Understanding the data memory behavior of benchmarks using Principal Components Analysis

by

Saket Kumar, B.E.

Report

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

Master of Science in Engineering

The University of Texas at Austin

May 2004

Understanding the data memory behavior of benchmarks using Principal Components Analysis

APPROVED BY
SUPERVISING COMMITTEE

Dedication

To my family and all my well-wishers who have always shown me the right way, due to whom, I am where I am now.

Acknowledgements

I would like to thank Dr. Lizy Kurian John for her invaluable guidance and advice during the course of this work. I would also like to express my gratitude to Dr. Tony Ambler for agreeing to be the reader for my report. I also want to thank all the members of the Laboratory for Computer Architecture for their invaluable help and encouragement.

May 2004

Understanding the data memory behavior of benchmarks using Principal Components Analysis

by

Saket Kumar, M.S.E

The University of Texas at Austin, 2004

SUPERVISOR: Lizy Kurian John

Minimizing simulation time and hence reducing the time to market is a very important issue in modern microprocessor designs. Benchmark programs take a considerable amount of time running on complex machine simulators. These benchmarks explore different areas of the design space and there is a possibility that some of the benchmarks end up testing the same aspect of the processor design.

For studying program characteristics like data memory behavior, it may not be necessary to run all the benchmark programs from a benchmark suite. We need to find out minimum number of program-input pairs from a benchmark suite, that

represent the whole suite in terms of its behavior. This helps in reducing the simulation time considerably.

The objective of this report is to study data memory behavior of different benchmark programs and find out how clustered or far away, they are in the workload design space. SPECCPU2000 and SPECJVM98 benchmarks are characterized for different cache parameters and their sensitivity to varying cache parameters is studied. A statistical data analysis technique called Principal Components Analysis (PCA) is used to identify the differences.

Table of Contents

List of Figures

List of Tables

1. Introduction

Reducing the simulation time, while running benchmarks during the design of a microprocessor is a very important concern from the time to market perspectives. The design of modern computer systems is based upon the experimental procedure of measuring the running time of different workloads on the machine to be designed.

Various simulation models at different levels of accuracy are created during the design phase of a computer. The models represent the structure and behavior of the microprocessor in various ways. The more detailed the model is, the more accurately it models the machine, at the same time, the longer it takes to simulate a cycle.

A workload could be considered as a benchmark program, given particular inputs. It has to satisfy certain criteria in order to increase the likelihood of a good design. It has to be representative for the target application domain of the system, i.e. it should exhibit similar properties as the applications that will actually be running on the system.

In the current scenario, workloads are continuously evolving to keep pace with the technological improvements. At the same time, they are becoming larger and larger requiring huge amount of simulation time. However, there is a certain amount of redundancy within a benchmark suite as well as within different benchmark suites, which if identified, could help significantly in reducing the simulation time. John et al [9] state that certain benchmark input pairs result in testing the same area in the potential workload domain.

Moreover, when we are studying certain program characteristics like data memory behavior, we need not run the whole benchmark suite with all the input pairs, as many of the program-input pairs incidentally target the same area in the workload design space. It saves a lot of simulation time, if we are able to identify a minimum number of program-input pairs from the benchmark suites, which are non-overlapping; at the same time they explore all the possible areas of the workload space.

By characterizing the benchmarks, the program-input pairs stressing upon a portion of the application space already tested by another program-input pair, can be identified, thereby eliminating the redundancy. Ideally, benchmarks should stress all locations of the design space, thereby evaluating the machine in all aspects.

If we consider a p-dimensional workload space, where each dimension represents one of the p workload characteristics, then each benchmark program-input pair can be mapped as a point, where the coordinates of the point are determined by the p workload characteristics. The benchmark suite corresponds to the cloud of points of its individual programs. Projecting both suites in the p-dimensional space and analyzing their corresponding clouds can analyze the differences in the two workloads. We need to find out, if there are regions containing points of one workload, but not of the other. We also want to find out how diverse one workload is compared to the other.

Considering the large amount of data to inspect, it is going to be very difficult to determine the similarity of data memory behavior from the p workload characteristics. Moreover, many of the workload characteristics are correlated, making it difficult to determine the true cause of the differences between the workloads. We tackle this problem using a statistical data reduction technique called as Principal Components Analysis (PCA) that reduces the dimensionality of the data from p to q ($q << p$), without losing important information.

Reducing the dimensionality makes our analysis much easier and helps in identifying the similarity between benchmark programs and the extent of diversity within a benchmark suite.

In this report, we try to study the data memory behavior of different benchmarks programs and try to find out how clustered or far away, they are in the workload design space. At first, we characterize the miss rates of various benchmarks of SPECJVM98 benchmark suite for different cache configurations and their sensitivity to different cache parameter changes is studied. The analysis is done using PCA. A similar analysis is performed for different program-input pairs of the SPECCPU2000 benchmark suite. Then a combined analysis is performed for the benchmarks of SPECJVM98 and SPECCPU2000 combined together to find out the difference between the two benchmark suites.

An attempt is made to come up with a small subset of benchmarks from each benchmark suite, in order to study the data memory behavior, without compromising upon the extent of coverage of the workload design space.

2. Background and Motivation

2.1 Motivation

Long time back, computers were designed based on intuition and individual experiences. During the last two decades, a more systematic approach has been followed by the micro-architects. Different simulation tools have come into existence and computers are designed based on the results provided by them. However, due to the increasing complexity of the microprocessors and the applications that runs on them, the simulators have become very time consuming and it has become very important to reduce the simulation time.

The memory behavior of programs is often explained using temporal and spatial localities. These characteristics are measured using distributions that make them hard to compare across programs. They are also not capable of predicting the conflict misses.

Memory behavior of workloads can be characterized using different metrics. We have chosen cache miss rate as the performance metric because it corresponds closely to the performance that can be expected with caches, and it is independent of other system parameters.

Moreover, data cache miss rates vary considerably between different programs, making it a very important metric towards characterizing different programs. A lot of studies have been done related to understanding the data memory behavior of different benchmark suites.

In this report, we have used Principal Components Analysis (PCA) to study the data memory behavior of benchmarks within a benchmark suite as well as the differences between two benchmark suites. PCA is a very powerful tool to find out the dependencies between different correlated variables and helps us to come up with a set of uncorrelated variables that can be used to study the behavior of benchmarks in the presence of a large amount of data. We also try to single out the eccentric benchmarks, if any, in the benchmark suites.

2.2 Related Work

John et al [10] have explained the short term and long term goals that can be achieved using workload characterization. In the short term, it can be used to impact the performance tuning of architectures for emerging workloads. It can also lead to tuning of compiler optimizations and application development. In the long term, workload characterization can be used to develop a program behavior model, which can be used along with a processor model to do the analytical performance modeling of computer systems.

Gee et al [8] have studied the cache performance of SPEC92 benchmarks for a variety of cache configurations. They found that the instruction cache miss ratios are generally very low, and that the data cache miss ratios for the integer benchmarks are also quite low. Data cache miss ratios for floating point benchmarks are more in line with the published measurements of real workloads.

Chow et al [2] have used PCA to compare the emerging Java workloads with non-Java workloads. The most significant difference was found in their density of indirect branches. This work showed the effectiveness of using PCA in screening and categorizing workload statistics as well as some interesting patterns of indirect branches of Java workloads.

Eeckhout et al [4] [5] [6] [7] have used PCA to analyze the impact of different inputs on the behavior of programs. They selected a limited set of representative program-input pairs with small dynamic instruction counts. They were able to substantiate their claims by showing that the program-input pairs that are close to each other in the principal components space indeed exhibit similar behavior as a function of micro-architectural changes.

Vandierendonck et al [21] have used PCA to study the data memory behavior of SPECCPU95 and SPECCPU2000 benchmark suites and identified the eccentric and fragile benchmarks present in the two suites. Eccentric benchmarks have a behavior that differs significantly from the other benchmarks present in the suite. Fragile benchmarks are weak benchmarks as their execution time is determined entirely by a single bottleneck. Removing that bottleneck can reduce their execution time to a significant extent.

3. Methodology

In this section, we are going to explain how data memory behavior is characterized, what workload characteristics we are taking into account, which benchmarks are being used, what principal components analysis is, and what the procedure for our experiment is.

3.1 Data memory Characterization

Data memory behavior of a workload can be characterized by its data cache miss rates in a wide range of cache configurations. In order to perform principal components analysis, we convert the measured data cache miss rates into ratios of miss rates by taking the ratio of miss rates in two different cache configurations with one cache parameter varying, others remaining the same.

This helps in a better interpretation of data, because each variable measures the influence of changing one cache parameter while keeping the other parameters intact. This transformation also helps in removing much of the variability between caches with a different size or block size.

The various cache parameters that we take into account are the cache size, associativity, block size, cache replacement policy and the write-back policy.

These parameters are sufficient enough to describe most of the cache configurations in the modern processors [16] [17].

We form 58 workload characteristics (variables), which are simply the ratios of the miss rates, varying one cache parameter at a time. These 58 variables for each of the benchmarks are fed into the PCA.

These variables are tabulated as below.

Variable	Size	Assoc.	Blk Size	Repl. Policy	alloc/non alloc
1	8	1->2	32	LRU	non alloc
2	8	2->4	32	LRU	non alloc
3	8	4->8	32	LRU	non alloc
4	8	1->2	32	Random	non alloc
5	8	2->4	32	Random	non alloc
6	8	4->8	32	Random	non alloc
7	8	2	32	LRU->Random	non alloc
8	8	4	32	LRU->Random	non alloc
9	8	8	32	LRU->Random	non alloc
10	32	1->2	32	LRU	non alloc
11	32	2->4	32	LRU	non alloc
12	32	4->8	32	LRU	non alloc
13	32	1->2	32	Random	non alloc

14	32	2->4	32	Random	non alloc
15	32	4->8	32	Random	non alloc
16	32	2	32	LRU->Random	non alloc
17	32	4	32	LRU->Random	non alloc
18	32	8	32	LRU->Random	non alloc
19	128	1->2	32	LRU	non alloc
20	128	2->4	32	LRU	non alloc
21	128	4->8	32	LRU	non alloc
22	128	1->2	32	Random	non alloc
23	128	2->4	32	Random	non alloc
24	128	4->8	32	Random	non alloc
25	128	2	32	LRU->Random	non alloc
26	128	4	32	LRU->Random	non alloc
27	128	8	32	LRU->Random	non alloc
28	8	2	32->64	LRU	non alloc
29	8	2	64->128	LRU	non alloc
30	32	2	32->64	LRU	non alloc
31	32	2	64->128	LRU	non alloc
32	128	2	32->64	LRU	non alloc
33	128	2	64->128	LRU	non alloc
34	32	1->2	64	LRU	non alloc
35	32	2->4	64	LRU	non alloc
36	32	4->8	64	LRU	non alloc
37	32	1->2	64	Random	non alloc
38	32	2->4	64	Random	non alloc
39	32	4->8	64	Random	non alloc
40	32	2	64	LRU->Random	non alloc
41	32	4	64	LRU->Random	non alloc
42	32	8	64	LRU->Random	non alloc
43	4->16	8	32	LRU	non alloc
44	16->64	8	32	LRU	non alloc
45	64->256	8	32	LRU	non alloc
46	4->16	8	32	Random	non alloc
47	16->64	8	32	Random	non alloc
48	64->256	8	32	Random	non alloc
49	8	2	64	LRU	non alloc->alloc

50	8	2	128	LRU	non alloc->alloc
51	32	2	64	LRU	non alloc->alloc
52	32	2	128	LRU	non alloc->alloc
53	128	2	64	LRU	non alloc->alloc
54	128	2	128	LRU	non alloc->alloc
55	32	1	32	LRU	non alloc->alloc
56	32	2	32	LRU	non alloc->alloc
57	32	4	32	LRU	non alloc->alloc
58	32	8	32	LRU	non alloc->alloc

Table 1: The different workload characteristics

As can be seen from the table 1, variables 1-3, 10-12 and 19-21 measure the impact of associativity changes for 8KB, 32KB and 128KB caches with a block size of 32 bytes and LRU replacement policy, respectively. Variables 4-6, 13-15 and 22-24 do the same for caches with random replacement policy. Variables 34-39 do the same for a 32KB cache with a block size of 64 bytes.

Variables 7-9, 16-18, 25-27 and 40-42 correspond to cache replacement policy changes for cache sizes of 8KB, 32KB and 128KB respectively for different associativities.

Variables 28-33 measure the impact of block size changes in 8KB, 32KB and 128KB caches with a degree of associativity of 2 and having LRU replacement policy.

Variables 43-45 account for the impact of cache size changes in caches having a degree of associativity of 8 and LRU replacement. Variables 46-48 do the same for random replacement policy.

Variables 49-58 study the impact of sensitivity to write back policies in different cache configurations with LRU replacement policy.

3.2 Description of the Benchmarks

3.2.1 SPEC CPU2000

The SPEC CPU2000 benchmark suite is a collection of 26 computation-intensive, non-trivial programs used to evaluate the performance of a computer's CPU, memory system, and compilers. The benchmarks in this suite were chosen to represent real-world applications, and thus exhibit a wide range of runtime behaviors. The integer benchmarks are written in C & C++, while the floating-point benchmarks are mostly in Fortran.

The different integer benchmarks used for our experiment are as follows:

1. 164.gzip: gzip (GNU zip) is a popular data compression program that uses Lempel-Ziv coding as its compression algorithm.

2. 175.vpr: VPR is a placement and routing program. It automatically implements a technology-mapped circuit (i.e. a netlist, or hypergraph, composed of FPGA logic blocks and I/O pads and their required connections) in a Field-Programmable Gate Array (FPGA) chip.

3. 176.gcc: 176.gcc is based on gcc Version 2.7.2.2. It generates code for a Motorola 88100 processor. The benchmark runs as a compiler with many of its optimization flags enabled.

4. 181.mcf: A benchmark derived from a program used for single-depot vehicle scheduling in public mass transportation. The program is written in C and the benchmark version uses almost exclusively integer arithmetic.

5. 186.crafty: Crafty is a high-performance Computer Chess program that is designed around a 64-bit word. It runs on 32-bit machines using the "long long" data type. It is primarily an integer code, with a significant number of logical operations such as and, or, exclusive or and shift.

6. 197.parser: The Link Grammar Parser is a syntactic parser of English, based on link grammar, an original theory of English syntax.

