 4.3: Dendrogram for CPU2006 integer benchmarks

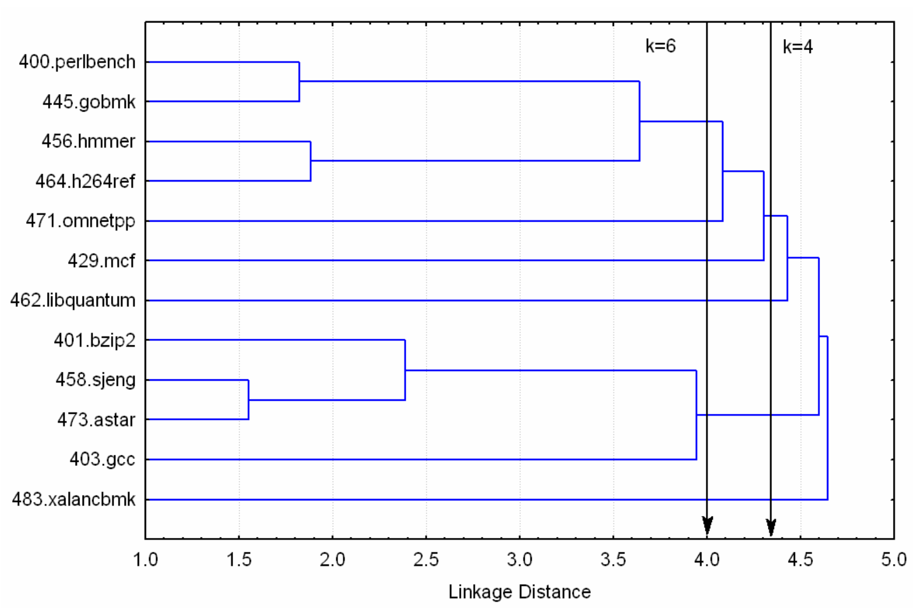


Table 4.4: Subsets for integer benchmarks from CPU2006

k=4	400.perlbench, 462.libquantum,473.astar,483.xalancbmk
k=6	400.perlbench, 471.omnetpp, 429.mcf, 462.libquantum, 473.astar, 483.xalancbmk

Figure 4.3 shows a dendrogram for CINT2006 benchmarks obtained after applying PCA and Hierarchical Clustering on the characteristics from Table 4.3. The Euclidean distance between the benchmarks is used as a measure of dissimilarity and single-linkage distance is computed to create a dendrogram. Seven Principal Components (PCs) are chosen which retain 94% of the variance. In the dendrogram in Figure 4.3 the horizontal axis shows the linkage distance indicating the dissimilarity between the benchmarks. The ordering on the y-axis does not have particular significance, except that benchmarks are positioned close to each other when the distance is smaller. Benchmarks that are outliers can be seen to have larger linkage distances with the rest of the clusters formed in a hierarchical way. If a researcher chooses to pick six benchmarks, then drawing a vertical line at linkage distance of four as shown in Figure 4.3 will give a subset of six benchmarks(k=6). Drawing a line at a point little less than 4.5 yields a subset of four (k=4). Table 4.4 shows the subsets. In clusters where there are more than two programs, the representative of cluster i.e. the benchmark closest to the center of the cluster is chosen as a representative. As the line moves from right to the left on the dendrogram the number of benchmarks in a subset goes on increasing. This helps the user to pick appropriate benchmarks when the time to simulate benchmarks is limited. The subsets formed are validated in the next section.

Figure 4.4 shows the dendrogram for floating point benchmarks in CPU 2006. Five PCs are chosen using the Kaiser criterion which retains 85% of the variance. The two vertical arrows show the points at which the subsets are formed. The resulting clusters are shown in Table 4.5. The distance of each of the benchmarks in the cluster to the center has to be recalculated and a representative can be chosen. In Figure 4.4 there are two main clusters which split at extreme right because the branch characteristics of the benchmarks. 447.dealII, 450. soplex and, 453.povray exhibit a comparatively higher branch misprediction rate.

Figure 4.4: Dendrogram for CPU2006 floating point benchmarks

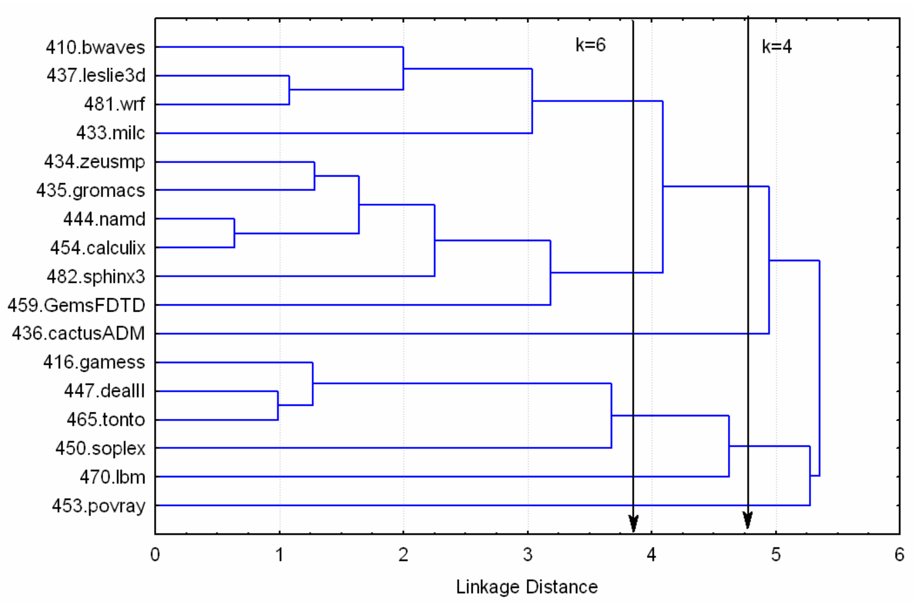


Table 4.5: Subsets for floating point benchmarks from CPU2006

K=4	482.sphinx3, 436.cactusADM, 447.dealII, 453.povray
K=6	437.leslie3d, 454.calculix, 436.cactusADM, 447.dealII, 470.lbm, 453.povray

Note that clustering and subsetting gives importance to unique features and differences. It helps to eliminate redundancy and reduce efforts in experimentation; however, one should not mistake the mix of program types in a subset as the mix of program types in the real-world. The mix of programs in the real-world may contain more normal cases as opposed to challenging corner cases which get emphasized in benchmark suites.

4.3.1 Validation of SPEC CPU2006 subsets

Validation of the subsets is based on actual performance scores of carefully chosen five commercial machines. The SPEC website [59] contains several CPU 2006 submissions from major commercial computer vendors. The execution times for each platform and baseline execution times on the reference machine can be obtained from the SPEC site for each benchmark program. The average speedup obtained based on the subset is compared against the average speedup from the entire component (CINT or CFP) of the suite. In accordance with SPEC practices, geometric mean is used to find the average.

Figure 4.5: Validation of the subset of integer benchmarks using 5 systems from SPEC website

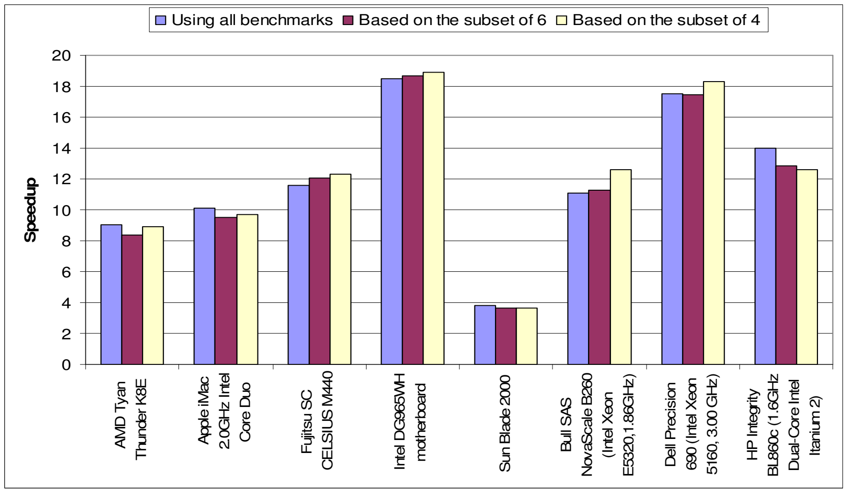
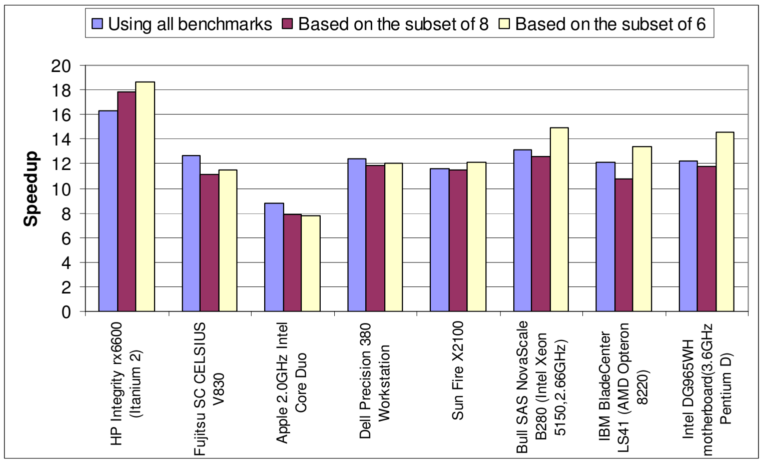


Figure 4.5 shows the comparison of both the subsets of CINT2006 benchmarks from Table 4.4. For CINT component the subset of 4 programs shows an average error of 5.8% and a maximum error of 10.1%. The subset of 6 benchmarks shows an average

error of 3.8% and a maximum error of 8%. This shows that even a subset of 4 benchmarks out of 12 CINT benchmarks has a very good predictive power in estimating the speedup shown by the entire suite.

Figure 4.6 shows the validation of CFP2006 benchmarks using both the subsets from Table 4.5. The subset is seen to predict the speedup very closely for the integer suite. The error in the floating point subset is higher than that in the integer; however, there is no change in ranking considering these machines. For a subset of 6 the average error is 10.8% with the maximum error of 19%. Hence we look at a subset of 8 benchmarks which shows the average error of 7% and the maximum error is 12%.

Figure 4.6: Validation of the subset of floating point benchmarks using 5 systems from SPEC website

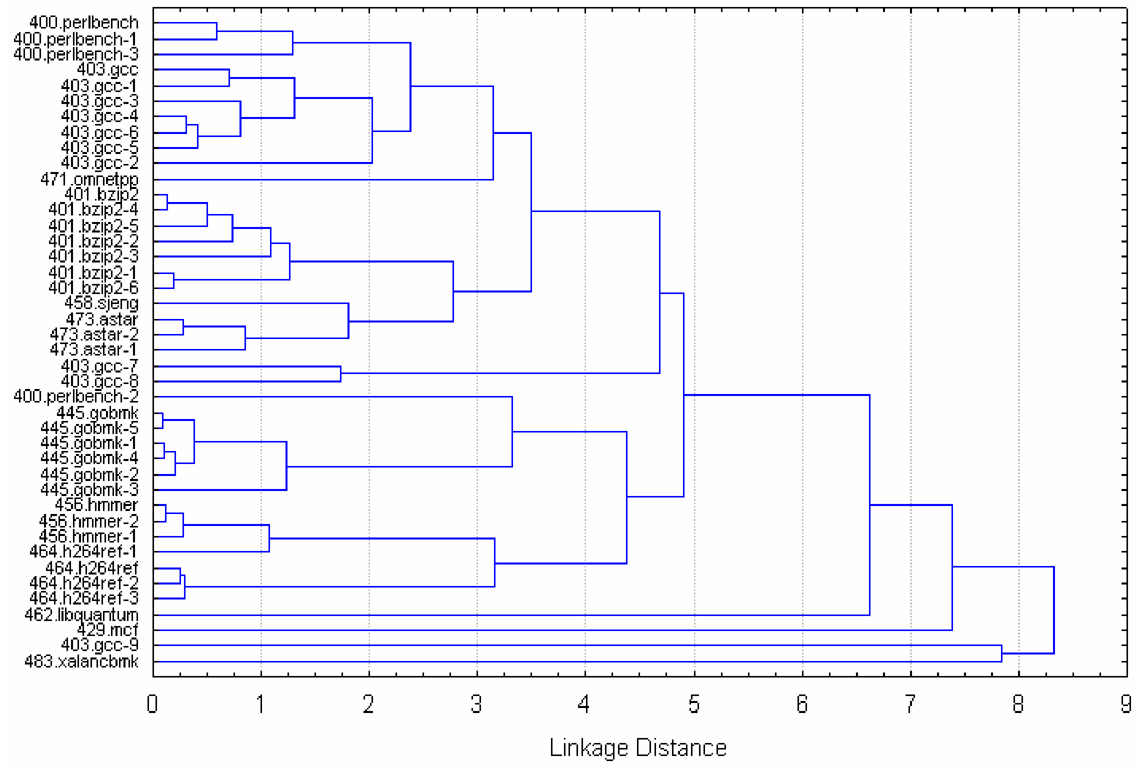


4.3.2 Selecting representative input set

Many benchmarks in the CPU2006 have multiple input sets. Hence forth in this sub-section a program-input pair will be referred to as workload and all the workloads of a program run together as a benchmark. A reportable SPEC run uses all the workloads for each benchmark; however it is possible to use PCA and clustering to identify a representative input set, helping architecture researchers to reduce simulation time and effort. In SPEC CPU2006 403.gcc benchmark has nine input sets. The program characteristics shown in Table 2.9 are measured for all the different workloads and for the benchmark, which are used in the analysis. Whenever data is reported for a benchmark, it is the aggregate behavior summing up all its input sets.

Figure 4.7 shows the dendrogram for input sets and the benchmarks for the integer component. Seven PCs covering 89% of variance are chosen for this analysis. Some benchmarks have only one input set and hence represented only by their name. In some benchmarks, all input sets appear clustered together, whereas in many cases, some input sets are very different from the other input sets of the same benchmark. As an example, the behavior of 403.gcc-9 is significantly different from its siblings.

Figure 4.7: Dendrogram of CPU2006 integer benchmarks with their input sets plotted separately



A benchmark's input set closest to the complete (aggregated run over all inputs one after another) run is marked as the representative input set. In CINT2006, the benchmarks that have multiple input sets are 400.perlbench, 401.bzip2, 403.gcc, 445.gobmk, 456.hmmer, 464.h264ref, 473.astar. For each of these benchmarks a representative input set is listed in Table 4.6. Figure 4.8 shows the dendrogram for CFP2006 benchmarks. Six PCs covering 88% of variance are chosen. In this category there are only two benchmarks with multiple input sets. i.e. 416.gamess and 450.soplex.

Figure 4.8: Dendrogram of CPU2006 floating point benchmarks with their input sets plotted separately

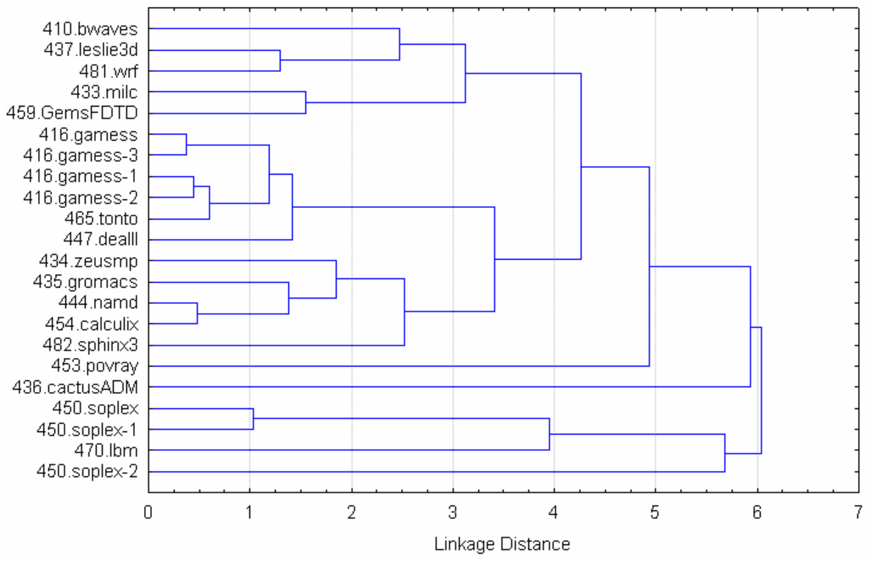


Table 4.6: List of representative input set for CPU2006 benchmarks with multiple inputs

CINT2006 benchmarks	464.h264avc - input set 2
400.perlbench - input set 1	473.astar - input set 2
401.bzip2 - input set 4	
403.gcc - input set 1	**CFP2006 benchmarks**
445.gobmk - input set 5	416.gamess - input set 3
456.hmmer - input set 2	450.soplex - input set 1

Table 4.7: Application areas with multiple programs in the CPU2006 benchmark suite

Application area	**Benchmarks**
Artificial Intelligence	**458.sjeng**, 445.gobmk,**473.astar**
Equation solver	436.cactusADM, 459.GemsFDTD
Fluid Dynamics	**410.bwaves**, 434.zeusmp, **437.leslie3D**, 470.lbm
Molecular Dynamics	**435.gromacs, 444.namd**
Quantum Chemistry	**465.tonto, 416.gamess**
Engineering and Operational Research	454.calculix, 447.dealII, 450.soplex, 453.povray

4.4 BALANCE IN THE BENCHMARK SUITE

Table 4.7show a list of the application areas and the integer benchmarks associated with each of them. There are multiple programs from certain application areas, e.g. in the integer suite, there are 3 programs (458.sjeng, 445.gobmk, 473.astar) from the artificial intelligence area, and in the floating point suite, there are 4 programs (410.bwaves, 434. zeusmp, 437.leslie3d, 470.lbm) from the fluid dynamics area, but none of the benchmarks are from the Electronic Design Automation (EDA) application area. The earlier SPEC suites contained EDA applications (vpr, twolf, espresso, eqntott). The goal of this experiment is to see if losing the applications from EDA domain is a weakness of CPU 2006 or do some other programs included in the suite cover the workload space where the EDA programs are positioned. Also it is interesting to see if the multiple programs from one area included in the suite have sufficient unique behavior to warrant their inclusion. This section shows how the process of measuring program similarity can be used to check the balance in the benchmark suite. Analysis of similarity between benchmarks will help in answering the concerns about EDA applications mentioned above. The program characteristics shown in Table 4.3 are measured for all the SPEC CPU2000 and SPEC CPU2006 integer programs and projected in the workload space after applying PCA.

