Quantifying the Impact of Input Data Sets on Program Behavior and its Applications

Lieven Eeckhout LEECKHOU@ELIS.RUG.AC.BE
Hans Vandierendonck HVDIEREN@ELIS.RUG.AC.BE
Koen De Bosschere KDB@ELIS.RUG.AC.BE
Department of Electronics and Information Systems (ELIS), Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium

Abstract

Having a representative workload of the target domain of a microprocessor is extremely important throughout its design. The composition of a workload involves two issues: (i) which benchmarks to select and (ii) which input data sets to select per benchmark. Unfortunately, it is impossible to select a huge number of benchmarks and respective input sets due to the large instruction counts per benchmark and due to limitations on the available simulation time. In this paper, we use statistical data analysis techniques such as principal components analysis (PCA) and cluster analysis to efficiently explore the workload space. Within this workload space, different input data sets for a given benchmark can be displayed, a distance can be measured between program-input pairs that gives us an idea about their mutual behavioral differences and representative input data sets can be selected for the given benchmark. This methodology is validated by showing that program-input pairs that are close to each other in this workload space indeed exhibit similar behavior. The final goal is to select a limited set of representative benchmark-input pairs that span the complete workload space. Next to workload composition, we discuss two other possible applications, namely getting insight in the impact of input data sets on program behavior and evaluating the representativeness of sampled traces.

1. Introduction

The first step when designing a new microprocessor is to compose a workload that should be representative for the set of applications that will be run on the microprocessor once it will be used in a commercial product [1, 2]. A workload then typically consists of a number of benchmarks with respective input data sets taken from various benchmarks suites, such as SPEC, TPC, MediaBench, etc. This workload will then be used during the various simulation runs to perform design space explorations. It is obvious that *workload design*, or composing a representative workload, is extremely important in order to obtain a microprocessor design that is optimal for the target environment of operation. The question when composing a representative workload is thus twofold: (i) which benchmarks and (ii) which input data sets to select. In addition, we have to take into account that even high-level architectural simulations are extremely time-consuming. As such, the total simulation time should be limited as much as possible to limit the time-to-market. This implies that the total number of benchmarks and input data sets should be limited without compromising the final design. Ideally, we would like to have a limited set of benchmark-input pairs spanning the complete workload space, which contains a variety of the most important types of program behavior.

Conceptually, the complete workload design space can be viewed as a p-dimensional space with p the number of important program characteristics that affect performance, e.g., branch prediction accuracy, cache miss rates, instruction-level parallelism, etc. Obviously, p will be too large to display the workload design space understandably. In addition, correlation exists between these variables which reduces the ability to understand what program characteristics are fundamental to make the diversity in the workload space. In this paper, we reduce the p-dimensional workload space to a q-dimensional space with $q \ll p$ ($q = 2$ to $q = 4$ typically) making the visualisation of the workload space possible without losing important information. This is achieved by using statistical data reduction techniques such as principal components analysis (PCA) and cluster analysis.

Each benchmark-input pair is a point in this (reduced) q-dimensional space obtained after PCA. We can expect that different benchmarks will be 'far away' from each other while different input data sets for a single benchmark will be clustered together. This representation gives us an excellent opportunity to measure the impact of input data sets on program behavior. Weak clustering (for various inputs and a single benchmark) indicates that the input set has a large impact on program behavior; strong clustering on the other hand, indicates a small impact. This claim is validated by showing that program-input pairs that are close to each other in the workload space indeed exhibit similar behavior. I.e., 'close' program-input pairs react in similar ways when changes are made to the architecture.

In this paper, which is an extended version of [3], we show that this methodology can be used for workload design. Indeed, strong clustering suggests that a single or only a few input sets should be selected to be representative for the cluster. This will reduce the total simulation time significantly for two reasons: (i) the total number of benchmark-input pairs is reduced; and (ii) we can select the benchmark-input pair with the smallest dynamic instruction count among all the pairs in the cluster. In addition, we show that this method is also useful in the context of trace sampling [4, 5, 6, 7, 8, 9, 10, 11], or the simulation of a restricted number of samples taken from a complete execution trace. For trace sampling to be successful, it is important that the sampled traces are representative. The representativeness of a sampled trace can be evaluated using the methodology presented in this paper, i.e., sampled traces that are close to the complete program execution can be considered as being representative.

Another potential application, next to getting insight in program behavior, workload composition and trace sampling, is profile-driven compiler optimizations. During profile-guided optimizations, the compiler uses information from previous program runs (obtained through profiling) to guide compiler optimizations. Obviously, for effective optimizations, the input set used for obtaining this profiling information should be representative for a large set of possible input sets. The methodology proposed in this paper can be useful in this respect because input sets that are close to each other in the workload space will exhibit similar behavior.

This paper is organized as follows. In section 2, the program characteristics used are enumerated. Principal components analysis, cluster analysis and their use in the context of this paper are discussed in section 3. In section 4, we show that these data reduction techniques are useful in the context of workload characterization. In addition, we discuss how input data sets affect program behavior. In section 5, we discuss two important applications, namely (i) workload composition, or the selection of representative program-input pairs, and (ii) the evaluation of the representativeness of sampled traces. Section 6 discusses related work. We conclude in section 7.

2. Workload Characterization

It is important to select program characteristics that affect performance for performing data analysis techniques in the context of workload characterization. Selecting program characteristics that do not affect performance, such as the dynamic instruction count, might discriminate benchmark-input pairs on such a characteristic yielding no information about the behavior of the benchmark-input pair when executed on a microprocessor. On the other hand, it is important to incorporate as many program characteristics as possible so that the analysis done on it will be predictive, i.e., we want strongly clustered program-input pairs to behave similarly so that a single program-input pair can be chosen as a representative of the cluster. The determination of what program characteristics to be included in the analysis in order to obtain a predictive analysis is a study on its own and is out of the scope of this paper. The goal of this paper is to show that data analysis techniques such as PCA and cluster analysis can be a helpful tool for getting insight in the workload space when composing a representative workload.

We have identified the following program characteristics:

- **Instruction mix.** We consider five instruction classes: integer arithmetic operations, logical operations, shift and byte manipulation operations, load/store operations and control operations.

- **Branch prediction accuracy.** We consider the branch prediction accuracy of three branch predictors: a bimodal branch predictor, a gshare branch predictor and a hybrid branch predictor. The bimodal branch predictor consists of an 8K-entry table containing 2-bit saturating counters which is indexed by the program counter of the branch. The gshare branch predictor is an 8K-entry table with 2-bit saturating counters indexed by the program counter xor-ed with the taken/not-taken branch history of 12 past branches. The hybrid branch predictor [12] combines the bimodal and the gshare predictor by choosing among them dynamically. This is done using a meta predictor that is indexed by the branch address and contains 8K 2-bit saturating counters.

- **Data cache miss rates.** Data cache miss rates were measured for five different cache configurations: an 8KB and a 16KB direct mapped cache, a 32KB and a 64KB two-way set-associative cache and a 128KB four-way set-associative cache. The block size was set to 32 bytes.

- **Instruction cache miss rates.** Instruction cache miss rates were measured for the same cache configurations mentioned for the data cache.

- **Sequential flow breaks.** We have also measured the number of instructions between two sequential flow breaks or, in other words, the number of instructions between two taken branches. Note that this metric is higher than the basic block size because some basic blocks 'fall through' to the next basic block.

- **Instruction-level parallelism.** To measure the amount of ILP in a program, we consider an infinite-resource machine, i.e., infinite number of functional units, perfect caches, perfect branch prediction, etc. In addition, we schedule instructions as soon as possible assuming unit execution instruction latency. The only dependencies considered between instructions are read-after-write (RAW) dependencies through registers as well as through memory. In other words, perfect register and memory renaming are assumed in these measurements.

For this study, there are $p = 20$ program characteristics in total on which the analyses are done.

3. Data Analysis

In the first two subsections of this section, we will discuss two data analysis techniques, namely principal components analysis (PCA) and cluster analysis. In the last subsection, we will detail how we used these techniques for analyzing the workload space in this study.

3.1 Principal Components Analysis

Principal components analysis (PCA) [13] is a statistical data analysis technique that presents a different view on the measured data. It builds on the assumption that many variables (in our case, program characteristics) are correlated and hence, they measure the same or similar properties of the program-input pairs. PCA computes new variables, called *principal components*, which are *linear combinations* of the original variables, such that all principal components are uncorrelated. PCA tranforms the p variables $X_1, X_2, \ldots, X_p$ into p principal components $Z_1, Z_2, \ldots, Z_p$ with $Z_i = \sum_{j=1}^{p} a_{ij} X_j$. This transformation has the properties (i) $Var[Z_1] > Var[Z_2] > \ldots > Var[Z_p]$ which means that Z_1 contains the most information and Z_p the least; and (ii) $Cov[Z_i, Z_j] = 0, \forall i \neq j$ which means that there is no information overlap between the principal components. Note that the total variance in the data remains the same before and after the transformation, namely $\sum_{i=1}^{p} Var[X_i] = \sum_{i=1}^{p} Var[Z_i]$.

As stated in the first property in the previous paragraph, some of the principal components will have a high variance while others will have a small variance. By removing the components with the lowest variance from the analysis, we can reduce the number of program characteristics while controlling the amount of information that is thrown away. We retain q principal components which is a significant information reduction since $q \ll p$ in most cases, typically $q = 2$ to $q = 4$. To measure the fraction of information retained in this q-dimensional space, we use the amount of variance $(\sum_{i=1}^{q} Var[Z_i])/(\sum_{i=1}^{p} Var[X_i])$ accounted for by these q principal components. Typically 85% to 90% of the total variance should be explained by the retained principal components.

In this study the p original variables are the program characteristics mentioned in section 2. By examining the most important q principal components, which are linear combinations of the original program characteristics ($Z_i = \sum_{j=1}^{p} a_{ij} X_j, i = 1, \ldots, q$), meaningful interpretations can be given to these principal components in terms of the original program characteristics. A coefficient a_{ij} that is close to +1 or -1 implies a strong impact of the original characteristic X_j on the principal component Z_i. A coefficient a_{ij} that is close to 0 on the other hand, implies no impact.

The next step in the analysis is to display the various benchmarks as points in the q-dimensional space built up by the q principal components. This can be done by computing the values of the q retained principal components for each program-input pair. As such, a view can be given on the workload design space and the impact of input data sets on program behavior can be displayed, as will be discussed in the evaluation section of this paper. Note that the projection on the q-dimensional space will be much easier to understand than a view on the original p-dimensional space for two reasons: (i) q is much smaller than p: $q \ll p$, and (ii) the q-dimensional space is uncorrelated.

During principal components analysis, one can either work with normalized or non-normalized data (the data is normalized when the mean of each variable is zero and its variance is one). In the case of non-normalized data, a higher weight is given in the analysis to variables with a higher variance. In our experiments, we have used normalized data because of our heterogeneous data; e.g., the variance of the ILP is orders of magnitude larger than the variance of the data cache miss rates.

3.2 Cluster Analysis

Cluster analysis [13] is another data analysis technique that is aimed at clustering n cases, in our case program-input pairs, based on the measurements of p variables, in our case program characteristics. The final goal is to obtain a number of groups, containing program-input pairs that have 'similar' behavior. There exist two commonly used types of clustering techniques, namely linkage clustering and K-means clustering. These two clustering techniques will be discussed in what follows.

