Performance Prediction using Program Similarity

Aashish Phansalkar Lizy K. John
{aashish, ljohn}@ece.utexas.edu
University of Texas at Austin

***Abstract* - Modern computer applications are developed at a very fast rate and new features are being added to the current ones everyday. A computer architect is always looking for quicker ways to evaluate performance of the design on these applications in a relatively shorter time. Simulation is a very popular tool used in the early design phase of a microprocessor. But applications run for a long time and microprocessors are getting complex which leads to very high simulation times. With a reasonably quick workload characterization we should be able to predict performance of the new application on the system(s). This paper proposes two simple techniques to predict performance based on the similarity of the new application with already characterized benchmarks whose performance numbers are available. Each of these techniques is then used to show how speedup and cache miss-rates can be predicted for a given program.**

1. Introduction

Computer benchmarking involves use of application programs as benchmarks. The process of performance evaluation includes using these programs as standard tests to evaluate performance of a computer system. Benchmarks are also used by computer architects to evaluate enhancements and various design options in the early phase of design space exploration. Due to increase in complexity of systems and many times, long running programs, cycle accurate simulation becomes a difficult option to choose. Also, simulating individual components like cache, although not very expensive need to be evaluated for a lot of different configurations. Ideally, to overcome this challenge in shortest possible time one can build a mathematical or analytical model for each of the design blocks which can estimate performance when the necessary parameters are plugged in. But it is well known that building a mathematical or analytical model is difficult because the system that needs to be modeled is extremely complex which makes the process non-trivial and a lot of times very difficult.

In such situation using the knowledge of similarity of programs in predicting performance can be helpful. New applications are being developed everyday. Our methodology is based on using the microarchitecture independent metrics to characterize programs in [1]. We also use Principal Components Analysis (PCA) to transform these metrics into set of variables that are uncorrelated with respect to each other as discussed in [1]. A benchmark suite like SPEC CPU 2000 has a wide variety of applications that may cover a significant amount of workload space. Hence we use SPEC CPU 2000 benchmarks in our study.

The pace of software development is on the rise. New features are being added to make incremental or drastic changes to the behavior of applications on a microprocessor. It is in prime interest of a microprocessor designer or an architect to measure performance of these new or modified programs. It is difficult to use cycle accurate simulation to do the same because of long simulation time. An architect or designer may have many other application programs from customers or other collaborative software vendors in his repository of benchmarks for which the performance is already known. If we measure similarity between the new application and the set of benchmarks in the repository one can predict performance of the new application without running cycle accurate simulation.

2. Methodology

Figure 1 shows the block diagram of the prediction methodology. We will first describe the microarchitecture independent metrics we used to characterize and measure similarity between the benchmarks and the new application and then briefly describe the statistical multivariate analysis technique called Principal Components Analysis (PCA) that we used in our methodology. We then briefly describe how we use the microarchitecture independent metrics to find similarity between programs and then predict performance of the new program based on its similarity with the programs in the repository.

2.1. Metrics used for measuring program similarity

Microarchitecture independent metrics allow for a comparison between programs by understanding the inherent characteristics of a program isolated from features of particular microarchitectural components. This section also provides an intuitive reasoning to illustrate how the measured metrics can affect the manifested performance. The metrics measured in this study are a subset of all the microarchitecture-independent characteristics that can be potentially measured, but we believe that our metrics cover a wide enough range of the program characteristics to make a meaningful comparison between the programs. The metrics that we use can be broadly classified into five categories: instruction mix, branch behavior, metrics to measure