Figure 4.9: Scatter plot of PC2 Vs PC1 to show the position of EDA applications in the workload space

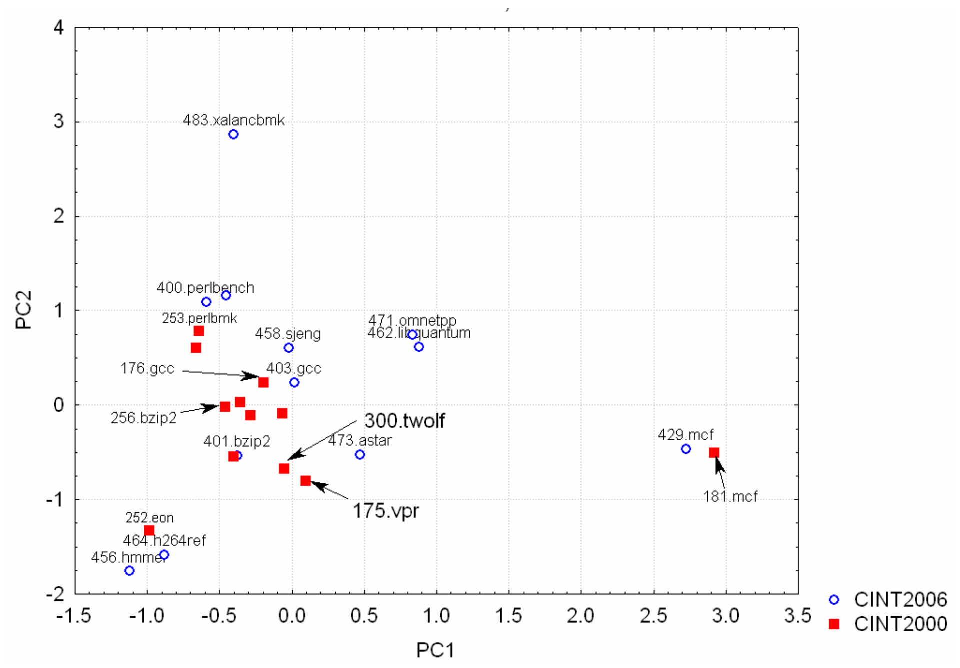
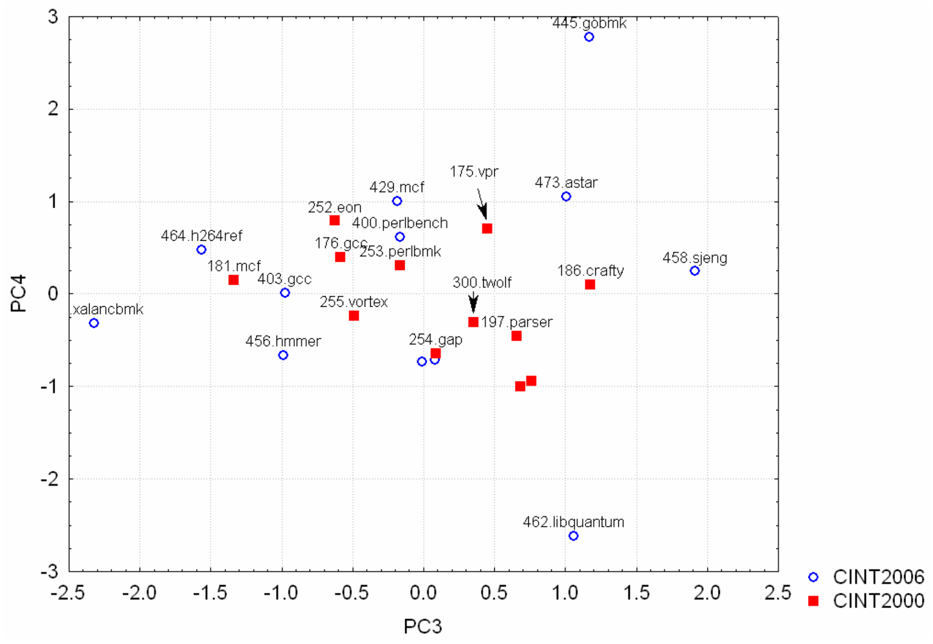


Figure 4.10: Scatter plot of PC4 Vs PC3 to show the position of EDA applications in the workload space



The benchmarks 175.vpr and 300.twolf in particular are the EDA applications used as benchmarks in CPU2000 suite. Figure 4.9 and Figure 4.10 show the projections of the workload space on the first four PCs. From these figures it is evident that the EDA tool benchmarks from CPU2000 i.e. 175.vpr and 300.twolf lie close to 473.astar from CPU2006 in both the projections and close to 401.bzip2 from the CPU2006 benchmark in Figure 4.9. Also the EDA tools are surrounded by other benchmarks which does not make them unique This shows that the EDA tools that were commonly used 5 to 6 years back, do not show very different behavior from some recent benchmarks which are from a different application area. Since EDA tool industry is evolving very fast with new features and capabilities added to the tool frequently, more recent EDA applications may show very different behavior. In summary, the behavior of older EDA application area benchmarks show similar performance behavior as some of the recent SPEC CPU2006 benchmarks but the latest EDA applications need to be studied and compared with the benchmarks in the new suite.

Any two benchmarks that belong to the same application area can show different behavior on certain architecture. The similarity analysis described in the previous subsection for subsetting can be used to compare the benchmarks from Table 4.7 from the same application area. Consider one application area from Table 4.7 at a time and then go back to Figure 4.3 for integer programs and Figure 4.4 for floating point benchmarks to observe similarity within a given application area. In case of artificial intelligence applications, 458.sjeng and 473.astar show very similar behavior and can be found quite close to each other in the workload space, while 445.gobmk is much further away from its siblings. The equation solver applications do not lie close to each other and hence justify the presence of both the benchmarks in the suite. 410.bwaves and 437.leslied, are relatively close to each other than the other two programs in their application area. Both the programs in molecular dynamics are different and relatively closer to each other with the linkage distance of less than 2 between them. 465.tonto and 416.gamess also have a linkage distance of less than 2. The last application area in Table 4.7 which has applications much spread out in the workload space compared to the others and have four programs which are significantly different from each other. To summarize this study, there are differences between programs that affect performance; however, if elimination of similar programs is desired by a user based on the application area, the programs marked in bold in Table 4.7 show highest redundancy (similarity to other existing programs in the same domain).

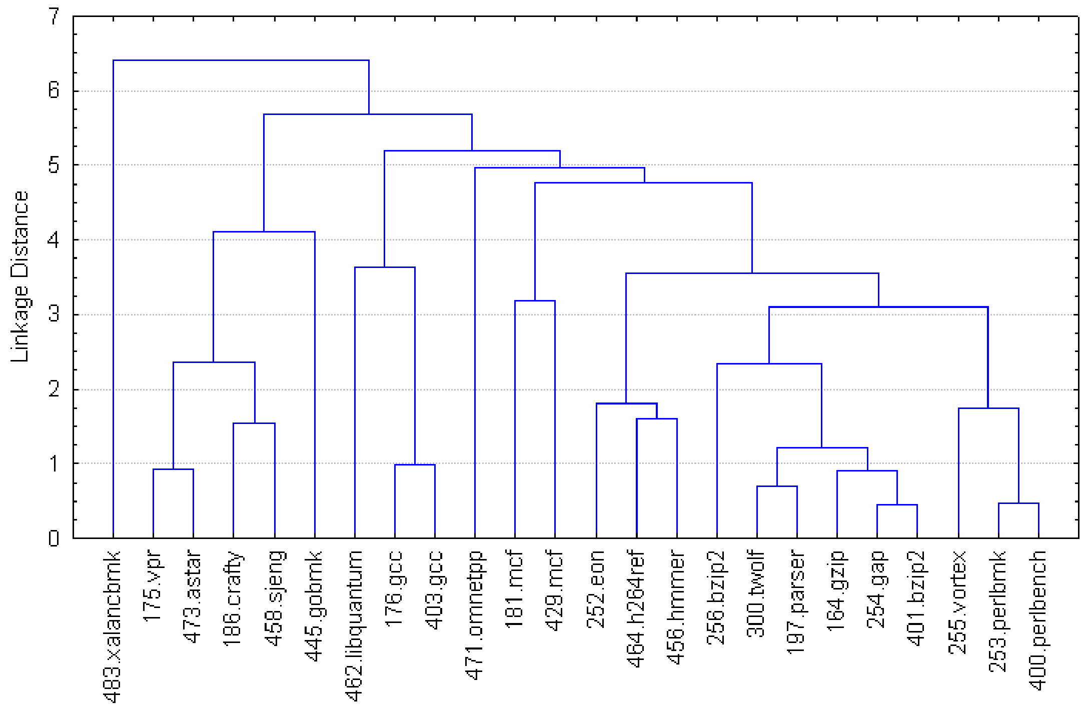
4.5 COMPARISON OF CPU2006 WITH THE CPU2000 BENCHMARK SUITE

The SPEC CPU2006 benchmark suite is the latest in the five generations of CPU benchmarks after the SPEC CPU2000 benchmarks. It is important to see how the benchmark suites compare with the benchmarks from its previous generation. The following experiment is done to compare the two suites. Microarchitecture dependent metrics listed in Table 4.3 are used to characterize the benchmarks. The microarchitecture dependent characteristics that are chosen for the experiment are measured on five different computer systems with different ISAs and compiled using state of the art compilers. The workload space is built using the six characteristics on each machine (30 characteristics in total).

4.5.1 Analysis of integer benchmarks

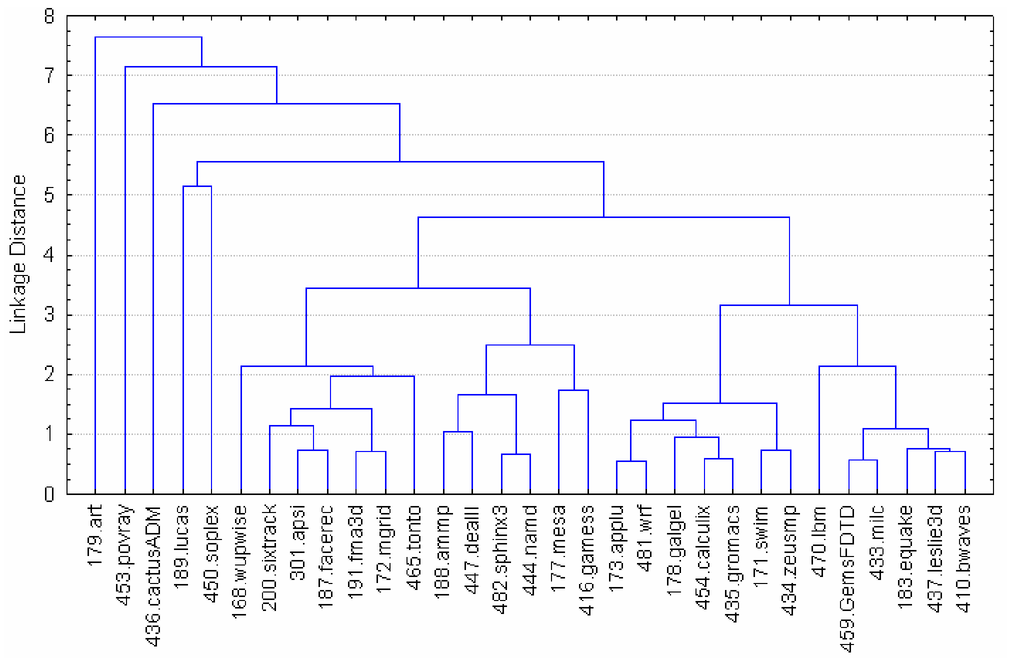
Figure 4.11 shows a dendrogram plotted using the microarchitecture dependent metrics for integer programs. The benchmarks with a suffix number starting with '4' are all the CPU2006 benchmarks. The rest are CPU2000 benchmarks e.g. 429.mcf belongs to the SPEC CPU2006 suite while 181.mcf belongs to CPU2000 suite. There are a few programs that are common between the two suites. Mcf, perl and gcc that are present in both the benchmark suites can be seen relatively close to each other in the common workload space. Although mcf benchmarks from the two suites are significantly away from the other benchmarks, they are quite similar to each other. There are a few CPU2006 benchmarks that are quite far away in the workload space from the other benchmarks i.e. xalancbmk, gobmk, libquantum and omnetpp. But xalancbmk is farther in the workload space from all other benchmarks. This does not signify good or poor performance for xalancbmk but just shows that its behavior is very different.

Figure 4.11: Dendrogram to compare integer benchmarks from SPEC CPU2006 and SPEC CPU2000 benchmark suite



The similarity analysis also found that most of the integer benchmarks from CPU2000 suite that were carried forward to CPU2006 suite e.g. mcf, gcc and perl are significantly similar. But the SPEC CPU2006 benchmarks run for a very long time. This raises a question whether the program like mcf and gcc are made to run longer without any significant change in the control flow behavior of the benchmarks. It may also mean that the later versions of these real-world applications do not show significantly different behavior as far as processor and memory performance are concerned but only operate on a bigger set of data. If the control flow behavior is similar and the programs are modified to run for a longer time, using older benchmark for simulation based studies should not be criticized in the research community.

Figure 4.12: Dendrogram to compare floating point benchmarks from SPEC CPU2006 and SPEC CPU2000 benchmark suite



4.5.2 Analysis of floating point benchmarks

Figure 4.12 shows a dendrogram for the floating point benchmarks from both the suites. None of the application programs were carried over from CPU2000 to CPU2006. The same rule that CPU2006 benchmarks star with a suffix of '4' and the rest are all

CPU2000 benchmarks. It is evident that there are mainly three groups of benchmarks which can be seen starting from the right side of the figure. There are five benchmarks on the left, which can be easily classified as outliers in the workload space. The benchmark 179.art is still the farthest one followed by 453.povray. For each floating benchmark in CPU2000 there is at least one CPU2006 benchmark nearby in the workload space. It shows that the overall coverage of floating point CFP2006 benchmarks is more than the CPU2000 floating point benchmarks.

4.6 SUMMARY

Recently, SPEC released a new benchmark suite called SPEC CPU2006. The objective of this chapter is to demonstrate the use of fast subsetting approach. Another objective is to demonstrate the different analysis techniques that can be used to study a newly released benchmark suite. The approach used for measuring program similarity in fast subsetting approach is different from the one used in Chapter 2. In this dissertation, microarchitecture dependent metrics from different computer systems with different ISAs, microarchitecture and compiler are used to measure program similarity. This is a faster way of characterizing benchmarks. The approach of using microarchitecture dependent metrics was also used as one of the criteria in selecting benchmarks to form the SPEC CPU2006 suite. The microarchitecture independent approach is better because it captures inherent behavior of programs.

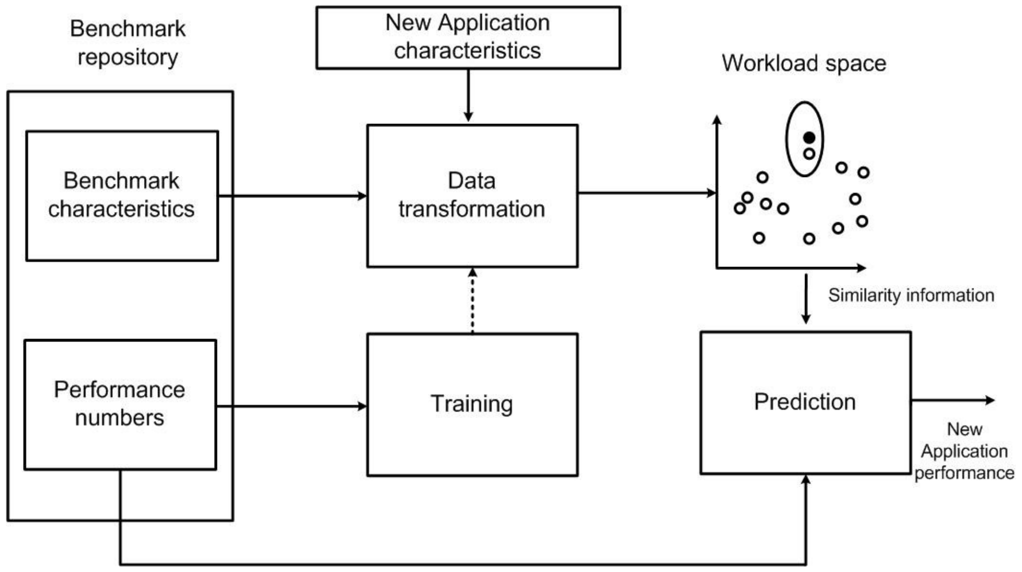
There a two potential issues with the balance of the suite. Some applications may not be represented in the suite and different programs from the same application domain are a part of the benchmark suite. The goal is to take an example and show how the analysis can be done. CPU2006 suite does not have a single application from the EDA (Electronic Design Automation) tool domain. Previous versions of SPEC CPU benchmarks had EDA tool applications in the suite. The experiment measures the similarity and finds if the EDA tool applications from older suite are close to other benchmarks in the workload space from the new suite. The experiment also studies whether the benchmarks from the same application domain are different to warrant their inclusion the suite.

Chapter 5: Performance Prediction Using Program Similarity

Customers who buy computer systems use the benchmark suites to compare different computer systems to make a purchase decision. Although there are other factors like cost which can affect the decision, finding the fastest computer system in the same price range is important for a customer. Many computer system manufacturers use a benchmark suite e.g. the SPEC CPU benchmark suite to report performance scores of their system. One of the scores reported for the SPEC CPU benchmark suite is reported as the average speedup of the system to a standard baseline system chosen by SPEC. This helps to find how fast one computer system is to the other. But the customer's application may not be a part of the benchmark suite and it is difficult to ensure whether a particular system will perform better than the other for the customer's application. Ideally, a customer's application is his best benchmark but numerous difficulties may force the customer to infer from the benchmark scores available from SPEC or TPC. One of these difficulties is porting the application to numerous platforms to measure performance of the application and even if that is possible, it is almost impossible for the customer to run the benchmark on all the different systems available in the market.

In this chapter, a methodology to estimate performance of a workload based on other workloads or benchmarks is presented. The program similarity information between the customer's application and the standard benchmarks is used to predict the performance that is specific to the customer's application. The methodology is presented in Figure 5.1. The benchmark repository is a set of benchmarks whose performance scores are available with the measured microarchitecture independent characteristics. A new application is then mapped into the workload space built using the benchmarks after transforming the characteristics. The position of the application relative to the benchmarks in the workload space is then used to predict performance. The prediction block involves giving more weight to the benchmarks that are similar to the customer's application.