Linkage clustering starts with a matrix of distances between the n cases or program-input pairs. As a starting point for the algorithm, each program-input pair is considered as a group. In each iteration of the algorithm, the two groups that are most close to each other (with the smallest distance in the p-dimensional space, also called the *linkage distance*) will be combined to form a new group. As such, close groups are gradually merged until finally all cases will be in a single group. This can be represented in a so called *dendrogram*, which graphically represents the linkage distance for each group merge at each iteration of the algorithm. Having obtained a dendrogram, it is up to the user to decide how many clusters to take. This decision can be made based on the linkage distance. Indeed, small linkage distances imply strong clustering while large linkage distances imply weak clustering. There exist several methods for calculating the distance between groups or clusters of program-input pairs all potentially leading to different clustering results. In this paper, we consider two possibilities that were found to produce results that are quite consistent with each other. We have used the *furthest neighbor* strategy (also known as *complete linkage*) and the *weighted pair-group average* strategy. In complete linkage, the distance between two clusters is computed as the largest distance between any two program-input pairs from the clusters (or thus the furthest neighbor). In the weighted pair-group average method, the distance between two clusters is computed as the weighted average distance between all pairs of program-input points in two different clusters. The weighting of the average is done by considering the cluster size, i.e., the number of program-input points in the cluster.

Next to linkage clustering we did also consider *K-means clustering*. K-means clustering produces exactly K clusters with the greatest possible distinction. The algorithm works as follows. In each iteration, the distance is calculated for each program-input pair to the center of each cluster. A program-input pair will then be assigned to the closest cluster. As such, new cluster centers can be computed. This algorithm is iterated until no more changes are observed. It is well known that the result of K-means clustering can be dependent on the choice of the initial cluster centers. In this paper we have maximized the distance between the various cluster centers (being single program-inputs pairs at this stage of the algorithm) as an initial estimate.

3.3 Workload Analysis

The workload analysis done in this paper is a combination of PCA and cluster analysis and consists of the following steps:

1. The $p = 20$ program characteristics as discussed in section 2 are measured by instrumenting the benchmark programs with ATOM [14], a binary instrumentation tool for the Alpha architecture. With ATOM, a statically linked binary can be transformed to an instrumented binary. Executing this instrumented binary on an Alpha machine yields us the program characteristics that will be used throughout the analysis. Measuring these $p = 20$ program characteristics was done for the 79 program-input pairs mentioned in section 4.1.

2. In a second step, these 79 (number of program-input pairs) $\times$ 20 ($= p$, number of program characteristics) data points are normalized so that for each program characteristic the average equals zero and the variance equals one. On these normalized data points, principal components analysis (PCA) is done using STATISTICA [15], a package for statistical computations. This works as follows. A 2-dimensional matrix is presented as input to STATISTICA that has 20 columns representing the original program characteristics. There are 79 rows in this matrix representing the various program-input pairs. On this matrix, PCA is performed by STATISTICA which yields us p principal components.

3. Now, it is up to the user to determine how many principal components need to be retained. This decision is made based on the amount of variance accounted for by the retained principal components.

4. The q retained principal components can be analyzed and a meaningful interpretation can be given to them. This is done based on the coefficients a_{ij}, also called the *factor loadings*, as they occur in the following equation $Z_i = \sum_{j=1}^{p} a_{ij} X_j$, with $Z_i, 1 \leq i \leq q$ the principal components and $X_j, 1 \leq j \leq p$ the original program characteristics. A positive coefficient a_{ij} means a positive impact of program characteristic X_j on principal component Z_i; a negative coefficient a_{ij} implies a negative impact. If a coefficient a_{ij} is close to zero, this means X_j has (nearly) no impact on Z_i.

5. The program-input pairs can be displayed in the workload space built up by these q principal components. This can easily be done by computing $Z_i = \sum_{j=1}^{p} a_{ij} X_j$ for each program-input pair.

6. Rescale the q principal components to unit variance.

7. Within this rescaled q-dimensional space the Euclidean distance can be computed between the various program-input pairs as a reliable measure for the way program-input pairs differ from each other. There are two reasons supporting this statement. First, the values along the axes in this space are uncorrelated since they are determined by the principal components which are uncorrelated by construction. The absence of correlation is important when calculating the Euclidean distance because two correlated variables—that essentially measure the same thing—will contribute a similar amount to the overall distance as an independent variable; as such, these variables would be counted twice, which is undesirable. Second, through rescaling the principal components (previous step), the principal components are placed on a common scale. Without rescaling, the variance of a principal component—which is a manifestation of the correlation in the original data—would give a higher weight in the calculation of the Euclidean distance to correlated characteristics in the original data.

8. Finally, cluster analysis can be done using the distance between program-input pairs as determined in the previous step. Based on the dendrogram a clear view is given on the clustering within the workload space.

The reason why we chose to first perform PCA and subsequently cluster analysis instead of applying cluster analysis on the initial data is as follows. The original variables are highly correlated which implies that an Euclidean distance in this space is unreliable due to this correlation as explained previously. First performing PCA alleviates this problem. Another approach would have been to use the Mahalanobis distance[1] [13] which also takes into account the correlation between variables. However, there are two advantages of using PCA instead of the Mahalanobis distance. First, PCA gives us the opportunity to visualize the workload space in an understandable way. Second, PCA helps us in explaining why program-input pairs differ from each other in terms of the original program characteristics. Note that the Mahalanobis distance is equivalent to the distance measure that is used in this paper if all the principal components would have been used in our calculation of the Euclidean distance. Since we keep the leading principal components, which account for all but a small fraction of the variance, our distance measure becomes a very close approximation of the real Mahalanobis distance.

4. Evaluation

In this section, we first present the program-input pairs that are used in this study. Second, we show the results of performing the workload analysis as discussed in section 3.3. Finally, the methodology is validated in section 4.3.

1. For calculating the Mahalanobis distance, the overall covariance matrix of the original characteristics X can be used.

benchmark	input	dyn. I-cnt. (M)	I-footprint	D-footprint (K)
gcc	amptjp	835	147,402	375
	c-decl-s	835	147,369	375
	cccp	886	145,727	371
	cp-decl	1,103	143,153	579
	dbxout	141	120,057	215
	emit-rtl	104	127,974	108
	explow	225	105,222	280
	expr	768	142,308	653
	gcc	141	129,852	125
	genoutput	74	117,818	104
	genrecog	100	124,362	133
	insn-emit	126	84,777	199
	insn-recog	409	105,434	357
	integrate	188	133,068	199
	jump	133	126,400	130
	print-tree	136	118,051	201
	protoize	298	137,636	159
	recog	227	123,958	161
	regclass	91	125,328	117
	reload1	778	146,076	542
	stmt-protoize	654	148,026	261
	stmt	356	138,910	250
	toplev	168	125,810	218
	varasm	166	139,847	168
postgres	Q2	227	57,297	345
	Q3	948	56,676	358
	Q4	564	53,183	285
	Q5	7,015	60,519	654
	Q6	1,470	46,271	1,080
	Q7	932	69,551	631
	Q8	842	61,425	11,821
	Q9	9,343	68,837	10,429
	Q10	1,794	62,564	681
	Q11	188	65,747	572
	Q12	1,770	65,377	258
	Q13	325	65,322	264
	Q14	1,440	67,966	448
	Q15	1,641	67,246	640
	Q16	82,228	58,067	389
	Q17	183	54,835	366

Table 1: Characteristics of the benchmarks used (part 1) with their inputs, dynamic instruction count (in millions), instruction footprint (number of instructions executed at least once) and data memory footprint in 64-bit words (in thousands).

4.1 Experimental Setup

In this study, we have used the SPECint95 benchmarks (`http://www.spec.org`) and a database workload consisting of TPC-D queries (`http://www.tpc.org`), see Tables 1 and 2. The reason why we chose SPECint95 instead of the more recent SPECint2000 is to limit the simulation time. SPEC opted to dramatically increase the runtimes of the SPEC2000 benchmarks compared to the SPEC95 benchmarks which is beneficial for performance evaluation on real hardware but impractical for simulation purposes. In addition, there are more reference inputs provided with SPECint95 than with SPECint2000. For gcc (GNU C compiler) and li (lisp interpreter), we have used all the reference input files. For ijpeg (image processing), penguin, specmun and vigo were taken from the reference input set. The other

benchmark	input	dyn. I-cnt. (M)	I-footprint	D-footprint (K)
li	boyer	226	9,067	36
	browse	672	9,607	39
	ctak	583	8,106	18
	dderiv	777	9,200	16
	deriv	719	8,826	15
	destru2	2,541	9,182	16
	destrum2	2,555	9,182	16
	div2	2,514	8,546	19
	puzzle0	2	8,728	19
	tak2	6,892	8,079	16
	takr	1,125	8,070	36
	triang	3	9,008	15
ijpeg	band (2362x1570)	2,934	16,183	5,718
	beach (512x480)	254	16,039	405
	building (1181x1449)	1,626	16,224	2,742
	car (739x491)	373	16,294	596
	dessert (491x740)	353	16,267	587
	globe (512x512)	274	16,040	436
	kitty (512x482)	267	16,088	412
	monalisa (459x703)	259	16,160	508
	penguin (1024x739)	790	16,128	1,227
	specmun (1024x688)	730	15,952	1,136
	vigo (1024x768)	817	16,037	1,273
compress	14000000 e 2231 (ref)	60,102	4,507	4,601
	10000000 e 2231	42,936	4,507	3,318
	5000000 e 2231	21,495	4,494	1,715
	1000000 e 2231	4,342	4,490	433
	500000 e 2231	2,182	4,496	272
	100000 e 2231	423	4,361	142
m88ksim	train	24,959	11,306	4,834
	ref	71,161	14,287	4,834
vortex	train	3,244	78,766	1,266
	ref	92,555	78,650	5,117
perl	jumble	2,945	21,343	5,951
	primes	17,375	16,527	8
	scrabbl	28,251	21,674	4,098
go	50 9 2stone9.in	593	55,894	45
	50 21 9stone21.in	35,758	62,435	57
	50 21 5stone21.in	35,329	62,841	57

Table 2: Characteristics of the benchmarks used (part 2) with their inputs, dynamic instruction count (in millions), instruction footprint (number of instructions executed at least once) and data memory footprint in 64-bit words (in thousands).

images that served as input to ijpeg were taken from the web. The dimensions of the images are shown between brackets. For compress (text compression), we have adapted the reference input '14000000 e 2231' to obtain different input sets. For m88ksim (microprocessor simulation) and vortex (object-oriented database), we have used the train and the reference inputs. The same was done for perl (perl interpreter): jumble was taken from the train input, and primes and scrabbl were taken from the reference input; as well as for go (game): '50 9 2stone9.in' from the train input, and '50 21 9stone21.in' and '50 21 5 stone21.in' from the reference input.

In addition to SPECint95, we used postgres v6.3 running the decision support TPC-D queries over a 100MB Btree-indexed database. For postgres, we ran all TPC-D queries except for query 1 because of memory constraints on our machine.