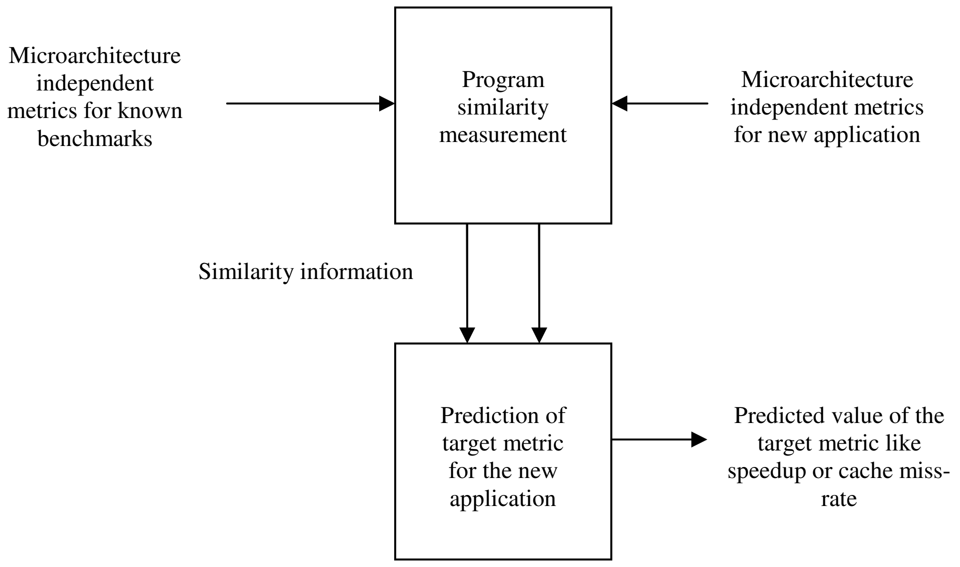
Microarchitecture independent metrics for known benchmarks
Program similarity measurement
Microarchitecture independent metrics for new application
Similarity information
Prediction of target metric for the new application
Predicted value of the target metric like speedup or cache miss-rate

Figure 1: Block diagram of prediction methodology

instruction level parallelism, data locality and instruction locality. Other program characteristics, such as value predictability can also be added to the analysis if they are exploited by the microarchitecture for performance benefit. The detailed list of all the microarchitecture-independent metrics we use in our study is as follows:

Instruction Mix:

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

Behavior of branches:

We used the following set of metrics to compare branch behavior of programs:

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

*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***:** We measure the fraction of taken forward branches in the dynamic instruction stream of the program.

Inherent Instruction Level Parallelism:

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

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

Data locality:

*Data Temporal Locality***:** Several locality metrics have been proposed in the past [4] [5] [6] [7] [8] [9] [10], however, many of them are computation and memory intensive. We picked the average memory reuse distance metric from [9] since it is more computationally feasible than other metrics. In this metric, locality is quantified by computing the average distance (in terms of number of memory accesses) between two consecutive accesses to the same address, for every unique address in the program. The evaluation is performed in four distinct window sizes, analogous to cache block sizes. This metric is calculated for window sizes of 16, 64, 256 and 4096 bytes. The choice of the *window* sizes is based on the experiments conducted by Lafage et.al. [9]. Their experimental results show that the above set of window sizes was sufficient to characterize the locality of the data reference stream with respect to a wide range of data cache configurations.

Example of 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 window sizes to that of window size 16 mentioned above. As the window size is increased, programs with a high spatial locality will have a lower value for the data temporal locality metric and vice versa. Therefore, the ratio of the data temporal locality metric for two different window sizes can be used to quantify the spatial locality in a program.
Instruction locality
Instruction Temporal Locality: The instruction temporal locality metric is quantified by computing the average distance (in terms of number of instructions) between two consecutive accesses to the same static instruction, for every unique static instruction in the program that is executed at least twice. The instruction temporal locality is calculated for *window* sizes of 16, 64, 256, and 4096 bytes. The reuse distance is calculated in the same fashion as shown in the example above
Instruction Spatial Locality: Spatial locality of the instruction stream is characterized by the ratio of instruction temporal locality metrics for higher window sizes to that of window size 16. As the window size is increased, programs with a high spatial locality will have a lower value for the data temporal locality metric and vice versa. Therefore, the ratio of the data temporal locality metric for two different window sizes can be used to quantify the spatial locality in a program.

