Performance Prediction based on Inherent Program Similarity

Kenneth Hoste[†], Aashish Phansalkar[‡], Lieven Eeckhout[†],
Andy Georges[†], Lizy K. John[‡] and Koen De Bosschere[†]

[†]ELIS, Ghent University, Belgium
[‡]ECE, The University of Texas at Austin

{kehoste,leeckhou,ageorges,kdb}@elis.UGent.be
{aashish,ljohn}@ece.utexas.edu

ABSTRACT

A key challenge in benchmarking is to predict the performance of an application of interest on a number of platforms in order to determine which platform yields the best performance. This paper proposes an approach for doing this. We measure a number of microarchitecture-independent characteristics from the application of interest, and relate these characteristics to the characteristics of the programs from a previously profiled benchmark suite. Based on the similarity of the application of interest with programs in the benchmark suite, we make a performance prediction of the application of interest. We propose and evaluate three approaches (normalization, principal components analysis and genetic algorithm) to transform the raw data set of microarchitecture-independent characteristics into a benchmark space in which the relative distance is a measure for the relative performance differences. We evaluate our approach using all of the SPEC CPU2000 benchmarks and real hardware performance numbers from the SPEC website. Our framework estimates per-benchmark machine ranks with a 0.89 average and a 0.80 worst case rank correlation coefficient.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling Techniques, Performance attributes

General Terms

Experimentation, Measurement, Performance

Keywords

Performance Modeling, Workload Characterization, Inherent Program Behavior

PACT'06, September 16–20, 2006 , Seattle, Washington, USA.

1. INTRODUCTION

From a benchmark consumer point-of-view, a key challenge is to determine the platform that yields the best performance for a given application of interest. Ideally, the user's application of interest is his best benchmark. However, in many practical circumstances the user has to rely on the performance scores of a standardized benchmark suite for estimating the performance of the application of interest for two reasons. First, it is too difficult or costly to port the application program of interest to a wide range of platforms. Second, there are many platforms for which the performance needs to be measured before making a choice about which platform yields to the best performance for the given application.

A popular tool for estimating performance of an application program on an unavailable platform is detailed cycle-accurate processor simulation. However, next to not solving the porting problem, simulation is very time consuming and thus is difficult to use in practice.

This motivates us to come up with a different solution to this ubiquitous problem in benchmarking. The methodology proposed in this paper uses the already known performance scores of standardized benchmark suites on the systems of our interest. As a part of our methodology we measure a set of microarchitecture-independent characteristics for the new application of interest and relate them to the same characteristics of the benchmarks in the standardized benchmark suite. The microarchitecture-independent characteristics capture the inherent program behavior that is unbiased towards a particular microarchitecture. We then use the knowledge of similarity between the application of interest and the corresponding benchmarks to predict the performance of the application of interest. In other words, we use the standardized benchmarks as proxies for our application of interest based on similarity.

The key issue in a methodology that uses program similarity based on microarchitecture-independent program characteristics is to determine how differences in microarchitecture-independent characteristics translate into differences in performance. We propose and evaluate three approaches for achieving that, namely normalization, principal components analysis and a genetic algorithm, of which the genetic algorithm shows to be the most accurate. A genetic algorithm learns how to rescale the benchmark space so that the Euclidean distance in the benchmark space becomes a more accurate measure for performance differences when running the benchmarks on a variety of platforms.

We evaluate our framework for predicting machine ranks using SPEC published speedup rates that cover various commercial machines with different ISAs, compiler settings and microprocessors.