Figure 5.1: Block diagram of methodology used for performance prediction



5.1 METHODOLOGY

This section mainly describes the general methodology of predicting performance of a new application in detail. Figure 5.1 shows a block diagram for the methodology of the technique. As a convention only specific to this chapter, the programs which are well characterized and whose performance is already known are called *benchmarks*. The new application for which the prediction of performance is desired is called the *application*. This infrastructure also makes an assumption that performance prediction happens only for one application at a time.

The benchmarks which form a repository of performance information are shown towards the left in Figure 5.1. The repository contains a number of benchmarks with their characteristics and performance scores. In this dissertation the characteristics are microarchitecture independent metrics which are discussed in Chapter 2 and performance score for a benchmark is its speedup, average Cycles per Instruction (CPI) count or metrics like cache hit-rate. The performance numbers of benchmarks can be in any form but relevant and appropriate benchmark characteristics should be measured. Also, it is important to have characteristics that do not take very long to measure. The application for which the performance needs to be predicted is also characterized with exactly the same microarchitecture independent metrics. The main idea of this technique is to map the application to the workload space of benchmarks and then in the workload space use the similarity information of the application with the other benchmarks from the repository to predict performance using the performance numbers of benchmarks.

The methodology can be divided into two main parts:

- Data transformation and training
- Prediction

These two main parts in the block diagram are described in detail in the subsequent sections. Before the application is mapped into the workload space of benchmarks, they are pre-processed with different techniques. The pre-processing stage is called 'Data Transformation' in Figure 5.1. The different data transformation schemes considered, are described in the next section. This step is similar to choosing the right characteristics to measure similarity. The process of choosing characteristics can be considered as giving weights to different characteristics or removing a certain set of unimportant characteristics. Finding weights or choosing characteristics needs the help of already known performance numbers of the benchmarks. This process is shown in Figure 5.1 with a block called 'Training'. The dotted line is drawn between the 'Training' block and the 'Data Transformation' block to show that it is optional and performance prediction can be done without training. Other techniques are also evaluated where training is not required. The last step involves finding similar benchmark(s) to the application and use the performance numbers to predict the applications performance. This is shown in Figure 5.1 by the block named 'Prediction'. The process of prediction can be done in multiple ways. This part of the prediction process essentially decides which benchmarks to choose to use in the prediction process based on their similarity to the application. One of the approaches is the *k-nearest neighbor* approach which is simple and commonly used for classification. The distance matrix is calculated which has the distance between the application and all the benchmarks from the repository. Then the neighbor(s) of the application and their performance numbers are used to calculate the predicted performance of the application.

5.1.1 Data transformation and training

Since the goal of the technique is to use the already available information to predict performance it is important to find the characteristics that affect performance. Microarchitecture independent metrics that are measured to characterize programs are very broad and it is possible that only a small set of these are needed to accurately predict performance of the application on a certain system. In fact, if the characteristics that do not affect performance as much are used, they will add inaccuracy to the prediction result. A solution to this problem is giving weights to each of the characteristics. Choosing characteristics is equivalent to giving a weight of zero or one.

The four different transformation techniques proposed in this dissertation are:

- Equal Weights (EW)
- Choosing characteristics based on correlation to performance (COR)
- Principal Components Analysis (PCA)
- Genetic Algorithm (GA)

The first technique gives equal weight to all characteristics and transforms each characteristic across benchmarks by normalizing them. The second method uses correlation coefficients of characteristics with performance number to weigh each characteristic. The third method uses Principal Components Analysis (PCA) which removes correlation and transforms the data into a set of uncorrelated variables called Principal Components. Same transformation has been used for subsetting earlier in this dissertation. The fourth method is based on Genetic Algorithm (GA) which finds weight for each characteristic using evolution theory. The genetic algorithm uses the 'Training' block in Figure 5.1 to find weight for each characteristic. Each of these transformation techniques is described in detail in the following sections.

5.1.1.1 Equal Weight (EW)

Normalization is also used in Chapter 2 which transforms each characteristic to so that the mean is zero and standard deviation of one.

$$X_t = \frac{(x - \bar{x})}{\delta}$$

where, X_t is the transformed value of each data point, x is the original value of the data point and $\bar{x}$ is the original mean value of all the data points which in this case would be the mean of a characteristic. Normalization removes the bias caused by the value of a variable. Some characteristics of benchmarks can have a value of the order of thousand while some of them can have the order of ten or hundred. This difference in the range of values can give more weight to the characteristics that have higher range when the distance between two programs is calculated. The distance between two programs is the measure of similarity between them. To remove the bias caused by the range of values of different characteristics, normalization is a basic step. After normalization, all characteristics get equal weight. This is a baseline case and all other techniques will be compared with this technique. All other techniques use the normalized data for transforming the characteristics. This technique is very simple and does not need any performance information in the transformation.

5.1.1.2 Choosing characteristics based on correlation to performance (COR)

The next step in choosing characteristics is to use the normalized data to assign weights. This technique is similar to the one used in Chapter 4 to select characteristics. Correlation coefficient of each characteristic with the performance numbers is calculated. The characteristics that show high correlation are chosen. The rest of the characteristics are ignored in the next step of prediction. The range of correlation coefficients is from -1 to 1. Characteristics that have their correlation coefficients close to 1 or -1 are selected.

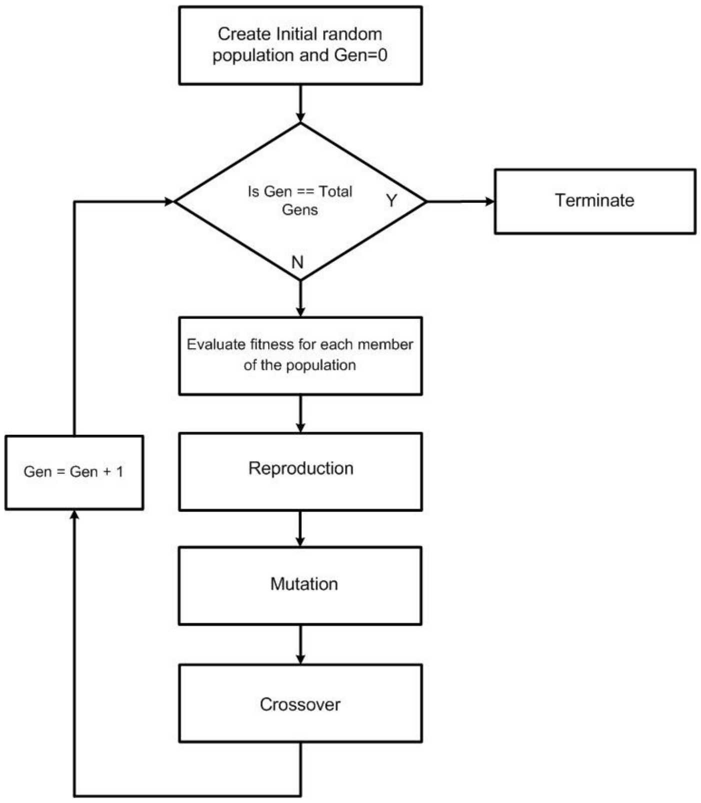
5.1.1.3 Principal Components Analysis (PCA)

PCA is described in detail in Chapter 2. PCA transforms the characteristics into another set of characteristics called the Principal Components (PCs). Then a set of top few PCs are chosen based on the amount of variance they cover and used in the analysis. PCs are a linear combination of all the variables but the top few PCs will give more weight to the characteristics that show higher variance. Thus the transformation using PCA will give more weight to characteristics that show higher variance. It also removes the correlated variables and avoids giving more weight to a feature of the program e.g. data locality which can have multiple characteristics like temporal locality and spatial locality.

5.1.1.4 Genetic Algorithm (GA)

Genetic algorithm is a technique of finding a solution based on the theory of evolution. In this dissertation GA is used to find weight for each characteristic. GA was first described by Holland [27] , and is used to find solutions in engineering and other science field using the natural evolution and selection process. Figure 5.2 shows the flow-chart for a general implementation of genetic algorithm.

Figure 5.2: Flow chart for genetic algorithm used in finding weights for characteristics.


Create Initial random population and Gen=0
Is Gen == Total Gens
Y
N
Terminate
Evaluate fitness for each member of the population
Reproduction
Mutation
Crossover
Gen = Gen + 1

In biological systems, genetic information that determines the individuality of an organism is stored in chromosomes. In this case the chromosomes are the vectors of weights where each element of a vector is a weight for each characteristic. Hence the length of each vector is equal to the number of characteristics. A population of weight vectors are replicated and passed onto the next generation with selection depending on fitness. The weight vectors then go through the phases of reproduction, mutation and crossover. The weight vectors are altered through genetic operations such as mutation and crossover to cover a broader space. Each weight vector forms a candidate solution to the problem. The passage of each vector to the next generation is determined by its relative fitness, i.e. the closeness of its properties to the goal. The fitness function in this case is the accuracy of prediction of performance. Random combinations and/or changes of the transmitted vectors produce variations in the next generation of offspring. The offspring is the derived solution to be used for next generation. The better the fitness (correspondence with desired properties), higher is the chance of being selected for next generation. By going through many generations, optimal or near optimal solutions are obtained.

Figure 5.3: Pseudo-code for fitness function used in the genetic algorithm for performance prediction

```
Input    : Vector of weights
Output   : Average CPI Prediction error (fitness value)

function fitness (weights)

# Transform the train benchmarks' characterisitcs using the weights

for bench = 1 to all_train_benchmarks
    weighted_train_metrics = train_metrics * weights
end for

# Use the leave one out technique and predict performance of the
#  left out benchmarking using the rest

for bench = 1 to all_train_benchmarks
    neighbors = find_k_neighbors(weighted_train_metrics, bench, k)
    predicted_perf = predict(bench, perf_numbers, k)
    prediction_error[bench] = modulus (predicted_perf -
    perf_numbers[bench] )  * 100 / perf_numbers[bench]
end for

error = average(prediction_error)

return error
```

The most important part in the genetic algorithm is the fitness function used to evaluate each candidate solution. The fitness function returns a score which the genetic algorithm uses to classify the potential solutions and decides which ones go through to the next generation. The fitness function is usually very unique to the problem being solved. In this case the fitness function should return the final error in predicting performance. The performance number for each benchmark in the repository shown in Figure 5.1 is known. These performance numbers for all benchmarks will be used with their characteristics from the repository to find the average prediction error which is an indication of fitness.

Figure 5.3 shows the pseudo-code for the fitness function used to predict and evaluate each candidate in GA. The candidate with lowest error is the fittest. The algorithm uses the leave one out technique and calculates the prediction error using K-neighbors technique for each train benchmark. The average of the individual errors is used as the fitness score. Every time a vector of weights is to be evaluated for its fitness, the fitness function is invoked. So the number of times the fitness function gets invoked in one generation is equal to the total population in each generation which is set to 20 in the experiment described in the subsequent section. The input to the fitness function is a vector of weights and the output is the average prediction error. A set of training benchmarks is used to evaluate the fitness. The choice of training benchmarks depends on the process of cross-validation. Leave-one-out cross-validation technique is used for the experiments, so the training benchmarks will be all the benchmarks except the one whose performance is being predicted. The first *for* loop transforms all the characteristics to a set of weighted characteristics. The second *for* loop then goes through each of the training benchmark and uses a leave-one-out method and finds prediction error for all the training benchmarks. Then an average prediction error is calculated for all the training benchmarks using their individual prediction errors. This average number is used as the fitness score for the weights that are fed as input to the fitness function.

Figure 5.4: Progress plot of GA while predicting performance for one application

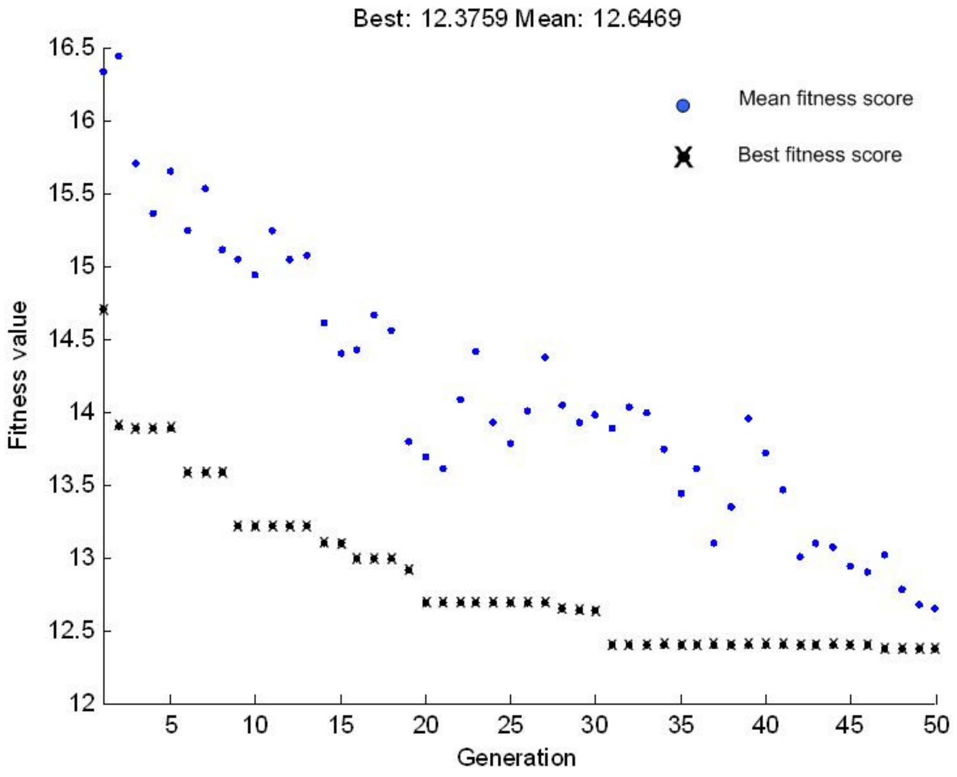


Figure 5.4 shows a progress chart of the GA as a population of weights is evaluated. The algorithm is run for 50 generations with a population of 20. There are two dots plotted at each generation. The darker one shows the fitness function of the best case. The other dot shows the average value of the fitness function over all the 20 members of the population. This plot can be drawn for every application that is predicted. At the end, the last dot at the rightmost bottom corner is the best case and is used to weigh the characteristics and predict performance based on similarity. The mean and best case dots come closer after every generation because the members adapt and move towards the best case.

5.1.2 Performance prediction from the workload space

After the characteristics are transformed, the benchmarks and the application of interest is mapped in the workload space. The workload space is an *n* dimensional space formed using the *n* characteristics. As shown in Figure 5.1 the prediction block analyzes the workload space and predicts the performance of the application. The k-nearest neighbor algorithm is used to do the prediction. In k-nearest neighbor algorithm the value of *k* is the number of neighbors used to do the prediction.

Figure 5.5: Illustration of k-nearest neighbor algorithm

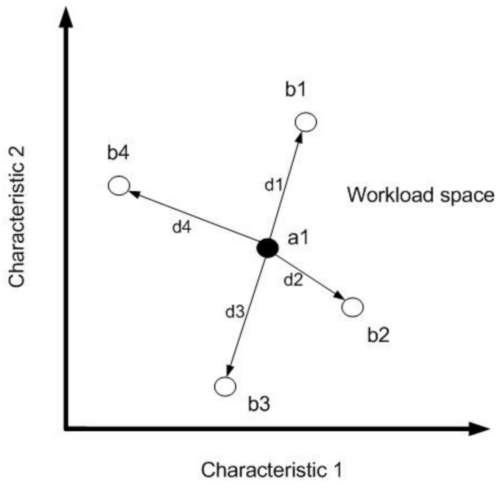


Figure 5.5 shows an illustration of the k-nearest neighbor algorithm. The illustration is only for a two dimensional space but the same algorithm is also applicable for more than two dimensional spaces. In Figure 5.5, the application of interest is shown by a black colored point labeled *a1*. The benchmarks used to predict the performance of *a1* are shown by *b1, b2, b3* and *b4* which are at a distance of *d1, d2, d3* and *d4* respectively. For *k=1,* the performance of the nearest benchmark which in the illustration is *d2* is reported as the predicted performance. In case of $k > 1$ a weight that is inversely proportional to the distance between a benchmark and application can be applied to each

of the neighbors used for prediction. The weight for each benchmark is calculated with the following equation:

$$w_i = r_i / sum(\ r_1,\ r_2,\ r_3,\ r_4\ \ldots r_n\ )$$

where $r_1, r_2, r_3,\ r_4$ are reciprocals of distances *d1, d2, d3*and *d4* respectively. Then a weighted mean is calculated to predict performance of the application *a1*. In this dissertation different values of *k* are evaluated.

5.2 PERFORMANCE PREDICTION EXPERIMENTS

5.2.1 Experiments for predicting machine ranks using speedup

5.2.1.1 Experimental setup for prediction of machine ranks

The experimental setup includes a repository of the SPEC CPU2000 benchmarks used in Chapter 2 with the microarchitecture independent characteristics and the speedup numbers of these benchmarks for ten different real machines reported on the SPEC website. These machines are from different computer vendors. Out of these systems there is at least one system with the x86, Itanium, IBM's POWER, Sun sparc ISA. The aim is to experiment with different computer systems with large variation in their design and configuration. Table 5.1 shows the list of these ten machines.