7. 252.eon: Eon is a probabilistic ray tracer. It sends a number of 3D lines (rays) into a 3D polygonal model. Intersections between the lines and the polygons are computed, and new lines are generated to compute light incident at these intersection points.

8. 253.perlbmk: 253.perlbmk is a cut-down version of Perl v5.005_03, the popular scripting language.

9. 254.gap: It implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming).

10. 255.vortex: VORTEX is a single-user object-oriented database transaction benchmark, which exercises a system kernel coded in integer C.

11. 256.bzip2: 256.bzip2 is based on Julian Seward's bzip2 version 0.1. The only difference between bzip2 0.1 and 256.bzip2 is that SPEC's version of bzip2 performs no file I/O other than reading the input. All compression and decompression happens entirely in memory. This is to help isolate the work done to only the CPU and memory subsystem.

12. 300.twolf: The TimberWolfSC placement and global routing package is used in the process of creating the lithography artwork needed for the production of microchips. Specifically, it determines the placement and global connections for groups of transistors (known as standard cells), which constitute the microchip.

3.2.2 SPECJVM98

The SPECJVM98 benchmark suite basically measures the performance of Java Virtual Machines. Most of the programs are real-world applications with high demand on the memory system. The various Java benchmark programs used for our experiment are as follows:

1. _201_compress: It is similar to 164.zip and uses modified Lempel-Ziv method. It basically finds common substrings and replaces them with a variable size code. This is deterministic, and can be done on the fly.

2. _209_db: It performs multiple database functions on memory resident database. It reads in a 1 MB file, which contains records with names, addresses and phone numbers of entities and a 19KB file called scr6, which contains a stream of operations to perform on the records in the file.

3. _213_javac: This is the Java compiler from the JDK 1.0.2.

4. _222_mpegaudio: This is an application that decompresses audio files that conform to the ISO MPEG Layer-3 audio specification.

5. _227_mtrt: This is a raytracer program that works on a scene depicting a dinosaur, where two threads each renders the scene in the input file time-test model, which is 340KB in size.

6. _202_jess: JESS is the Java Expert Shell System, based on NASA's CLIPS expert shell system. The benchmark workload solves a set of puzzles commonly used with CLIPS.

7. _228_jack: It is a Java parser generator that is based on the Purdue Compiler Construction Tool Set.

3.3 Principal Components Analysis (PCA)

Appendix A shows the data cache miss rates for 7 SPECJVM98 benchmarks for different data cache configurations. As we can see from there, there is a huge chunk of data and it is not an easy task to interpret it and draw some meaningful conclusions regarding the sensitivity of the benchmarks to different cache parameter changes. Moreover, there is a large correlation between the variables, if we transform these miss rates into 58 variables as described in section 3.1.

In order to interpret such a large amount of data and make some meaningful conclusion from it, we need to reduce the number of variables to be analyzed from such a large value to a much smaller number, which could be easily interpreted using 2-dimensional plots. Principal Components Analysis helps us achieve that, without losing much of the information.

Principal components analysis is a multi-variate data analysis technique that reduces the dimensionality of a data set consisting of strongly correlated variables, to a set of uncorrelated variables called as Principal Components.

Since the principal components are uncorrelated, each one makes an original contribution towards accounting for the variance of the original variables.

The principal components are arranged in decreasing order of their variance. It is often found that the first few principal components account for most of the information present in the original data set. This helps in reducing the dimensionality of the data and makes the analysis simpler with smaller set of variables.

The p original variables, X_i, i = 1 to p are linearly transformed into p principal components, Z_i, i = 1 to p. The principal components are constructed such that Z_1 has the maximum variance and then Z_2 is chosen such that it has the maximum variance under the constraint that it is not correlated to Z_1. The same procedure is followed to form the other principal components. Consequently, the principal components are arranged in the order of decreasing variance and are uncorrelated, i.e. the covariance between one principal component and the other is equal to zero. Covariance is a measure of the extent to which the deviations of two variables match.

The geometrical properties of principal components can be elucidated by some two dimensional figures. Let us assume that we have a sample of observations on two standardized variables X1 and X2. We can use X1 and X2 as coordinate axes and plot the standardized variables as in figure 1.

From the shape of the scatterplot, we can see that there is a substantial correlation between X1 and X2. There are two variables, and if the variables are not perfectly correlated, two principal components are required to completely account for the variation in the two variables. The first principal component is a new coordinate axis in the variable space which is oriented in a direction that maximizes the variation of the projections of the points on the new coordinate axis, the first principal component Z1 (Figure 2). Since the second principal component Z2 is not correlated with Z1, it is orthogonal to Z1.

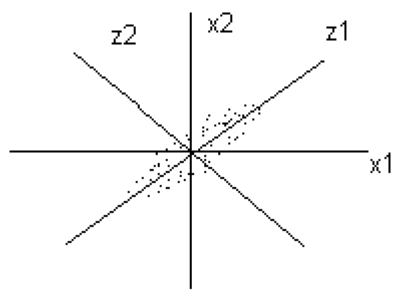

Figure 1: Scatter Plot of two standardized variables

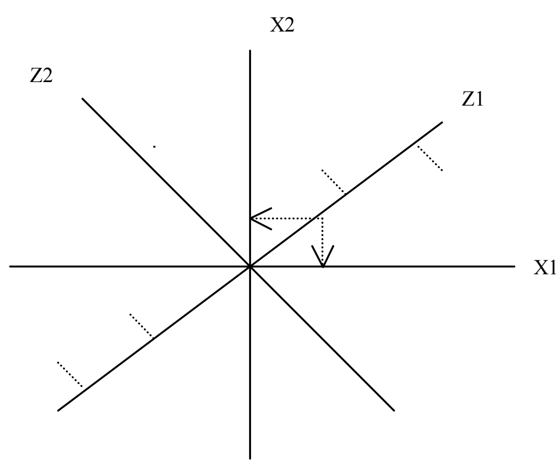

Figure 2: One-dimensional representation by largest principal components of two dimensional data

Retaining only those principal components that have the maximum variance brings down the dimensionality of the data set. The number of retained principal components, depends upon what fraction of the variance in the original data set, we want to explain.

It is advisable to standardize the variables before applying principal components analysis. By standardizing, we mean that the variables are rescaled such that they have zero mean and unit variance. This ensures that the variable having higher variance doesn't have higher impact on the first few principal components.

The main idea behind reducing the dimensionality is that, by having say $q = 3$ or 4 variables, makes it much easier to understand the differences between the benchmarks, compared to the case when the benchmarks can differ in say $p = 50$ different ways.

If q is small, the user can visualize the reduced space by means of a scatter plot that shows the position of each benchmark with respect to the principal components. The eccentricity of the benchmarks with respect to the analyzed benchmark suite determines their position on the scatter plot. Benchmarks that are close to the origin of the q-dimensional space are average benchmarks, i.e. when one of the parameters is changed, the benchmark will see a change similar to the average over the entire suite. Benchmarks that are far away from the origin are very sensitive to the changes in the parameter.

Factor loadings are used to determine the parameters that play an important role in each principal component. Naturally, only a few parameters play an important role in each principal component. The factor loadings are the coefficients aij in the linear combination, $Z_i = \sum_{j=1}^{p} aijXj$. The larger a_{ij} is in magnitude, the stronger it influences the principal component. The closer it is to zero, the lesser or nil impact it has on the principal component. Thus, the benchmarks with large values of X_j will score positively on Z_i when a_{ij} is positive, while those that have small values for X_j will score negatively.

Principal components analysis can also be used to judge the impact of the input on a program as well. The inputs usually have a small impact, when their workload characteristics do not differ much, while the programs are affected much by the inputs, if they are widely separated in the scatter plots.

Consequently, these program-input pairs will be close to each other in the original p-dimensional space as well in the q-dimensional space of the principal components. It is also possible to find groups of benchmarks that are internally close, but externally distant from other clusters. It can be said that inputs have little effect on the behavior of the program, if all the instances of the same program run on different inputs are in the same cluster.

Principal components analysis can also be used to compare benchmark suites. It can be used to find out whether two benchmark suites differ significantly depending upon their relative positions of their benchmarks in the scatter plot. When the benchmark suites behave entirely different, they will occupy disjoint areas in the q-dimensional space of principal components.

In reality, it can be expected that the benchmarks overlap, thereby a few benchmarks exhibiting similar behavior. When a region of space contains benchmarks from only one suite, then those benchmarks are characteristically different from the benchmarks in the other suite that are not present in that region.

3.4 Procedure

The SPECCPU2000 and SPECJVM98 benchmarks were run on SUN machines for different cache configurations. We used shade-analyzer's cache simulator '*cachesim5*' for measuring the data cache miss rates. The configuration of the Instruction Cache was fixed to be of 8KB, 32 bytes block size, direct mapped cache.

The miss rates obtained for the various different cache configurations was transformed into ratios of miss rates and we obtained 58 different variables for each benchmark.

Each variable was normalized to have a zero mean and unit variance. Then principal components analysis was performed on the data, delivering 58 (if number of benchmarks > 58 else equal to number of benchmarks) uncorrelated principal components sorted in the order of decreasing variance.

The eigenvalues and the fraction of variance contained in all the principal components are calculated. A proper choice of the number of principal components, q to be retained is made based on the percentage of variance of the actual data that we want to retain.

The benchmarks are plotted in the q-dimensional space with first q principal components as the axes. The plots and the factor loadings are analyzed and they are used to figure out the differences in behavior of different benchmark programs.

4. Results

This section summarizes the results of the study that characterizes the SPECJVM98 and SPECCPU2000 benchmark programs in terms of their data memory behavior. At first, we analyze the SPECJVM98 and SPECCPU2000 benchmark suites using PCA in sections 4.1 and 4.2 respectively. Then we perform the combined analysis of SPECJVM98 and SPECCPU2000 programs taken together. The results of the combined analysis are discussed in section 4.3. Section 4.4 makes an attempt towards selecting a subset of benchmark programs from both the benchmark suites for study of data memory behavior.

4.1 Analysis of SPECJVM98

SPECJVM98 benchmarks were run on Sun machines for different data cache configurations and the miss rates were obtained using Shade Analyzer's cachesim. The benchmarks were run for 1 billion instructions after skipping 400 million instructions at the beginning.

58 variables were formed using the procedure given in the methodology section. Principal components analysis was performed on the 58 variables,

describing the data memory behavior, for seven of the SPECJVM98 benchmarks. Table 2 shows the percentage of variance accounted by each of the seven Principal Components (PCs). The Eigen value of a principal component reflects the amount of variance it accounts for.

	Eigen Value	**%Variance**	**Cumulative %**
PC1	18.460	37.133	37.133
PC2	13.326	26.805	63.938
PC3	10.427	20.973	84.911
PC4	4.461	8.973	93.884
PC5	2.279	4.583	98.467
PC6	0.762	1.533	100.000
PC7	0.000	0.000	100.000

Table 2: Fraction of total variance explained by the PCs (SPECJVM98)

As seen from the table, the principal components are ordered in decreasing amount of variance and the first principal component, PC1 accounts for 37% of the total variance. It can also be seen that the first 3 principal components account for almost 85% of the total variance. So, we can explain the 85% of the variance present in the original 58 variables with the first 3 principal components.

We can exclude the other components from the analysis because they include comparatively much less information and are relatively harder to interpret.

Table 3 shows the factor loadings for the all the 58 variables corresponding to the first 3 principal components. In this analysis, we look for weights having an absolute value greater than 0.15 and they have been displayed in bold. The impact of the variables with smaller weights is ignored in the explanation of the principal components.

X_i	PC1	PC2	PC3	X_i	PC1	PC2	PC3
1	0.001	0.092	**-0.168**	30	0.018	-0.140	**0.162**
2	-0.010	**-0.188**	**-0.178**	31	0.051	**-0.171**	**0.195**
3	-0.114	**-0.171**	0.034	32	-0.106	**-0.161**	0.119
4	0.008	0.111	-0.129	33	-0.104	**-0.179**	0.137
5	-0.064	**-0.207**	-0.053	34	0.028	**-0.228**	-0.055
6	0.069	-0.107	**-0.239**	35	-0.114	**-0.204**	-0.044
7	0.015	0.002	0.148	36	-0.127	**-0.191**	-0.073
8	-0.030	0.003	**0.249**	37	-0.005	**-0.234**	0.049
9	0.112	0.086	0.094	38	-0.111	**-0.204**	-0.030
10	0.138	**-0.161**	-0.046	39	-0.030	-0.106	**0.243**
11	-0.144	**-0.175**	0.061	40	-0.081	0.008	**0.253**
12	0.052	-0.096	**0.219**	41	-0.071	0.011	**0.260**
13	-0.083	**-0.212**	-0.016	42	-0.028	0.018	**0.273**
14	**-0.171**	-0.142	-0.041	43	0.140	**-0.150**	-0.025
15	-0.082	-0.038	**0.252**	44	0.123	0.036	0.180
16	**0.209**	-0.011	-0.035	45	**0.171**	0.122	0.016
17	**-0.212**	0.002	-0.006	46	0.148	-0.136	-0.022
18	**-0.210**	0.002	0.049	47	**0.162**	0.025	0.132
19	-0.116	-0.081	0.107	48	**0.171**	0.121	0.010
20	**0.207**	0.014	0.076	49	0.146	-0.099	-0.030
21	**0.211**	0.020	0.055	50	**0.174**	-0.055	-0.050
22	**-0.190**	-0.057	0.050	51	0.142	**-0.165**	-0.066
23	**0.212**	0.010	0.049	52	**0.153**	**-0.156**	-0.051
24	**0.170**	-0.020	**0.167**	53	0.119	**-0.204**	-0.032
25	**-0.205**	-0.003	-0.045	54	0.116	**-0.207**	-0.038
26	**-0.171**	-0.018	**-0.156**	55	0.138	**-0.157**	-0.068
27	**-0.208**	-0.039	-0.056	56	0.126	**-0.179**	-0.062
28	0.116	-0.027	**0.210**	57	0.116	**-0.188**	-0.070
29	0.043	-0.115	**0.248**	58	0.111	**-0.191**	-0.075

Table 3: Factor Loadings (SPECJVM98)

The factor loadings help us finding out what each principal component correspond to.

As can be seen from table 3, PC1 scores heavily for variables X43 to X48, which account for cache size variation. It can also be seen that PC1 has high factor loadings for variables X16 to X18 and X19 to X24, that correspond to cache replacement policy changes and associativity changes respectively. So, we can conclude that PC1 doesn't represent a single variable; rather it accounts for cache-size, replacement policy and associativity variations.