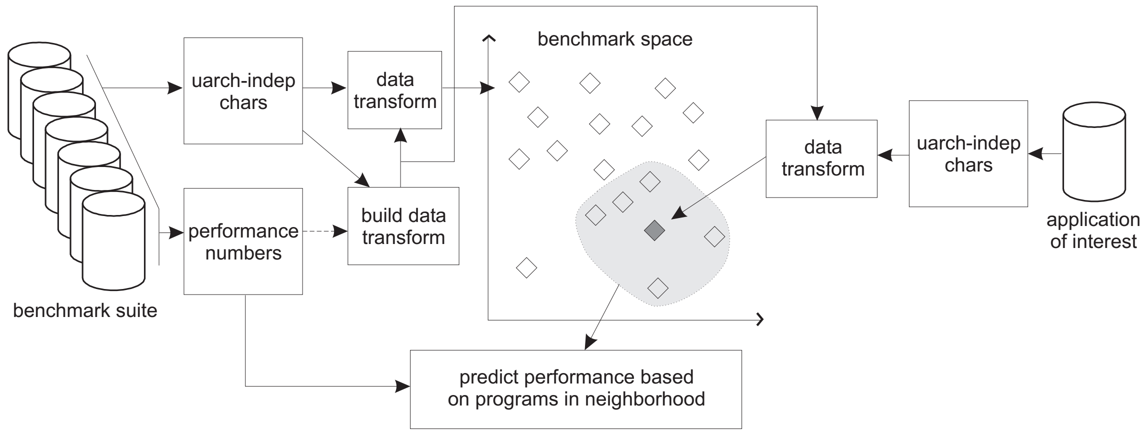

Figure 1: The framework proposed in this paper for predicting performance based on microarchitecture-independent program characteristics.

Current practice, which uses the average rank across all benchmarks for predicting ranks for specific applications of interest, achieves an average 0.83 and a worst case 0.64 rank correlation coefficient for the estimated speedups versus the measured speedups. Our framework based on inherent program similarity achieves an average correlation coefficient of 0.89; the worst case correlation coefficient that we observe is 0.79. These results demonstrate that our framework is indeed capable of tracking performance differences across platforms with different ISAs, compilers and microprocessors. To the best of our knowledge, this paper is the first to propose a methodology for predicting machine ranks for individual programs based on microarchitecture-independent program similarity.

This paper is organized as follows. We first detail on our performance prediction framework. We then present our experimental setup followed by the evaluation of our framework. Finally, we discuss related work and conclude.

2. PERFORMANCE PREDICTION FRAMEWORK

Figure 1 illustrates the framework that we propose in this paper for predicting performance based on microarchitecture-independent program similarity. The framework assumes a collection of programs which we call the *benchmark suite*. For each of these benchmarks, we have a collection of microarchitecture-independent characteristics as well as performance numbers on a (number of) platform(s). The performance numbers could be obtained from simulation or from real hardware execution. These microarchitecture-independent characteristics along with the performance numbers are then used to build a data transformation matrix — building the data transformation matrix can also be done without using performance numbers, thus using microarchitecture-independent characteristics solely (hence the dashed line between the 'performance numbers' box and the 'build data transform' box in Figure 1). Once the data transformation matrix is computed, the original microarchitecture-independent data matrix is transformed using the data transformation matrix. The benchmarks can now be viewed as points in a transformed data space which we call the *benchmark space*.

For an application of interest for which we want to predict performance, we then compute a set of microarchitecture-independent characteristics — this is the same set of characteristics that we used to build the benchmark space. We subsequently transform the microarchitecture-independent characteristics using the same data transformation matrix as above. This locates the application of interest in the benchmark space. Performance is then predicted by appropriately weighting the performance numbers of the benchmarks in the neighborhood of the application of interest.

We now discuss a number of aspects of this framework: (i) the microarchitecture-independent characteristics, (ii) how to build the data transformation matrix, and (iii) how to compute a performance number for the application of interest.

2.1 Microarchitecture-independent characteristics

Ideally, the program characteristics that serve as input to our framework should be platform-independent characteristics. In other words, they should be compiler-independent, ISA-independent and microarchitecture-independent in order to capture the true inherent program behavior. Since this is difficult to do, we take a pragmatic approach and use microarchitecture-independent characteristics. The characteristics that we collect are specific to a given ISA and a given compiler, however they are independent of a given microarchitecture, *i.e.*, the characteristics are independent of cache size, branch predictor size, processor core configuration, *etc*. As will be shown in the evaluation section of this paper, these characteristics, inspite of being ISA-dependent and compiler-dependent, are accurate enough for tracking performance across different platforms with different ISAs and compilers.

Table 1 summarizes the 47 microarchitecture-independent characteristics that we use in this paper. The range of microarchitecture-independent characteristics is fairly broad in order to cover all major program behaviors such as instruction mix, inherent ILP, working set sizes, memory strides, branch predictability, *etc*. Measuring these program characteristics can be done efficiently through instrumentation which is substantially faster than simulation. We include the following characteristics:

Instruction mix. We include the percentage of loads, stores, control transfers, arithmetic operations, integer multiplies and floating-point operations.