2.2. Principal Components Analysis

There are many variables (29 microarchitecture-independent characteristics) involved in our study. It is humanly impossible to simultaneously look at all the data and draw meaningful conclusions from them. We thus use multivariate statistical data analysis techniques, namely *Principal Component Analysis.* We first normalize the data to have the mean equal to zero and standard deviation of one for each variable. Principal components analysis (PCA) [11] is a classic multivariate statistical data analysis technique that is used to reduce the dimensionality of a data set while retaining most of the original information. It builds on the assumption that many variables (in our case, microarchitecture-independent program characteristics) are correlated. PCA computes new variables, so called principal components, which are linear combinations of the original variables, such that all the principal components are uncorrelated. PCA transforms *p* variables $X_1, X_2,...., X_p$ into *p* principal components (PC) $Z_1,Z_2,\ldots,Z_p$ such that:

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

This transformation has the property $Var\ [Z_1] \geq \text{Var}\ [Z_2] \geq \ldots \geq \text{Var}\ [Z_p]$ which means that Z_1 contains the most information and Z_p the least. Given this property of decreasing variance of the principal components, we can remove the components with the lower values of variance from the analysis. This reduces the dimensionality of the data set while controlling the amount of information that is lost. In other words, we retain *q* principal components (q << p) that explain at least 75% to 90 % of the total information.

We propose two methods which use the microarchitecture independent metrics to measure similarity and then predict performance of the new program on microprocessor component/system.

1) Assign weights to each of the benchmark programs based on their similarity to new application and then use the weighted mean to predict the performance.
2) Cluster similar benchmarks in the microarchitecture independent workload space and then use the representative program from the cluster which contains the new application program to predict performance.

In the next section we discuss our results where we estimate speedup and cache miss-rates of programs using the two techniques mentioned above and discuss the techniques in more detail applied to predict each of the metrics.

3. Results

In this section we show the results of using the two methods discussed above to predict the speedup of a new application program and also predict cache miss-rate of a new application program for one choice of cache configuration. For brevity of results we show it for only one cache configuration.

3.1. Knowledge based prediction of speedup

The standard benchmark suites like the SPEC CPU2000 have programs from many different application domains. Almost all big computer system manufacturers report performance results for their systems. Here is another scenario where the prediction of performance using the knowledge of benchmarks is useful. A user can compare performance of different computer systems by referring to the reported scores. But many times, the user has his own application(s) for which he needs to predict performance for a

Statistics	% Error in predicted speedup of bzip2 using Weighted GM	% Error in predicted speedup of bzip2 using Weighted HM	% Error in predicted speedup of bzip2 using Weighted AM	% Error in predicted speedup of bzip2 using GM
Average	14.37	11.31	18.05	16.41
Std. deviation	9.69	8.58	11.22	10.15
Standard Error	0.387	0.342	0.449	0.406
Lower CI	13.72	10.74	17.30	15.73
Upper CI	15.02	11.89	18.80	17.09

Table 1: Percentage error in predicting speedup for 256.bzip2 based on other integer programs from the SPEC CPU2000 suite

particular system. In a very rare case the same application is a part of the benchmark suite. In short the way SPEC calculates the aggregate performance metric, involves calculating the geometric mean of speedups for all benchmark programs ran in conformation with the run rules. In this case SPEC gives equal importance to all the programs. But the user might only be concerned about his application program's performance on a particular machine. In using SPEC recorded aggregate metric of speedup for predicting performance of user's application might have large error. Also, giving equal importance to all programs might lead to misleading conclusion. To demonstrate the two methods we carefully pick fifty different machines from different manufacturers and their scores for SPEC CPU2000 programs from the SPEC website [12]. For the simplicity of study and ease of validation, we consider one of the benchmarks bzip2 from the SPECint2000 suite as the user's application program. The aim of this study is to improve the prediction of performance of bzip2. With the available information from SPEC website the user's best prediction for his application program (bzip2 in this case) is the geometric mean of speedup of all benchmark programs.

Method I

An important part of this study is to find how similar each of the benchmark programs is to the user's application program. To quantify similarity between the application program and all the benchmark programs, we use the twenty nine microarchitecture independent characteristics of programs described in Section 2.1 for the nine SPECint2000 benchmark programs and the bzip2 program. The measure of similarity is based on distance between two programs in the transformed space (PCA space) of these twenty nine metrics. Similarity has inverse relation with distance in our research. We use the reciprocal of distance as a measure to assign weights to each of the programs. Let d_1 , d_2 , d_3 ... d_n be the distances between the application program and each of the benchmarks programs and r_1 , r_2 , r_3 ... r_n be their reciprocals respectively. Since the distance is inversely proportional to the weight we calculate each weight w_i in the following way