The benchmarks were compiled with optimization level -O4 and linked statically with the -non_shared flag for the Alpha architecture.

4.2 Results

In this section, we will first perform PCA on the data for the various inputs of gcc. Subsequently, the same will be done for li and postgres. Finally, PCA and cluster analysis will be applied on the data for all the benchmark-input pairs of Tables 1 and 2. We present the data for gcc, li and postgres before presenting the analysis of all the program-input pairs because these three benchmarks illustrate different aspects of the techniques in terms of the number of retained principal components, clustering, etc.

4.2.1 Gcc

Based on Figure 1, we retained two principal components for the 24 input sets of gcc. These two principal components together account for 88.3% of the total variance; the first and the second component account for 67.9% and 20.4% of the total variance, respectively. In Figure 2, the factor loadings are presented for these two principal components. E.g., this means that the first principal component is computed as $PC1 = -0.141 \times ILP + 0.908 \times bimodal + 0.852 \times gshare + \ldots$. The first component is positively dominated, see Figure 2, by the branch prediction accuracy, the percentage arithmetic, logical and control operations and the D-cache miss rates; and negatively dominated by the number of instructions between two taken branches, the percentage load/store operations and the I-cache miss rates. The second component is positively dominated by the percentage shift operations; and negatively dominated by the ILP. Figure 3 presents the various input sets of gcc in the 2-dimensional space built up by these two components. Data points in this graph with a high value along the first component, have high branch prediction accuracies, high percentages of arithmetic, logical and control operations and high D-cache miss rates compared to the other data points; in addition, these data points also have a low number of instructions between two taken branches, a low percentage load/store operations and low I-cache miss rates. Note that only relative distances are important. For example, emit-rtl and insn-emit are relatively closer to each other than insn-emit and varasm.

Figure 3 shows that gcc executing input explow exhibits a different behavior than the other inputs. This is due to its high D-cache miss rates, its high branch prediction accuracies, its high percentage arithmetic, logical, shift and control operations; and its low ILP, its low percentage load/store operations, its low number of instructions between two taken branches, and its low I-cache miss rates. The difference in program behavior for inputs emit-rtl and insn-emit is mainly due to its high I-cache miss rate, its high percentage load/store operations, its low branch prediction accuracy, its low percentage arithmetic, logical and control operations and its low D-cache miss rates. This can be concluded from the factor loadings presented in Figure 2; we also verified that this is true by inspecting the original data. The strong cluster in the middle of the graph contains the inputs gcc, genoutput, genrecog, jump, regclass, stmt and stmt-protoize. Note that although the characteristics

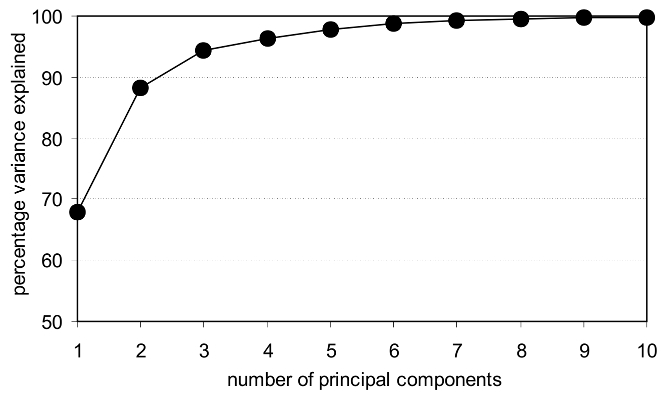


Figure 1: Amount of variance explained for gcc.

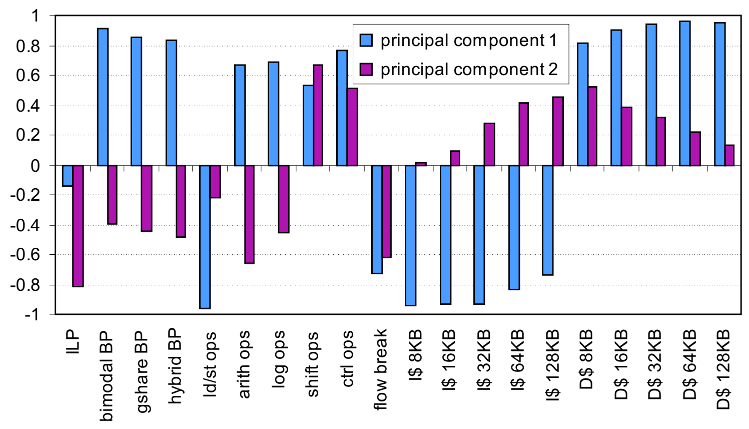


Figure 2: Factor loadings for gcc.

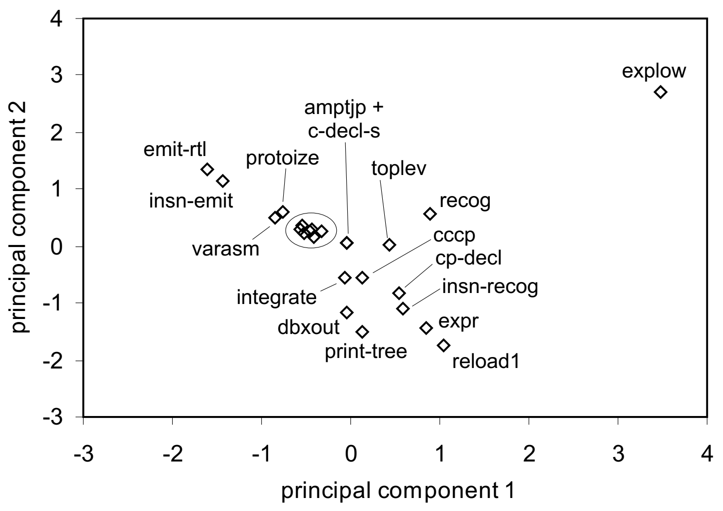


Figure 3: Workload space for gcc.

mentioned in Tables 1 and 2 (i.e., dynamic instruction count, I-footprint and D-footprint) are significantly different, these inputs result in a quite similar program behavior.

4.2.2 Li

Based on Figure 4, we retained three principal components for the lisp interpreter li. These three principal components together account for 87.0% of the total variance; the first component explains 42.2% of the total variance, the second component 33.4% and the third component 11.4%, respectively. The first component is positively dominated, see Figure 5, by the percentage shift operations and the miss rates for D-caches larger than 16KB; and negatively dominated by the miss rates for I-caches smaller than 16KB. The second component is positively dominated by the percentage arithmetic and logical operations and the I-cache miss rates for caches larger than 32KB; and negatively dominated by the percentage load/store operations and the number of instructions between two taken branches. The third component is negatively dominated by the amount of ILP and the percentage control operations.

Figure 6 presents the various input sets of li in the 3-dimensional space built up by the three retained principal components: the first component versus the second component on the left and the third component versus the second on the right. Seven input sets result in a behavior that is different from the other input sets. Three of these, namely takr, browse and boyer, have a higher miss rate for larger D-caches, a higher percentage shift operations, and a lower miss rate for the small I-caches. Two of these input sets, namely puzzle0 and triang, have a higher I-cache miss rate (larger than 16KB), a higher percentage arithmetic and logical operations, a lower percentage load/store operations, a smaller number of instructions between two taken branches. The two remaining input sets that show a different behavior from the other input sets, namely destru2 and destrum2, have a low value along the third principal component. As such, we conclude that these two inputs have a relatively high ILP, a relatively high percentage control operations, and a relatively high branch prediction accuracy for the bimodal branch predictor. The remaining five input sets show a similar behavior, namely dderiv, tak2, deriv, ctak and div2.

4.2.3 TPC-D

Based on Figure 7, we retained four principal components for postgres running 16 TPC-D queries, accounting for 84.5% of the total variance; the first component accounts for 45.8% of the total variance and is positively dominated, see Figure 8, by the percentage of arithmetic operations, the I-cache miss rate and the D-cache miss rate for small cache sizes; and negatively dominated by the percentage of logical operations. The second component accounts for 18.1% of the total variance and is positively dominated by the branch prediction accuracy. The third component accounts for 12.1% of the total variance and is negatively dominated by the D-cache miss rates for large cache sizes. The fourth component accounts for 8.5% of the total variance and is positively dominated by the percentage of shift operations and negatively dominated by the percentage memory operations.

Figure 9 shows the data points of postgres running the TPC-D queries in the 4-dimensional space built up by these four components. To display this 4-dimensional space understandably, we have shown the first principal component versus the second in one graph; and the

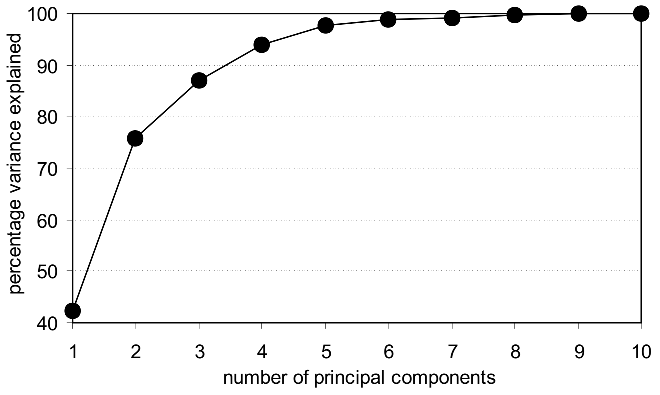


Figure 4: Amount of variance explained for li.

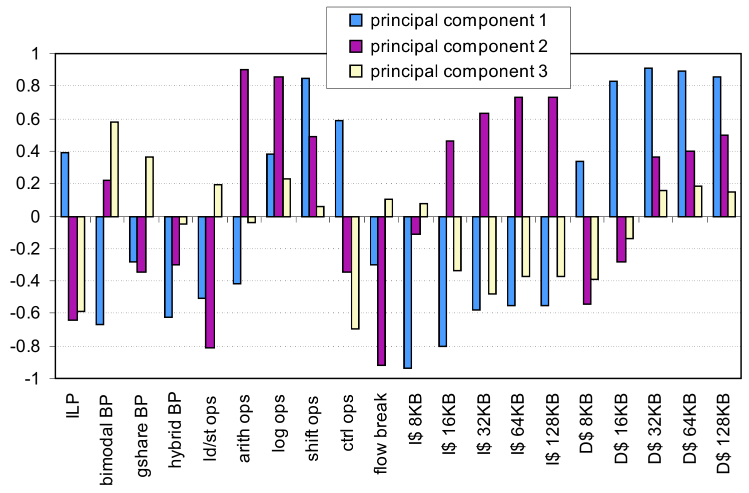


Figure 5: Factor loadings for li.