ILP. In order to quantify the amount of instruction-level parallelism (ILP), we consider an out-of-order processor model in which everything is idealized and unlimited except for the window size — we assume perfect caches, perfect branch prediction, infinite number of functional units, *etc*. We measure the amount of IPC that can

category	no.	characteristic
instruction mix	1	percentage loads
	2	percentage stores
	3	percentage control transfers
	4	percentage arithmetic operations
	5	percentage integer multiplies
	6	percentage fp operations
ILP	7	32-entry window
	8	64-entry window
	9	128-entry window
	10	256-entry window
register traffic	11	avg. number of input operands
	12	avg. degree of use
	13	prob. register dependence $= 1$
	14	prob. register dependence ≤ 2
	15	prob. register dependence ≤ 4
	16	prob. register dependence ≤ 8
	17	prob. register dependence ≤ 16
	18	prob. register dependence ≤ 32
	19	prob. register dependence ≤ 64
working set size	20	I-stream at the 32B block level
	21	I-stream at the 4KB page level
	22	D-stream at the 32B block level
	23	D-stream at the 4KB-page level
data stream strides	24	prob. local load stride $= 0$
	25	prob. local load stride ≤ 8
	26	prob. local load stride ≤ 64
	27	prob. local load stride ≤ 512
	28	prob. local load stride ≤ 4096
	29	prob. local store stride $= 0$
	30	prob. local store stride ≤ 8
	31	prob. local store stride ≤ 64
	32	prob. local store stride ≤ 512
	33	prob. local store stride ≤ 4096
	34	prob. global load stride $= 0$
	35	prob. global load stride ≤ 8
	36	prob. global load stride ≤ 64
	37	prob. global load stride ≤ 512
	38	prob. global load stride ≤ 4096
	39	prob. global store stride $= 0$
	40	prob. global store stride ≤ 8
	41	prob. global store stride ≤ 64
	42	prob. global store stride ≤ 512
	43	prob. global store stride ≤ 4096
branch predictability	44	GAg PPM predictor
	45	PAg PPM predictor
	46	GAs PPM predictor
	47	PAs PPM predictor

Table 1: Microarchitecture-independent characteristics.

be achieved for an idealized processor with a given window size of 32, 64, 128 and 256 in-flight instructions.

Register traffic characteristics. We collect a number of characteristics concerning registers [6]. Our first characteristic is the average number of input operands to an instruction. Our second characteristic is the average degree of use, or the average number of times a register instance is consumed (register read) since its production (register write). The third set of characteristics concerns the register dependency distance. The register dependency distance is defined as the number of dynamic instructions between writing a register and reading it.

Working set. We characterize the working set size of the instruction and data stream. For each benchmark, we count how many unique 32-byte blocks were touched and how many unique 4KB pages were touched for both instruction and data accesses.

Data stream strides. The data stream is characterized with respect to local and global data strides [10]. A global stride is defined as the difference in the data memory addresses between temporally adjacent memory accesses. A local stride is defined identically except that both memory accesses come from a single instruction — this is done by tracking memory addresses for each memory operation. When computing the data stream strides we make a distinction between loads and stores.

Branch predictability. The final characteristic we want to capture is branch behavior. The most important aspect would be how predictable the branches are for a given benchmark. In order to capture branch predictability in a microarchitecture-independent manner we used the Prediction by Partial Matching (PPM) predictor proposed by Chen *et al.* [2], which is a universal compression/prediction technique.

A PPM predictor is built on the notion of a Markov predictor. A Markov predictor of order k predicts the next branch outcome based upon k preceding branch outcomes. Each entry in the Markov predictor records the number of next branch outcomes for the given history. To predict the next branch outcome, the Markov predictor outputs the most likely branch direction for the given k-bit history.

An m-order PPM predictor consists of $(m+1)$ Markov predictors of orders 0 up to m. The PPM predictor uses the m-bit history to index the mth order Markov predictor. If the search succeeds, i.e. the history of branch outcomes occurred previously, the PPM predictor outputs the prediction by the mth order Markov predictor. If the search does not succeed, the PPM predictor uses the $(m-1)$-bit history to index the $(m-1)$th order Markov predictor. In case the search misses again, the PPM predictor indexes the $(m-2)$th order Markov predictor, etc. Updating the PPM predictor is done by updating the Markov predictor that makes the prediction and all its higher order Markov predictors. In this paper, we consider four variations of the PPM predictor: GAg, PAg, GAs and PAs. 'G' means global branch history whereas 'P' stands for per-address or local branch history; 'g' means one global predictor table shared by all branches and 's' means separate tables per branch. The order of all of these predictors equals 13 in our measurements. We want to emphasize that these characteristics for computing the branch predictability are microarchitecture-independent. The reason is that the PPM predictor is to be viewed as a theoretical basis for branch prediction — it attains upper-limit performance — rather than an actual predictor that is to be built in hardware.