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

All the weights should add up to the number of benchmark programs *n.* The benchmark program that is closest to the user's application gets the highest weight and so on. If the application program is exactly identical to one of the benchmark programs, then the distance between them becomes zero and that benchmark program gets all the weight (*n*). In a very rare case if all the benchmark programs are equidistant from the user's application program in the workload space, then all the weights will be equal to one and the weighted mean will be equal to the mean of speedups of all the benchmark programs. After we have the weights for each benchmark, we calculate the weighted mean of speedups of all the benchmark programs. We experimented with different means i.e. weighted geometric mean, weighted harmonic mean and weighted arithmetic mean to see which mean gives a better estimate of the speedup. We then find the average error shown by each of these weighted means in predicting the speedup of bzip2 for all the fifty machines. We also find the confidence interval for average error calculated using each different type of mean over fifty different machines. Table 1 shows the results for method I. The last column shows the statistics for percentage error in predicting speedup of bzip2 if the aggregate metric of geometric mean reported on the SPEC website was used. An average error of 16.41% can be seen over all fifty machines. At 95% confidence level the confidence interval is (15.73%, 17.09%). This can be considered as the base case to which the rest of the weighted means will be compared. We observe that the weighted harmonic mean shows the lowest average error of 11.31%. The confidence interval of average error in predicting speedup for weighted harmonic mean is (10.74%, 11.89%). We can see that the confidence interval of the best case (using weighted HM) and none of the other cases(weighted AM/GM) overlap, which shows that they are statistically distinct.

Cluster 1	parser, twolf, vortex
Cluster 2	bzip2, gzip
Cluster 3	eon, vpr
Cluster 4	mcf
Cluster 5	crafty
Cluster 6	gcc

Table 2: Clusters for SPECint2000 programs using overall program characteristics

Method II

In this method we consider the same fifty machines from method I. We use the twenty nine microarchitecture independent characteristics of programs for the nine SPECint2000 benchmark programs and the bzip2 program. We then put all these programs together and apply Principal Components Analysis and then k-means clustering [13] to the transformed data. We use BIC (Baysian Information Criterion) [13] to find optimal number of clusters. In this method the process of prediction is simpler. We look at the cluster results and see which cluster contains program bzip2. If bzip2 forms its own cluster this method fails. In such a case method I can be used. But if bzip2 has more than one program in its cluster then the representative of the cluster i.e. the speedup of program which is closest to the centre of the cluster is used to predict the speedup of bzip2. Using the BIC we get 6 optimal clusters. Table 2 shows the six optimal clusters. In this case Cluster 2 contains bzip2 and gzip. Since there are only two programs in the cluster, any program can be the representative of the cluster. We predict the speedup of bzip2 to be equivalent to gzip. We find that the average error in predicting the speedup of bzip2 is 20.29% and with 95% level of confidence the average error will be between (19.36%, 21.22%).

3.2. Knowledge based prediction of cache miss-rate

Performance of caches depends significantly on locality of the memory reference stream of a program. If the user knows the cache miss-rates and memory reference behavior of few benchmark programs, the user can predict the cache miss-rates of his application program based on its similarity of locality characteristics to the benchmark programs. We capture the information about locality of memory reference stream of a program using microarchitecture independent metrics. Predicting cache miss-rate of a program for several configurations can save time in early design space exploration. We modify the microarchitecture independent metric described in Section 2.1 to quantify locality of a program. In section 2.1 we calculate the weighted average of all the reuse distances and have a single number for each block size. We modify this metric to put the reuse distances in three different buckets of small, medium and large values for each block size. The modified metric is only measured for block size of 16 and 64 bytes. We use this metric to measure similarity between memory reference behavior of programs. We describe two methods to use the microarchitecture independent metric and predict cache miss-rates for different configurations of cache for a given program.