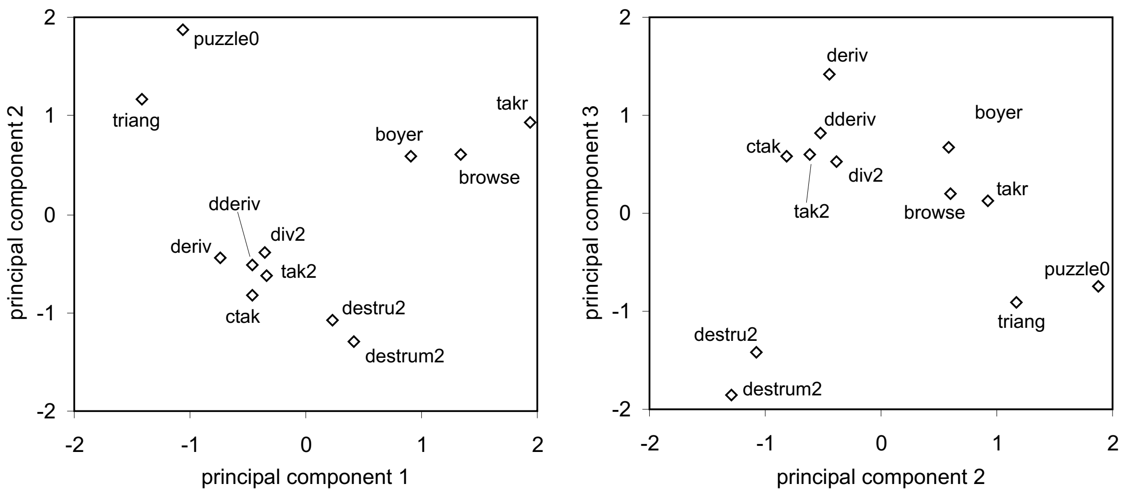


Figure 6: Lisp interpreter.

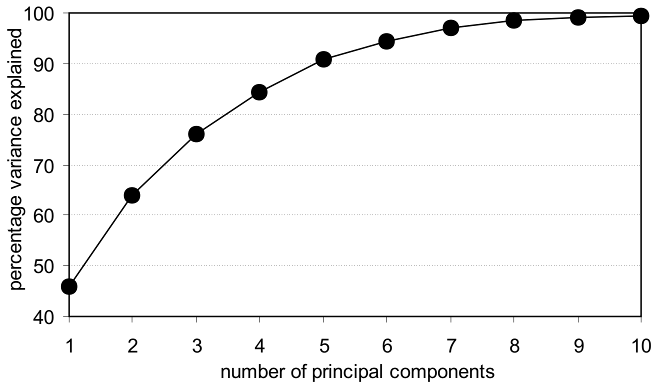


Figure 7: Amount of variance explained for postgres.

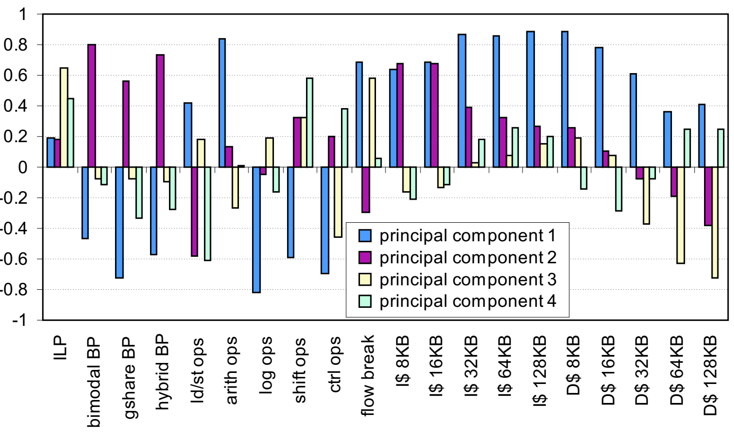


Figure 8: Factor loadings for postgres.

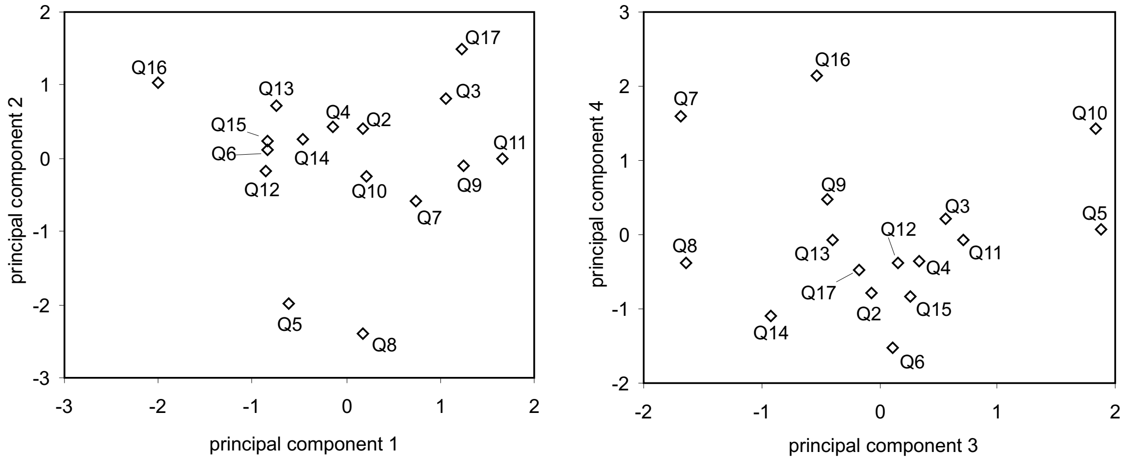


Figure 9: Workload space for postgres: first component vs. second component (graph on the left) and third vs. fourth component (graph on the right).

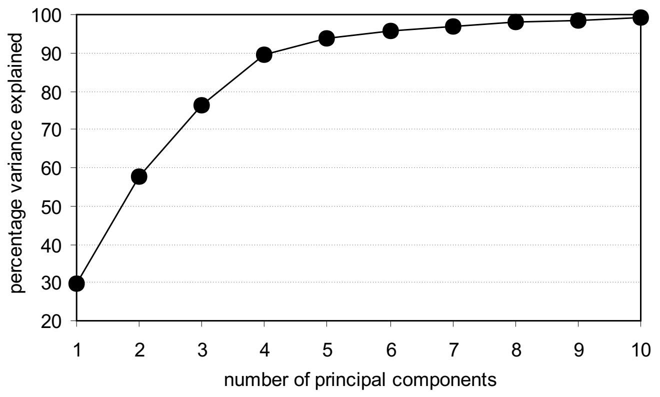


Figure 10: Amount of variance explained for all the program-input pairs.

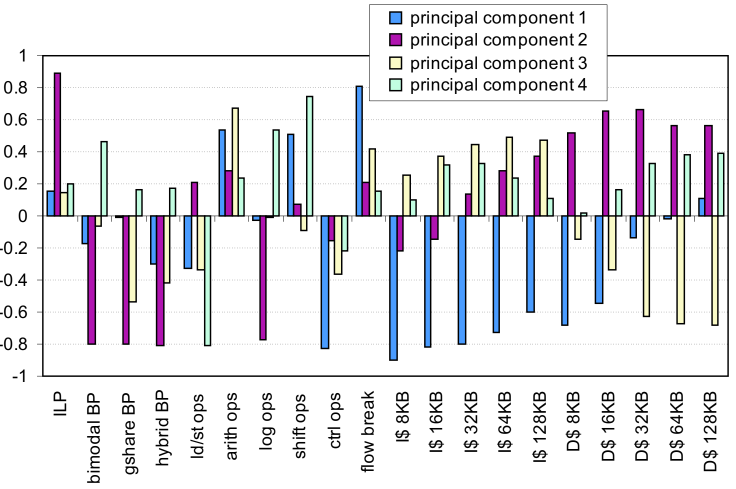


Figure 11: Factor loadings for all the program-input pairs.

third versus the fourth in another graph. These graphs do not reveal a strong clustering among the various queries. From this graph, we can also conclude that some queries exhibit a significantly different behavior than the other queries. For example, queries 7 and 8 have significantly higher D-cache miss rates for large cache sizes. Query 16 has, along the first principal component, a relatively low percentage arithmetic operations, relatively low I-cache miss rates, relatively low D-cache miss rates for small cache sizes, and a relatively high percentage logical operations; along the fourth principal component, query 16 has a higher percentage of shift operations and a lower percentage of load/store operations.

4.2.4 Workload Space

Now we change the scope to the entire workload space, i.e., by considering all the 79 program-input pairs from Tables 1 and 2. Based on Figure 10, we retain four principal components accounting for 89.5% of the total variance. The first component accounts for 29.7% of the total variance and is positively dominated, see Figure 11, by the number of instructions between two taken branches; and negatively dominated by the percentage control operations and the I-cache miss rates. The second principal component accounts for 28.0% of the total variance and is positively dominated by the amount of ILP and negatively dominated by the branch prediction accuracy and the percentage of logical operations. The third component accounts for 18.5% of the total variance and is positively dominated by the percentage arithmetic operations and negatively dominated by the D-cache miss rates for large cache sizes. The fourth component accounts for 13.3% of the total variance and is positively dominated by the percentage shift operations and negatively dominated by the percentage load/store operations.

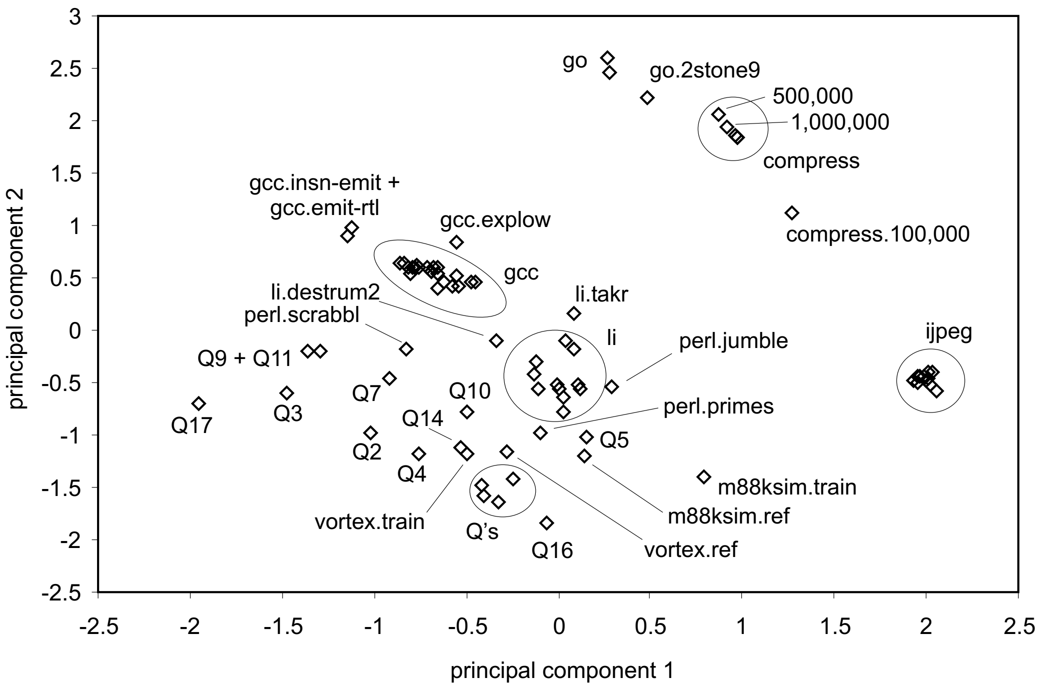
The results of the analyses that were done on these data, are shown in Figures 12 to 14. Figure 12 represents the program-input pairs in the 4-dimensional workload space built up by the four retained principal components. The dendrograms corresponding to the cluster analyses are shown in Figures 13 and 14 using the complete linkage rule and the weighted pair-group average linkage rule, respectively. Program-input pairs connected by small linkage distances are clustered in early iterations of the analysis and thus, exhibit similar behavior. Program-input pairs on the other hand, connected by large linkage distances exhibit different behavior.