2.2 The Data Transformation Matrix

As a second step in our methodology, the raw data matrix, which is a matrix where the rows are the benchmarks and where the columns are the microarchitecture-independent characteristics, needs to be transformed. This is done by multiplying the raw data matrix with the data transformation matrix. We now propose three different methods of data transformation, namely normalization, principal components analysis and a genetic algorithm.

2.2.1 Normalization

An important issue with the raw data matrix is that some microarchitecture-independent characteristics vary in the range 10 ± 1 whereas other characteristics vary in the range 1 ± 0.1, *e.g.*, the variance of the ILP metric is orders of magnitude larger than the variance of the instruction mix metric. Using the Euclidean distance in the benchmark space built from the raw data matrix would thus give higher weight to characteristics that take larger values. Normalization so that the mean is zero and the variance is one for all microarchitecture-independent characteristics across all benchmarks alleviates this issue. Normalization gives an equal weight to all the microarchitecture-independent characteristics.

2.2.2 *Principal Components Analysis*

A second important issue is that some dimensions in the benchmark space (even after normalization) can be correlated. The Euclidean distance gives higher weight to correlated characteristics. In other words, the underlying program characteristic that causes the microarchitecture-independent characteristics to correlate, gets a higher weight in the Euclidean distance. Principal components analysis (PCA) [7] is a statistical data analysis technique that extracts uncorrelated dimensions from a data set.

The input to PCA is a matrix in which the rows are the *cases* and the columns are the *variables*. In this paper, the cases are the various benchmarks; the columns are the 47 normalized microarchitecture-independent characteristics. 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) $Var[Z_1] \geq Var[Z_2] \geq \ldots \geq Var[Z_p]$ — this means Z_1 contains the most information and Z_p the least; and (ii) $Cov[Z_i, Z_j] = 0, \forall i \neq j$ — this means there is no information overlap between the principal components. Note that the total variance in the data (variables) remains the same before and after the transformation, namely $\sum_{i=1}^{p} Var[X_i] = \sum_{i=1}^{p} Var[Z_i]$. In this paper, X_i is the ith microarchitecture-independent characteristic; Z_i then is the ith principal component after PCA. $Var[X_i]$ is the variance of the original microarchitecture-independent characteristic X_i computed over all benchmarks. Likewise, $Var[Z_i]$ is the variance of the principal component Z_i over all benchmarks.

Some of the principal components account for a higher variance than others. By removing the principal components with the lowest variance from the analysis, we can reduce the dimensionality of the data set while controlling the amount of information that is lost. We retain q principal components which is a significant information reduction since $q \ll p$ in most cases. 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. For example, criteria such as '80% of the total variance should be explained by the retained principal components' could be used for data reduction. An alternative criterion is to retain all principal components for which the individual retained principal component explains a fraction of the total variance that is at least as large as the minimum variance of the original variables.

The output obtained from PCA is a matrix in which the rows are the various benchmarks and the columns are the retained principal components. We subsequently normalize the principal components, *i.e.* we rescale the principal components to unit variance. This gives equal weight to all of the principal components [5].

2.2.3 *Genetic Algorithm*

Since we use the Euclidean distance as a distance measure in the benchmark space, we implicitly assume that the Euclidean distance in the (microarchitecture-independent) benchmark space is proportional to the performance differences across a variety of platforms. Normalization and PCA only partially address this issue. Normalization assumes that all normalized microarchitecture-independent characteristics have an equal impact on overall performance; PCA assumes that all normalized underlying (and uncorrelated) program characteristics have an equal impact. However, some program characteristics have a much larger impact on performance than others. For example, the branch prediction accuracy typically has a much larger impact on overall performance than the fraction multiply operations. As such, an appropriate distance measure should give a higher weight to the branch prediction accuracy metric than to the fraction multiply operations.

A higher or lower impact to a particular program characteristic can be given by multiplying the program characteristic by a given factor. This scaling gives a higher or lower weight to the given program characteristic when computing the Euclidean distance in the benchmark space.

We propose a genetic algorithm (GA) for computing these weights. A genetic algorithm is an evolutionary optimization method that starts from a population of solutions. For each solution in the population, a fitness score is computed and the solutions with the highest fitness score are selected for constructing the next generation. This is done by applying mutation and crossover on the selected solutions from the previous generation. Mutation randomly changes a single solution; crossover generates new solutions by mixing existing solutions. This algorithm is repeated, *i.e.*, new generations are constructed, until no more improvement is observed for the fitness score.