Table 5.1: List of machines used in the experiment of machine rank prediction

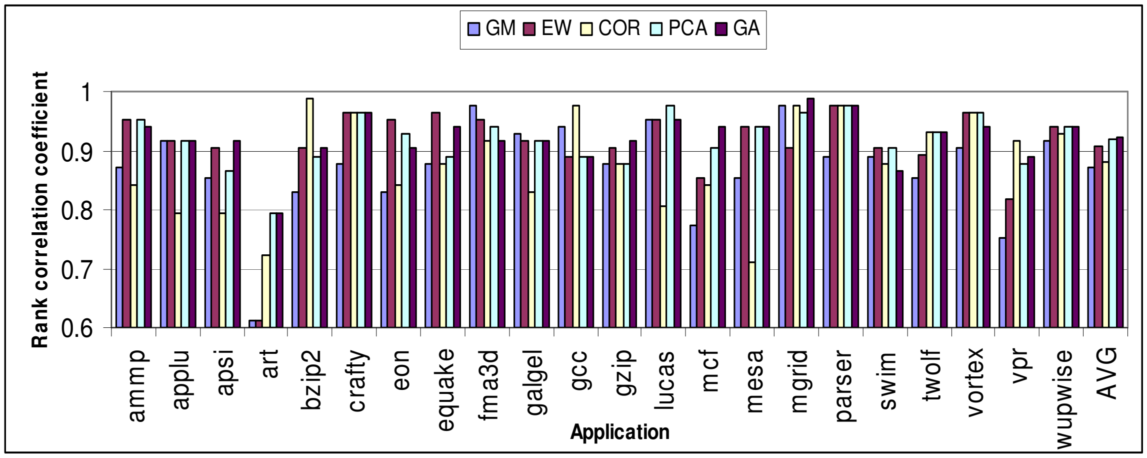
Computer system name
AMD-TyanThunderK8QSPro(S4882), AMDOpteron(TM)850
AlphaServerDS106-600
AltosG520(3.6GHzIntelXeon)
Dell-PrecisionWorkStation340(1.5GHzP4)
Fujitsu-PRIMEPOWER650(1890MHz)
HP Integrityrx4640-8(1.6GHz-9MBItanium2)
IntelD850EMV2motherboard(2.53GHz,Pentium4processor)
SGI-Origin200360MHzR12k
SunBlade2500(1.6GHz)
X6DH8-E-G2Motherborad-Intel Xeon 3.6GHz 2MCache

As mentioned earlier in this chapter it is important for a customer to find ranks of machine based on their performance that are specific to his application. In this experiment each benchmark is considered as a customer's application using the leave-one-out technique and the other programs in SPEC CPU2000 suite act as the benchmarks in the repository. The speedup of each application is predicted on all the ten machines using the methodology described in the previous section. The four data transformation techniques are used independently and the KNN method is used to find benchmark(s) similar to the application for prediction. The predicted speedup on each machine can be used to find the predicted ranking of the machines. This ranking is then compared with the original measured ranking from the SPEC website using the rank correlation coefficient. The rank correlation coefficient has a range from -1 to 1 where -1 shows that the ranks are completely flipped and 1 shows that the predicted ranks are exactly the same as the measured ranks. Instead of predicting speedup on one machine and comparing that with the actual speedup, the rankings of different machines are compared. If this prediction technique is not available, customers would just look at the geometric mean (GM) of the speedup scores of all the benchmarks and rank the machines. In this experiment the GM based speedups are calculated by taking the geometric mean of speedups of all the benchmarks in the benchmark repository. The result of the prediction technique using the four different data transformations is compared with the rank predictions using GM.

The goals of this experiment as follows:

1) To compare the ability of machine rank prediction of the GM method with prediction using program similarity information
2) To evaluate the different data transformation techniques, i.e. EW, COR, PCA and GA and also evaluate the different values of *k* for the KNN method.

Figure 5.6: Comparison of rank correlation coefficient of all the benchmarks for ten machines

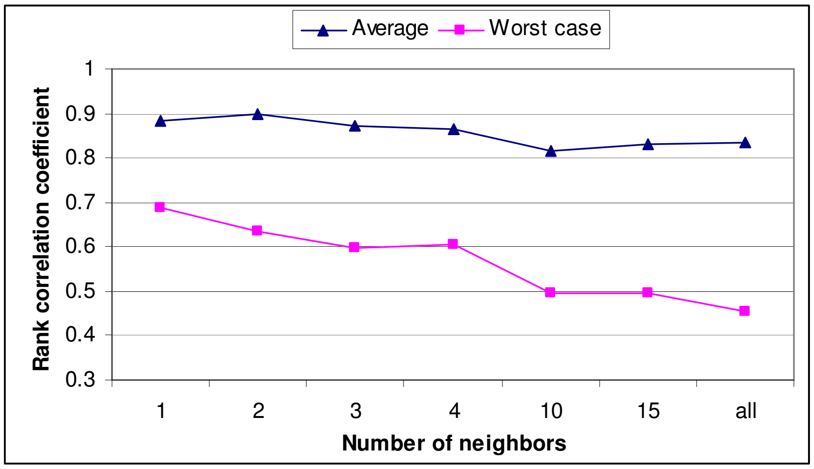


5.2.1.2 Results of prediction of machine ranks

Figure 5.6 shows the rank correlation coefficients of all the benchmarks, at a time one benchmark is considered as a customer application and its rank correlation coefficient is estimated over all ten machines. The last set of columns (AVG) show the average rank correlation coefficient over all the benchmarks. The AVG case shows that all the techniques that predict machine ranks using similarity between programs (EW, COR, PCA, GA) show higher rank correlation coefficient than GM which is the simplest technique that would be used by the customer if the program similarity information is not available. There are only very few case where the GM shows higher rank correlation coefficient.

The second main goal of this experiment is to compare the different data transformation techniques. From Figure 5.6, it is observed that the GA and PCA data transformation techniques show higher rank correlation coefficient of 0.92 and 0.91 respectively as compared to EW and COR.

Figure 5.7: Evaluation of different values of *k* for the K-nearest neighbor algorithm for predicting the ranks of machines



Different value of *k* are evaluated to see how many neighbors should be used to predict the speedup and hence the rank of the machine. Figure 5.7 shows the different values of *k* used, on the horizontal axis and the vertical axis shows the rank correlation coefficient. The two lines show the average rank correlation coefficient and the worst rank correlation coefficient over all the data transformation techniques for different values of *k*. If *k* is more than one then a weighted average speedup is calculated where the weight is proportional to the distance of the application of its neighbors. The average rank correlation coefficient is highest for *k=2* but *k=1* is also not too far away. But the worst case rank correlation coefficient goes on decreasing steadily. The result shown in Figure 5.6 was using *k=2*.

5.2.2 Experiments for predicting CPI

5.2.2.1 Experimental setup for CPI prediction

The main part of this experimental setup is the benchmark repository. It is important to have as many benchmarks as possible with their performance numbers. Cycles per instruction (CPI) shows the performance of a CPU and is inversely proportional to performance. The performance numbers used throughout this experiment that form a part of the benchmark repository are the CPI counts for each benchmark. Sim-outorder simulator from Simplescalar [2] tool set is used to measure the CPI counts on an out-of-order processor model. Table 5.1 shows the details of the configuration.

Table 5.2: Configuration of cycle accurate processor simulator model used to measure CPI

Issue width	2-way
RUU/LSQ	32/16
Memory System	8KB 2-way L1 I/D, 256K 4-way L2
ITLB/DTLB	4-way 16 entries/ 4-way 32 entries 30 cycle misses
L1/L2/mem latency	1/6/36 cycles
Functional Units	2 I-ALU, 1 I-MUL/DIV, 2FP-ALU, 2 FP-MUL/DIV
Branch Predictor	Combined 2k tables 4 cycle misprediction penalty

The Simpoint methodology [51][52] shows that programs have large-scale phases. It is extremely time consuming to run cycle accurate simulations for long running benchmarks. To limit the time spent on the data collection phase, the experiments are done on the phases of the SPEC CPU2000 benchmarks, instead of their complete run. These phases provide more elements of the data for the repository of benchmarks. The CPI prediction for the whole benchmark can be done using the CPI numbers predicted for the whole benchmark. Each phase is about 100 million instructions long. Instead of the complete benchmarks, these phases form the repository of the benchmarks and hence forth in this chapter the phases will be referred to as benchmarks. The microarchitecture independent metrics described in Chapter 2 are measured for these benchmarks and form a part of the benchmark repository. The tool for performance prediction is written in

Matlab [41]. Also, the genetic algorithm tool from Matlab is used to implement the fourth technique (GA) described in section 5.3. After building the repository of benchmarks, a leave-one-out cross-validation technique is used to find the average predicted CPI. Each benchmark is treated as an unknown application with rest of benchmarks used for similarity analysis. The *k* nearest neighbor approach is used to find the most similar benchmark and its CPI is used for prediction. The accuracy of CPI prediction is expressed in terms of percentage of error for each benchmark and then an average prediction error is equal to the average over all benchmarks.

The three main goals of this experiment are:

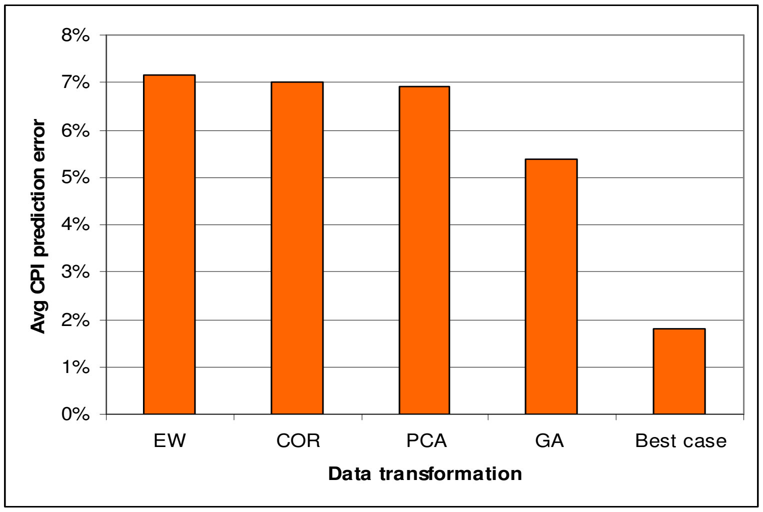
1) To compare the four different techniques of data transformation described in section 5.3. The baseline technique is EW since it takes the least of the effort and is the first step for all other techniques.
2) To see the effect of fewer benchmarks in the repository. Since the methodology is based on finding a similar program to predict performance, it is intuitive that error goes up as the number of benchmarks in the repository goes down.
3) To evaluate the different values of *k* in the k-nearest neighbor algorithm.

The results of the experiment that are discussed in the next section compare different techniques to transform the data and map the benchmarks in the workload space.

5.2.2.2 Results for CPI prediction

Figure 5.8 shows the average prediction error for multiple data transformation techniques discussed previously in section 5.3. Prediction is done using the nearest neighbor technique. In this technique CPI of the nearest neighbor is used to predict the CPI of the unknown application. Training for GA is also done using nearest neighbor prediction technique. These results are obtained using the leave-one-out cross-validation. Each column in Figure 5.8 is the average of prediction errors for all benchmarks after the leave-one-out cross-validation. The last column shows the best-case average prediction error due to availability of similar program(s). The best case assumes that the benchmark that has CPI closest to the CPI of the unknown application is also the closest benchmark in the workload space. This best-case is for the repository of benchmarks used in the experiment and will change if there are any changes in the repository.

Figure 5.8: Comparison of average prediction error in CPI for multiple data transformation techniques



The results with prediction errors for each of the individual phases using the leave-one-out validation technique are presented in the appendix. The phases are split into two tables due to space constraints on a single page. The technique of choosing characteristics based on correlation coefficient of CPI with the microarchitecture independent metrics shows the highest average error while the prediction based on GA shows the best results. GA and COR are the techniques that require training and use the CPI scores to find weights and choose characteristics respectively. Prediction based on PCA chooses the top few PCs based on the amount of variance covered by each PC. In this study top ten PCs are retained which cover about 95% of variance. The PCA transformation of data performs better than EW which involves only normalizing the data. The best-case result gives an idea about the lower bound on the prediction error in this case.

Figure 5.9: Comparison of maximum prediction error in CPI for multiple data transformation techniques

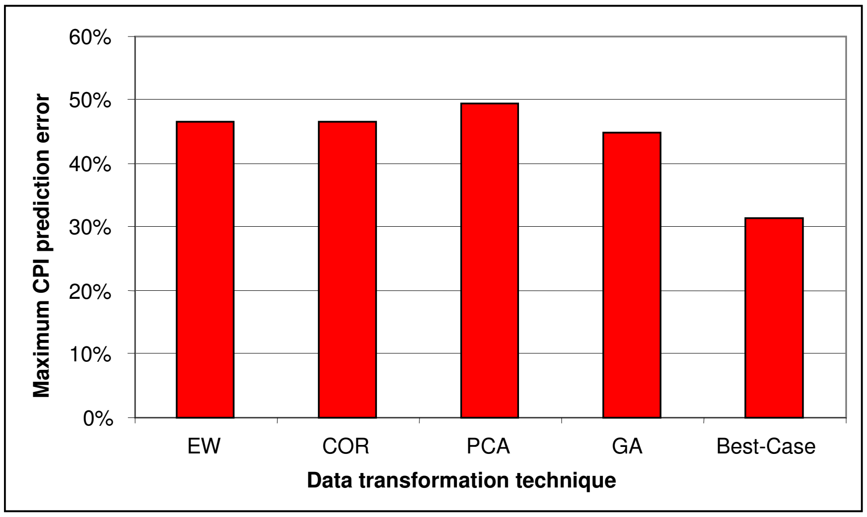
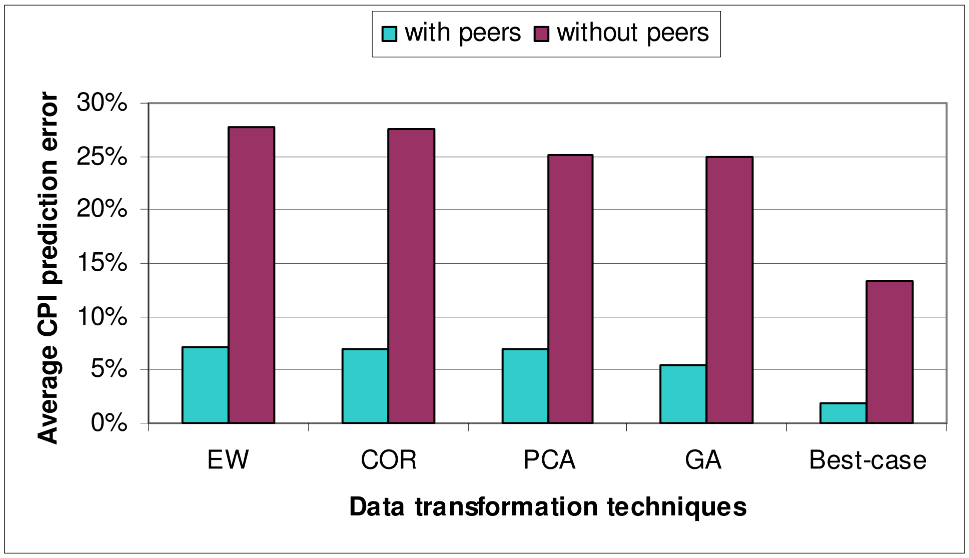


Figure 5.9 shows the maximum error seen in predicting CPI using different data transformation techniques. The maximum error for GA is lower than the one for the normalized data. But for PCA, although the average prediction error is less than EW, it has a slightly higher maximum error. The best-case maximum error is also about 30% which means that there is at least one benchmark that has about 30% prediction error.

Figure 5.10 shows the effect of having fewer benchmarks in the repository. In this experiment the phases from the same benchmark are excluded from the repository of benchmarks and the prediction is based only on the phases from the other programs. While predicting CPI for bzip2_1 phases bzip2_2 to bzip2_8 are excluded leaving only the other sixty phases that can be seen in the figures in appendix. Since the number of phases used for prediction decreased as compared to the previous experiment it is intuitive that the average prediction error will increase because the opportunity to find similar phases reduces.

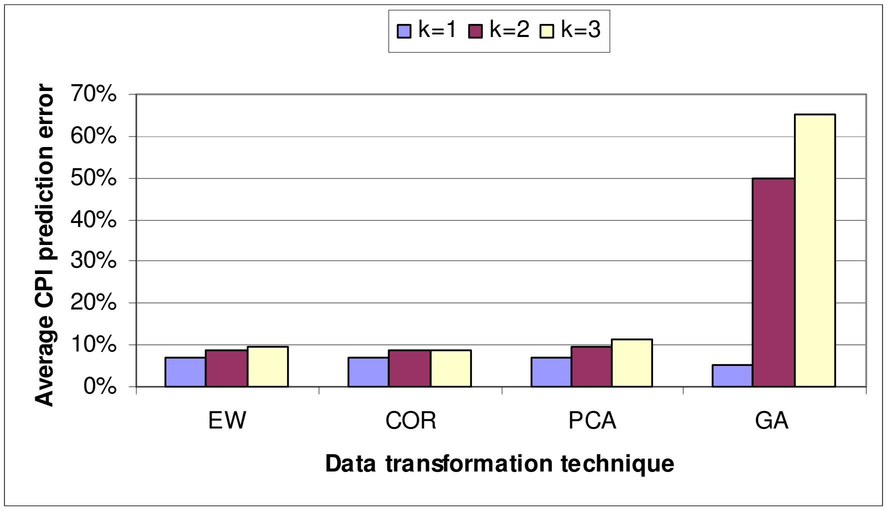
Figure 5.10: Increase in prediction error seen with decrease in similar benchmarks in the repository



In Figure 5.10 the prediction method where the phases from the same program are not considered is called ‘without peers’ and the one considering all the rest of the phases is called ‘with peers’. The results for ‘with peers’ are exactly the same as shown in Figure 5.8. The prediction errors for GA and PCA are almost the same. The best-case error for ‘without peers’ is about 13% which is significantly higher than 1.7% for ‘with peers’. The availability of similar benchmark affects the prediction error significantly.

The third experiment evaluates the different values of *k* in the k-nearest neighbor approach of prediction. So far in the previous experiments, the nearest neighbor approach has been used. In this experiment for each of the four data transformation the prediction of CPI is done using *k=1, k=2* and *k=3*. The average CPI prediction errors are shown in Figure 5.11. It is clear that the *k=1* nearest neighbor approach works better than the *k=2* and *k=3*. In case of *k=2* and *k=3* each neighbor considered is given a weight that is proportional to the distance of the application of interest from its neighbor. The error is higher for *k=2* and *k=3* than the nearest neighbor approach.

Figure 5.11: Evaluation of different values of k in k-nearest neighbor approach for CPI prediction



5.2.3 Experiments for predicting cache hit-rate

In this experiment cache hit-rate is predicted instead of CPI using the microarchitecture independent metrics for data locality. Also, the four different data transformation techniques EW, COR, PCA, GA are compared. The data locality metrics proposed in Chapter 2, are used as microarchitecture independent metrics in the prediction methodology. These metrics are based on the reuse distance of memory accesses. As such, it is important to compare the different forms of reuse distance metrics used to model locality. The reuse distance measured across all the memory accesses can be represented as a distribution of reuse distances instead of an average number. The distribution is then reduced to lesser dimensions by aggregating the buckets into classes of small, medium and large. The distribution is first made of buckets of reuse distance of 2, 4, 8, 16 and so on, up to 4096 and the last bucket is greater than 4096. This vector shows the distribution of reuse distance for a program or a phase of a program in this case. These buckets are then aggregated in such a way that the first 6 buckets i.e. up to the distance of 64 is aggregated into a bucket called ‘small reuse distance’. The distribution from 128 to 8192 is aggregated into a bucket called ‘medium reuse distance’ and the remaining buckets are aggregated into a bucket called ‘large reuse distance’. Note that, all the sizes of original buckets are power of 2.