PC2 has high factor loadings for the variables X51 to X58, which correspond to cache allocate/non-allocate policies. So, PC2 measures the impact of changing from non-allocate to allocate caches.

PC3 scores high for variables X28 to X33, which measure the impact of increasing the block size. Hence we can conclude that PC3 primarily measures the spatial locality of the benchmarks.

The workload space can be visualized by means of scatter plots. The scatter plots show that the different benchmarks have different sensitivity to the cache parameters. As we have retained 3 principal components in our analysis, we can have three possible plots, viz. PC1 vs. PC2, PC1 vs. PC3, PC2 vs. PC3.

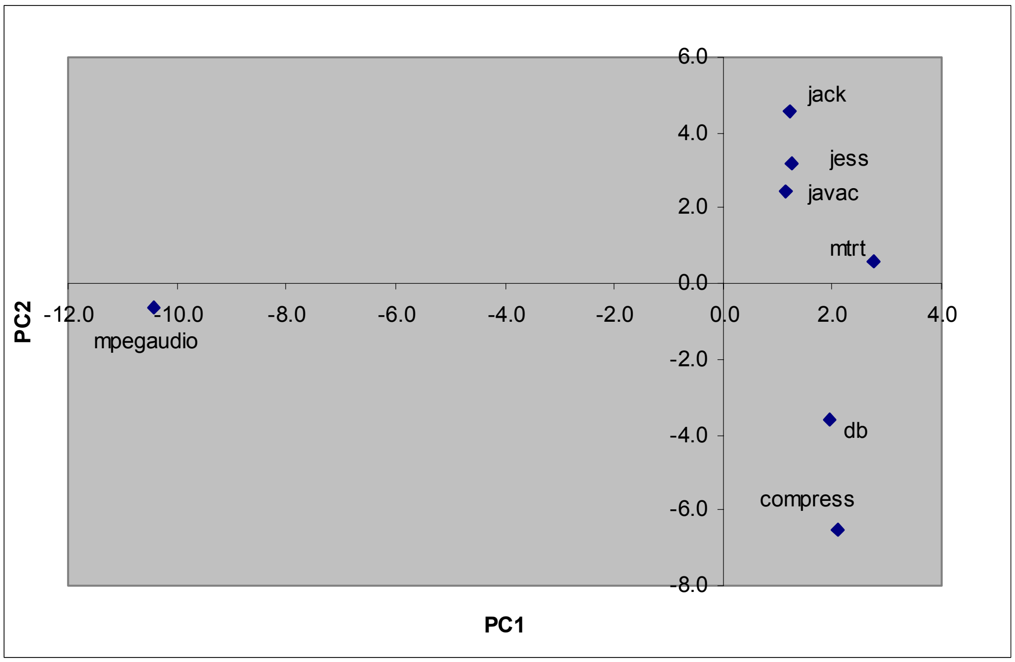

Figure 3: Scatter Plot of PC1 vs. PC2 (SPECJVM98)

Figure 3 shows the scatter plot between the first two principal components that account for 64% of the total variance contained between the 58 variables. As can be seen from the plot, mpegaudio is far away from the other benchmarks, that means it is much more distinct that the other benchmarks in the suite, if we are considering their data memory behavior.

The benchmarks having positive value of PC2, i.e. jack, jess, javac and mtrt are benefited by write allocate caches while mpegaudio, compress and db are favored by no write-allocate caches.

Having larger size data–cache, benefits the benchmarks that have positive values of PC1. That means all the benchmarks except mpegaudio; perform well with larger sized data caches. Mpegaudio is very much sensitive to cache-size and associativity variations.

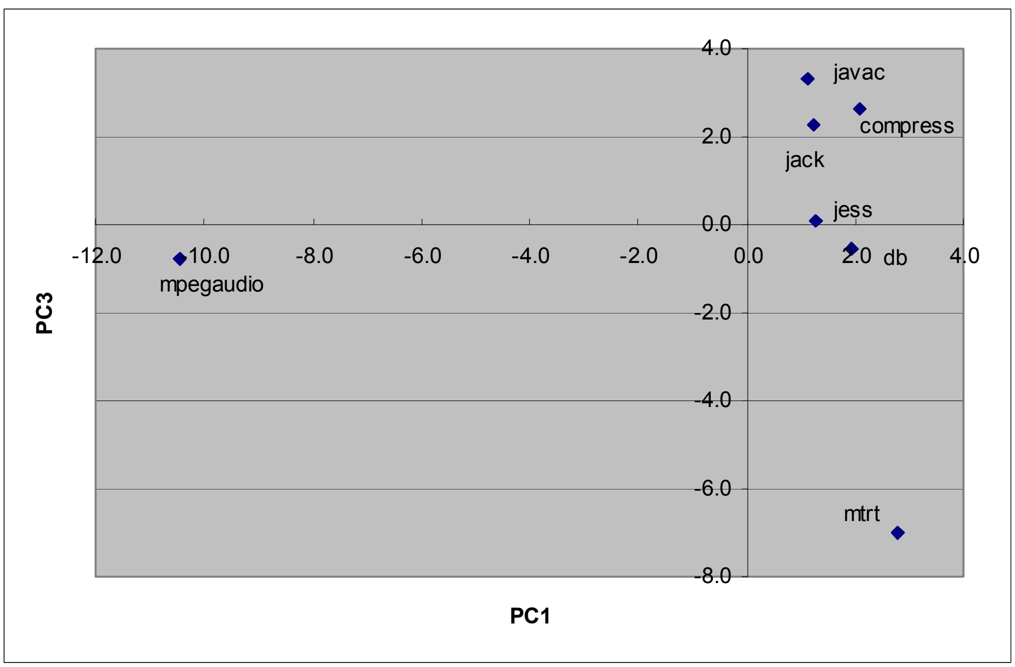

Figure 4: Scatter Plot of PC1 vs. PC3 (SPECJVM98)

Figure 4 shows the scatter plot of PC1 vs. PC3. As PC3 accounts for sensitivity of the benchmarks towards block-size variations, we can see that jess, db and mpegaudio, that have very low value of PC3, are almost insensitive to block-size variations. Larger block sized caches benefit javac, compress and jack while mtrt performs well for smaller block sizes.

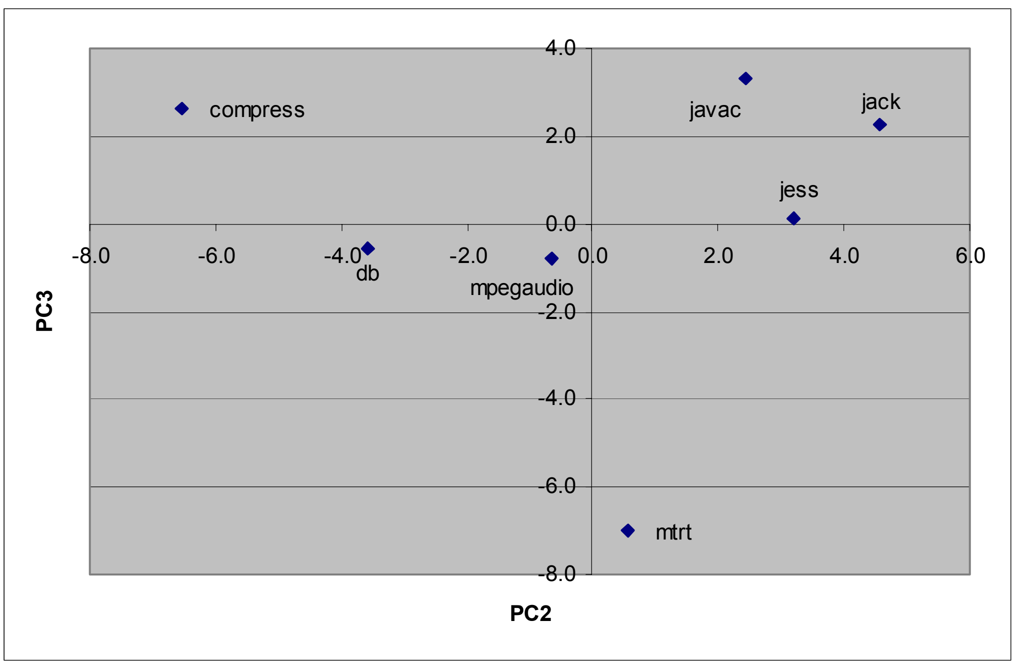

Figure 5: Scatter Plot of PC2 vs. PC3 (SPECJVM98)

Figure 5 shows the scatter plot between PC2 and PC3. Here, we can see that all the benchmarks appear to be scattered far apart. Since PC2 and PC3 account for lesser variance than PC1, so we can conclude that mpegaudio is far distinct from the other benchmarks in the suite as it scores highly on PC1.

Eccentric Benchmarks: It can be concluded from the scatter plots that mpegaudio is an eccentric benchmark as it has a relatively much larger absolute value of PC1 compared to the other benchmarks in the suite. It is very much sensitive to data cache size variations. Compress and mtrt are somewhat eccentric as they have relatively distinct values of PC2 and PC3, with respect to the other members of the benchmark suite.

4.2 Analysis of SPECCPU2000

SPECCPU2000 benchmarks were run on Sun machines for different data cache configurations and the miss rates were obtained using Shade Analyzer's cachesim. The benchmarks were run for 1 billion instructions after skipping 1 billion instructions at the beginning for initialization.

58 variables were formed using the procedure given in the same way as done for the java benchmarks. Principal components analysis was performed on the 58 variables, describing the data memory behavior, for 12 of the

SPECCPU2000 benchmarks for 33 program-input pairs. Table 4 shows the percentage of variance accounted by the first six Principal Components (PCs). As stated earlier, the Eigen value of a principal component reflects the amount of variance it accounts for.

	Eigen Value	%Variance	Cumulative %
PC1	23.46431	41.72	41.72
PC2	9.40455	16.72	58.44
PC3	6.10782	10.86	69.30
PC4	4.33553	7.71	77.01
PC5	2.99607	5.33	82.34
PC6	2.36802	4.21	86.55

Table 4: Fraction of total variance explained by the PCs (SPECCPU2000)

The Principal Components are ordered in decreasing amount of variance and the first principal component, PC1 accounts for nearly 42% of the total variance. It can also be seen that the first 4 principal components account for almost 77% of the total variance. So, we can explain the 77% of the variance present in the original 58 variables with the first 4 principal components.

We exclude the other principal components as they include much lesser information and therefore their exclusion wouldn't affect the analysis much. Thus, we have reduced the dimensionality of the data from 58 to 4 variables, without losing much of the variability in the data.

The factor loadings for the all the 58 variables corresponding to the first 4 principal components are shown in table 5. All the weights having an absolute value greater than 0.15 are significant in the analysis and they have been marked bold.

X_i	PC1	PC2	PC3	PC4	X_i	PC1	PC2	PC3	PC4
1	0.093	**-0.177**	**0.237**	-0.040	30	-0.065	0.089	**0.230**	**-0.191**
2	0.006	**-0.256**	0.127	0.145	31	-0.146	0.045	-0.015	**-0.162**
3	0.029	**-0.193**	0.002	-0.030	32	0.017	**0.158**	**0.194**	-0.048
4	0.082	-0.143	**0.274**	-0.037	33	-0.062	**0.169**	-0.020	**0.183**
5	0.007	-0.116	0.002	**0.391**	34	**0.180**	-0.057	0.105	0.043
6	0.082	0.033	-0.027	0.038	35	**0.177**	0.049	-0.096	-0.005
7	-0.088	**0.201**	-0.072	0.015	36	**0.167**	0.106	-0.087	0.013
8	-0.057	**0.249**	-0.144	0.098	37	**0.186**	-0.029	0.010	0.092
9	-0.035	**0.270**	-0.118	0.077	38	**0.172**	0.093	-0.090	-0.041
10	**0.188**	-0.049	-0.048	0.107	39	**0.164**	0.063	-0.081	-0.142
11	**0.187**	0.044	-0.100	-0.003	40	-0.077	0.075	**-0.318**	0.129
12	**0.184**	-0.009	-0.066	-0.081	41	-0.105	0.133	**-0.240**	0.016
13	**0.180**	-0.047	-0.082	0.116	42	-0.149	0.043	**-0.176**	-0.132
14	**0.181**	0.062	-0.097	-0.047	43	**0.175**	0.013	-0.094	0.143
15	**0.160**	-0.007	-0.089	-0.140	44	**0.183**	-0.075	-0.096	-0.025
16	**-0.183**	0.036	-0.049	-0.005	45	-0.065	-0.109	-0.060	-0.147
17	**-0.188**	0.015	0.046	0.001	46	**0.179**	0.013	-0.079	0.141
18	**-0.182**	0.006	0.031	0.030	47	**0.183**	-0.094	-0.078	-0.031
19	**0.186**	-0.065	-0.078	-0.101	48	0.014	-0.098	-0.074	**-0.222**
20	**0.163**	-0.096	-0.064	**-0.177**	49	0.105	**0.184**	-0.101	0.005
21	0.059	**-0.172**	-0.022	-0.005	50	0.082	**0.221**	-0.049	-0.138
22	**0.176**	-0.077	-0.087	-0.138	51	0.107	**0.186**	**0.169**	0.036
23	**0.156**	-0.061	-0.075	**-0.252**	52	0.058	**0.203**	0.140	-0.094
24	0.040	-0.138	-0.026	-0.066	53	0.120	**0.185**	0.130	0.113
25	**-0.161**	-0.037	0.050	**0.207**	54	0.078	**0.226**	0.094	0.014
26	-0.099	0.114	-0.016	**-0.273**	55	0.125	**0.155**	**0.205**	-0.028
27	-0.092	0.072	-0.031	**-0.329**	56	0.120	**0.183**	**0.190**	0.050
28	-0.053	0.095	**0.177**	**-0.247**	57	0.120	**0.202**	**0.165**	0.076
29	-0.019	0.066	**-0.356**	0.027	58	0.124	**0.204**	**0.156**	0.075

Table 5: Factor Loadings (SPECCPU2000)

The factor loadings help us to analyze what cache parameter variation, each principal component represents.

As we can see from table 5, PC1 has high factor loadings for variables X43 to X48, which corresponds to cache size variation. PC1 scores highly for the variables X10 to X15 and X34 to X39, which correspond to cache associativity variation. Hence PC1 accounts for the sensitivity of the benchmarks to cache size and associativity variations.