The fitness score that we use here is the prediction accuracy of our framework to predict performance speedups across a wide range of machines. As such, the genetic algorithm learns how the distance measure in the benchmark space correlates with performance across a variety of platforms. The genetic algorithm thus uses performance numbers for building the data transformation matrix; the normalization and PCA approaches do not use performance numbers.

2.3 Performance Prediction

Once the data transformation matrix is computed using one of the approaches discussed in the previous section, we transform the raw data matrix by multiplying it with the data transformation matrix. Each benchmark then is a point in the multidimensional benchmark space. Predicting performance for an application of interest then requires that microarchitecture-independent characteristics are measured and that these characteristics are transformed using the data transformation matrix. This locates the application of interest in the benchmark space.

Predicting performance for the application of interest is done by taking a weighted average over the performance numbers of the benchmarks in the neighborhood of the application of interest. All the benchmarks that are part of the neighborhood are called *proxies* of the application of interest. The weighting is done based on the distance between the proxy and the application of interest. In fact, the weight w_i is inversely proportional to the distance d_i. The weight w_i is computed as

$$w_i = \frac{\sum_{i=1}^{n} \frac{1}{d_i}}{d_i}, \qquad (1)$$

with n being the number of proxies of the application of interest. In this paper, we focus on predicting performance speedups, rather than predicting raw performance. Predicting relative performance differences is often more important in practice. The performance speedup of the application of interest is computed as the weighted harmonic average over the speedups of the proxies:

$$S = \frac{1}{\sum_{i=1}^{n} \frac{w_i}{S_i}}. \qquad (2)$$

2.4 Discussion

An inherent limitation with this performance prediction framework is that accurate performance prediction is difficult for an application of interest that is isolated in the benchmark space. The fact that an application of interest is isolated in the benchmark space indicates that the application of interest is dissimilar to all of the programs in the benchmark suite in terms of its microarchitectiure-independent characteristics. As such, it is to be expected that an accurate performance prediction will be difficult to make based on the almost non-existing similarity of the application of interest with the programs in the benchmark suite.

As a result, an important issue to our performance prediction framework is which programs to select for inclusion in the benchmark suite, *i.e.*, the benchmark suite should be diverse enough to cover a wide range of program behaviors. In this paper, we use SPEC CPU2000 as our benchmark suite because the SPEC website records performance numbers for all of the SPEC CPU2000 benchmarks for a large variety of platforms. Showing that our performance prediction framework works using a standardized benchmark suite has a lot of practical appeal. People can compare their application of interest versus the SPEC CPU benchmarks based on inherent program behavior and make performance predictions using the publicly available SPEC CPU results for a large number of commercial machines.

As a part of our future work, we will study how to build a benchmark suite that reduces the number of weak spots in the benchmark space in order to make accurate predictions for a wider range of applications of interest. One potential avenue could be to look at program phases of existing benchmarks to populate the benchmark space.

3. EXPERIMENTAL SETUP

In this paper, we use all of the SPEC CPU2000 benchmarks with all of their reference inputs. The binaries were taken from the SimpleScalar website; they are compiled for the Alpha ISA. Measuring the microarchitecture-independent characteristics discussed in section 2.1 is done using ATOM [15]. ATOM is a binary instrumentation tool that allows for instrumenting functions, basic blocks and instructions. The instrumentation itself is done offline, *i.e.*, an instrumented binary is stored on disk. Then the microarchitecture characteristics are measured by running the instrumented binary.

In the evaluation section of this paper, we use the real hardware performance numbers reported on the SPEC CPU website[1]. We use the speedup ratios with base optimization compared to the reference SPEC CPU machine which is a SUN Ultra5_10 workstation with a 300MHz SPARC processor with 256MB main memory. We use speedup numbers for 36 machines with different ISAs, processors and machine configurations from a variety of computer manufacturers such as AMD, Intel, Alpha, HP, IBM, SUN, etc. Table 2 enumerates all the machines that we use in this study.

Recall that the genetic algorithm uses performance numbers for learning how to scale the microarchitecture-independent characteristics for accurate performance predictions. As such, in order to make a fair evaluation of the genetic algorithm, we use a leave-one-out methodology. The leave-one-out methodology leaves a benchmark out of the data set for building the model using the genetic algorithm; the model is then subsequently used for predicting the performance for the left-out benchmark.