5.2.3.1 Experimental setup for cache hit-rate prediction

The experimental setup for cache hit-rate prediction is similar to the one described in the previous section about prediction of CPI. The phases of nine benchmarks from SPEC CPU2000 are used in the experiment. The data locality of phases is measured using the two different ways described above. The average reuse distance metric is referred to as ‘Average_RD’ in this experiment and the reuse distance distribution based metric is called ‘RDD’. Average_RD was used before in Chapter 2 to subset benchmarks and also in predicting CPI in the previous section. In this experiment the Average_RD metric and RDD are compared by calculating the accuracy of each of these metrics to predict cache hit-rates for the phases of SPEC CPU2000 benchmarks using the leave-one-out validation technique. The comparison is done using all the four data transformation techniques i.e. EW, COR, PCA, GA. The cache configuration used in this experiment is 8KB, 64 bytes block, direct mapped cache.

In summary the two main goals of this experiment are:

1) To compare the four different techniques of data transformation described in section 5.3 for predicting cache hit-rates. The baseline technique is EW since it takes the least of the effort and is the first step for all other techniques. The Average_RD metric is used to characterize data locality of the phases.

2) To compare Average_RD and RDD metrics by comparing their ability to predict cache hit-rates for phases of CPU2000 benchmarks.

5.2.3.2 Results for cache hit-rate prediction

Figure 5.12 shows the average prediction error for multiple data transformation techniques discussed previously in section 5.1.1. Prediction is done using the nearest neighbor approach. Similar to the previous experiment of predicting CPI, the nearest neighbor approach shows more accurate results as compared to the two and three near neighbor approach. The GA data transformation technique shows a lower average cache hit-rate prediction error as compared to EW, COR and PCA but the EW, COR and PCA transformation techniques show very similar prediction errors. The best-case error is calculated using the same method as described in the CPI prediction experiment and is about 0.5%.

Figure 5.12: Comparison of average prediction error in cache hit-rate for different data transformation techniques

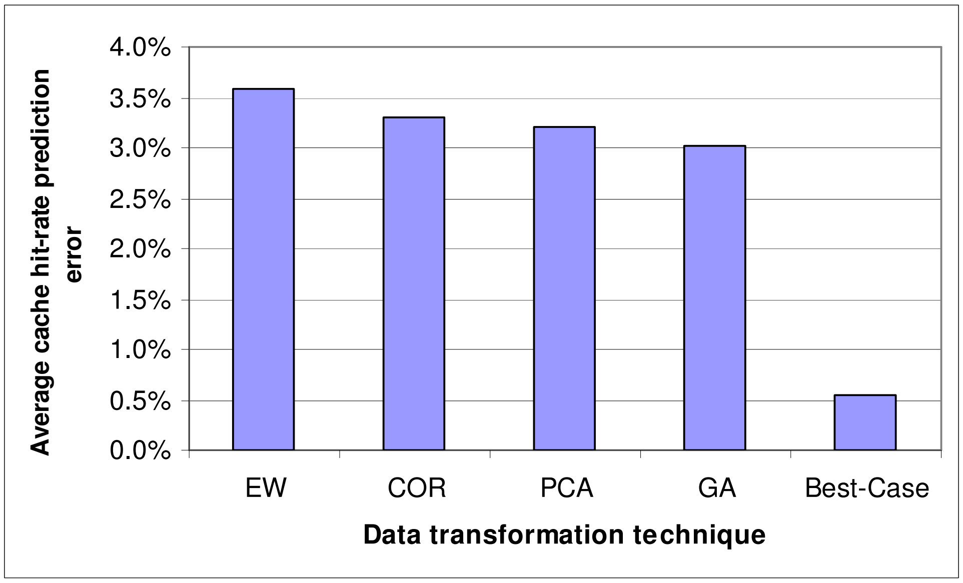
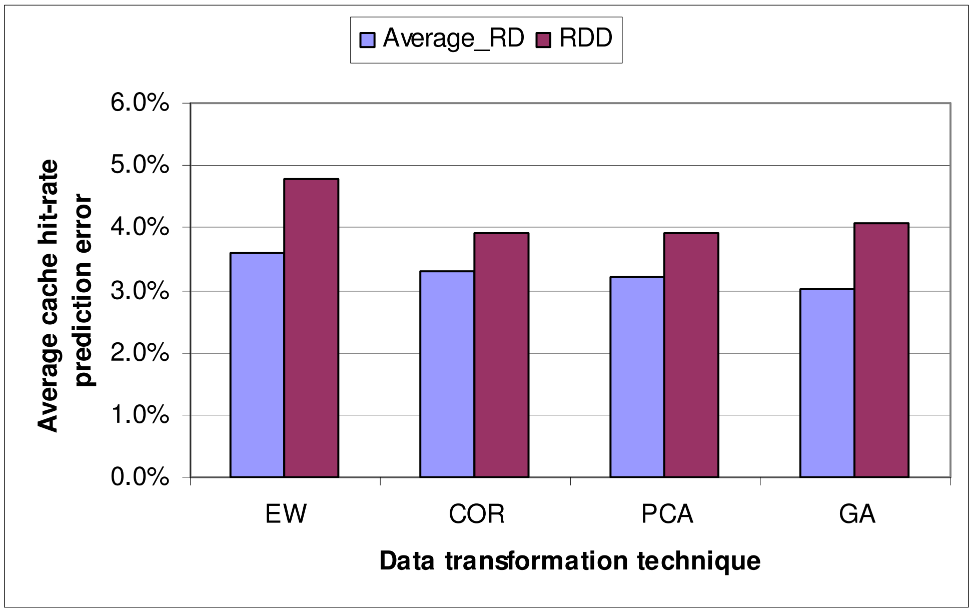


Figure 5.13 shows the comparison of the ability of Average_RD and RDD to predict cache hit-rate. The Average_RD metric can predict the cache hit-rates with much more accuracy as compared to the RDD approach. Also, for RDD the GA data transformation technique shows the best result with about 42% prediction error. It can be observed from Figure 5.13 that the data transformation technique does not affect the prediction errors for Average_RD as much as it does for RDD. The best case for RDD which is achieved using GA is not as good as any of the predictions using Average_RD. This shows that the Average_RD metric shows better ability to predict cache hit-rates as compared to RDD.

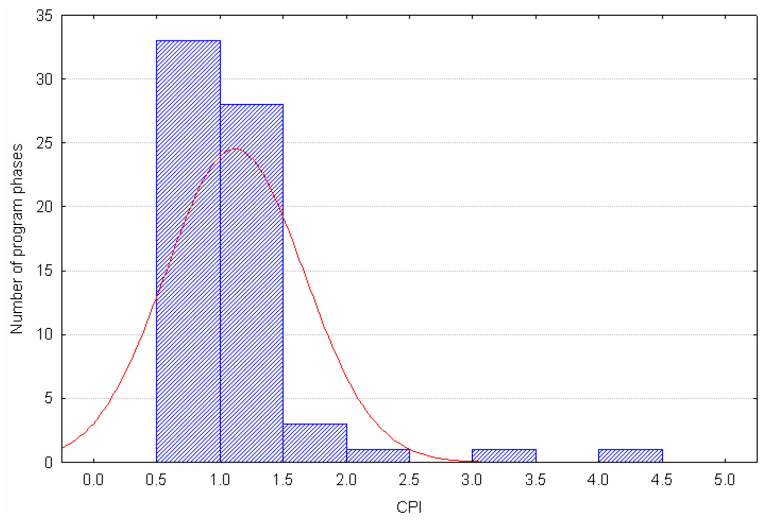
Figure 5.13: Comparison of two ways to measure data locality using microarchitecture independent metrics for cache hit-rate prediction



5.3 DISCUSSION

This section describes the main challenges faced by the performance prediction technique using similarity between programs. Unlike the other techniques like simulation and analytical modeling, this technique needs the similarity information about other benchmarks and hence the accuracy depends significantly on the availability of a large number of diverse benchmarks. Further in this section another challenge about finding an upper bound on the error has been discussed. Each of these challenges is described in detail.

Figure 5.14: Histogram of CPI of program phases to illustrate the skew in the distribution of CPI



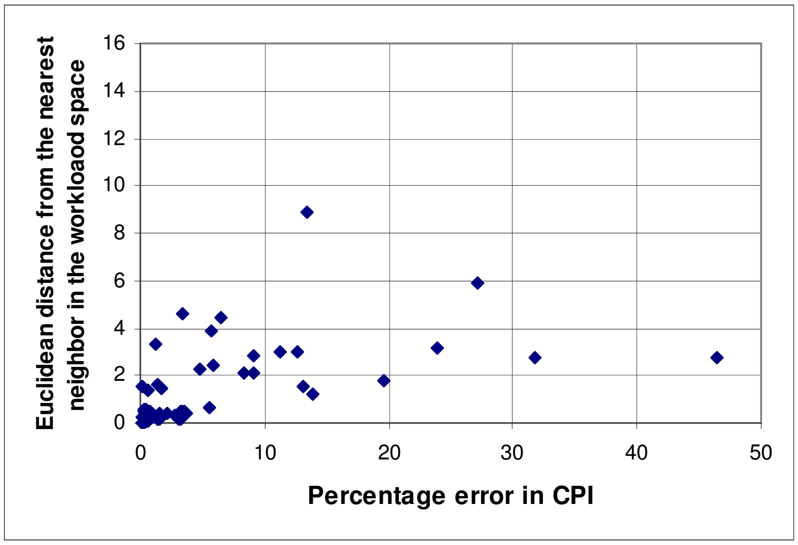
5.3.1 Distribution of benchmarks in the workload space

In case of SPEC CPU benchmarks a benchmark that stresses the CPU or memory the most is considered as a good test. These benchmarks are inherently difficult to optimize and hence are considered as tough benchmarks. It is difficult to find such benchmarks and a common example is *mcf* from the SPEC CPU2000 benchmark suite. There is a tendency to include more tough benchmarks so that the test cases are strong and if a certain system is evaluated with these benchmarks, the designer and customer have more confidence about design performing well on the tough benchmarks. Usually such programs are rare and when plotted in the workload space, they become the corner cases or outliers in the space covered by the suite. But there are many programs that do not stress the system and they are the common cases which are available easily. Also, if a program shows very good performance on the system it is not considered as a good test case and may be excluded from the benchmark suite. If a histogram of programs is plotted using their CPI numbers, the distribution looks to be skewed to the right as shown in Figure 5.14. The CPI numbers shown in the histogram are used in the CPI prediction experiments described before in this chapter. Finding phases of benchmarks similar to the application (or phases within the application) is central to accurate prediction of performance. Now, the question essentially is about what kind of distribution the benchmarks should show to assist in the better prediction of performance. Ideally, a uniform distribution for benchmark performance numbers in building a benchmark repository shown in Figure 5.1 will be very useful. Uniform distribution will help the application of interest to find similar benchmarks. Bell et.al. [4][5] proposed the development of synthetic benchmarks from a real benchmark for efficient performance evaluation and validation. Joshi et.al. [34] proposed a technique to extract characteristics from a real benchmark and generate synthesize programs with the extracted ones. This technique can be used to populate the workload space and cover different areas in the workload space with a uniform distribution. To ensure that there are benchmarks of broad range of behavior or performance, it is important that a histogram that is plotted as shown in Figure 5.14 shows a uniform distribution. The repository of benchmarks with a uniform or close to uniform distribution may help to increase the accuracy of performance prediction using program similarity.

5.3.2 Threshold of distance for predicting performance in the workload space

For a performance evaluation technique it is important to have an idea about the range of error. In case of simulation, the upper bound on the error typically depends on the extent of details that are implemented in the simulator to model the system. It is difficult to find an upper bound on the error for performance prediction based on program similarity but a threshold of distance can be used to judge the error before doing the prediction. The threshold will be certain distance above which the prediction errors can be more than the user's margin of error. One way to judge the threshold is by looking at the distance of the nearest neighbor from the application of interest whose performance is to be predicted. The error is proportional to the distance between the application of interest and the nearest benchmark. The following experiment is done to see how the distance between two benchmarks correlates with the error in CPI. The data used in this experiment is from the CPI prediction experiment where a benchmark is a phase from the SPEC CPU2000 programs. From the repository, each benchmark is mapped on a plot of CPI error and distance of the benchmark from its nearest neighbor. CPI error is the percentage difference in CPI seen with the nearest neighbor. Figure 5.15 shows the plot for data transformed using genetic algorithm (GA).

Figure 5.15: Correlation of distance and error in CPI prediction using nearest neighbor for data transformed using genetic algorithm (GA)



The next step in determining the threshold distance is to calibrate it with the nearest neighbor to the prediction error. This process depends on the designer's margin of error e.g. from Figure 5.15 if the designer wants the maximum error to be 10%, according to the data available for calibration the maximum distance allowed from the nearest neighbor should be approximately 1 in the workload space formed using GA. In the bottom left two blocks seen in the grid in Figure 5.15, all the benchmarks with the margin of error of 10% in the calibration experiment lie within the euclidean distance of 1.

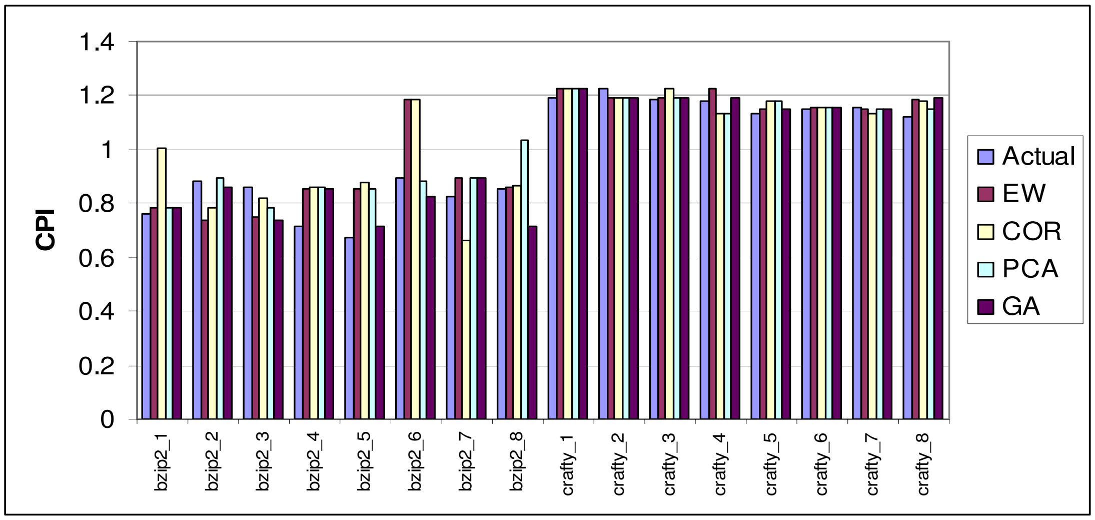
5.3.3 Comments on CPI prediction errors for individual phases

In the previous section on CPI prediction the results were described concisely with only the average prediction error over all the phases of the benchmarks shown in the results. This sub-section looks at individual phases and the prediction error of the outliers. All the results of individual phases are presented in the appendix. Some of the figures used in the discussion are included in this chapter but are also presented in the appendix. The CPI prediction numbers from the experiment described before in this chapter are shown for each phase using leave-one-out cross-validation technique. Figure 15.16 shows the CPI prediction done using the peers from the same benchmark to which the phase belongs to. Figure 15.17 shows the similar result but with the peer phases excluded from the analysis. Since there are many phases, they are split into four plots (a), (b), (c), (d). The first bar for each phase within a benchmark is its actual measured CPI. Each of the remaining four bars shows the predicted CPI using the four different data transformation technique EW, COR, PCA, GA respectively.

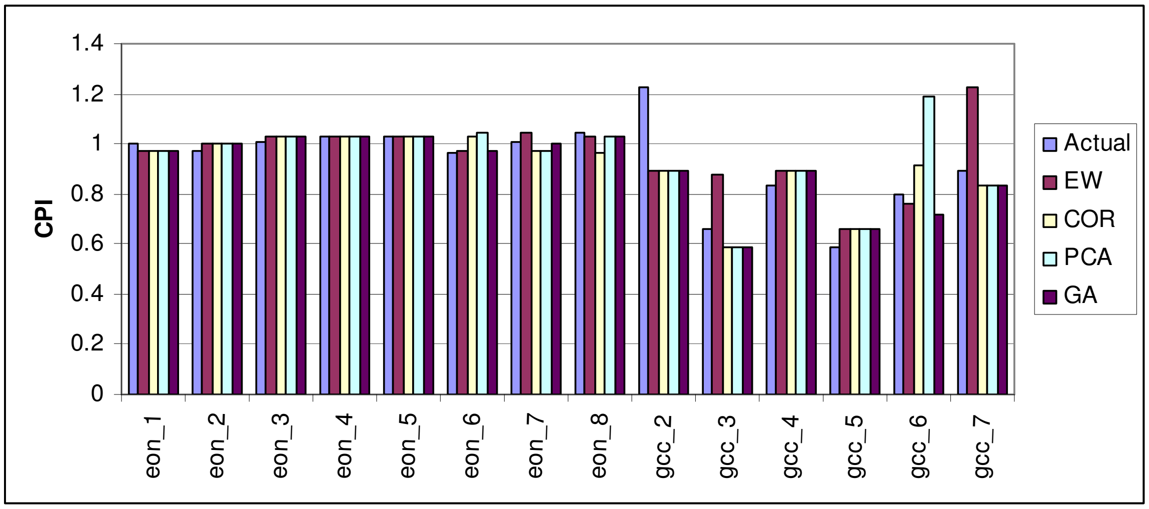
It can be seen clearly that in Figure 15.16, for many phases the peers are the nearest neighbors for a phase for all four data transformation techniques e.g. for *eon, gzip* and *twolf* in Figure 15.16 their peer phases are very similar and hence lead to a very accurate prediction. The phases of *mcf* show quite diverse behavior but most of its phases find their peers to predict the CPI. In Figure 15.17 where the peers are not used in the analysis, each of the phases finds a phase from other benchmarks to predict its CPI. Specifically the phases from *mcf* become outliers as their peers are not used in the analysis. It can be seen from Figure 15.17 (c) that all *mcf* phases show a large difference in actual and predicted errors. It is clear from this observation that *mcf* phases are outliers in the workload space and all the peers are close to each other and away from the rest of the phases. The *mcf_7* phase has the highest error. Each of the phases can be studied individually. The microarchitecture independent metrics for the phases in *twolf* are very similar which explains why their CPIs are similar. The inherent behavior of the phases is measured using the microarchitecture independent metrics captures the CPI quite well for similarity analysis.