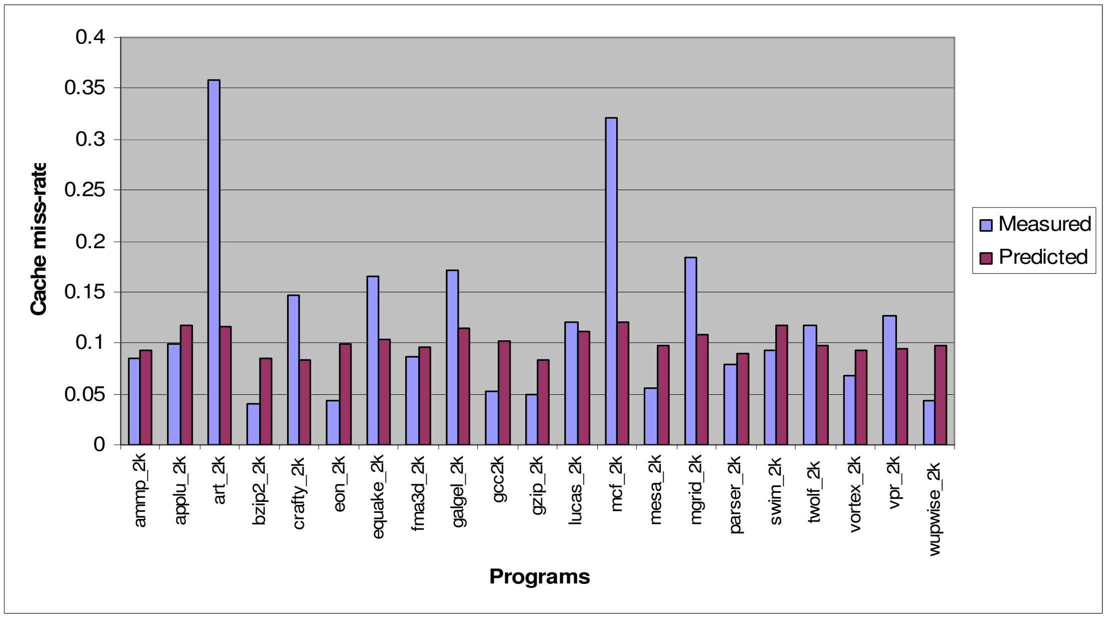

Figure 2: Comparison of measured and predicted data cache miss-rate for SPEC CPU2000 programs using method I

Method I

We describe the methodology using an example where we predict cache miss-rates of each of the SPEC CPU2000 benchmark program assuming that the cache miss-rates of the rest of the programs are known. Each SPEC CPU2000 program is considered as the user's application and its cache miss-rate is predicted based on it similarity to other SPEC CPU2000 programs. This helps us ease our validation efforts. For the microarchitecture independent metric described above we transform them to principal components space using PCA. We use the principal component scores in the transformed workload space to obtain the weights for the programs for which the cache miss-rates are known. The weights are calculated as described in Method I of Section 3.1. The distance in principal components space is inversely proportional to similarity in data memory reference behavior of programs. After assigning weights to each program we calculate the weighted arithmetic mean of the cache miss-rates. This mean is used as the predicted cache miss-rate of the program. Figure 2 shows the actual and predicted cache miss-rates of SPEC CPU2000 programs for a single cache configuration of 8KB, 64 byte block, direct mapped cache.

Method II

In this method, instead of giving a weight to each program we use k-means clustering to cluster programs in the principal component space which we obtain by transforming the microarchitecture independent locality metric. We use the Bayesian Information Criterion (BIC) [13] to obtain optimal clusters. Table 3 shows the 4 optimal clusters and the programs in each cluster.

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

Table 3: Clusters obtained using microarchitecture independent metrics for data locality

When we predict cache miss-rate of equake which lies in cluster 1 we simply look at the cache miss-rate of applu which is the representative of its cluster. But if we want to predict the cache miss-rate of applu, we choose the cache miss-rate of program among the rest that is closest to the centre of the cluster. If there are two programs in a cluster e.g. cluster 2 then the other program's cache miss-rate is used as prediction. Figure 3 shows the measured and predicted cache miss-rates of all the programs for 8KB, 64 byte block size, direct mapped cache. Except a few programs e.g. mgrid the absolute error in predicting cache miss-rate is relatively small. The error introduced for a program in predicting cache miss-rate has two sources, one due to not having a program similar to itself in the repository and the other if the microarchitecture independent metric does not capture locality behavior of access stream.