PC2 has high value of factor loadings for the variables X49 to X58, which correspond to cache write allocate/non-allocate policies. So, PC2 measures the impact of changing from non-allocate to allocate caches.

PC3 scores high for variables X28 to X30, which correspond to the sensitivity of the benchmarks to block size variations of the data cache. Hence it primarily measures the spatial locality of the various program-input pair of the SPECCPU2000 benchmark suite.

PC4 has high value of factor loadings for variables X25 to X27, which correspond to the sensitivity of the benchmarks to changes in cache replacement policy from LRU to random.

As we have retained the first four principal components in our analysis, we can have a maximum possible of 6, 2-dimensional scatter plots. The scatter plots between PC1 vs. PC2, PC1 vs. PC3, PC1 vs. PC4 and PC3 vs. PC4 are discussed in our analysis.

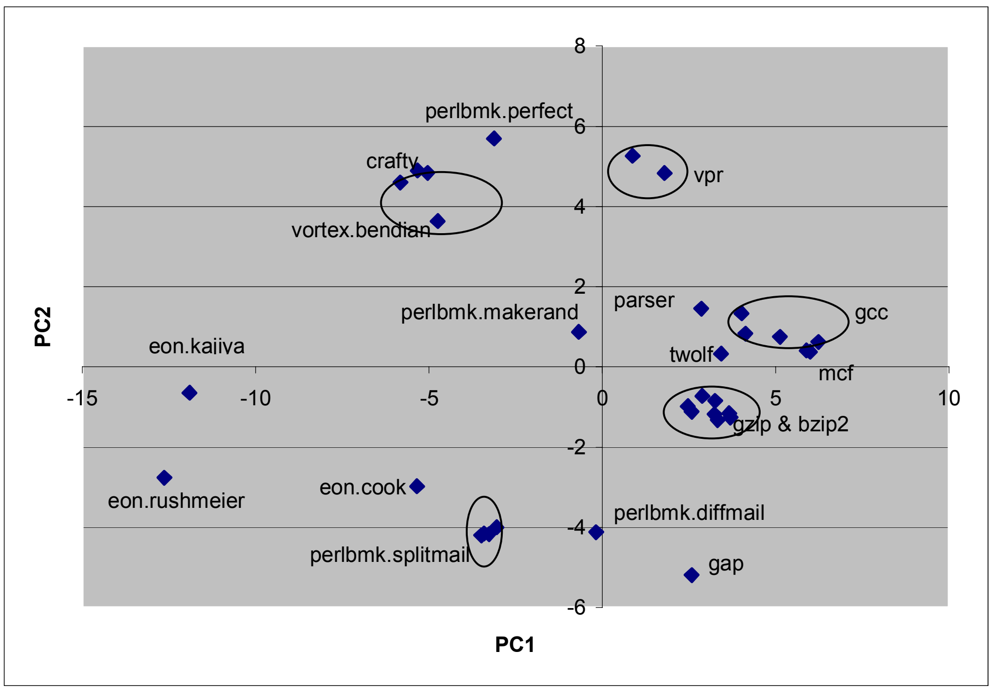

Figure 6: Scatter Plot of PC1 vs. PC2 (SPECCPU2000)

Figure 6 shows the scatter plot between the first two principal components. PC1 and PC2 account for 58% of the total variance contained in the data. As we can see here, the benchmarks in the SPECCPU2000 suite are much more scattered compared to those in the SPECJVM98 suite. That means that SPECCPU2000 benchmarks represent a much more diversified set of programs, if we intend to study the data memory behavior.

It can also be observed that the different inputs corresponding to the same benchmark program have almost similar behavior except for the perlbmk benchmark whose different program-input pairs are scattered.

The other interesting behavior that can be seen is the close proximity of gzip and bzip2 program-input pairs. As we can see here, the different program-input pairs of the two benchmarks are closely clustered. That implies that the two benchmark programs have similar data memory behavior. The same can be said about gcc and mcf.

The benchmarks vpr, parser, twolf, gcc, mcf, crafty, vortex and some program input pairs of perlbmk (perfect and makerand) and eon (kajiya) have a positive value of PC2, that means that they are benefited by write allocate caches. The others are favored by non-allocate policy in the data cache.

The benchmarks that have positive values of PC1 are benefited by having larger data cache with higher associativities. The benchmarks vpr, parser, twolf, gcc, mcf, gzip, bzip2 and gap perform well with larger data caches. For others, it is advisable to have smaller cache size.

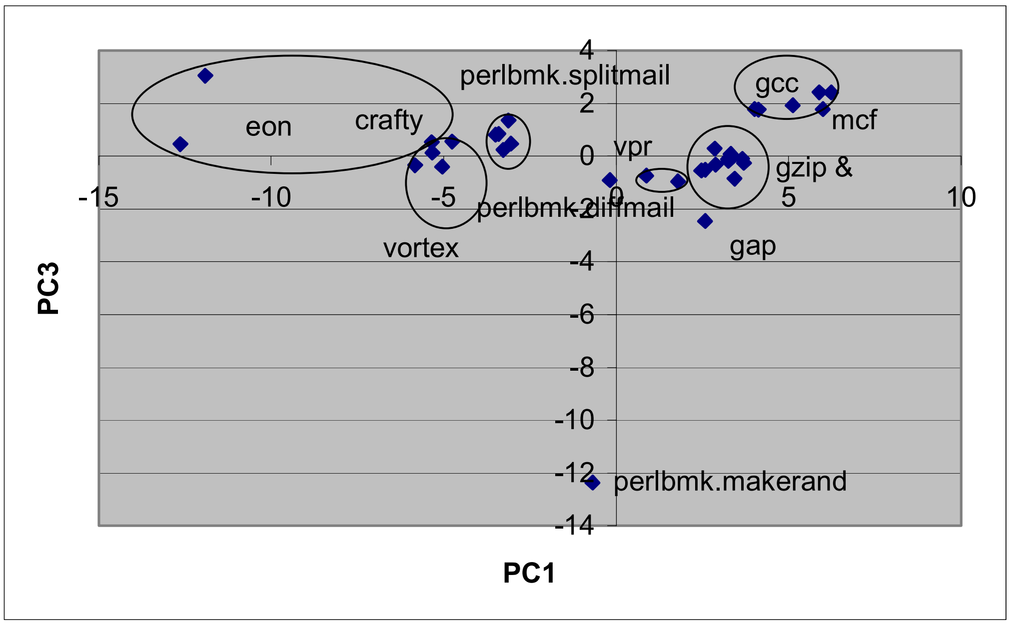

Figure 7: Scatter Plot of PC1 vs. PC3 (SPECCPU2000)

Figure 7 shows the scatter plot of PC1 vs. PC3 for the various program-input pairs of the SPECCPU2000 benchmark suite. Since PC3 corresponds to the sensitivity of the benchmarks to block-size variations, we can see that makerand input of the perlbmk benchmark program, having high values of PC3, is very sensitive to block-size variations. The other benchmarks are relatively less sensitive to block-size variations. The different input pairs of the same benchmark are seen to be clustered together except for perlbmk and eon to some extent.

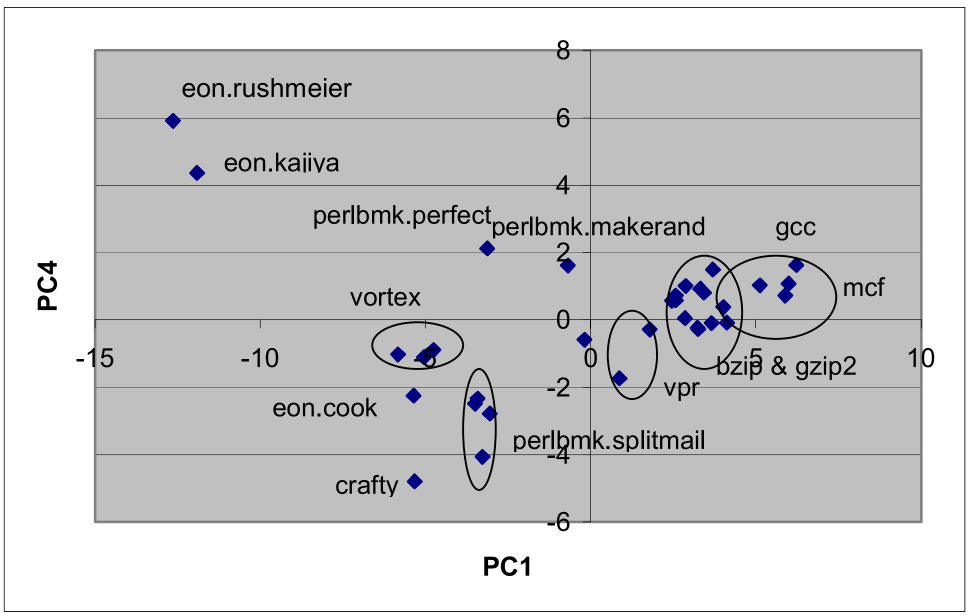

Figure 8: Scatter Plot of PC1 vs. PC4 (SPECCPU2000)

Figure 8 shows the scatter plot between PC1 and PC4. As PC4 corresponds to the sensitivity of the benchmarks to the change in the cache replacement policy from LRU to Random, we can see that gcc, mcf, bzip2, gzip and vpr benefit from having a random replacement policy while vortex, eon, crafty, and perlbmk benefit from having least recently used (LRU) replacement algorithm.

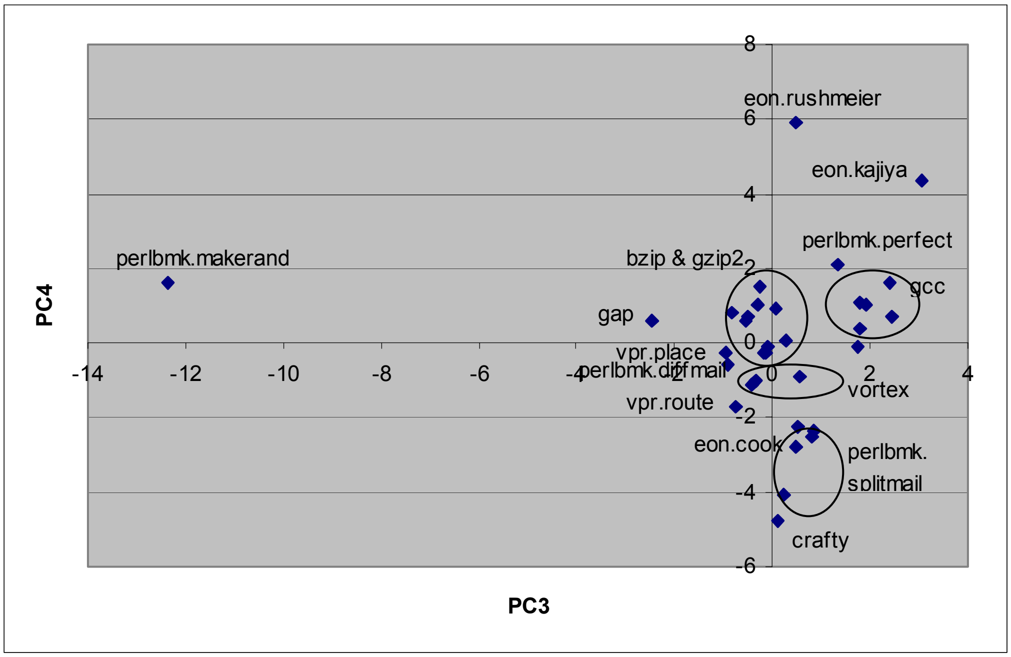

Figure 9: Scatter Plot of PC3 vs. PC4 (SPECCPU2000)

Figure 9 shows the scatter plot between PC3 and PC4 that account for 19% of the variance within them. As seen in the earlier plots, the benchmarks seem to be clustered. The different program input pairs of the same benchmark behave similar for data cache parameter variation. From all these plots, it can be seen that various program-input pairs of gzip and bzip2 have almost identical data memory behavior. The same can be said about gcc and mcf.

Eccentric Benchmarks: Eon is an eccentric benchmark as it is very sensitive to cache size variations, hence has very distinct value of PC1 compared to the other program-input pairs. Vpr, crafty and some inputs of perlbmk also exhibit somewhat eccentric behavior as their response is much affected by cache write allocate policies compared to the other benchmark programs.

4.3 Combined Analysis of SPECCPU2000 & SPECJVM98

We have analyzed the data memory behavior of SPECJVM98 and SPECCPU2000 benchmark programs in the earlier two sections. Now, we perform the same analysis for all the benchmark programs of SPECJVM98 and SPECCPU2000, combined together.

Using the procedure given in the methodology section, we formed 58 variables for 12 of the SPECCPU2000 benchmarks for 33 different program-input pairs and 7 SPECJVM98 benchmark programs. So, we have 40 different program-input pairs for the analysis. Principal components analysis was performed on those 58 variables. Table 6 shows the percentage of variance accounted by the first six principal components. As mentioned earlier, the Eigen value of a principal component reflects the amount of variance present in the total data, it accounts for.

	Eigen Value	%Variance	*Cumulative %*
PC1	22.798	40.31	40.31
PC2	9.688	17.13	57.44
PC3	6.035	10.67	68.11
PC4	4.190	7.41	75.52
PC5	3.564	6.30	81.82
PC6	2.424	4.29	86.11

Table 6: Fraction of total variance explained by the PCs (SPECCPU2000 & SPECJVM98 combined)

As we can see from the table, first principal component PC1 accounts for nearly 40% of the total variance contained in the data. The first 4 principal components account for almost 76% of the total variance. So, we can drop in the other principal components and consider the first 4 components for our analysis.

So, we can explain the 76% of the variance present in the original 58 variables with the first 4 principal components.

As explained earlier as well, the principal components are arranged in decreasing order of their variance. By excluding the lower principal components, we wouldn't be losing much of the variability in the data, but our analysis becomes much simpler as we are reduced to 4 variables from original 58 variables.

The factor loadings for the all the 58 variables corresponding to the first 4 principal components are shown in table 7. As done in the earlier analyses, all the weights having an absolute value greater than 0.15 are marked as bold and they are considered to be significant in the analysis.