[1] `http://www.spec.org/cpu2000`

AMD Epox 8KHA Motherboard, AMD Athlon(TM) XP2100+
AMD TYAN Thunder K8QS Pro (S4882), AMD Opteron(TM) 850
Compaq AlphaServer DS10 6/600
Acer Altos G520 (3.6GHz Intel Xeon)
Acer Altos G710 (3.0GHz Intel Xeon)
Dell Precision WorkStation 340 (1.5GHz Pentium 4)
Dell Precision WorkStation 340 (2.2GHz Pentium 4)
Dell Precision Workstation 380 (3.8GHz Pentium 4, 2MB L2)
Fujitsu PRIMEPOWER650 (1890MHz)
Fujitsu PRIMEPOWER900 (2160MHz)
Fujitsu Siemens Celcius 460
Fujitsu Siemens Celsius V810, Opteron(TM) 252, Linux64-bit
Fujitsu Siemens PRIMERGY BX620S2, 64-bit Intel Xeon 3.60GHz
AMD Gigabyte GA-7DX Motherboard, 1.2GHz Athlon Processor
HP Integrity rx4640-8 (1.6GHz/9MB L2 Itanium2)
HP AlphaServer GS1280 7/1300
HP ProLiant BL25p, AMD Opteron(TM) 252
IBM eServer Bladecenter HS20 (3.8GHz Intel Xeon, 2MB L2 Cache)
IBM eServer e326 (AMD Opteron(TM) 246)
IBM eServer p5 575 (1900MHz, 1 CPU)
IBM eServer pSeries 690Turbo (1700MHz, 1 CPU)
ION SR2300WV2 (3.2GHz Intel Xeon processor w. 2MB L3 cache)
Intel D850EMV2 motherboard (2.53GHz, Pentium4 processor)
Intel D850MD motherboard (2.0GHz, Pentium 4 processor)
Intel D875PBZ motherboard (3.2GHz, Pentium 4 processor)
Intel D925XECV2 motherboard (3.46GHz, Pentium 4 processor)
Pathscale ASUS SK8N Motherboard, AMD Opteron (TM) Model 248
AMD Rioworks HDAMA Motherboard, AMD Opteron 246
SGI Altix3700 Bx2 (1600MHz 9MB L3, Itanium2)
SGI Altix3000 (1500MHz, Itanium2)
SGI Origin200 360MHz R12k
Sun Blade 1000 Model1900
Sun Blade 2500 (1.6GHz)
Sun Fire V1280 (1200 MHz)
Sun Java Workstation W1100z
Supermicro X6DH8/E-G2 Motherboard (Intel Xeon 3.6GHz 2M Cache)

Table 2: Real hardware systems used in this paper.

4. EVALUATION

We now evaluate if our framework can predict the speedup ranks of real systems for an application of interest. To demonstrate the results we initially plot the estimated speedup numbers versus the actual speedup numbers for four example benchmarks, namely art, gap, gcc and mesa as shown in Figure 2. Each dot in these scatter plots represents one machine; there are 36 machines plotted in total. The graphs in the right column are for the GA data transformation method; the graphs on the left are for the 'average speedup' method. The 'average speedup' approach is what people would use in current practice given the SPEC CPU data; they would choose a machine that achieves high average performance on all of the benchmarks. We observe that the estimated speedups correlate very well with the measured speedups for the average speedup method. However, the estimated speedup results for the GA data transformation method correlate better with the measured speedups than the average speedup approach.

4.1 Predicting machine ranks

To quantify the accuracy of predicting the ranks of different systems we compute rank correlation coefficients. For a given application of interest we rank all the machines based on the predicted speedups. A similar rank can be computed based on the measured speedups. We then compute the Spearman rank correlation coefficient which is a measure for how well the estimated rank corresponds to the measured rank. The closer to 1, the better the estimated rank. The results are shown in Figure 3 for the three data transformation methods that we evaluate in this paper: normalization, PCA and genetic algorithm. For PCA, we retain 6 principal components which explain slightly more than 80% of the total variance; we experimented with different number of principal components, however, 6 principal components showed to yield