Isolated points. From the data presented in Figures 12 to 14, it is clear that benchmarks go, ijpeg and compress are isolated in this 4-dimensional space. Indeed, in the dendrogram shown in Figure 14, these three benchmarks are connected to the other benchmarks through long linkage distances. E.g., go is connected to the other benchmarks with a linkage distance of 4.6 which is much larger than the linkage distance for more strongly clustered pairs. An explanation for this phenomenon can be found in Figure 12. Indeed, for go the discrimination is made along the second and third component. In other words, this is due to its low branch prediction accuracy, its low percentage logical operations, its high amount of ILP, its high percentage arithmetic operations, and its low D-cache miss rates for larger cache sizes. Compress discriminates itself along the third component which is mainly due to its high D-cache miss rates for large caches. For ijpeg, the different behavior is due to, along the first and fourth component, the high percentage of arithmetic, shift and control operations, the high number of instructions between two taken branches, the low percentage of load/store and control operations, and the low I-cache miss rates.

Strong clusters. There are also several strong clusters which suggests that only a small number (or in some cases, only one) of the input sets should be selected to represent the whole cluster. This will ultimately reduce the total simulation time since only a few (or only one) program-input pairs need to be simulated instead of all the pairs within that cluster. We can identify several strong clusters:

- The data points corresponding to the gcc benchmark are strongly clustered, except for the input sets emit-rtl, insn-emit and explow. These three input sets exhibit a different behavior from the rest of the input sets. However, emit-rtl and insn-emit have a quite similar behavior.

- The data points corresponding to the lisp interpreter li except for browse, boyer, takr, triang and puzzle0 are strongly clustered as well. This can be clearly seen from Figures 13 and 14 where this group is clustered with a linkage distance that is smaller



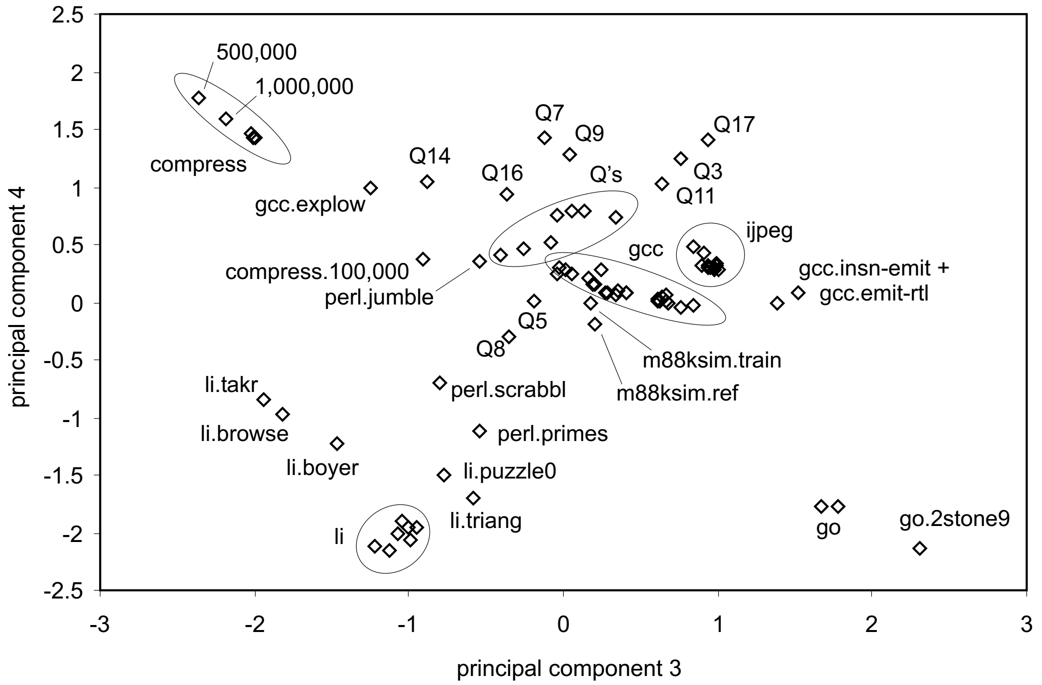


Figure 12: Workload space for all the program-input pairs: first component vs. second component (upper graph) and third vs. fourth component (bottom graph).

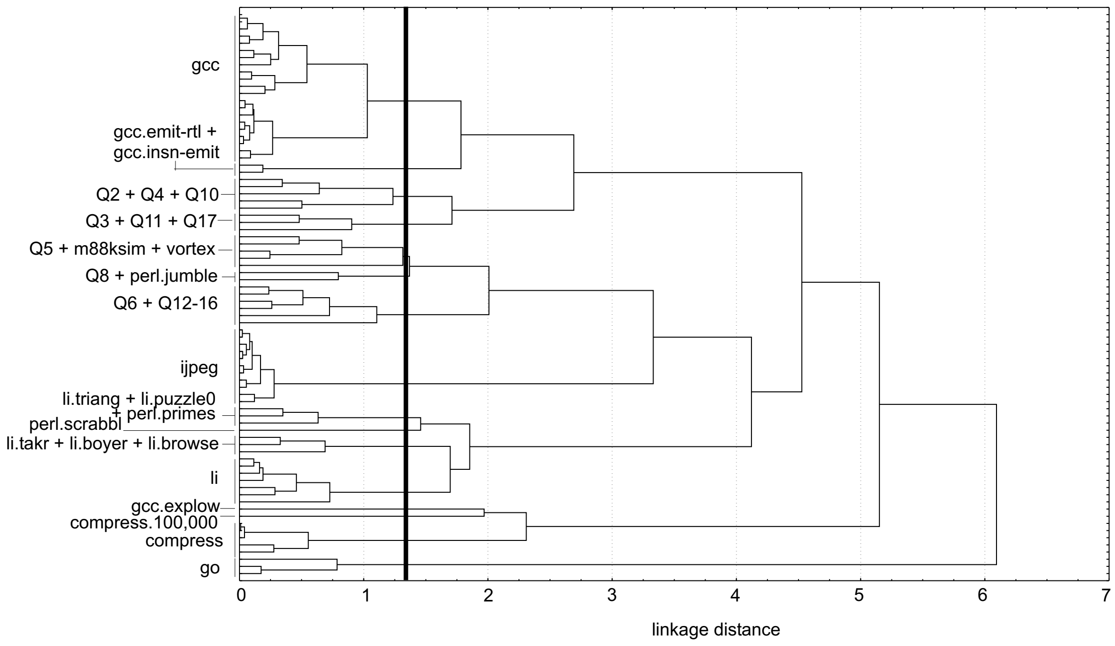


Figure 13: Dendrogram obtained through cluster analysis using the *complete* linkage rule. The thick vertical line shows the point where 16 clusters are formed (see section 5.1).

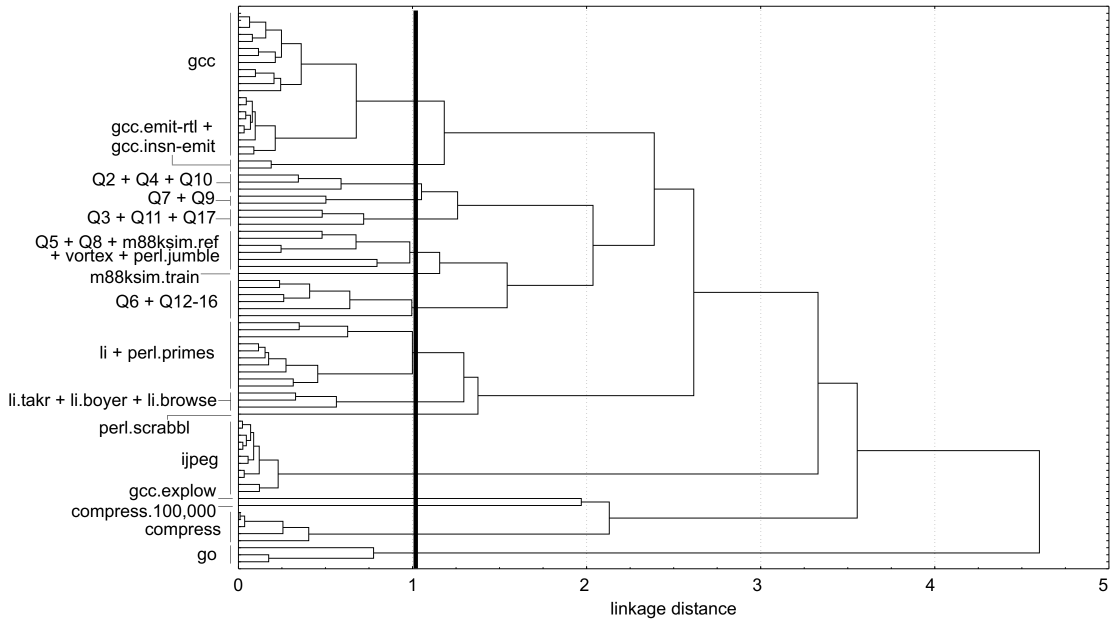


Figure 14: Dendrogram obtained through cluster analysis using the *weighted pair-group average* linkage rule. The thick vertical line shows the point where 16 clusters are formed (see section 5.1).

than 1. The three input sets browse, boyer and takr are grouped with the other li input sets with a linkage distance that is much larger.

- All input sets for ijpeg result in similar program behavior since all input sets are clustered in one group. An important conclusion from this analysis is that in spite of the differences in image dimensions, ranging from small images (512x482) to large images (2362x1570), the behavior of ijpeg remains quite the same.
- The input sets for compress are strongly clustered as well except for '100000 e 2231'.

Reference vs. train inputs. Along with its benchmark suite SPECint, SPEC releases reference and train inputs. The purpose for the train inputs is to provide input sets that should be used for profile-based compiler optimizations. The reference input is then used for reporting results. Within the context of this paper, the availability of reference and train input sets is important for two reasons. First, when reference and train inputs result in similar program behavior we can expect that profile-driven optimizations will be effective. Second, train inputs have smaller dynamic instruction counts which make them candidates for more efficient simulation runs. I.e., when a train input exhibits a similar behavior as a reference input, the train input can be used instead of the reference input for exploring the design space which will lead to a more efficient design flow.

In this respect, we take the following conclusions:

- The train and reference input for vortex exhibit similar program behavior with a linkage distance that is smaller than 0.4.
- For m88ksim on the other hand, this is less the case—the linkage distance is larger than 1.
- For go, the train input '50 9 2stone9.in' leads to a behavior that is slightly different from the behavior of the reference inputs '50 21 9stone21.in' and '50 21 5stone21.in'. The two reference inputs on the other hand, lead to similar behavior.
- All three inputs for perl (two reference inputs and one train input) result in quite different behavior.