X_i	PC1	PC2	PC3	PC4	X_i	PC1	PC2	PC3	PC4
1	0.099	**-0.153**	**0.255**	-0.038	30	-0.071	0.084	**0.197**	**-0.206**
2	0.016	**-0.231**	**0.167**	**0.157**	31	-0.149	0.041	-0.012	**-0.156**
3	0.030	**-0.151**	0.060	-0.004	32	0.015	0.135	0.139	-0.059
4	0.086	-0.124	**0.280**	-0.040	33	-0.064	0.138	-0.060	**0.179**
5	0.015	-0.110	0.020	**0.399**	34	**0.185**	-0.047	0.094	0.039
6	0.081	0.048	-0.006	0.055	35	**0.180**	0.047	-0.103	-0.001
7	-0.097	**0.176**	-0.107	0.003	36	**0.169**	0.094	-0.110	0.010
8	-0.067	**0.221**	**-0.184**	0.089	37	**0.190**	-0.019	0.006	0.090
9	-0.044	**0.240**	**-0.163**	0.066	38	**0.174**	0.087	-0.106	-0.039
10	**0.194**	-0.039	-0.043	0.106	39	**0.167**	0.060	-0.097	-0.145
11	**0.190**	0.041	-0.108	-0.001	40	-0.083	0.069	**-0.298**	0.138
12	**0.188**	0.000	-0.061	-0.075	41	-0.113	0.121	**-0.231**	0.021
13	**0.185**	-0.041	-0.076	0.118	42	-0.153	0.044	-0.149	-0.120
14	**0.183**	0.057	-0.107	-0.042	43	**0.180**	0.019	-0.090	0.143
15	**0.163**	0.003	-0.086	-0.138	44	**0.188**	-0.064	-0.081	-0.023
16	**-0.186**	0.022	-0.048	0.003	45	-0.049	-0.120	-0.048	**-0.150**
17	**-0.193**	0.008	0.048	0.008	46	**0.183**	0.019	-0.077	0.140
18	**-0.186**	-0.001	0.032	0.032	47	**0.188**	-0.078	-0.058	-0.030
19	**0.191**	-0.053	-0.065	-0.093	48	0.028	-0.106	-0.065	**-0.218**
20	**0.169**	-0.069	-0.038	**-0.177**	49	0.084	**0.214**	-0.071	0.015
21	0.065	-0.135	0.023	-0.007	50	0.067	**0.235**	-0.044	-0.132
22	**0.182**	-0.064	-0.071	-0.127	51	0.072	**0.226**	**0.183**	0.040
23	**0.162**	-0.039	-0.056	**-0.252**	52	0.027	**0.230**	**0.153**	-0.070
24	0.047	-0.120	0.003	-0.064	53	0.088	**0.224**	**0.146**	0.112
25	**-0.165**	-0.041	0.059	**0.210**	54	0.045	**0.250**	0.112	0.025
26	-0.107	0.100	-0.031	**-0.269**	55	0.095	**0.202**	**0.216**	-0.020
27	-0.097	0.058	-0.042	**-0.322**	56	0.089	**0.223**	**0.197**	0.052
28	-0.059	0.098	0.149	**-0.260**	57	0.093	**0.237**	**0.169**	0.076
29	-0.021	0.060	**-0.342**	0.038	58	0.097	**0.238**	**0.160**	0.076

Table 7: Factor Loadings (SPECCPU2000 & SPECJVM98 combined)

As can be seen from table 7, PC1 has high factor loadings for variables X34 to X39 and X10 to X15 that implies that PC1 reflects the sensitivity of the benchmarks to associativity variation. PC1 also scores highly for X43 to X44 and X46 to X47 that correspond to data cache size increase. Hence, PC1 accounts for the sensitivity of the benchmarks to the associativity and size of the data cache variations.

The factor loadings for the variables X49 to X58 are pretty high for PC2. As these variables correspond to cache allocate/non-allocate policies, PC2 measures the impact of changing from non-allocate to allocate policy in data cache.

PC3 scores high for variables X28 to X30, which correspond to the sensitivity of the benchmarks to block size variations of the data cache. Hence it primarily measures the spatial locality of the various program-input pairs of the SPECCPU2000 and SPECJVM98 benchmark suites.

PC4 has high value of factor loadings for variables X25 to X27, which correspond to the sensitivity of the benchmarks, while the cache replacement policy is changed from LRU to Random.

Now that we have retained first 4 principal components out of total 40 components, we can have 6 scatter plots possible. As used in the earlier analysis, we will consider scatter plots between PC1 vs. PC2, PC1 vs. PC3, PC1 vs. PC4 and PC3 vs. PC4.

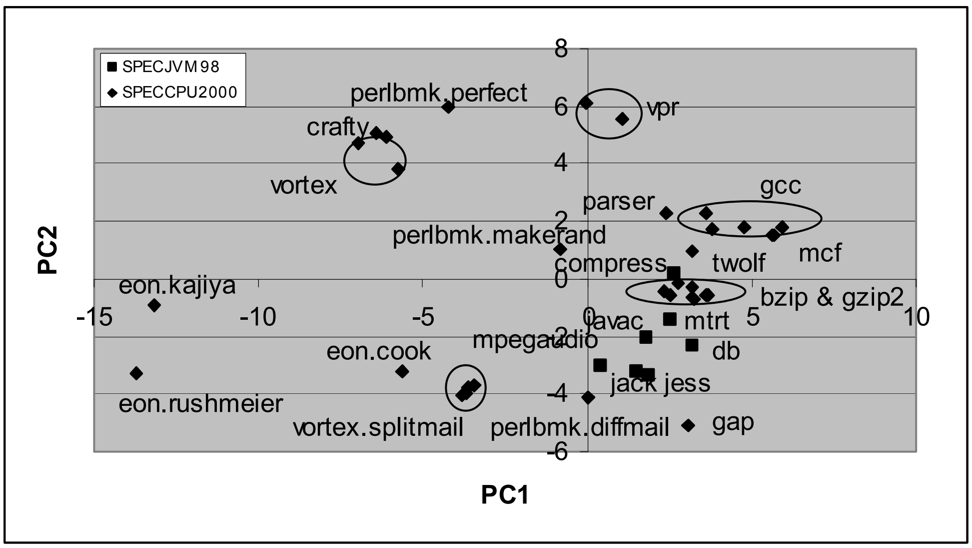

Figure 10: Scatter Plot of PC1 vs. PC2 (SPECCPU2000 & SPECJVM98 combined)

Figure 10 shows the scatter plot between the first two principal components that account for 57% of the total variability contained in the data. We can clearly see the extent of diversity in the SPECCPU2000 benchmarks with respect to SPECJVM98 benchmark programs, while we are studying the data memory behavior. The SPECJVM98 benchmark programs are clustered within a small area in the two dimensional space of PC1 and PC2.

As observed in the analysis of SPECCPU2000, here also, we can see that the different inputs corresponding to the same benchmark program exhibit similar behavior. Of course, perlbmk is an exception, as it is scattered all around the space.

The various program-input pairs of bzip2 and gzip can be seen to be closely clustered. That means that the two benchmarks are similar in terms of their dealings with the data cache. The different input pairs of gcc behave quite similar to the mcf benchmark present in the SPECCPU2000 benchmark suite.

As PC1 corresponds to the sensitivity of the benchmarks to data cache associativity and size, we can see that all the SPECJVM98 benchmarks perform better for increasing cache size and associativities. Amongst the SPECCPU2000 benchmarks, vpr, parser, gcc, mcf, twolf, bzip2, gzip and gap have positive value of PC1, implying their likings for larger data cache. The other programs perform better if the cache size is reduced.

Since PC2 reflects the sensitivity of a benchmark while changing from non-allocate to allocate cache write policy, all the programs having positive value of PC2 perform better when write allocate policy is incorporated in the data cache. Others behave just the opposite.

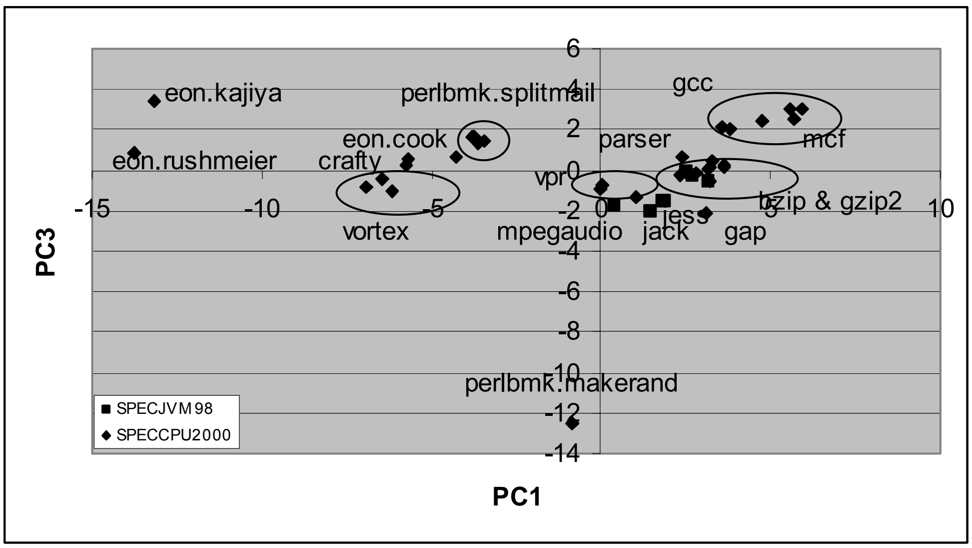

Figure 11: Scatter Plot of PC1 vs. PC3 (SPECCPU2000 & SPECJVM98 combined)

Figure 11 shows the scatter plot between PC1 and PC3. As PC3 corresponds to the sensitivity of the benchmarks to increase in block size, all the benchmarks having positive value of PC3 are supportive of increasing block size while the ones having negative value of PC3 perform better when the block size is decreased. Here also, we can observe the close proximities of the different input pairs of the same benchmark.

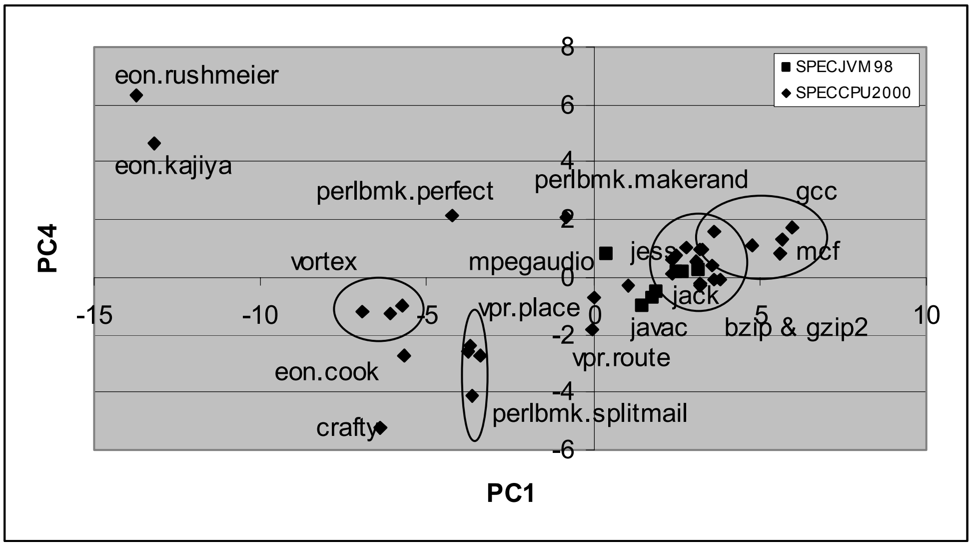

Figure 12: Scatter Plot of PC1 vs. PC4 (SPECCPU2000 & SPECJVM98 combined)

Figure 12 shows the scatter plot between PC1 and PC4. Since PC4 corresponds to the response of a benchmark when the cache replacement algorithm changes from LRU to Random, all the benchmarks having positive value of PC4 are supportive of Random replacement algorithm while the others perform better with LRU algorithm.

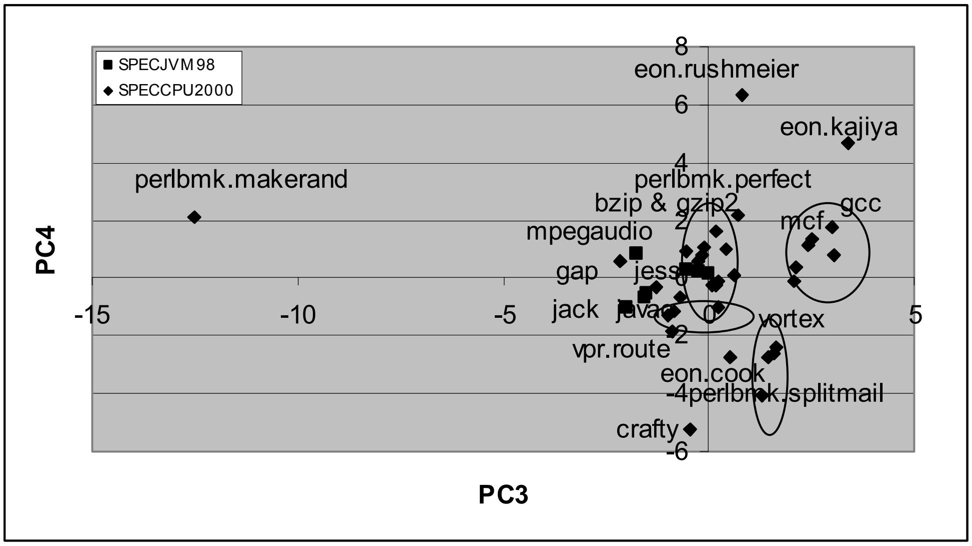

Figure 13: Scatter Plot of PC3 vs. PC4 (SPECCPU2000 & SPECJVM98 combined)

PC3 and PC4 account for almost 18% of the variance contained in the data. The scatter plot of PC3 vs. PC4 is shown in Figure 13. Here also, we can see the clustering of the java benchmarks as well as the clustering of different program inputs of the same benchmark program.

4.4 Selecting a subset of benchmarks for studying data memory behavior

SPECJVM98: In order to stress the data cache of a machine using java programs, we need not run all the programs of the SPECJVM98 benchmark suite. Based upon the scatter plots obtained using PCA, we would select mpegaudio, compress, mtrt and javac.

SPECCPU2000: We observed the similarity in the data memory behavior of different inputs of the same program. Perlbmk was an exception in this case as it behaved differently for different inputs applied to it.