Figure 15.16: Percentage errors in predicting CPI using the four different data transformation techniques for each phase using the leave one out technique **(with phases from the same benchmark included)**. Note: The figure is split into (a), (b), (c), and (d) due to space limitation on X-axis for all the phases.

(a)

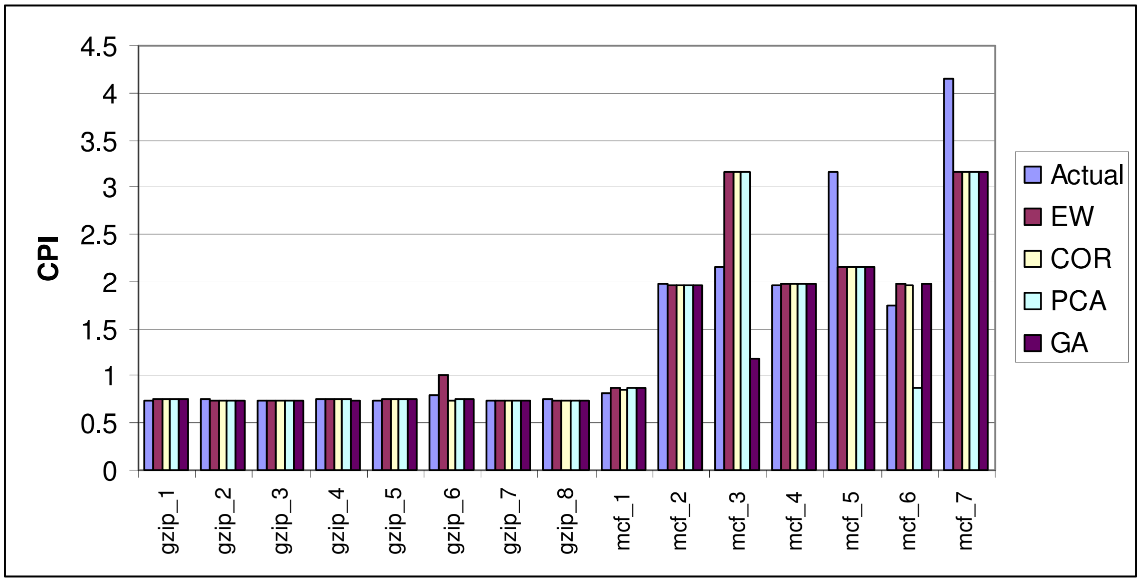


(b)



CPI
1.4
1.2
1
0.8
0.6
0.4
0.2
0
eon_1
eon_2
eon_3
eon_4
eon_5
eon_6
eon_7
eon_8
gcc_2
gcc_3
gcc_4
gcc_5
gcc_6
gcc_7
Actual
EW
COR
PCA
GA

(c)



CPI
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
gzip_1
gzip_2
gzip_3
gzip_4
gzip_5
gzip_6
gzip_7
gzip_8
mcf_1
mcf_2
mcf_3
mcf_4
mcf_5
mcf_6
mcf_7
Actual
EW
COR
PCA
GA

(d)

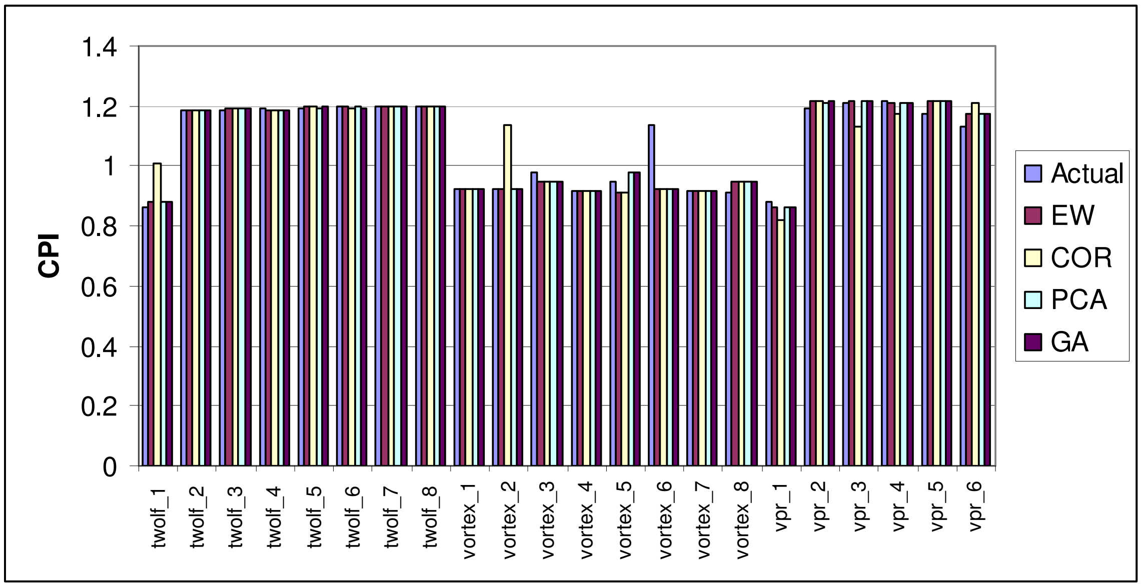
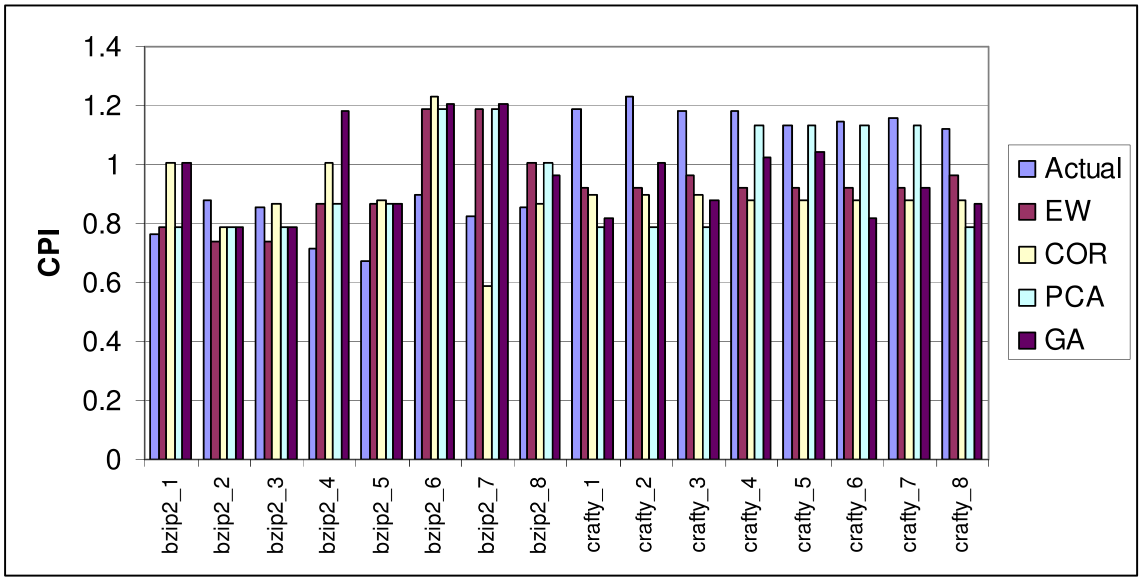
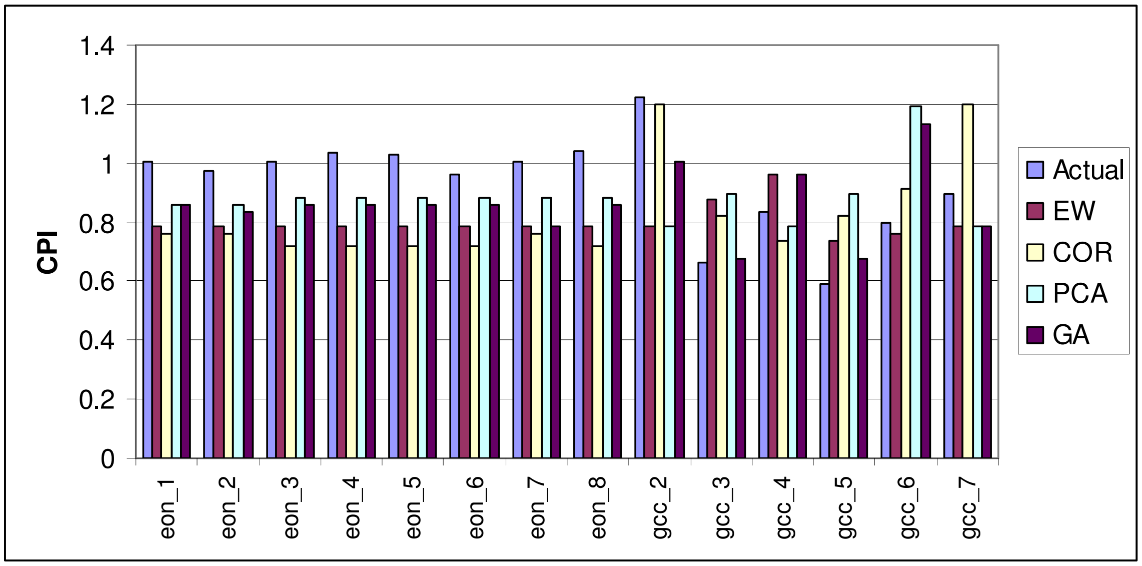


Figure 15.17: Percentage errors in predicting CPI using the four different data transformation techniques for each phase using the leave one out technique.**(without the phases from the same benchmarks)** Note: The figure is split into (a), (b), (c), and (d) due to space limitation on X-axis for all the phases.

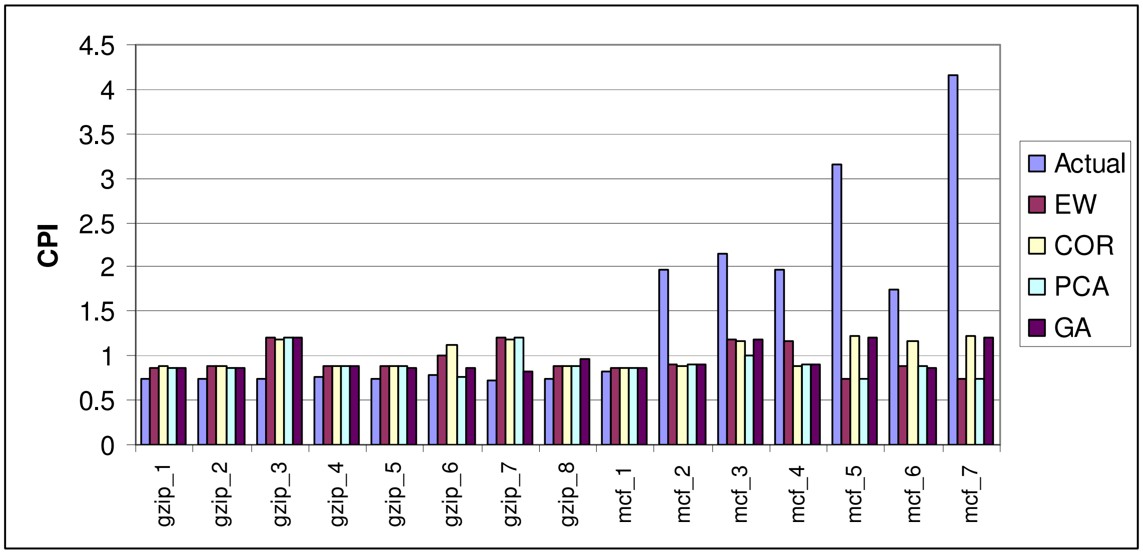
(a)



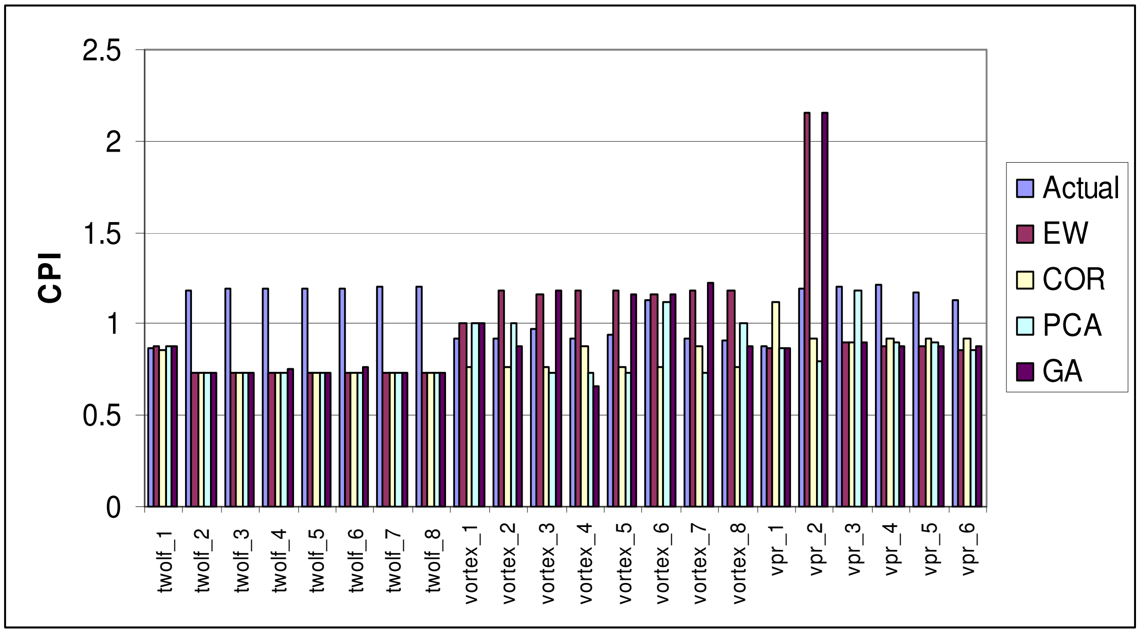
(b)


1.4
1.2
1
0.8
0.6
0.4
0.2
0
CPI
eon_1
eon_2
eon_3
eon_4
eon_5
eon_6
eon_7
eon_8
gcc_2
gcc_3
gcc_4
gcc_5
gcc_6
gcc_7
Actual
EW
COR
PCA
GA

(c)


4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
CPI
gzip_1
gzip_2
gzip_3
gzip_4
gzip_5
gzip_6
gzip_7
gzip_8
mcf_1
mcf_2
mcf_3
mcf_4
mcf_5
mcf_6
mcf_7
Actual
EW
COR
PCA
GA

(d)



5.4 SUMMARY

Performance prediction using program similarity is a technique that can be used in case of a classical problem in benchmarking and performance evaluation where a customer's application is not a part of the benchmark suite. Performance of the application can be estimated by finding an already well characterized benchmark in the workload space that is similar to the application. The performance score(s) of the similar benchmark(s) can then be used to estimate the performance of the application. Microarchitecture independent metrics are used to characterize the benchmarks and the application. The application is then mapped into the workload space of benchmarks and its performance is predicted.

There are many microarchitecture independent characteristics but only a small set of characteristics may affect performance or the degree at which they affect performance might be different. This dissertation explores different ways of choosing characteristics or giving weights to the characteristics. Three different data transformation techniques are evaluated, including the PCA, genetic algorithm and correlation based characteristic selection. The genetic algorithm and correlation based characteristic selection technique use the performance numbers of already characterized benchmarks to train the weights or select characteristics respectively.

To demonstrate and validate the technique, it is applied to predicting (i) Speedup (ii) CPI and (iii) Cache hit-rate. The speedup prediction experiment uses the SPEC CPU2000 benchmarks but the CPI and cache hit-rate prediction use the phases of SPEC CPU2000 benchmarks to do the experiments in manageable time. The prediction and validation in each of the experiments is done using the leave-one-out cross-validation.

Finally, some of the inherent challenges of this chapter are discussed in the chapter with examples from the experiments done for CPI prediction. The challenges mainly involve finding a similar benchmark to the application for which the performance is to be predicted. In a benchmark suite usually the distribution of performance scores is skewed by the outlying benchmarks. A uniform distribution will definitely help improve the chances of finding similar benchmark and hence improve the accuracy of performance prediction. The second challenge is about finding a way to judge the upper bound on the error of prediction. To illustrate a possible way of judging the upper bound on the error a short experiment is done to calibrate the distance between benchmarks to the prediction error in CPI.

Performance prediction using program similarity uses a practical approach of reusing the previously gathered information of several benchmarks or programs to make performance prediction of a new application. This approach will need some time to ramp as building the repository of benchmarks may take some time. The update to the repository should be done as new applications can be added to it over time.

Chapter 6: Previous Research

This chapter provides a brief summary of previous work related to program similarity, characteristics used for program similarity and subsetting. Later part of this chapter provides a summary of related work to performance prediction. Although, performance prediction using microarchitecture independent metrics has not been studied before, literature related to relevant techniques has been summarized. The summary also includes related work which show correlation between some microarchitecture independent metrics that have been proposed before

6.1 PROGRAM CHARACTERISTICS AND PROGRAM SIMILARITY

Weicker [61] used characteristics such as statement distribution in program, distribution of operand data types, and distribution of operations, to study the behavior of several stone-age benchmarks. Saveedra and Smith [50] characterized Fortran applications in terms of number of various fundamental operations, and predicted their execution time. They also develop a metric for program similarity that makes it possible to classify benchmarks with respect to a large set of characteristics. Source code level characterization has not gained popularity due to the difficulty in standardizing and comparing the characteristics across various programming languages. Moreover, nowadays, programmers rely on compilers to perform even basic optimizations, and hence source code level comparison may be unfair.

Conte [11] uses kiviat views to qualitatively compare program behavior based on microarchitecture-dependent characteristics such as cache miss-rates, branch mispredict rates, etc. Yi *et al.* [66] use a Plackett-Burman design for classifying benchmarks based on how the benchmarks stress the same processor components to similar degrees. Plackett-Burman design helps to find the bottlenecks in smaller number of simulation runs and narrows down the problem of number of simulation runs. The goal of this classification is different.

A first major step in workload characterization is essentially collecting various characteristics that define the model. In the past, studying benchmark characteristics involved 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 [18][19][60][66]. 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.