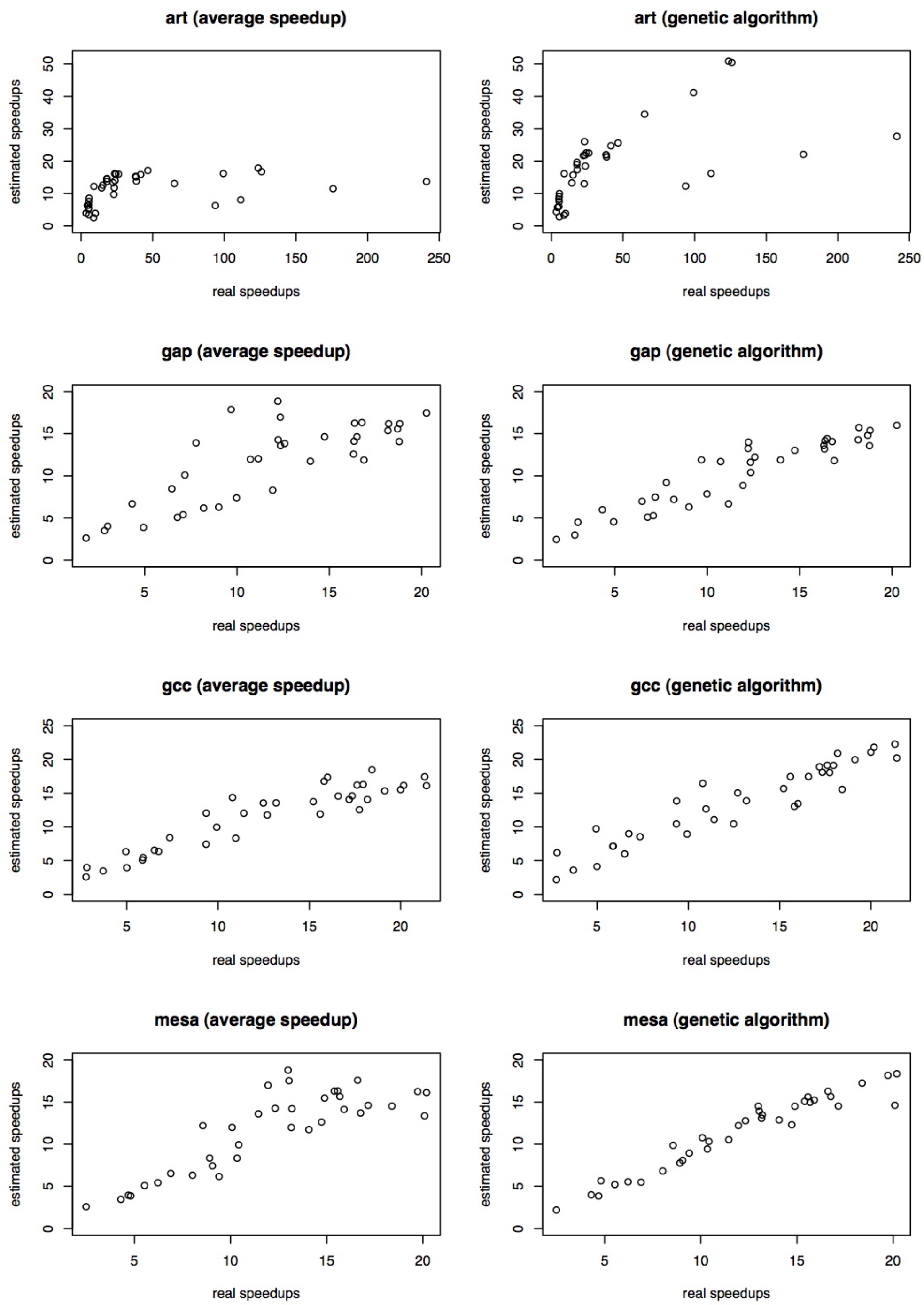

Figure 2: Scatter plots showing the estimated speedups versus the measured speedups for all 36 machines for four benchmarks: art, gap, gcc **and** mesa**; this is for the average speedup approach (left column) and for the GA data transformation method (right column).**