We also observed that bzip2 and gzip behaved similar to each other and they were clustered together in all the scatter plots. So, we could choose either of the two with one input set.

We need not run mcf, if are studying the data cache miss rates. Its behavior is much similar to that of different program-input pairs of gcc.

Since SPECCPU2000 is a very diversified benchmark suite, we can chose one program-input pair for each benchmark program, taking into account the above observations. Perlbmk needs to be run with different input combinations as its data memory behavior is much dependent upon the input set.

SPECCPU2000 & SPECJVM98: As we observed from the combined scatter plots, SPECCPU2000 is a much more diversified benchmark suite as compared to SPECJVM98. When we are studying the data memory behavior of a machine, SPECCPU2000 benchmarks cover a much wider area in the 4-dimensional space of the principal components compared to the SPECJVM98 benchmarks. Moreover, there are some or other benchmark programs of the SPECCPU2000 benchmark suite that are very close in behavior to the SPECJVM98 benchmarks. So, we can conclude that SPECCPU2000 benchmarks are much more diverse as compared to SPECJVM98 programs and they are able to measure the same workload characteristics that the java benchmarks measure.

The clustering of the Java benchmarks can be attributed to the fact that, here the properties of the Java Virtual Machine (JVM) are dominating instead of the actual program. The Java compiler converts the Java code into bytecodes and puts them into a ".class" file. This ".class" file can be interpreted on any machine that has a Java Virtual Machine on it. The JVM processes each of the bytecodes and executes them. When a JIT (Just In Time) compiler is present, it takes the bytecodes and compiles them into the native code for the machine that we are running the program upon. It can actually be faster to grab the bytecodes, compile them, and run the resulting executable than it is to interpret them. So, in the presence of JIT compilation, the properties of the Java Virtual Machine dominate and it is actually the behavior of the JVM rather than the benchmarks that is reflected, when the different benchmarks are characterized. Hence, all the Java benchmarks appear to be closely clustered.

5. Conclusion

We studied and analyzed the data memory behavior of the SPECJVM98 and SPECCPU2000 benchmark suites using principal components analysis. We studied the sensitivity of the benchmark programs of the two suites towards changes in the various data cache parameters.

We also studied the effect of different inputs applied to the same benchmark program for SPECCPU2000. We performed the analysis for the benchmarks of the two benchmark suites taken together to carve out the differences between them.

We found that different inputs to the same program behave almost identical, in terms of study of the data memory behavior. This helps us to conclude that we need not run all the program-input pairs for the same program, when we are studying the data cache behavior. We also found out that perlbmk is sensitive to input variations and its inputs are very diverse compared to other program-input pairs.

We also found that certain benchmarks have almost identical sensitivities to changes in data cache parameters. Our experiment helped us to find out such benchmark pairs. This can help us in eliminating the redundancy existing between different benchmark programs and help us chose a minimum number of benchmark programs to be run on a machine to explore the entire workload space while studying data memory. This can help us considerably in reducing the simulation time spent on running the benchmarks and hence the time to market.

This analysis also helped us figure out some of the eccentric benchmarks in the two benchmark suites. An eccentric benchmark has a significantly different behavior with respect to the other benchmarks. They are useful when constructing benchmark suites, as it is possible to obtain a large coverage of the behavior space with a few eccentric benchmarks. It is important to include them whenever we are sub-setting a benchmark suite in order to reduce the simulation time.

Appendix A

Data Cache Miss Rates for SPECJVM98 programs for different cache configurations.

	compress	jess	db	javac	mpgaudio	mtrt	jack
8Kb32s1wt	11.626	21.774	21.146	16.006	11.247	16.004	19.358
8Kb32s2rlruwt	9.014	18.881	19.214	13.552	9.650	14.238	16.530
8Kb32s4rlruwt	8.738	17.972	18.900	12.747	9.334	13.937	15.382
8Kb32s8rlruwt	8.656	17.586	18.567	12.388	9.306	13.396	14.565
8Kb32s1wt	11.626	21.774	21.146	16.006	11.247	16.004	19.358
8Kb32s2rrandomwt	9.618	19.624	19.890	14.833	10.124	14.900	17.315
8Kb32s4rrandomwt	9.522	19.051	19.852	14.095	10.021	14.443	16.563
8Kb32s8rrandomwt	9.424	18.751	19.660	13.707	9.812	14.567	16.089
32Kb32s1wt	7.969	17.018	18.457	12.133	6.890	11.552	14.110
32Kb32s2rlruwt	7.462	15.268	17.729	10.709	5.959	10.617	12.638
32Kb32s4rlruwt	7.368	14.777	17.346	10.341	5.919	10.219	12.219
32Kb32s8rlruwt	7.346	14.662	17.279	10.232	5.832	9.990	12.113
32Kb32s1wt	7.969	17.018	18.457	12.133	6.890	11.552	14.110
32Kb32s2rrandomwt	7.696	15.614	17.982	11.208	6.679	10.816	13.121
32Kb32s4rrandomwt	7.654	15.247	17.916	10.905	6.818	10.623	12.809
32Kb32s8rrandomwt	7.650	15.128	17.821	10.896	6.829	10.141	12.733
128Kb32s1wt	5.982	14.263	16.917	9.749	3.240	8.762	12.477
128Kb32s2rlruwt	5.678	13.395	16.551	9.439	3.169	8.129	11.761
128Kb32s4rlruwt	5.641	13.174	16.246	9.332	2.499	7.764	11.648
128Kb32s8rlruwt	5.625	13.102	16.135	9.298	2.322	7.691	11.620
128Kb32s1wt	5.982	14.263	16.917	9.749	3.240	8.762	12.477
128Kb32s2rrandomwt	5.780	13.614	16.678	9.603	3.352	8.299	11.945
128Kb32s4rrandomwt	5.774	13.457	16.599	9.549	2.671	8.137	11.885
128Kb32s8rrandomwt	5.789	13.408	16.626	9.561	2.569	7.958	11.861
8Kb64s2rlruwt	9.369	17.756	17.213	12.927	7.983	12.124	16.430
8Kb128s2rlruwt	10.020	17.485	17.588	13.399	7.821	11.304	16.579

32Kb64s2rlruwt	7.530	13.848	15.588	9.316	5.299	8.947	11.447
32Kb128s2rlruwt	7.704	12.201	14.820	8.501	4.648	7.416	10.446
128Kb64s2rlruwt	5.672	11.838	14.524	8.032	3.066	6.643	10.209
128Kb128s2rlruwt	5.679	10.281	13.737	7.012	3.026	5.299	8.920
32Kb64s1wt	8.116	16.206	16.534	10.945	5.995	10.014	13.273
32Kb64s2rlruwt	7.530	13.848	15.588	9.316	5.299	8.947	11.447
32Kb64s4rlruwt	7.399	12.987	15.250	8.893	5.266	8.622	10.752
32Kb64s8rlruwt	7.362	12.807	15.118	8.753	5.258	8.536	10.562
32Kb64s1wt	8.116	16.206	16.534	10.945	5.995	10.014	13.273
32Kb64s2rrandomwt	7.811	14.190	15.947	9.805	5.518	8.943	11.846
32Kb64s4rrandomwt	7.755	13.451	15.841	9.473	5.541	8.693	11.289
32Kb64s8rrandomwt	7.748	13.299	15.761	9.457	5.498	8.474	11.149
4Kb32s8rlruwt	9.545	20.110	19.583	14.841	10.961	15.630	19.431
16Kb32s8rlruwt	7.958	15.690	17.848	11.004	6.875	12.121	12.837
64Kb32s8rlruwt	6.607	13.786	16.701	9.738	4.698	8.524	11.822
256Kb32s8rlruwt	4.279	12.601	15.553	8.903	2.091	7.256	11.456
4Kb32s8rrandomwt	10.787	22.013	21.572	17.051	12.880	16.954	20.790
16Kb32s8rrandomwt	8.465	16.532	18.662	11.972	7.837	12.519	13.845
64Kb32s8rrandomwt	6.815	14.138	17.171	10.096	4.616	8.828	12.179
256Kb32s8rrandomwt	4.466	12.903	15.957	9.137	2.164	7.544	11.615
8Kb64s2rlruwbwa	5.680	7.507	7.658	6.537	2.878	6.974	7.486
8Kb128s2rlruwbwa	6.170	9.039	9.272	7.527	3.443	7.083	9.067
32Kb64s2rlruwbwa	3.962	3.198	5.929	2.830	0.643	4.171	2.141
32Kb128s2rlruwbwa	4.173	3.086	5.722	2.700	0.536	3.454	2.301
128Kb64s2rlruwbwa	2.224	1.049	4.832	1.463	0.165	1.807	0.756
128Kb128s2rlruwbwa	2.249	0.819	4.350	1.065	0.130	1.422	0.550
32Kb32s1wbwa	4.349	4.990	7.748	4.760	1.367	5.904	3.226
32Kb32s2rlruwbwa	3.919	3.367	7.008	3.504	0.943	4.855	2.182
32Kb32s4rlruwbwa	3.828	2.834	6.826	3.119	0.900	4.600	1.732
32Kb32s8rlruwbwa	3.809	2.632	6.772	2.975	0.882	4.511	1.560

Appendix B

Data Cache Miss Rates for SPECCPU2000 programs for different cache configurations.

	gap	parser	twolf	vpr.place	vpr.route	mcf	crafty
8Kb32s1wt	3.341	12.857	22.008	19.150	8.549	37.756	12.055
8Kb32s2rlruwt	2.959	10.828	19.194	12.023	5.278	37.381	9.255
8Kb32s4rlruwt	2.850	9.987	17.960	10.288	4.400	37.349	7.242
8Kb32s8rlruwt	2.843	9.810	17.237	9.631	4.213	37.317	6.274
8Kb32s1wt	3.341	12.857	22.008	19.150	8.549	37.756	12.055
8Kb32s2rrandomwt	3.141	11.659	21.113	14.278	5.989	37.810	9.955
8Kb32s4rrandomwt	3.085	11.051	20.990	13.329	5.386	38.393	8.710
8Kb32s8rrandomwt	3.058	10.930	20.812	13.224	5.239	38.100	8.318
32Kb32s1wt	3.022	8.336	17.095	8.942	4.266	35.322	4.354
32Kb32s2rlruwt	2.846	7.115	15.845	7.596	3.019	35.128	2.238
32Kb32s4rlruwt	2.842	6.818	15.628	7.351	2.818	35.156	1.614
32Kb32s8rlruwt	2.842	6.695	15.506	7.271	2.760	35.371	1.293
32Kb32s1wt	3.022	8.336	17.095	8.942	4.266	35.322	4.354
32Kb32s2rrandomwt	2.940	7.463	16.783	8.460	3.343	35.268	2.548
32Kb32s4rrandomwt	2.927	7.235	16.704	8.420	3.223	35.308	2.118
32Kb32s8rrandomwt	2.924	7.154	16.658	8.343	3.197	35.369	2.031
128Kb32s1wt	2.888	5.201	13.655	6.211	2.751	32.711	1.605
128Kb32s2rlruwt	2.835	4.656	13.076	5.795	2.254	31.840	1.077
128Kb32s4rlruwt	2.833	4.508	12.906	5.631	2.184	31.457	0.764
128Kb32s8rlruwt	2.830	4.469	12.880	5.587	2.161	31.303	0.740
128Kb32s1wt	2.888	5.201	13.655	6.211	2.751	32.711	1.605
128Kb32s2rrandomwt	2.873	4.845	13.422	6.119	2.394	32.286	1.161
128Kb32s4rrandomwt	2.868	4.763	13.386	6.025	2.354	32.104	1.064
128Kb32s8rrandomwt	2.867	4.745	13.390	6.011	2.341	32.068	1.039
8Kb64s2rlruwt	2.278	9.328	14.585	13.228	6.428	28.548	12.538
8Kb128s2rlruwt	1.749	9.310	12.630	16.179	9.435	23.521	15.216

32Kb64s2rlruwt	2.094	5.574	10.989	7.168	2.831	27.195	2.920
32Kb128s2rlruwt	1.298	4.871	8.421	7.357	3.176	22.366	3.909
128Kb64s2rlruwt	2.084	3.342	8.739	5.368	1.881	23.972	1.152
128Kb128s2rlruwt	1.276	2.732	6.309	5.145	1.735	20.004	1.292
32Kb64s1wt	2.298	6.986	12.341	8.780	4.570	27.462	5.414
32Kb64s2rlruwt	2.094	5.574	10.989	7.168	2.831	27.195	2.920
32Kb64s4rlruwt	2.088	5.268	10.572	6.864	2.509	27.148	2.125
32Kb64s8rlruwt	2.088	5.139	10.525	6.773	2.434	27.053	1.656
32Kb64s1wt	2.298	6.986	12.341	8.780	4.570	27.462	5.414
32Kb64s2rrandomwt	2.184	5.929	11.854	8.225	3.208	27.421	3.244
32Kb64s4rrandomwt	2.170	5.671	11.684	8.211	2.988	27.442	2.758
32Kb64s8rrandomwt	2.167	5.581	11.612	8.118	2.934	27.456	2.575
4Kb32s8rlruwt	2.852	11.833	22.108	13.611	5.455	37.829	13.527
16Kb32s8rlruwt	2.842	8.091	16.175	8.048	3.312	36.629	2.769
64Kb32s8rlruwt	2.842	5.484	14.538	6.501	2.421	32.286	0.868
256Kb32s8rlruwt	2.828	3.494	10.153	4.354	1.928	30.743	0.673
4Kb32s8rrandomwt	3.280	13.744	25.190	18.244	7.484	39.035	15.215
16Kb32s8rrandomwt	2.971	8.793	18.309	10.059	3.966	37.215	4.003
64Kb32s8rrandomwt	2.894	5.813	15.210	7.142	2.697	33.410	1.299
256Kb32s8rrandomwt	2.851	3.812	10.704	4.710	2.040	31.115	0.888
8Kb64s2rlruwbwa	0.457	7.553	9.754	9.633	5.945	26.142	9.903
8Kb128s2rlruwbwa	0.500	7.593	9.686	10.970	8.856	18.312	12.989
32Kb64s2rlruwbwa	0.268	4.142	7.004	5.780	2.409	25.200	1.743
32Kb128s2rlruwbwa	0.143	3.539	6.037	5.744	2.731	17.452	2.692
128Kb64s2rlruwbwa	0.262	2.179	5.350	4.167	1.469	22.718	0.336
128Kb128s2rlruwbwa	0.132	1.642	4.343	3.934	1.307	15.944	0.465
32Kb32s1wbwa	0.636	6.537	9.629	7.273	3.821	33.155	2.780
32Kb32s2rlruwbwa	0.529	5.463	8.630	6.261	2.619	33.043	1.168
32Kb32s4rlruwbwa	0.527	5.204	8.489	6.068	2.422	33.214	0.740
32Kb32s8rlruwbwa	0.527	5.099	8.467	5.996	2.364	33.512	0.603