From these observations, we can state that for some benchmarks the train input behaves similarly to the reference input. For other benchmarks this might not be true. As such, using train inputs when reporting performance results in architectural research might be reliable in some cases and unreliable in other cases.

Reduced inputs. KleinOsowski *et al.* [16] propose to reduce the simulation time of benchmarks by using reduced input sets. The final goal of their work is to identify a reduced input for each benchmark that results in similar behavior as the reference input but with a significant reduction in dynamic instruction counts and thus simulation time. From the data in Figures 12 to 14, we can conclude that, e.g., for ijpeg this is a viable option since small images result in quite similar behavior as large images. For compress on the other hand, we have to be careful: the reduced input '100000 e 2231' which was derived from the reference input '14000000 e 2231' results in quite different behavior. The other reduced inputs for compress lead to a behavior that is similar to the reference input.

Impact of input set on program behavior. As stated before, this analysis is useful for identifying the impact of input sets on program behavior. For example:

- The data points corresponding to postgres running the TPC-D queries are weakly clustered. For example, the spread along the first principal component is very large. As such, a wide range of different I-cache behavior can be observed when running the TPC-D queries. Note also that all the queries result in an above-average branch prediction accuracy, a high percentage of logical operations and low ILP (negative value along the second principal component).
- The difference in behavior between the input sets for compress is mainly due to the difference in the data cache miss rates (along the third principal component).
- In general, the variation between programs is larger than the variation between input sets for the same program. Thus, when composing a workload, it is more important to select different programs with a well chosen input set than to include various inputs for the same program. For example, the program-input pairs for gcc (except for explow, emit-rtl and insn-emit) and ijpeg are strongly clustered in the workload space. In some cases however, for example postgres and perl, the input set has a relatively high impact on program behavior.

4.3 Preliminary validation

As stated before, the purpose of the analysis presented in this paper is to identify clusters of program-input pairs that exhibit similar behavior. We will show that pairs that are close to each other in the workload space indeed exhibit similar behavior when changes are made to the microarchitecture on which they run.

In this section, we present a preliminary validation in which we observe the behavior of several input sets for gcc and one input set of each of the following benchmarks: go and li. The reason for doing a validation using a selected number of program-input pairs instead of all 79 program-input pairs is to limit simulation time. The simulations that are presented in this section already took several weeks. As a consequence, simulating all program-input pairs would have been impractically long[2]. However, since gcc presents a very diverse behavior (strong clustering versus isolated points, see Figure 3), we believe that a succesful validation on gcc with some additional program-input pairs can be extrapolated to the complete workload space with confidence.

We have used seven input sets for gcc, namely explow, insn-recog, gcc, genoutput, stmt, insn-emit and emit-rtl. According to the analysis done in section 4.2.1, emit-rtl and insn-emit should exhibit a similar behavior; the same should be true for gcc, genoutput and stmt. explow and insn-recog on the other hand, should result in a different program behavior since they are quite far away from the other input sets that are selected for this analysis. For go and li, we used 50 9 2stone9.in and boyer, respectively.

We used SimpleScalar v3.0 [17] for the Alpha architecture as simulation tool for this analysis. The baseline architecture has a window size of 64 instructions and an issue width of 4.

2. This is exactly the problem we are trying to solve.

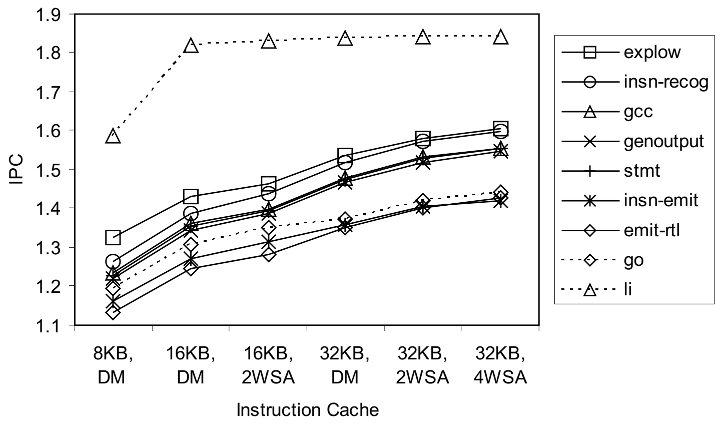


Figure 15: IPC as a function of the I-cache configuration; 16KB DM D-cache and 8K-entry bimodal branch predictor.

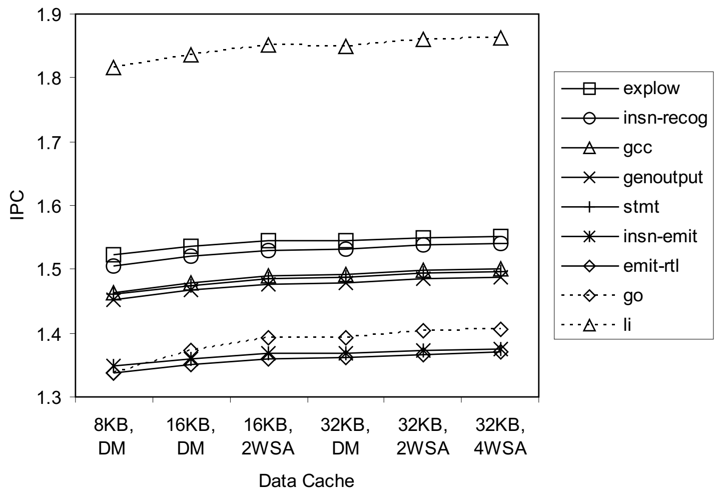


Figure 16: IPC as a function of the D-cache configuration; 32KB DM I-cache and 8K-entry bimodal branch predictor.

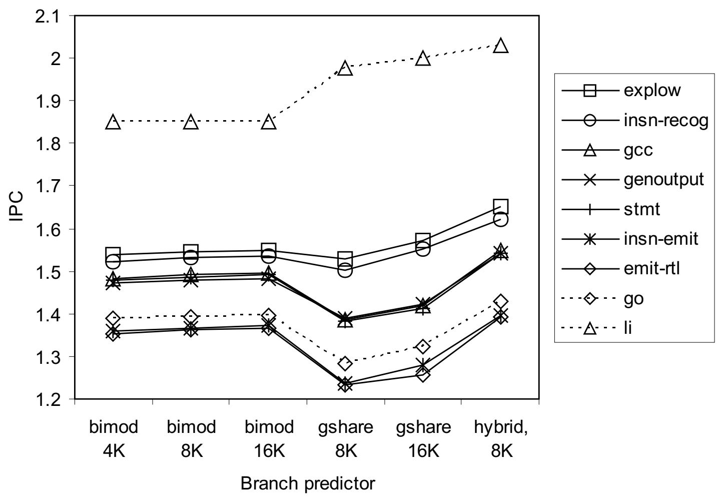


Figure 17: IPC a function of the branch predictor configuration; 32KB DM I-cache and 32KB DM D-cache.

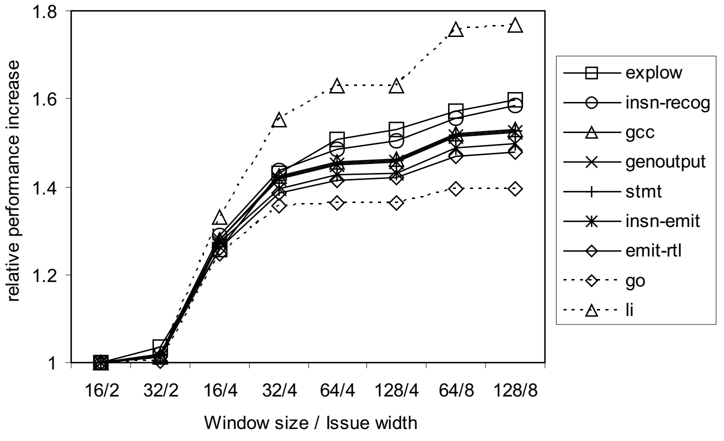


Figure 18: Performance increase w.r.t. the 16/2 configuration as a function of the window size and the issue width; 32KB DM I-cache, 32KB DM D-cache and 8K-entry branch predictor.

In Figures 15, 16, 17 and 18, the number of instructions retired per cycle (IPC) is shown as a function of the I-cache configuration, the D-cache configuration, the branch predictor and the window size versus issue width configuration, respectively. We will first discuss the results for gcc. Afterwards, we will detail on the other benchmarks.

For gcc, we clearly identify three groups of input sets that have similar behavior, namely (i) explow and insn-recog, (ii) gcc, genoutput and stmt, and (iii) insn-emit and emit-rtl. For example, in Figure 17, the branch behavior of group (i) is significantly different from the other input sets. Or, in Figure 18, the scaling behavior as a function of window size and issue width is quite different for all three groups. This could be expected for groups (ii) and (iii) as discussed earlier. The fact that explow and insn-recog exhibit similar behavior on the other hand, is unexpected since these two input sets are quite far away from each other in the workload space, see Figure 3. Note that this reveals a property of this methodology that was discussed is section 2, namely that not all program characteristics included in the analysis may equally influence performance. This might lead to a discrimation of program-input pairs on characteristics that have a minor impact on performance.

The other two benchmarks, go and li, clearly exhibit a different behavior on all four graphs. This could be expected from the analysis done in section 4.2.4 since PCA and cluster analysis pointed out that these benchmarks have a different behavior. Most of the mutual differences can be explained from the analysis done in this paper. For example, go has a different D-cache behavior than gcc which is clearly reflected in Figure 16. Also, li has a different I-cache behavior than gcc and go which is reflected in Figure 15. Other differences however, are more difficult to explain. Again, this phenomenon is due to the fact that some microarchitectural parameters have a minor impact on performance for a given microarchitectural configuration. However, for other microarchitectural configurations we can still expect different behavior. For example, go has a different branch behavior than gcc, according to the analysis done in section 4.2.4; in Figure 17, go and gcc exhibit a comparable behavior.

5. Applications

As discussed in the introduction, the methodology presented so far has several interesting applications. In section 4.2.4, we have extensively detailed on one particular application, namely getting insight in the impact of input data sets on program behavior. As such, we have also touched on its relationship with workload design, or the composition of a representative workload while taking into account the total simulation time. In this section, we will further detail on this important application by discussing (i) the selection of a restricted number of representative program-input pairs and (ii) the use of this methodology in the context of trace sampling.