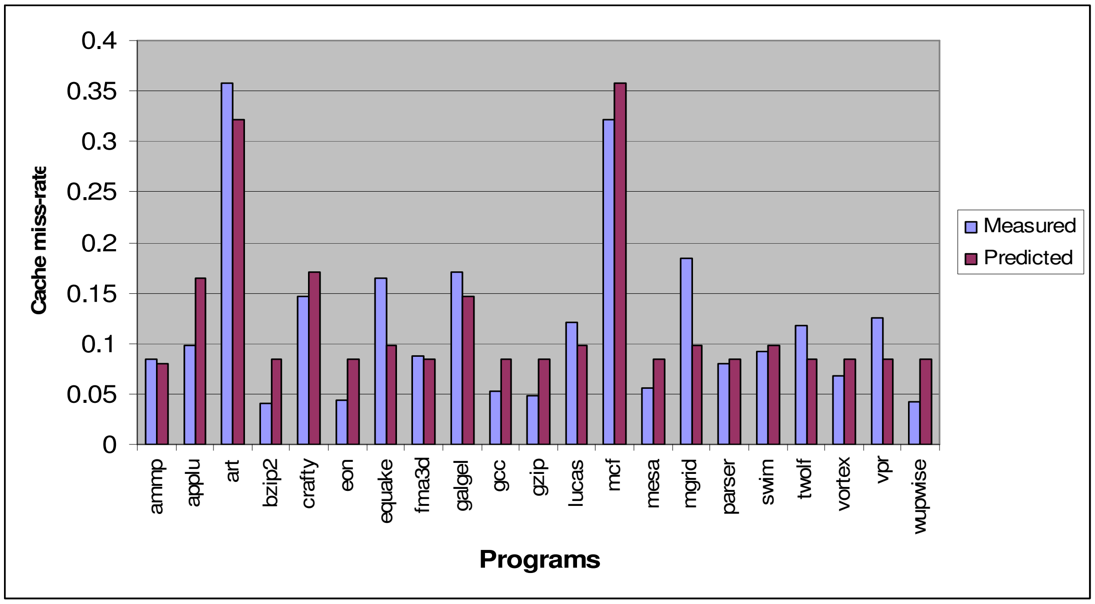

Figure 3: Comparison of measured and predicted data cache miss-rates for SPEC CPU2000 programs using method II

4. Discussion

The two methods discussed in this paper are two simple ways to predict performance based on the similarity of programs. In method II we predict performance based on only a few similar programs and ignore the rest of the programs that are far away or distantly similar. In a way we assign a weight of zero to such programs. But this method fails if we do not have a program that is similar to the new programs in the repository. If there are only a few programs in the repository there is a lesser chance of finding a similar program and hence method I may work better. If there are many benchmarks and the range of target performance metric is quite high, then clustering will give a smaller set of programs which show similar behavior and hence reduce the range over which we predict. If the new application program is not similar to any or few of the benchmark programs, then the error of performance prediction will be quite large if we use weights.

5. Conclusion

This paper demonstrates the use of two simple techniques to predict the performance of a new application. First technique uses the weights calculated based on similarity of new application program with the benchmark programs to find weights and then use the weighted means to predict performance. The second technique uses clustering to find benchmarks that lie in the same cluster as the application program and use only those benchmarks to predict the performance of the new application program. The strength of the techniques lay in the fact that microarchiture independent metrics are used to characterize all the programs and same metrics are used to predict performance on different configurations of a system instead of running multiple cycle accurate simulations or using microarchitecture dependent metrics which might limit the configurations over which the results are valid or accurate.

Acknowledgement

This research was supported in part by NSF grant 0429806, IBM, Intel and AMD Corporations.

Reference

[1] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. "Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites" *IEEE International Symposium on Performance Analysis of Systems and Software. March 2005*

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

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

[4] L. John, P. Vasudevan and J. Sabarinathan, "Workload Characterization: Motivation, Goals and methodology", pages 3 to 12 in *"Workload Characterization: Methodology and Case Studies"*, IEEE Computer Society, 1999

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

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

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

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

[9] 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.

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

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

[12] http://www.spec.org/cpu2000/results/

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