Data Cache Miss Rates (Contd.)

	gzip.src	gzip.log	gzip.gra	gzip.rand	gzip.prog
8Kb32s1wt	11.537	7.249	15.114	18.848	13.300
8Kb32s2rlruwt	9.894	6.404	13.809	16.590	11.291
8Kb32s4rlruwt	9.636	6.288	13.622	16.449	10.874
8Kb32s8rlruwt	9.577	6.254	13.581	16.383	10.783
8Kb32s1wt	11.537	7.249	15.114	18.848	13.300
8Kb32s2rrandomwt	10.329	6.645	14.222	18.409	11.852
8Kb32s4rrandomwt	10.152	6.562	14.105	18.285	11.561
8Kb32s8rrandomwt	10.120	6.536	14.071	18.251	11.479
32Kb32s1wt	8.269	5.493	12.300	14.705	9.463
32Kb32s2rlruwt	7.342	5.144	11.501	13.732	8.393
32Kb32s4rlruwt	7.256	5.148	11.526	13.714	8.216
32Kb32s8rlruwt	7.252	5.154	11.546	13.722	8.174
32Kb32s1wt	8.269	5.493	12.300	14.705	9.463
32Kb32s2rrandomwt	7.760	5.388	11.852	14.708	8.796
32Kb32s4rrandomwt	7.746	5.404	11.897	14.537	8.779
32Kb32s8rrandomwt	7.732	5.394	11.901	14.682	8.724
128Kb32s1wt	4.900	3.372	6.357	7.421	5.353
128Kb32s2rlruwt	4.411	3.399	5.953	7.066	4.826
128Kb32s4rlruwt	4.594	3.562	6.370	7.468	5.018
128Kb32s8rlruwt	4.702	3.620	6.430	7.496	5.128
128Kb32s1wt	4.900	3.372	6.357	7.421	5.353
128Kb32s2rrandomwt	4.617	3.423	6.202	7.391	5.056
128Kb32s4rrandomwt	4.696	3.450	6.578	7.699	5.153
128Kb32s8rrandomwt	4.714	3.466	6.663	7.754	5.201
8Kb64s2rlruwt	9.775	5.750	13.608	16.312	11.227
8Kb128s2rlruwt	9.908	5.476	13.761	16.206	11.521
32Kb64s2rlruwt	6.417	4.041	10.536	12.586	7.429
32Kb128s2rlruwt	6.189	3.629	10.410	12.466	6.989

128Kb64s2rlruwt	3.462	2.612	4.771	5.777	3.798
128Kb128s2rlruwt	3.072	2.243	4.526	5.556	3.372
32Kb64s1wt	7.586	4.567	11.645	13.750	8.668
32Kb64s2rlruwt	6.417	4.041	10.536	12.586	7.429
32Kb64s4rlruwt	6.343	4.056	10.586	12.577	7.211
32Kb64s8rlruwt	6.385	4.077	10.590	12.595	7.206
32Kb64s1wt	7.586	4.567	11.645	13.750	8.668
32Kb64s2rrandomwt	6.941	4.379	11.055	13.560	7.884
32Kb64s4rrandomwt	6.951	4.416	11.168	13.707	7.925
32Kb64s8rrandomwt	6.949	4.418	11.198	13.854	7.886
4Kb32s8rlruwt	10.399	6.619	13.983	17.216	11.851
16Kb32s8rlruwt	8.709	5.900	13.087	15.657	9.756
64Kb32s8rlruwt	6.020	4.561	9.717	11.571	6.769
256Kb32s8rlruwt	2.316	1.603	2.592	3.737	2.466
4Kb32s8rrandomwt	11.361	7.161	14.979	20.007	13.033
16Kb32s8rrandomwt	8.971	5.996	13.158	16.644	10.134
64Kb32s8rrandomwt	6.405	4.664	9.954	11.947	7.185
256Kb32s8rrandomwt	2.575	1.929	2.603	4.083	2.627
8Kb64s2rlruwbwa	5.707	3.357	7.227	8.638	6.612
8Kb128s2rlruwbwa	6.107	3.051	7.678	8.801	7.440
32Kb64s2rlruwbwa	3.276	2.371	5.524	6.719	3.674
32Kb128s2rlruwbwa	3.004	1.750	5.307	6.480	3.479
128Kb64s2rlruwbwa	1.259	1.254	1.927	2.216	1.305
128Kb128s2rlruwbwa	0.892	0.755	1.624	1.929	0.995
32Kb32s1wbwa	4.619	3.459	6.644	7.566	5.200
32Kb32s2rlruwbwa	3.750	3.164	5.937	7.133	4.053
32Kb32s4rlruwbwa	3.626	3.130	5.904	7.090	3.880
32Kb32s8rlruwbwa	3.588	3.114	5.903	7.089	3.805

Data Cache Miss Rates (Contd.)

	gcc.166	gcc.200	gcc.expr	gcc.integ	gcc.scilab
8Kb32s1wt	51.071	18.460	23.917	38.099	21.032
8Kb32s2rlruwt	49.064	15.788	21.420	35.901	18.883
8Kb32s4rlruwt	48.510	14.902	20.332	34.974	18.199
8Kb32s8rlruwt	48.319	14.631	19.946	34.690	18.126
8Kb32s1wt	51.071	18.460	23.917	38.099	21.032
8Kb32s2rrandomwt	49.594	16.508	22.121	36.557	19.514
8Kb32s4rrandomwt	49.220	16.013	21.417	35.909	19.272
8Kb32s8rrandomwt	49.136	15.860	21.183	35.755	19.208
32Kb32s1wt	47.974	13.158	18.971	34.536	17.190
32Kb32s2rlruwt	47.076	12.019	18.028	33.592	16.035
32Kb32s4rlruwt	46.891	11.592	17.605	33.346	15.880
32Kb32s8rlruwt	46.839	11.466	17.460	33.290	15.850
32Kb32s1wt	47.974	13.158	18.971	34.536	17.190
32Kb32s2rrandomwt	47.263	12.119	17.627	33.809	16.221
32Kb32s4rrandomwt	47.142	11.732	17.225	33.622	16.062
32Kb32s8rrandomwt	47.104	11.621	17.074	33.563	16.841
128Kb32s1wt	47.343	8.683	12.075	27.843	11.992
128Kb32s2rlruwt	46.799	8.225	11.139	30.072	11.892
128Kb32s4rlruwt	46.767	8.046	10.342	32.957	12.033
128Kb32s8rlruwt	46.763	8.038	10.164	32.944	12.413
128Kb32s1wt	47.343	8.683	12.075	27.843	11.992
128Kb32s2rrandomwt	45.821	8.231	11.573	27.627	11.721
128Kb32s4rrandomwt	45.438	8.160	11.464	27.461	11.697
128Kb32s8rrandomwt	45.266	8.127	11.418	27.313	11.733
8Kb64s2rlruwt	49.127	15.960	21.373	35.903	18.441
8Kb128s2rlruwt	40.971	15.103	19.285	31.163	16.712
32Kb64s2rlruwt	47.006	11.660	17.375	33.617	15.270
32Kb128s2rlruwt	37.728	9.882	14.089	26.922	12.678

128Kb64s2rlruwt	46.609	7.771	10.434	30.111	11.291
128Kb128s2rlruwt	37.238	6.956	9.618	25.070	10.379
32Kb64s1wt	47.867	12.864	18.556	34.441	16.568
32Kb64s2rlruwt	47.006	11.660	17.375	33.617	15.270
32Kb64s4rlruwt	46.676	11.100	16.975	33.145	15.107
32Kb64s8rlruwt	46.652	10.959	16.865	33.114	15.064
32Kb64s1wt	47.867	12.864	18.556	34.441	15.568
32Kb64s2rrandomwt	47.168	11.769	17.050	33.765	15.486
32Kb64s4rrandomwt	46.962	11.274	16.599	33.431	15.317
32Kb64s8rrandomwt	46.932	11.149	16.436	33.401	15.249
4Kb32s8rlruwt	50.352	20.678	22.638	37.259	23.397
16Kb32s8rlruwt	47.123	12.315	18.453	33.548	16.060
64Kb32s8rlruwt	46.795	10.104	13.336	33.158	14.796
256Kb32s8rlruwt	46.676	5.660	8.595	18.642	8.219
4Kb32s8rrandomwt	51.667	21.504	24.465	38.809	23.802
16Kb32s8rrandomwt	47.585	13.104	19.113	34.164	16.966
64Kb32s8rrandomwt	46.774	9.976	13.552	32.490	14.396
256Kb32s8rrandomwt	36.569	7.092	10.441	20.328	9.944
8Kb64s2rlruwbwa	47.501	13.588	18.661	34.432	16.039
8Kb128s2rlruwbwa	26.072	9.869	12.504	20.007	9.506
32Kb64s2rlruwbwa	45.609	10.326	15.442	32.003	14.316
32Kb128s2rlruwbwa	23.289	6.048	8.414	16.628	7.533
128Kb64s2rlruwbwa	45.208	6.853	6.239	30.414	10.985
128Kb128s2rlruwbwa	22.859	3.677	3.394	15.498	5.693
32Kb32s1wbwa	46.444	11.729	16.661	32.962	15.771
32Kb32s2rlruwbwa	45.845	10.653	15.755	32.185	15.067
32Kb32s4rlruwbwa	45.747	10.360	15.523	32.041	14.969
32Kb32s8rlruwbwa	45.720	10.270	15.467	32.014	14.942

Data Cache Miss Rates (Contd.)

	vortex.in1	vortex.in2	vortex.in3	bzip2.src	bzip2.gra
8Kb32s1wt	14.734	18.027	16.990	8.760	7.013
8Kb32s2rlruwt	10.725	10.865	10.768	7.745	6.299
8Kb32s4rlruwt	9.622	9.701	9.606	7.530	6.104
8Kb32s8rlruwt	9.247	9.395	9.165	7.444	6.078
8Kb32s1wt	14.734	18.027	16.990	8.760	7.013
8Kb32s2rrandomwt	12.167	13.206	13.516	8.027	6.581
8Kb32s4rrandomwt	11.481	12.196	12.141	7.855	6.376
8Kb32s8rrandomwt	11.528	12.048	11.747	7.801	6.358
32Kb32s1wt	7.572	10.244	7.447	6.786	5.777
32Kb32s2rlruwt	3.653	3.766	3.810	6.334	5.567
32Kb32s4rlruwt	2.636	2.781	2.695	6.228	5.559
32Kb32s8rlruwt	2.188	2.234	2.406	6.189	5.562
32Kb32s1wt	7.572	10.244	7.447	6.786	5.777
32Kb32s2rrandomwt	4.143	4.369	4.341	6.467	5.651
32Kb32s4rrandomwt	3.366	3.621	3.560	6.407	5.655
32Kb32s8rrandomwt	3.113	3.183	3.296	6.394	5.660
128Kb32s1wt	4.337	3.806	3.650	5.756	4.899
128Kb32s2rlruwt	1.558	1.445	1.818	5.569	4.747
128Kb32s4rlruwt	1.103	1.103	1.092	5.519	4.699
128Kb32s8rlruwt	1.025	1.030	1.028	5.499	4.671
128Kb32s1wt	4.337	3.806	3.650	5.756	4.899
128Kb32s2rrandomwt	1.790	1.661	1.984	5.632	4.813
128Kb32s4rrandomwt	1.307	1.319	1.320	5.606	4.809
128Kb32s8rrandomwt	1.262	1.257	1.258	5.599	4.814
8Kb64s2rlruwt	11.295	11.598	11.639	7.711	6.121
8Kb128s2rlruwt	12.595	12.731	12.789	8.312	6.691
32Kb64s2rlruwt	4.321	4.561	4.410	6.095	5.220
32Kb128s2rlruwt	4.902	5.157	5.172	5.799	4.992

128Kb64s2rlruwt	1.697	1.456	2.002	5.297	4.485
128Kb128s2rlruwt	1.748	1.640	2.202	4.910	4.291
32Kb64s1wt	9.299	11.607	8.915	6.716	5.553
32Kb64s2rlruwt	4.321	4.561	4.410	6.095	5.220
32Kb64s4rlruwt	3.345	3.400	3.346	5.975	5.194
32Kb64s8rlruwt	3.134	3.201	3.230	5.931	5.187
32Kb64s1wt	9.299	11.607	8.915	6.716	5.553
32Kb64s2rrandomwt	5.003	5.301	5.208	6.238	5.328
32Kb64s4rrandomwt	4.206	4.519	4.824	6.161	5.325
32Kb64s8rrandomwt	3.991	4.201	4.352	6.142	5.328
4Kb32s8rlruwt	13.098	13.373	13.345	8.026	6.300
16Kb32s8rlruwt	5.753	5.651	5.712	6.699	5.811
64Kb32s8rlruwt	1.209	1.204	1.191	5.838	5.185
256Kb32s8rlruwt	0.961	0.957	0.961	5.025	4.347
4Kb32s8rrandomwt	17.433	18.022	17.910	8.566	6.774
16Kb32s8rrandomwt	6.954	7.198	7.189	6.983	5.994
64Kb32s8rrandomwt	1.651	1.655	1.648	5.981	5.256
256Kb32s8rrandomwt	1.087	1.083	1.090	5.040	4.442
8Kb64s2rlruwbwa	6.775	7.163	7.020	4.886	3.613
8Kb128s2rlruwbwa	8.250	8.442	8.308	5.848	4.451
32Kb64s2rlruwbwa	2.551	2.687	2.603	3.058	2.651
32Kb128s2rlruwbwa	3.045	3.132	2.949	3.019	2.523
128Kb64s2rlruwbwa	0.721	0.646	0.927	2.102	2.017
128Kb128s2rlruwbwa	0.909	0.816	1.105	1.876	1.828
32Kb32s1wbwa	4.120	4.576	4.200	3.840	3.283
32Kb32s2rlruwbwa	1.969	2.028	2.134	3.330	3.031
32Kb32s4rlruwbwa	1.507	1.663	1.645	3.197	3.010
32Kb32s8rlruwbwa	1.251	1.306	1.421	3.147	3.003