5.1 Workload composition

Consider the case where we assume that all the 79 program-input pairs given in Tables 1 and 2 are representative for the target domain of operation. Obviously, simulating these 79 program-input pairs on an architectural simulator is infeasible. Indeed, the total dynamic instruction count of all these program-input pairs together exceeds 593 billions of instructions, or 23 days of simulation when using SimpleScalars out-of-order simulator at a

cluster	number	benchmarks	representative	dyn (M)
1	5	Q2, Q4, Q7, Q9 and Q10	Q4	564
2	6	Q6 and Q12-Q16	Q13	325
3	3	Q3, Q11 and Q17	Q3	948
4	5	Q5, m88ksim and vortex	vortex.train	3,244
5	2	Q8 and perl.jumble	Q8	842
6	21	all the inputs associated with gcc except for insn-emit, emit-rtl and explow	gcc.protoize	298
7	2	gcc.emit-rtl and gcc.insn-emit	gcc.emit-rtl	104
8	1	gcc.explow	gcc.explow	225
9	7	all inputs associated with li except for takr, browse, boyer, triang and puzzle0	li.ctak	583
10	3	li.takr, li.browse and li.boyer	li.browse	672
11	3	li.triang, li.puzzle0 and perl.primes	li.puzzle0	2
12	5	all inputs associated with compress except for compress.100,000	compress.1,000,000	4,342
13	1	compress.100,000	compress.100,000	423
14	11	all inputs associated with ijpeg	ijpeg.kitty	267
15	3	all inputs associated with go	go.5stone21	35,329
16	1	perl.scrabbl	perl.scrabbl	28,251
				76,419

Table 3: Selecting a workload consisting of 16 program-input pairs: via complete linkage cluster analysis.

cluster	number	benchmarks	representative	dyn (M)
1	3	Q2, Q4 and Q10	Q4	564
2	6	Q6 and Q12-Q16	Q13	325
3	6	Q5, Q8, perl.jumble, m88ksim.ref and vortex	vortex.train	3,244
4	3	Q3, Q11 and Q17	Q3	948
5	2	Q7 and Q9	Q7	932
6	21	all the inputs associated with gcc except for insn-emit, emit-rtl and explow	gcc.protoize	298
7	2	gcc.emit-rtl and gcc.insn-emit	gcc.emit-rtl	104
8	1	gcc.explow	gcc.explow	225
9	10	all inputs associated with li except for takr, browse and boyer; perl.primes	li.ctak	583
10	3	li.takr, li.browse and li.boyer	li.browse	672
11	5	all inputs associated with compress except for compress.100,000	compress.1,000,000	4,342
12	1	compress.100,000	compress.100,000	423
13	11	all inputs associated with ijpeg	ijpeg.kitty	267
14	3	all inputs associated with go	go.5stone21	35,329
15	1	m88ksim.train	m88ksim.train	24,959
16	1	perl.scrabbl	perl.scrabbl	28,251
				101,366

Table 4: Selecting a workload consisting of 16 program-input pairs: via weighted pair-group average linkage cluster analysis.

cluster	number	benchmarks	representative	dyn (M)
1	5	Q2, Q4, Q10 and vortex	Q4	564
2	6	Q6 and Q12-Q16	Q13	325
3	3	Q3, Q11 and Q17	Q3	948
4	2	Q7 and Q9	Q7	932
5	3	Q5, Q8 and perl.jumble	Q8	842
6	2	perl.primes and perl.scrabbl	perl.pimes	17,375
7	11	the following gcc inputs: emit-rtl, gcc, jump genoutput, genrecog, insn-emit, protoize regclass, stmt, stmt-protoize and varasm	gcc.protoize	298
8	12	the following gcc inputs: amptjp, cccp, c-decl-s cp-decl, dbxout, expr, insn-recog, integrate print-tree, recog, reload1, toplev	print-tree	136
9	1	gcc.explow	gcc.explow	225
10	9	all inputs associated with li except for takr, browse and boyer	li.ctak	583
11	3	li.takr, li.browse and li.boyer	li.browse	672
12	5	all inputs associated with compress except for compress.100,000	compress.1,000,000	4,342
13	1	compress.100,000	compress.100,000	423
14	11	all inputs associated with ijpeg	ijpeg.kitty	267
15	3	all inputs associated with go	go.5stone21	35,329
16	2	m88ksim.train and m88ksim.ref	m88ksim.train	24,959
				88,220

Table 5: Selecting a workload consisting of 16 program-input pairs: via K-means clustering.

speed of 300,000 instructions per second [18]. As such, three weeks of simulation yield us a performance metric of one single microarchitectural design point. If we take into account that a large number of design points need to be evaluated, we can conclude that this approach is impractical. One possible solution to this problem would be to run a huge number of simulations in parallel on a huge number of machines. Since machines are quite cheap nowadays, the equipment cost can be modest. However, the simulations might still be too time-consuming. For example, simulating one single microarchitectural configuration using the vortex.ref program-input pair, which has a dynamic instruction count of more than 92 billion instruction, still takes several days.

Therefore, we propose to reduce this large number of program-input pairs to a limited number, say 16, in order to reduce the total simulation time. For this purpose, we can apply the methodology presented in this paper. We have studied three possible clustering strategies: (i) linkage clustering using the complete linkage rule (see Figure 13), (ii) linkage clustering using the weighted pair-group average linkage rule (see Figure 14) and (iii) K-means clustering with K set to 16. The reason why we consider three different clustering strategies is to investigate how much the influence is of the applied clustering techniques on the final result. The results of this experiment are shown in Tables 3 to 5. For each cluster, the number of program-input pairs in each cluster is given, the program-input pairs themselves, a representative for each cluster and the dynamic instruction count (in millions) for each representative. The representative for each cluster was chosen by taking the program-input pair with the minimal dynamic instruction count that is as close as possible to the center of the cluster it belongs to. Another approach that can be used is to pick a limited number of extreme program-input pairs for each cluster—an extreme point in a cluster is a point that is situated at the 'boundary' of the cluster. The rationale behind this approach would be that the behavior of program-input pairs in the middle of a cluster can be extracted from the behavior of the extremes, for example through interpolation. In this paper, we did take the latter approach because we believe that processor performance of a program-input pair in the middle of a cluster cannot be accurately estimated by using extremes and interpolation because determining the interpolation curve is extremely difficult. The reason for this is that the influence of a program characteristic in one processor configuration can be completely different from the influence in case of another processor configuration. For this reason, we used the first approach, namely selecting a representative that is close enough to the center of its cluster.

We can make several interesting observations from Tables 3 to 5:

- the TPC-D queries are spread over 5 clusters. As discussed previously, this comes from the fact that the input set has a large impact on program behavior for postgres;
- the inputs for gcc also result in a spreading over multiple clusters. The two linkage clustering techniques, Tables 3 and 4, group all the input sets of gcc together except for the three inputs, insn-emit, emit-rtl and explow. The K-means clustering approach, see Table 5 divides gcc into two major clusters plus a cluster containing only explow. Roughly, we can state that these two major clusters correspond to the left part and the right part of the graph in Figure 3.
- several SPECint95 benchmarks are often classified with TPC-D queries. For example, vortex is classified with TPC-D queries Q2, Q4 and Q10 by the K-means clustering technique, see Table 5. The SPECint95 benchmarks perl and m88ksim are often classified with Q5 and Q8, although slightly different under the various clustering strategies.
- in case of, e.g., the complete linkage clustering, the total dynamic instruction count is reduced by a factor 7.8.

Note that in general the classifications made by the three clustering approaches are quite consistent. Indeed, most clusters occur in all three classifications. However, there are a number of program-input pairs that are classified in slightly different ways under the various clustering techniques. This is due to the fact that these program-input pairs are borderline cases that are somehow difficult to classify.

5.2 Trace sampling

Another interesting approach to the simulation problem is trace sampling [4, 5, 6, 7, 8, 9, 10, 11]. In trace sampling, several samples are taken from a program execution so that the total number of instructions in the samples is significantly less than the total number of instructions of a complete execution. In order to make viable design decisions based on these sampled traces, a sampled trace should be representative for the complete program execution. The methodology presented here could also be used to validate sampled traces.

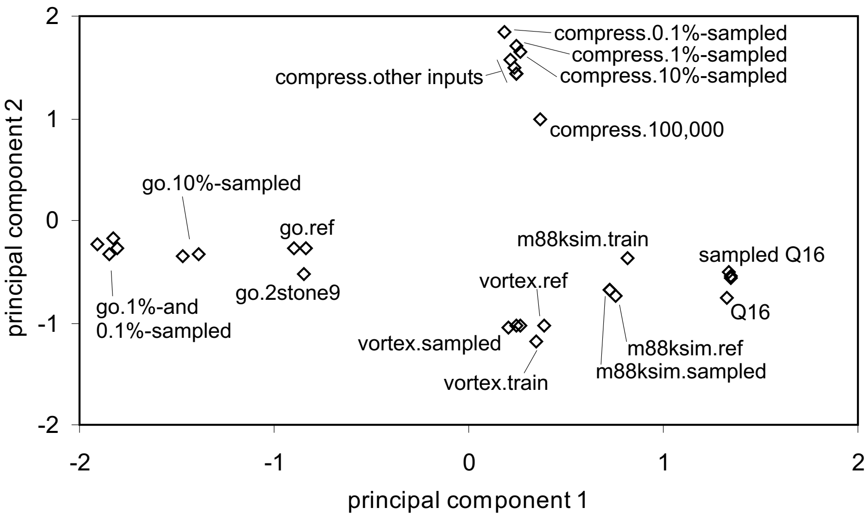


Figure 19: Workload space for the six long running program-input pairs and their sampled traces.

Indeed, a sampled trace that is situated close to its reference trace in the workload space could be considered as being representative.

To demonstrate the applicability of the methodology presented in this paper for trace sampling, we have set up the following experiment. We have considered sampled traces for six long running program-input pairs: Q16, m88ksim.ref, vortex.ref, compress, go.5stone21 and go.9stone21. For each of these program-input pairs, we have considered three sampled traces, with a sampling rate of 10%, 1% and 0.1%, respectively. This was done by taking samples of 1 million instructions every 10 million, 100 million and 1 billion instructions, respectively. Note that we assume a perfect warmup, i.e., perfectly warmed-up caches and perfectly warmed-up branch predictors, at the beginning of each sample. As such, we focus on the representativeness of the sampled traces.

For each of these sampled traces we have measured the program characteristics as mentioned in section 2. Subsequently, we have done a principal components analysis. The 2-dimensional space that results from this analysis is displayed in Figure 19. The total variance accounted for by the two principal components is 84.1%; the first principal component accounts for 56.6% and the second component accounts for 27.6%. Several interesting conclusions can be taken from this graph. First, as expected, the sampled trace with a sampling rate of 10% is closer to the reference input than the 1% and the 0.1% sampled traces in general, see for example go and compress. Second, in some cases a sampled trace seems to be a better option than a reduced input, e.g., for m88ksim the sampled traces are closer to the reference input than the train input. On the other hand, this seems not to be true for go; indeed, the train input 2stone9 is closer to the reference inputs than the sampled traces. Third, in some cases, e.g., for TPC-D query Q16, the 0.1% sampled trace seems to be nearly as representative as the 10% sampled trace. In conclusion, we can state that the methodology presented in this paper can be useful for measuring the representativeness of sampled traces.

6. Related work

Saavedra and Smith [19] addressed the problem of measuring benchmark similarity. For this purpose they presented a metric that is based on dynamic program characteristics for the Fortran language, for example the instruction mix, the number of function calls, the number of address computations, etc. For measuring the difference between benchmarks they used the squared Euclidean distance. The methodology in this paper differs from the one presented by Saavedra and Smith [19] for two reasons. First, the program characteristics measured here are more suited for performance prediction of contemporary architectures since we include branch prediction accuracy, cache miss rates, ILP, etc. Second, we prefer to work with uncorrelated program characteristics (obtained after PCA) for quantifying differences between program-input pairs, as extensively argued in section 3.3.