Very recently and also in the past, some work has been done to find redundancy in benchmark suites. This work has primarily used microarchitecture-dependent metrics such as execution time or SPEC peak performance rating for characterizing programs. Vandierendonck et. al. [60] 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. In [60], the author quantifies redundancy as the ability of a program to show different speedup on two different machines. The programs that do not show very different speedups are considered redundant. In other words [60] concludes that there is no need of such redundant programs to rank the predecided 340 machines. According to [60] the top ten redundant programs from SPEC CPU 2000 suite are *vpr, ammp, sixtrack, bzip2, vortex, gcc, mgrid, equake, wupwise, galgel.* The top ten important benchmarks are *apsi, lucas, mcf, gap, facerec, mesa, art, eon, parser* and *fma3d.* Dujmovic and Dujmovic [16] developed a quantitative approach to evaluate benchmark suites. They used the execution time of a program on several machines 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 inherent 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 poor) on them, and not because that benchmark was not unique. Although *gcc* is considered to have complex control flow and considered to be an interesting benchmark, the relatively lower rank of *gcc* in [60] 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. [11][12][31][36][55][56]. Generic measures of parallelism were used by Noonburg et. al. [45] and Dubey et. al. [15] based on a profile of dependency distances in a program. Sherwood et. al. [51] 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 [22][32].

Eeckhout et.al. [18][19] proposed measuring program similarity based on microarchitecture dependent metrics and showed relative positions of benchmarks in the workload space built by the measured characteristics. A subset that generated using this analysis may be biased due to a fixed configuration used to measure the metrics similar to the previous work mentioned above.

Hoste and Eeckhout [29] compare microarchitecture independent metrics with microarchitecture dependent metrics. They compare the distances from workload spaces built using microarchitecture independent metrics and microarchitecture dependent metrics. The correlation coefficient comes out to be 0.46 which shows that the characterization of programs is different and it will be misleading to measure similarity based on microarchitecture dependent metrics.

Another stream of work reduces simulation time of benchmarks by finding representative phases within a program [52][64]. These techniques are orthogonal to the one presented in this paper and can be used to further reduce the simulation time of the subset of programs selected from the suite. Simpoint work uses a metric called Basic Block Vector (BBV) to find phases within a program. BBV forms a code signature of a program. It is a vector which has frequency of each static basic block in the code. The BBV is measured for each chunk of a certain fixed number of instructions. Then these chunks are clustered based on their BBV. Each cluster represents a phase within the program. This work also uses microarchitecture independent metrics but BBV cannot be used to compare two different programs because BBV is based on the static basic blocks of a program.

6.2 PERFORMANCE PREDICTION USING PROGRAM SIMILARITY

A large body of work has also been done on the correlation between microarchitecture independent program characteristics and processor performance; see for example [1][37][53]. However, these techniques do not predict performance for an application of interest based on cross-program similarity. Instead, these techniques predict performance based on intra-program phase-level similarities. This requires that particular phases of the application need to be executed for making a performance prediction. This is not the case with work described in this dissertation.

The top few principal components after doing PCA are based on the variance within a dataset. Characteristics that have higher variance get chosen in the top few principal components or get higher weight. But there might be some characteristics in the data that might be more useful to classify or form good clusters but have small variance. Yueng and Ruzzo [65] propose a technique using greedy algorithm to choose principal components while using PCA for clustering gene expression data. In [65] the authors tackle this problem by using greedy algorithm to choose principal components, not based on the degree of variance but by empirically checking each one of the components to see if the classification improves. This is very relevant to the work on performance prediction using program similarity because the program characteristics that are not very well correlated can have higher variance and vice-versa.

Chapter 7: Conclusions and Future Research

7.1 CONCLUSIONS

The process of performance evaluation is a very important part of the design process. Benchmarks are used with simulation infrastructure to evaluate performance in early design phase of microprocessors. Modern benchmarks are developed from real world applications which are getting complex and run for a long time. The complexity of microprocessor design is increasing due to added features for better performance and optimizations. This increases the effort of performance evaluation and may effectively lead to longer time to market for newly designed microprocessors or computer systems. Usually, researchers and computer architects randomly choose benchmarks to evaluate their idea which may lead to misguiding results. Performance analysts build benchmark suites by choosing programs from a set of candidates. Each of these benchmarks need to analyzed to find a set that is diverse in the performance behavior to form a benchmark suite. This dissertation contributes towards solving these problems by measuring program similarity to reduce the effort in performance evaluation. Measuring program similarity will help to find a subset of benchmarks that are representative of the complete benchmark suite and can be used instead of the whole suite in the process of performance evaluation. The similarity information between programs can also used to predict performance of an unknown application if its similarity is measured with the already well characterized benchmarks.

- **Subsetting using program similarity**

This dissertation proposes the use of microarchitecture independent metrics to measure similarity between benchmarks. The use of microarchitecture independent metrics helps to find a subset that applicable to a wide range of architectures. To demonstrate the methodology, SPEC CPU2000 and media applications are used which belong to different application domains. Twenty-nine metrics are measured for each benchmark and all the benchmarks are analyzed together using statistical techniques like PCA and clustering. The subsets are validated using IPC, cache miss-rate and speed-up. The subsets formed showed that the average error of IPC projection for the SPEC CPU2000 and media benchmark suites is less than 5%. The average error in projection of speedup for the SPEC CPU2000 benchmarks is less than 10%.

- **Analysis of workload space coverage**

  Multiple benchmark suites often need to be compared and a representative set of benchmarks from multiple suites should be used for performance analysis. Program similarity analysis can be used to compare different benchmark suites and evaluate their coverage of workload space. Benchmark suites like the SPEC CPU evolve over time. Microarchitecture independent metrics are used to measure the similarity between programs and compare the four generations of SPEC CPU suites. The similarity of the benchmarks from four suites is also analyzed separately for four different characteristics i.e. data locality, instruction locality, ILP and branch behavior. All the benchmarks from the four suites are plotted on a scatter plot and a dendrogram. The SPEC CPU suites have evolved over time for the data locality characteristics but not much ILP and branch behavior. The instruction locality characteristics have almost remained the same over the four generations

- **Fast subsetting using microarchitecture dependent metrics**

  Development of a benchmark suite is a process where the benchmarks source code or inputs are changed rapidly over time to account for high standards on issues like portability. The benchmarks should not be an easy target for small tweaks and compiler optimizations for achieving high performance. The process of subsetting to find a representative set using microarchitecture independent metrics may take a few weeks which may not be possible with the rapidly changing benchmarks. This dissertation proposes a fast subsetting approach which uses microarchitecture dependent metrics. The microarchitecture dependent metrics are measured on five different state-of-the-art systems with different ISAs, compilers and architectures. This makes the subset applicable on a wide range of platforms. Two programs that show similar behavior using the metrics essentially show similar behavior on all five systems. To demonstrate the methodology the SPEC CPU2006 benchmarks are used to find a subset that is representative of the suite. This subset can also be used for simulations and performance analysis.

- **Performance prediction using program similarity**

A customer who plans to purchase a computer system to run his application does not find his application in the standard benchmark suites. But performance scores of different computer systems in the market are available for standard benchmarks like SPEC CPU benchmarks. In this dissertation the information of similarity between the customer's application and the benchmarks is used to find a more accurate performance score of the application on the computer systems. This technique uses the microarchitecture independent metrics to characterize the benchmarks and the application. The application is then mapped into the workload space of benchmarks and its performance is predicted. There are many microarchitecture independent characteristics but only a small set of characteristics may affect performance or the degree at which they affect performance might be different. To choose characteristics that affect performance three different data transformation techniques are evaluated i.e. PCA, selection based on correlation coefficient and genetic algorithm. To demonstrate the techniques and validate the idea, an experiment that predicts the speedup on ten different machines is done. Rank correlation coefficient is used to evaluate the correct ranking of machines with the value of 1 being the ideal case. SPEC CPU2000 benchmarks ran to completion are used in this experiment and leave-one-out cross-validation technique is used. The best result is obtained using GA data transformation technique which shows a rank correlation coefficient of 0.92. The second experiment predicts CPI of individual phases within a program using all the microarchitecture independent characteristics and the best average prediction error for CPI was 5.7%. The third experiment predicts cache hit-rate using only the data locality characteristics and the GA data transformation shows an average hit-rate prediction error of 3%. The experimental setup in the second and third experiment uses the phases of the SPEC CPU2000 benchmarks instead of the whole benchmarks and validates the prediction accuracy using the leave-one-out cross-validation. Performance prediction using program similarity reuses the previously gathered information of several benchmarks or programs to make performance prediction of a customer's application without porting the application to the given platform.

7.2 DIRECTIONS FOR FUTURE WORK

- **Scheduling on heterogeneous multi-core system**

Heterogeneous multi-core systems are made from multiple processing cores with varying strengths. Multiple threads running at a time have different requirements for performance and resources e.g. one thread may need out-of-order processing but the other thread may run well on an in-order processor. But scheduling the threads on a particular core is crucial to get the benefit of saving on power consumption without losing on performance. Static scheduling is one option especially for embedded and ASIC processors where the applications that run on the system may not change over time. Identifying program characteristics and program similarity analysis can help in making scheduling decisions. Architects can come up with strategies to map each application based on a certain strategy developed in conjunction with the similarity analysis. Characteristics that differentiate the processors can be measured for each program and then clustering can be used to find a number of clusters equal to that of the number of cores on the systems. Based on the position of the programs, each cluster of programs or applications can be mapped on to a particular core.

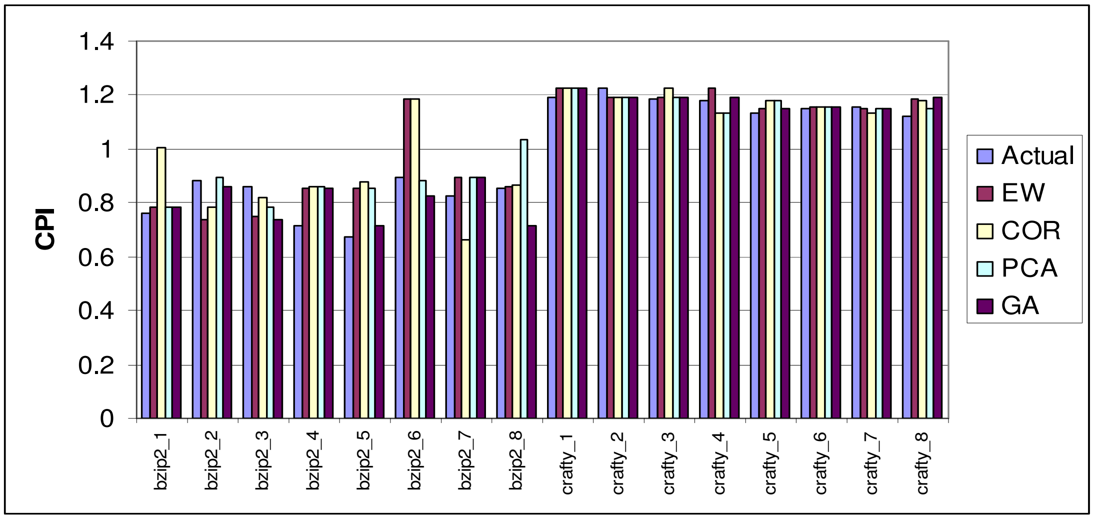
- **Measuring program similarity for multi-threaded applications**

  With the advent of multi-core systems the application development is rapidly moving towards multi-threaded applications. Multi-threaded programs may share data or may have a very high thread level parallelism. The research work in this dissertation can be directed towards finding similarity between such applications. One of the major challenges for this work lies in coming up with microarchitecture independent metrics that model communication between threads. Different communication paradigms are used to implement the data sharing between threads e.g. shared memory, message passing interface (MPI). The applications written using different paradigms may have completely different code structure and may need to be analyzed separately. But still the challenge remains the same and coming up with microarchitecture independent metrics will be crucial to the analysis.

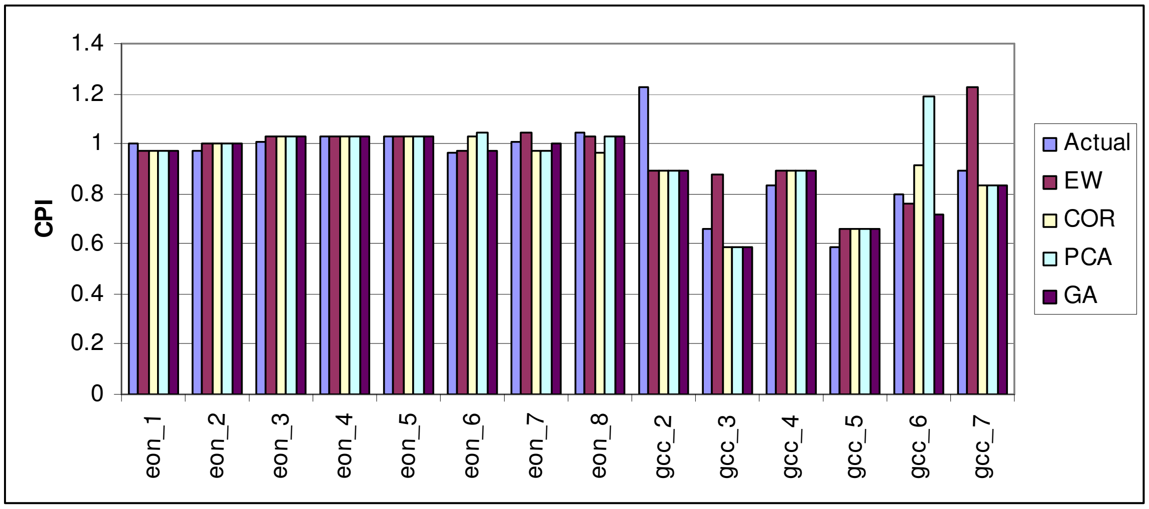
Appendix

Figure A1: Percentage errors in predicting CPI using the four different data transformation techniques for each phase using the leave one out technique **(with phases from the same benchmark included)**. Note: The figure is split into (a), (b), (c), and (d) due to space limitation on X-axis for all the phases.

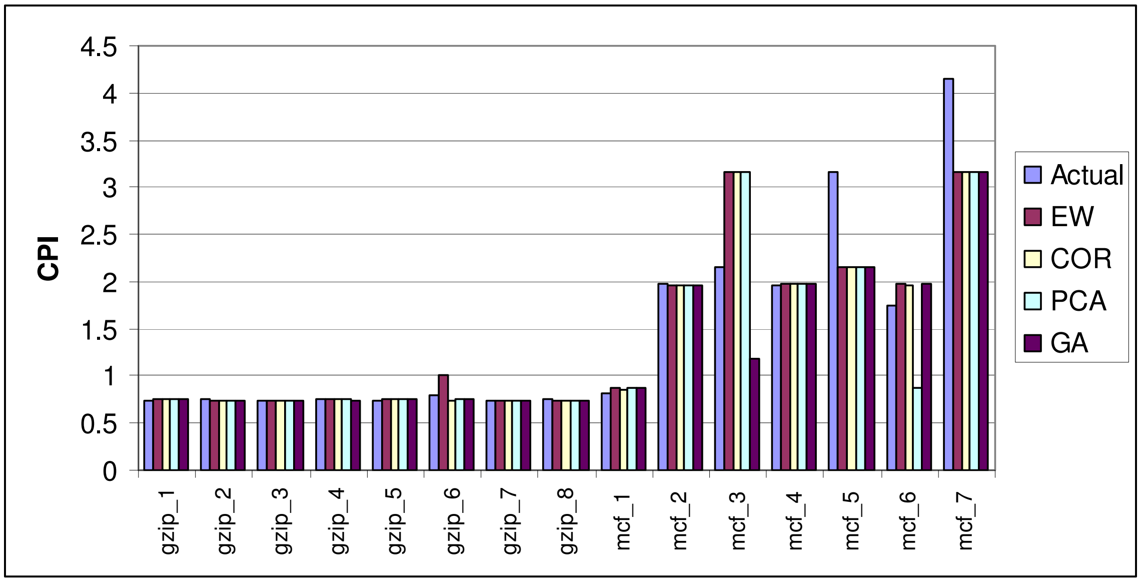
(a)



(b)

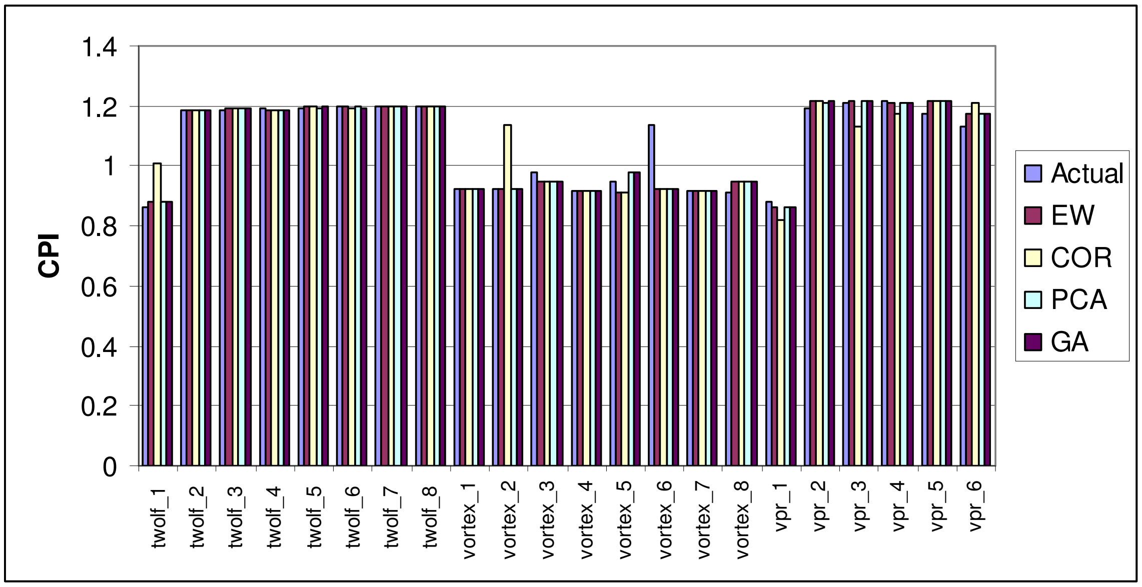


(c)



CPI
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
gzip_1
gzip_2
gzip_3
gzip_4
gzip_5
gzip_6
gzip_7
gzip_8
mcf_1
mcf_2
mcf_3
mcf_4
mcf_5
mcf_6
mcf_7
Actual
EW
COR
PCA
GA