Data Cache Miss Rates (Contd.)

	bzip2.prog	eon.cook	eon.rush	eon.kajiya	perlbmk.diff
8Kb32s1wt	7.215	9.746	15.349	12.753	18.568
8Kb32s2rlruwt	6.308	7.302	10.860	9.785	15.267
8Kb32s4rlruwt	6.027	6.890	11.576	9.773	15.107
8Kb32s8rlruwt	5.965	6.158	11.118	9.772	15.237
8Kb32s1wt	7.215	9.746	15.349	12.753	18.568
8Kb32s2rrandomwt	6.630	8.283	12.147	10.557	15.526
8Kb32s4rrandomwt	6.375	7.676	13.429	11.151	15.028
8Kb32s8rrandomwt	6.341	6.544	12.160	11.364	14.840
32Kb32s1wt	5.638	3.524	6.682	2.416	9.936
32Kb32s2rlruwt	5.273	2.216	3.175	0.956	8.424
32Kb32s4rlruwt	5.202	1.153	1.262	0.186	8.341
32Kb32s8rlruwt	5.167	0.954	0.739	0.135	7.421
32Kb32s1wt	5.638	3.524	6.682	2.416	9.936
32Kb32s2rrandomwt	5.400	2.498	4.029	1.118	8.661
32Kb32s4rrandomwt	5.365	1.267	1.873	0.300	8.429
32Kb32s8rrandomwt	5.345	1.213	1.611	0.195	7.909
128Kb32s1wt	4.746	1.422	4.806	1.120	7.161
128Kb32s2rlruwt	4.561	0.757	0.426	0.043	6.189
128Kb32s4rlruwt	4.502	0.744	0.118	0.013	5.980
128Kb32s8rlruwt	4.476	0.744	0.118	0.013	5.884
128Kb32s1wt	4.746	1.422	4.806	1.120	7.161
128Kb32s2rrandomwt	4.637	0.759	0.791	0.064	6.400
128Kb32s4rrandomwt	4.613	0.744	0.119	0.013	6.235
128Kb32s8rrandomwt	4.606	0.744	0.118	0.013	6.252
8Kb64s2rlruwt	6.130	9.168	10.658	8.481	15.334
8Kb128s2rlruwt	6.633	8.205	11.492	7.693	15.576
32Kb64s2rlruwt	4.967	2.818	1.983	1.174	8.704
32Kb128s2rlruwt	4.759	2.715	2.428	1.336	9.151

128Kb64s2rlruwt	4.239	0.601	0.153	0.050	6.268
128Kb128s2rlruwt	3.992	0.393	0.172	0.060	6.205
32Kb64s1wt	5.457	3.809	5.808	2.788	10.089
32Kb64s2rlruwt	4.967	2.818	1.983	1.174	8.704
32Kb64s4rlruwt	4.889	1.414	0.674	0.345	8.824
32Kb64s8rlruwt	4.852	1.035	0.367	0.166	7.553
32Kb64s1wt	5.457	3.809	5.808	2.788	10.089
32Kb64s2rrandomwt	5.101	2.832	2.591	1.385	9.003
32Kb64s4rrandomwt	5.053	1.429	0.758	0.470	8.874
32Kb64s8rrandomwt	5.030	1.398	0.462	0.233	8.074
4Kb32s8rlruwt	6.516	17.217	20.791	17.215	17.439
16Kb32s8rlruwt	5.524	1.837	3.433	1.272	9.424
64Kb32s8rlruwt	4.819	0.744	0.124	0.016	6.501
256Kb32s8rlruwt	4.130	0.744	0.118	0.013	5.748
4Kb32s8rrandomwt	7.126	18.333	24.476	20.310	18.844
16Kb32s8rrandomwt	5.769	2.283	4.164	1.938	9.847
64Kb32s8rrandomwt	4.968	0.746	0.385	0.019	6.853
256Kb32s8rrandomwt	4.228	0.744	0.118	0.013	5.948
8Kb64s2rlruwbwa	3.830	2.887	2.967	2.675	4.673
8Kb128s2rlruwbwa	4.520	3.048	2.916	2.929	5.041
32Kb64s2rlruwbwa	2.584	0.548	0.683	0.476	1.590
32Kb128s2rlruwbwa	2.447	0.663	0.788	0.580	1.888
128Kb64s2rlruwbwa	1.818	0.026	0.026	0.014	0.378
128Kb128s2rlruwbwa	1.601	0.030	0.031	0.015	0.397
32Kb32s1wbwa	3.345	1.021	1.333	0.792	2.476
32Kb32s2rlruwbwa	2.946	0.480	0.664	0.386	1.588
32Kb32s4rlruwbwa	2.867	0.101	0.105	0.076	1.209
32Kb32s8rlruwbwa	2.829	0.050	0.068	0.042	0.843

Data Cache Miss Rates (Contd.)

	pbmk.mr	pbmk.perf	pbmk.s1	pbmk.s2	pbmk.s3	pbmk.s4
8Kb32s1wt	8.453	14.740	17.795	17.939	18.587	17.919
8Kb32s2rlruwt	3.824	11.661	16.599	16.962	17.395	16.960
8Kb32s4rlruwt	3.287	9.765	17.383	17.570	16.206	17.640
8Kb32s8rlruwt	3.274	9.208	18.223	18.098	18.421	18.155
8Kb32s1wt	8.453	14.740	17.795	17.939	18.587	17.919
8Kb32s2rrandomwt	4.153	12.957	16.839	17.446	17.273	17.443
8Kb32s4rrandomwt	4.141	11.625	16.128	16.523	14.760	16.483
8Kb32s8rrandomwt	4.154	11.445	15.806	15.847	15.408	15.786
32Kb32s1wt	3.621	6.513	7.574	8.378	8.386	8.386
32Kb32s2rlruwt	3.270	4.921	4.664	5.490	4.613	5.479
32Kb32s4rlruwt	3.265	3.847	3.269	3.176	3.180	3.151
32Kb32s8rlruwt	3.268	3.573	2.964	3.004	2.990	2.984
32Kb32s1wt	3.621	6.513	7.574	8.378	8.386	8.386
32Kb32s2rrandomwt	3.645	5.609	5.466	5.947	5.385	5.961
32Kb32s4rrandomwt	3.595	4.664	4.250	4.142	4.144	4.133
32Kb32s8rrandomwt	3.622	4.472	4.045	4.023	4.019	4.018
128Kb32s1wt	3.318	1.801	3.795	3.617	3.735	3.683
128Kb32s2rlruwt	3.222	0.641	2.975	2.922	2.930	2.930
128Kb32s4rlruwt	3.088	0.254	2.853	2.815	2.837	2.824
128Kb32s8rlruwt	3.064	0.194	2.851	2.811	2.836	2.821
128Kb32s1wt	3.318	1.801	3.795	3.617	3.735	3.683
128Kb32s2rrandomwt	3.310	0.742	3.467	3.300	3.367	3.348
128Kb32s4rrandomwt	3.239	0.323	3.422	3.223	3.296	3.287
128Kb32s8rrandomwt	3.179	0.187	3.435	3.200	3.313	3.262
8Kb64s2rlruwt	2.179	11.570	17.156	16.957	16.909	16.985
8Kb128s2rlruwt	7.777	12.152	16.798	16.758	15.646	16.772
32Kb64s2rlruwt	1.237	4.918	3.955	5.655	3.952	5.628
32Kb128s2rlruwt	1.235	4.630	3.694	6.217	5.543	6.185

128Kb64s2rlruwt	1.195	0.610	1.841	1.869	1.818	1.841
128Kb128s2rlruwt	1.211	0.582	0.754	1.609	0.819	1.569
32Kb64s1wt	2.903	6.392	6.704	8.323	7.561	8.315
32Kb64s2rlruwt	1.237	4.918	3.955	5.655	3.952	5.628
32Kb64s4rlruwt	1.227	3.478	2.478	2.522	3.504	2.423
32Kb64s8rlruwt	1.227	3.050	1.866	1.959	1.930	1.931
32Kb64s1wt	2.903	6.392	6.704	8.323	7.561	8.315
32Kb64s2rrandomwt	2.190	5.664	4.393	5.532	4.403	5.502
32Kb64s4rrandomwt	2.136	4.462	2.951	3.000	3.441	2.912
32Kb64s8rrandomwt	2.109	4.196	2.580	2.702	2.685	2.649
4Kb32s8rlruwt	3.639	13.146	22.842	22.613	22.704	22.698
16Kb32s8rlruwt	3.273	5.606	4.568	5.260	4.045	5.125
64Kb32s8rlruwt	3.209	1.548	2.872	2.864	2.874	2.856
256Kb32s8rlruwt	3.064	0.165	2.829	2.742	2.792	2.774
4Kb32s8rrandomwt	4.645	16.350	25.157	25.056	25.239	25.117
16Kb32s8rrandomwt	3.875	7.324	5.657	5.942	5.308	5.828
64Kb32s8rrandomwt	3.346	1.606	3.666	3.543	3.599	3.557
256Kb32s8rrandomwt	3.141	0.165	3.229	2.988	3.058	3.075
8Kb64s2rlruwbwa	2.715	7.113	7.650	7.673	7.522	7.650
8Kb128s2rlruwbwa	5.728	7.910	8.574	8.689	8.608	8.675
32Kb64s2rlruwbwa	0.319	2.800	1.804	1.860	1.790	1.834
32Kb128s2rlruwbwa	0.323	2.457	2.563	2.691	2.660	2.667
128Kb64s2rlruwbwa	0.300	0.445	0.316	0.338	0.322	0.327
128Kb128s2rlruwbwa	0.305	0.414	0.287	0.390	0.297	0.381
32Kb32s1wbwa	0.331	4.069	3.636	3.687	3.714	3.674
32Kb32s2rlruwbwa	0.319	3.125	1.457	1.537	1.495	1.498
32Kb32s4rlruwbwa	0.316	2.671	0.601	0.585	0.557	0.555
32Kb32s8rlruwbwa	0.317	2.558	0.439	0.495	0.468	0.473

References

[1] Pradeep Bose, and T. M. Conte, *"Performance Analysis and its Impact on Design"*, IEEE Computer, 31(5): 41-49, May 1998.

[2] K. Chow, A. Wright, and K. Lai, *"Characterization of Java Workloads by Principal Components Analysis and Indirect Branches"*, In Proceedings of the Workshop on Workload Characterization (WWC-1998), held in conjunction with the 31st Annual ACM/IEEE International Symposium on Micro-architecture (MICRO-31), pp. 11–19, Nov. 1998

[3] G.H. Dunteman, *"Principal Components Analysis"*, SAGE Publications, 1989.

[4] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, *"Workload Design: Selecting Representative Program-Input Pairs"*, In Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE CS Press, pp. 83-94, 2002.

[5] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, *"Designing Computer Architecture Research Workloads"*, IEEE Computer Magazine, pp. 65-71, Feb. 2003.

[6] L. Eeckhout, A. Georges, and K. De Bosschere, *"How Java Programs interact with Virtual Machines at the microarchitectural level"*, OOPSLA 2003.

[7] 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, 5:1-33, 2 2003.

[8] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan J. Smith, *"Cache Performance of the SPEC92 Benchmark Suite"*, IEEE MICRO, 1993.

[9] L. John, *"Performance Evaluation: Techniques, Tools and Benchmarks"*, The Computer Engineering Handbook, CRC Press, 2001.

[10] L. John, P. Vasudevan and J. Sabarinathan, *"Workload Characterization: Motivation, Goals and Methodology",* in Workload Characterization: Methodology and Case Studies, IEEE Computer Society, 1999.

[11] Tao Li, Lizy John, N. Vijaykrishnan, Anand Sivasubramaniam, Jyotsana Sabarinathan, and A. Murthy, *"Using Complete System Simulation to Characterize SPECjvm98 Benchmarks"*, In Proceedings of ACM International Conference on Supercomputing, pp. 22-23, 2000.

[12] David L. Lilja *"Measuring Computer Performance – A Practitioner's Guide"*, Cambridge University Press, 2000.

[13] B. F. J. Manly, *"Multivariate Statistical Methods: A Primer"*, Chapman & Hall, Second edition, 1994.

[14] Patterson and Hennessy, Computer Architecture: The Hardware/Software Approach, by Hennessy and Patterson, Morgan Kaufman Publishers, 2nd edition, 1998, ISBN 1558604286.

[15] R. Radhakrishnan, N. Vijaykrishnan, Lizy John, and Anand Sivasubramaniam, *"Architectural Issues in Java Runtime Systems"*, In Proceedings of the International Symposium on High Performance Computer Architecture, pp. 387-398, 2000.

[16] A. J. Smith, *"Cache Memories"*, ACM Computing Surveys, 14(3): 473-530, Sept. 1982.

[17] A. J. Smith, *"Cache evaluation and the impact of workload choice"*, In Proceedings of the 12th Annual Symposium on Computer Architecture, pp. 64-73, 1985.

[18] J. E. Smith, *"Characterizing Computer Performance with a Single Number"*, Communications ACM, vol. 31, no. 10, pp. 1202-1206, 1998.

[19] SPEC Benchmarks, http://www.spec.org/

[20] Shade Analyzer User's Manual, http://www.sun.com/

[21] Hans Vandierendonck, and Koen De Bosschere, *“Eccentric and Fragile Benchmarks”*, ISPASS 2004.

[22] Reinhold P. Weicker, *"An Overview of Common Benchmarks"*, IEEE Computer, pp. 65-75, December 1990.

[23] Reinhold Weicker, *"On the Use of SPEC Benchmarks in Computer Architecture Research"*, Computer Architecture News, pp. 19-22, March 1997.

VITA

Saket Kumar, the son of Girish Kumar Verma and Prabha Verma, was born in Ranchi, India on July 27, 1979. After completing his work at Kendriya Vidyalaya, Patna, in 1996, he entered into the Indian Institute of Technology Roorkee (erstwhile University of Roorkee), Roorkee, India. He received the Bachelor of Engineering from Indian Institute of Technology Roorkee in May 2000. After working for C-DOT, New Delhi, India for two years, he joined the graduate school at the University of Texas at Austin in August 2002.

Permanent Address: 9,Kesrinagar,
Patna-800024,
Bihar,
India.

This report was typed by Saket Kumar.