Hsu *et al.* [20] studied the impact of input data sets on program behavior using high-level metrics, such as procedure level profiles and IPC, as well as low-level metrics, such as the execution paths leading to data cache misses. They conclude that the test input set as provided by SPEC is not suitable for simulation purposes because the execution profile is quite different from the profile obtained from the reference input. The train input was found to be better than the test input. However, they observed that the execution paths leading to data cache misses are very different between the train input and the reference input.

Yi, Lilja and Hawkins [21] propose a technique for classifying benchmarks with similar behavior, i.e., by grouping benchmarks that stress the same processor components to similar degrees. Their method is based on a Plackett-Burman design.

KleinOsowski *et al.*[16] propose to reduce the simulation time of the SPEC 2000 benchmark suite by using reduced input data sets. Instead of using the reference input data sets provided by SPEC, which result in unreasonably long simulation times, they propose to use smaller input data sets that accurately reflect the behavior of the full reference input sets. For determining whether two input sets result in more or less the same behavior, they used the chi-squared statistic based on the function-level execution profiles for each input set. Note that a resemblance of function-level execution profiles does not necessarily imply a resemblance of other program characteristics which are probably more directly related to performance, such as instruction mix, cache behavior, etc. The latter approach was taken in this paper for exactly that reason. KleinOsowski *et al.* also recognized that this is a potential problem. The methodology presented in this paper can be used as well for selecting reduced input data sets. A reference input set and a resembling reduced input set will be situated close to each other in the q-dimensional space built up by the principal components.

As discussed in section 5.2, trace sampling is also closely related to the topic of this paper. Iyengar *et al.* [6] propose an R-metric for measuring the representativeness of a sampled trace. Lafage and Seznec [8] propose to choose representative samples using cluster analysis. They applied their method for data cache simulations. Characterizing the individual samples is done using two microarchitecture-independent metrics, one that captures the temporal locality of the memory reference stream and one that captures the spatial locality of the memory reference stream. Recently, Sherwood *et al.* [10, 11] characterize the large scale behavior (as seen over billions of instructions) of computer programs using one microarchitecture-independent metric, namely the *Basic Block Vector*. In essence, the BBV quantifies the basic block execution profile. By measuring a BBV for each program slice (containing for example 100 million instructions) the various program slices can be characterized. Subsequently, the program slices with similar BBVs and thus similar behavior are grouped together through clustering. For each cluster, a representative sample can be chosen that can be used for trace sampling.

Trace sampling and reduced input sets are compared in [22]. The authors conclude, completely consistent with our conclusions made in section 5.2, that both approaches can lead to significant prediction errors when compared to the execution of the reference input. However, both approaches have their own benefits. Reduced inputs allow the execution of a program from the beginning to the end; trace sampling allows flexibility by varying the sample rate, the sample length, the number of samples, etc.

Recently, a new fast simulation technique was introduced, namely statistical simulation [23, 24, 25, 26]. In statistical simulation, a statistical profile is extracted from a program execution which is subsequently fed into a synthetic trace generator. The synthetic trace being generated can then be executed on a trace-driven simulator which yields performance estimates. Due to the statistical nature of the technique, the total number of instructions in a synthetic trace can be limited since the performance characteristics while simulating a synthetic trace quickly converge. Typically, no more than one million instructions need to be simulated to obtain a stable performance estimate. Statistical simulation is related to the research topic presented in this paper, since the success of both techniques relies on choosing relevant program characteristics to be incorporated in the analysis. For statistical simulation, relevant program characteristics are needed to obtain a high accuracy; for the technique presented in this paper, relevant program characteristics are needed to construct a reliable workload space reduction.

Another possible application of using a data reduction technique such as principal components analysis, is to compare different workloads. In [27], Chow *et al.* used PCA to compare the branch behavior of Java and non-Java workloads. The interesting aspect of using PCA in this context is that PCA is able to identify why two workloads differ. This can be done by analyzing the principal components. They conclude for example that Java workloads tend to have more indirect calls while non-Java workloads tend to have more direct and indirect jumps.

Huang and Shen [28] quantify the impact of input data sets on the bandwidth spectrum of computer programs. The bandwidth spectrum measures the average bandwidth requirements of a program's instruction and data stream as a function of the available local memory. They conclude that the basic shape of the bandwidth spectrum does not change much with varying inputs.

Changes in program behavior due to different input data sets are also important for profile-guided compilation [29], where profiling information from a past run is used by the compiler to guide its optimizations. Fisher and Freudenberger [30] studied whether branch directions from previous runs of a program (using different input sets) are good predictors of the branch directions in future runs. Their study concludes that branches generally take the same directions in different runs of a program. However, they warn that some runs of a program exercise entirely different parts of the program. Hence, these runs cannot be used to make predictions about each other. By using the average branch direction over a number of runs, this problem can be avoided. Wall [31] studied several types of profiles such as basic block counts and the number of references to global variables. He measured the usefulness of a profile as the speedup obtained when that profile is used in a profile-guided compiler optimization. Seemingly, the best results are obtained when the same input is used for profiling and measuring the speedup. This implies that every input is different in some sense and leads to different compiler optimizations.

7. Conclusion

In microprocessor design, it is important to have a representative workload to make correct design decisions. This paper proposes the use of principal components analysis and cluster analysis to efficiently explore the workload space. In this workload space, benchmark-input pairs can be displayed and a distance can be computed that gives us an idea of the behavioral differences between these benchmark-input pairs. This representation can be used to measure the impact of input data sets on program behavior. In addition, our methodology was succesfully validated by showing that program-input pairs that are close to each other in the principal components space, indeed exhibit similar behavior as a function of microarchitectural changes. Interesting applications for this technique are the composition of workloads and the validation of sampled traces.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments which definitely improved the quality of the paper. Lieven Eeckhout and Hans Vandierendonck are supported by a grant from the Flemish Institute for the Promotion of the Scientific-Technological Research in the Industry (IWT).

References

[1] P. Bose and T. M. Conte, "Performance analysis and its impact on design," *IEEE Computer*, vol. 31, pp. 41–49, May 1998.

[2] L. K. John, P. Vasudevan, and J. Sabarinathan, "Workload characterization: Motivation, goals and methodology," in *Workload Characterization: Methodology and Case Studies* (L. K. John and A. M. G. Maynard, eds.), IEEE Computer Society, 1999.

[3] 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-2002)*, pp. 83–94, Sept. 2002.

[4] T. M. Conte, M. A. Hirsch, and K. N. Menezes, "Reducing state loss for effective trace sampling of superscalar processors," in *Proceedings of the 1996 International Conference on Computer Design (ICCD-96)*, pp. 468–477, Oct. 1996.

[5] P. K. Dubey and R. Nair, "Profile-driven sampled trace generation," Tech. Rep. RC 20041, IBM Research Division, T. J. Watson Research Center, Apr. 1995.

[6] V. S. Iyengar, L. H. Trevillyan, and P. Bose, "Representative traces for processor models with infinite cache," in *Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA-2)*, pp. 62–73, Feb. 1996.

[7] R. E. Kessler, M. D. Hill, and D. A. Wood, "A comparison of trace-sampling techniques for multi-megabyte caches," *IEEE Transactions on Computers*, vol. 43, pp. 664–675, June 1994.

[8] T. Lafage and A. Seznec, "Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream," in *IEEE 3rd Annual Workshop on Workload Characterization (WWC-2000) held in conjunction with the International Conference on Computer Design (ICCD-2000)*, Sept. 2000.

[9] G. Lauterbach, "Accelerating architectural simulation by parallel execution of trace samples," Tech. Rep. SMLI TR-93-22, Sun Microsystems Laboratories Inc., Dec. 1993.

[10] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, pp. 3–14, Sept. 2001.

[11] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pp. 45–57, Oct. 2002.

[12] S. McFarling, "Combining branch predictors," Tech. Rep. WRL TN-36, Digital Western Research Laboratory, June 1993.

[13] B. F. J. Manly, *Multivariate Statistical Methods: A primer.* Chapman & Hall, second ed., 1994.

[14] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," Tech. Rep. 94/2, Western Research Lab, Compaq, Mar. 1994.

[15] StatSoft, Inc. STATISTICA for Windows. Computer program manual. 1999. `http://www.statsoft.com`.

[16] A. J. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja, "Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research," in *Workload Characterization of Emerging Computer Applications, Proceedings of the IEEE 3rd Annual Workshop on Workload Characterization (WWC-2000) held in conjunction with the International Conference on Computer Design (ICCD-2000)*, pp. 83–100, Sept. 2000.

[17] D. C. Burger and T. M. Austin, "The SimpleScalar Tool Set." Computer Architecture News, 1997. See also `http://www.simplescalar.com` for more information.

[18] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE Computer*, vol. 35, pp. 59–67, Feb. 2002.

[19] R. H. Saavedra and A. J. Smith, "Analysis of benchmark characteristics and benchmark performance prediction," *ACM Transactions on Computer Systems*, vol. 14, pp. 344–384, Nov. 1996.

[20] W. C. Hsu, H. Chen, P. Y. Yew, and D.-Y. Chen, "On the predictability of program behavior using different input data sets," in *Proceedings of the Sixth Workshop on Interaction between Compilers and Computer Architectures (INTERACT 2002), held in conjunction with the Eighth International Symposium on High-Performance Computer Architecture (HPCA-8)*, pp. 45–53, Feb. 2002.

[21] J. J. Yi, D. L. Lilja, and D. M. Hawkins, "A statistically rigorous approach for improving simulation methodology," in *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, Feb. 2003.

[22] J. W. Haskins Jr., K. Skadron, A. J. KleinOsowski, and D. J. Lilja, "Techniques for accurate, accelerated processor simulation: An analysis of reduced inputs and sampling," Tech. Rep. CS-2002-01, University of Virginia—Dept. of Computer Science, Jan. 2002.

[23] L. Eeckhout and K. De Bosschere, "Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, pp. 25–34, Sept. 2001.

[24] D. B. Noonburg and J. P. Shen, "A framework for statistical modeling of superscalar processor performance," in *Proceedings of the third International Symposium on High-Performance Computer Architecture (HPCA-3)*, pp. 298–309, Feb. 1997.

[25] S. Nussbaum and J. E. Smith, "Modeling superscalar processors via statistical simulation," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, pp. 15–24, Sept. 2001.

[26] M. Oskin, F. T. Chong, and M. Farrens, "HLS: Combining statistical and symbolic simulation to guide microprocessor design," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, pp. 71–82, June 2000.

[27] 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 Microarchitecture (MICRO-31)*, pp. 11–19, Nov. 1998.

[28] A. S. Huang and J. P. Shen, "The intrinsic bandwidth requirements of ordinary programs," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pp. 105–114, Oct. 1996.

[29] M. D. Smith, “Overcoming the challenges to feedback-directed optimization (keynote talk),” in *Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pp. 1–11, 2000.

[30] J. Fisher and S. Freudenberger, “Predicting conditional branch directions from previous runs of a program,” in *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 85–95, 1992.

[31] D. W. Wall, “Predicting program behavior using real or estimated profiles,” in *Proceedings of the 1991 International Conference on Programming Language Design and Implementation (PLDI-1991)*, pp. 59–70, 1991.