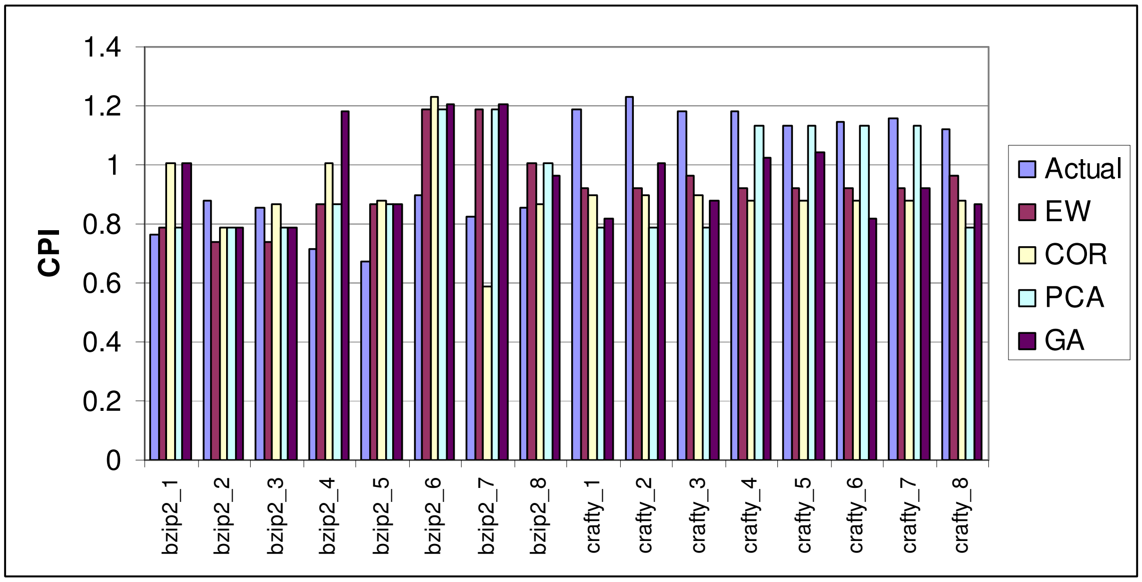
(d)



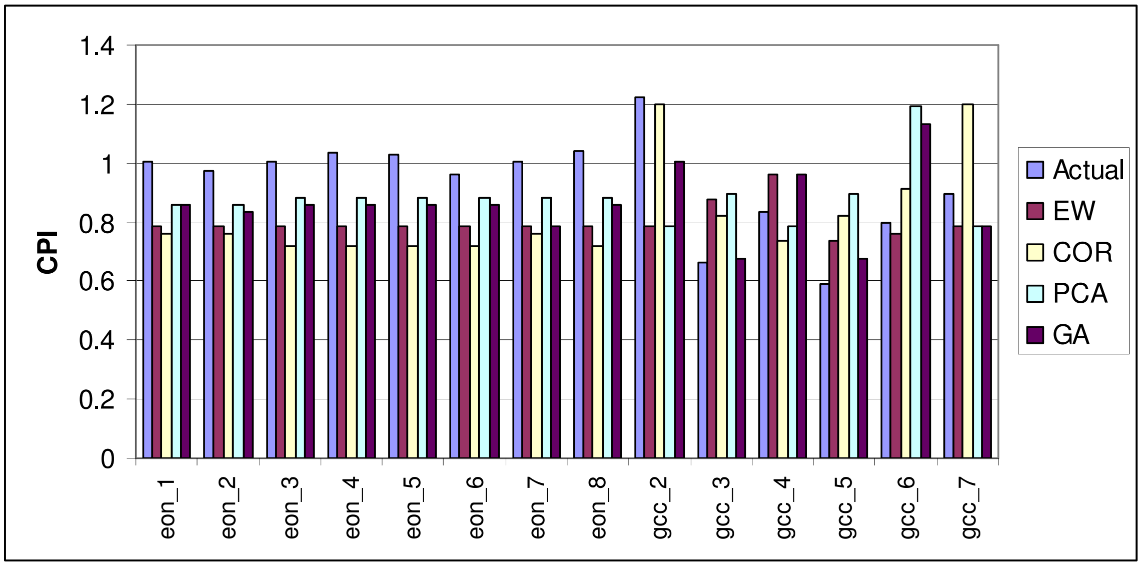
CPI
1.4
1.2
1
0.8
0.6
0.4
0.2
0
twolf_1
twolf_2
twolf_3
twolf_4
twolf_5
twolf_6
twolf_7
twolf_8
vortex_1
vortex_2
vortex_3
vortex_4
vortex_5
vortex_6
vortex_7
vortex_8
vpr_1
vpr_2
vpr_3
vpr_4
vpr_5
vpr_6
Actual
EW
COR
PCA
GA

Figure A2: Percentage errors in predicting CPI using the four different data transformation techniques for each phase using the leave one out technique.**(without the phases from the same benchmarks)** Note: The figure is split into (a), (b), (c), and (d) due to space limitation on X-axis for all the phases.

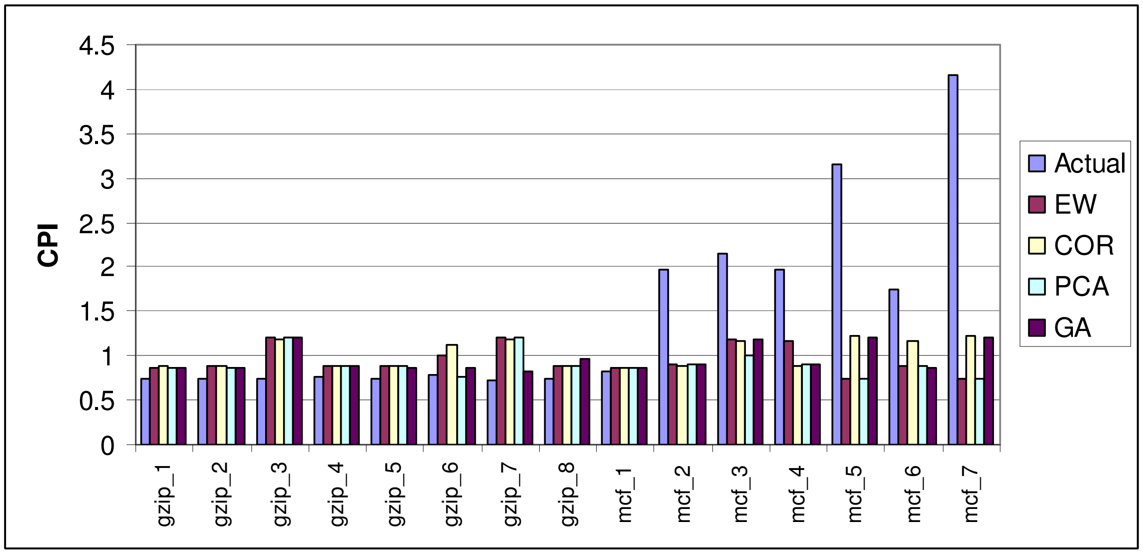
(a)



(b)

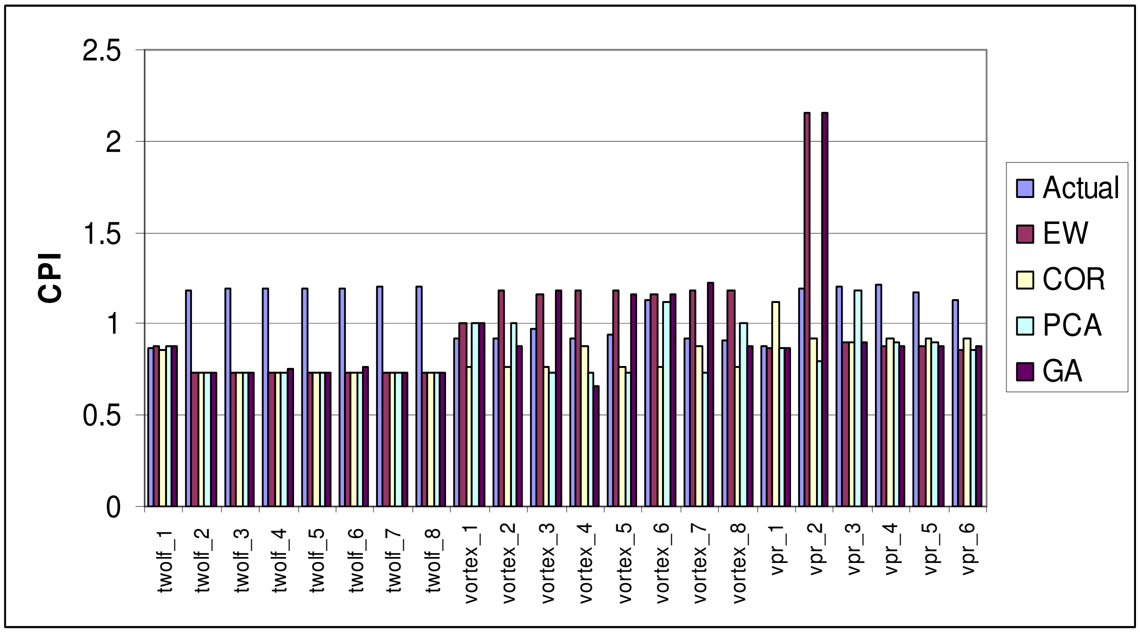


(c)



CPI
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
gzip_1
gzip_2
gzip_3
gzip_4
gzip_5
gzip_6
gzip_7
gzip_8
mcf_1
mcf_2
mcf_3
mcf_4
mcf_5
mcf_6
mcf_7
Actual
EW
COR
PCA
GA

(d)



CPI
2.5
2
1.5
1
0.5
0
twolf_1
twolf_2
twolf_3
twolf_4
twolf_5
twolf_6
twolf_7
twolf_8
vortex_1
vortex_2
vortex_3
vortex_4
vortex_5
vortex_6
vortex_7
vortex_8
vpr_1
vpr_2
vpr_3
vpr_4
vpr_5
vpr_6
Actual
EW
COR
PCA
GA

Bibliography

[1] M. Annavaram, R. Rakvic, M. Polito, J. Bouguet, R. Hankins, and B. Davies. "The fuzzy correlation between code and performance predictability". In Proceedings of the 37th International Symposium on Microarchitecture (MICRO), pages 93-104, Dec.2004

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

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

[4] R. Bell, Jr., R.R Bhatia,. L.K. John, J. Stuecheli, J. Griswell, P. Tu, L. Capps, A. Blanchard, R. Thai, "Automatic testcase synthesis and performance model validation for high performance PowerPC processors", 2006 International Symposium on Performance Analysis of Systems and Software (ISPASS-06), pages 154-165, March 2006

[5] R. Bell, Jr. and L. K. John, "Improved Automatic Testcase Synthesis for Performance Model Validation", Proceedings of the 19th Annual International conference on Supercomputing (ICS), pages 111-120, 2005

[6] S. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis", The ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA), Portland, OR, pp. 191-208, October 2006.

[7] P. Bose, "Workload characterization: A key aspect of microarchitecture design", IEEE Micro, March-April 2006, Volume 26, pp 5-6.

[8] H. Cain, K. Lepak, B. Schwartz, and M. Lipasti, "Precise and Accurate Processor Simulation", 5th Workshop On Computer Architecture Evaluation Using Commercial Workloads (CAECW), February 2002.

[9] J. Cantin, and M. Hill, "Cache Performance for SPEC CPU2000 Benchmarks," http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/.

[10] D. Citron, "MisSPECulation: Partial and Misleading Use of SPEC CPU2000 in Computer Architecture Conferences," in Proceedings of International Symposium on Computer Architecture, 2003, pp. 52-61.

[11] T. Conte, and W. Hwu, “Benchmark Characterization for Experimental System Evaluation,” in Proceedings of Hawaii International Conference on System Science, vol. I, Architecture Track, pp. 6-18, 1990.

[12] P. Denning, “The Working Set Model for Program Behavior,” Communications of the ACM, vol 2(5), pp. 323-333, 1968.

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

[14] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, “Using PAPI for hardware performance monitoring on Linux Systems” Conference on Linux Clusters: The HPC Revolution, Linux Clusters Institute, June 2001.

[15] P. Dubey, G. Adams, and M. Flynn, “Instruction Window Size Trade-Offs and Characterization of Program Parallelism,” IEEE Transactions on Computers, vol. 43(4), pp. 431-442, 1994.

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

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

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

[19] 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.

[20] R. Giladi and N. Ahituv, “ SPEC as a Performance Evaluation Measure,” IEEE Computer, pp. 33-42, Aug 1995.

[21] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown, “MiBench: A Free, Commercially Representative Embedded Benchmark Suite,” in Proceedings of 4th Annual Workshop on Workload Characterization, 2001.

[22] D. Hammerstrom and E. Davdison, “Information content of CPU memory referencing behavior,” in Proceedings of International Symposium on Computer Architecture, 1997, pp. 184-192.

[23] J. Henning, “Performance Counters and Development of SPEC CPU2006”. ACM SIGARCH Computer Architecture News, March 2007.

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

[25] J. Henning, “SPEC CPU2000: Measuring CPU Performance in the New Millenium”, IEEE Computer, July 2000.

[26] J. Henning, “SPEC CPU2006 Benchmark Descriptions”, ACM SIARCH Computer Architecture News, Vol. 34, No. 4, September 2006.

[27] J. Holland., “Adaptation in natural and artificial systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence”. University of Michigan Press, 1975.

[28] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L.K. John, K.D. Bosschere, “Performance Prediction Based on Inherent Program Similarity”, Proceedings of the 15th international conference on Parallel architectures and compilation techniques (PACT), pages 114-122, 2006.

[29] K. Hoste, L. Eeckhout, “Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics” IEEE International Symposium on Workload Characterization, Oct 2006.

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

[31] 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...................

[32] L. John, V. Reddy, P. Hulina, and L. Coraor, “Program Balance and its impact on High Performance RISC Architecture,” in Proceedings of the International Symposium on High Performance Computer Architecture, pp.370-379, Jan 1995..

[33] A. Joshi, A. Phansalkar, L. Eeckhout, L. John, “Measuring Program Similarity Using Inherent Program Characteristics,” Laboratory of Computer Architecture Technical Report TR-060201, The University of Texas at Austin, February 2006...

[34] A. Joshi, L. Eeckhout, R. H. Bell, Jr, L. K. John, “Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks”, 2006 IEEE International Symposium on Workload Characterization (IISWC), pages 105-115, Oct 2006.

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

[36] 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.

[37] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. “The strong correlation between code signatures and performance.” In Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), Mar.2005.

[38] C. Lee, M. Potkonjak, W.H Mangione-Smith “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems,” in Proceedings Of International Symposium on Microarchitecture, 1997.

[39] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, “PIN:Building Customized Program Analysis Tools with Dynamic Instrumentation”, Proceedings of 2005 ACM SIPLAN Conference on Programming Language Design and Implementation, pp 190-200, 2005 .

[40] Y. Luo, A. Joshi, A. Phansalkar, L.K. John, J. Ghosh, “Analyzing and Omroving Clustering Based Sampling for Microprocessor Simulation” 17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005), pages 193-200, Issue 24-27, Oct 2005.

[41] Mathworks, Matlab, http://www.mathworks.com.

[42] H. McGhan, SPEC CPU2006 Benchmark Suite, Microprocessor Report, October 10, 2006.

[43] N. Mirghafori, M. Jacoby, and D. Patterson, “Truth in SPEC Benchmarks,” Computer Architecture News vol. 23 (5), pp. 34-42, Dec 1995

[44] S. Mukherjee, S. Adve, T. Austin, J. Emer, and P. Magnusson, “Performance Simulation Tools,” IEEE Computer, Feb 2002

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

[46] A. Phansalkar, A. Joshi, L. Eeckhout, L. John, “Measuring Program Similarity – Experiments with SPEC CPU benchmark suites,” in Proceedings of International Symposium on Performance Analysis of Systems and Software, 2005.

[47] A. Phansalkar, A. Joshi, L.K. John, “Analysis of Redundancy and Application Balance in SPEC CPU2006 Benchmark Suite”, International Symposium on Computer Architecture (ISCA), June 2007.

[48] A. Phansalkar, A. Joshi, L.K. John, “Subsetting the SPEC CPU2006 Benchmark Suite”, ACM SIGARCH Computer Architecture News, March 2007.

[49] A. Phansalkar, L.K. John, “Performance Prediction using Program Similarity” 2006 SPEC Benchmark Workshop, 2006.

[50] R. Saveedra and A. Smith, “Analysis of benchmark characteristics and benchmark performance prediction,” in Proceedings of ACM Transactions on Computer Systems, vol. 14 (4), pp. 344-384, 1996.

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

[52] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically Characterizing Large Scale Program Behavior,” in Proceedings of International Conference on Architecture Support for Programming Languages and Operating Systems, 2002, pp. 45-57.

[53] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. “Automatically characterizing large scale program behavior.” In Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), pages 45-57, Oct.2002.

[54] K. Skadron, M. Martonosi, D.August, M.Hill, D.Lilja, and V.Pai. "Challenges in Computer Architecture Evaluation," IEEE Computer, pp. 30-36, Aug.2003.

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

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

[57] SPEC webpage “All published for SPEC CPU2000 Benchmark suite” web page: http://www.spec.org/cpu2000/results/cpu2000.html.

[58] J. Spirn and P. Denning, “Experiments with Program Locality,” The Fall Joint Conference, pp. 611-621, 1972.

[59] Standard Performance Evaluation Corporation (SPEC) website, CPU benchmarks, http://www.spec.org/benchmarks.html.

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

[61] R. Weicker, “An Overview of Common Benchmarks,” IEEE Computer, pp. 65-75, Dec 1990.

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

[63] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” in Proceedings of International Symposium on Computer Architecture, June 1995, pp. 24-36.

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

[65] K. Yeung, W. Ruzzo, “An Empirical Study on Principal Components Analysis for Clustering of Gene Expression Data”, Tech Report UW-CSE-2000-11-03, November 2000.

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

[67] J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D. Lilja, and L. K. John, "Evaluating Benchmark Subsetting Approaches," International Symposium on Workload Characterization, October 2006.

Vita

Aashish Shreedhar Phansalkar was born in Pune, India, on July 1st, 1978 as the son of Shreedhar Vishwanath Phansalkar and Vaishali Shreedhar Phansalkar. He received his Higher Secondary-school Certificate (HSC) from Fergusson College, Pune and his Bachelor of Engineering (BE) degree in Electrical Engineering from the Government College of Engineering, Pune affiliated to the University of Pune, India. After finishing his BE, he entered the Master's program in the Department of Electrical and Computer Engineering at University of Texas at Austin. After receiving his MSE degree at University of Texas at Austin, he joined the PhD program. During the summer of 2004 and 2006 he interned at Intel Corporation in Oregon. He is a student member of IEEE.

Permanent Address:	33, Shantiban Society, Kothrud,
	Pune 411038, Maharashtra
	India

This dissertation was typed by the author.