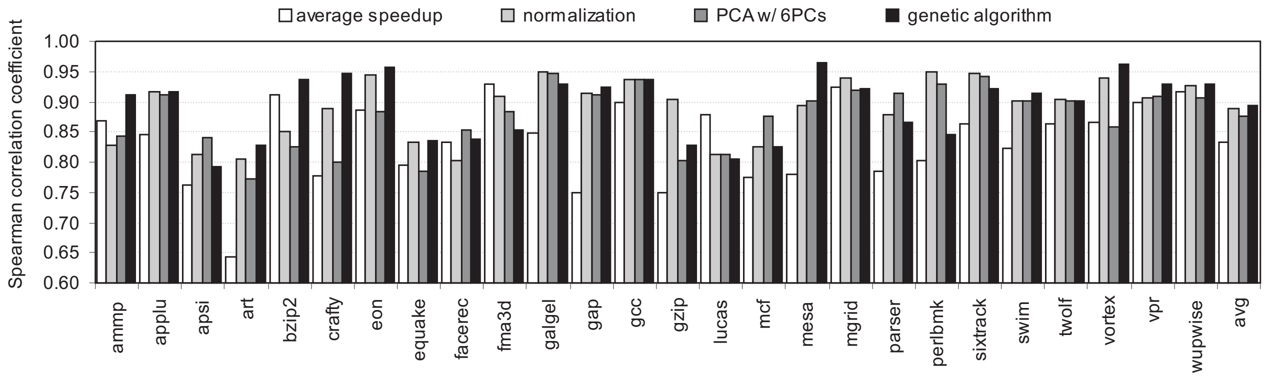

Figure 3: The Spearman correlation coefficients for estimating the ranks for the average benchmark suite speedup results, and the normalization, PCA and GA data transformation methods.

	avg speedup	normalization	PCA	GA
normalization	21	–	17	10
PCA	17	9	–	10
GA	23	16	16	–

Table 3: Summarizing the number of benchmarks out of the 26 SPEC CPU benchmarks for which a data transformation method in the rows outperforms a data transformation method in the columns.

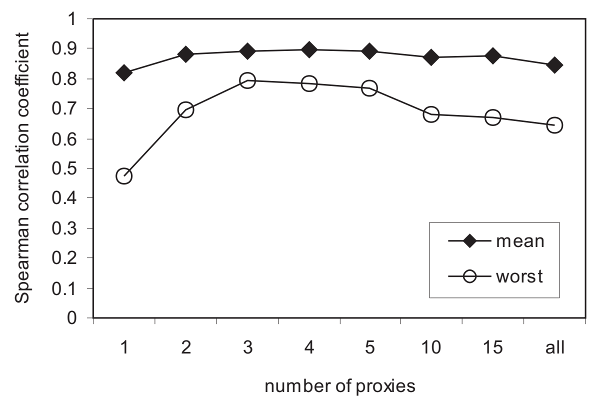

Figure 4: The average and worst Spearman rank correlation coefficient as a function of the number of proxies.

the best results. The baseline Spearman rank correlation coefficient that we compare against is obtained from a rank based on average speedup numbers across all benchmarks. This baseline rank correlation coefficient is 0.83 on average. The normalization and PCA data transformation methods achieve a higher average correlation coefficient, namely 0.889 and 0.876, respectively. The genetic algorithm achieves a slightly higher correlation coefficient, namely 0.892. Also important, next to achieving a good average correlation coefficient, is that the minimum correlation coefficient for our data transformation techniques (0.80 for normalization, 0.77 for PCA and 0.79 for the genetic algorithm) is significantly higher than the minimum correlation coefficient for the average speedup method (0.64).

Table 3 summarizes the number of benchmarks for which one data transformation method achieves a higher rank correlation coefficient than another data transformation method. The genetic algorithm which outperforms the normalization and PCA data transformation methods for 16 out of the 26 benchmarks, and outperforms the average speedup method for 23 of the 26 benchmarks, clearly is the best performing data transformation method.

In order to further quantify the significance of our results, we have done the following experiment. We quantified the performance loss by picking the machine with the highest rank according to the average speedup method compared to the best performing machine for a given application of interest. The average performance loss using this approach is 20%. Using the genetic algorithm to point to the machine with the highest rank yields a performance loss of only 13.6%. Doing the same experiment with the top 3 highest machine ranks yields 19.7% versus 11.1% performance loss, respectively. As such, we conclude that the small differences in rank correlation coefficient can make a big difference in practice.

4.2 Number of proxies

As mentioned in section 2.3, we compute the estimated speedup for the application of interest as a weighted average over a number of proxies. Figure 4 quantifies the average and the worst Spearman rank correlation coefficient as a function of the number of proxies. Using a single proxy for the application of interest yields relatively poor results; the worst rank correlation coefficient is 0.48. The best results are obtained for three proxies — and we used three proxies for all the other results presented in this paper. More proxies degrade the prediction accuracy.

Table 4 shows the three proxies along with their weights for all of the SPEC CPU2000 benchmarks. It is interesting to observe that the weights for all of the proxies are very close to $1/3$. In other words, the distance between the proxies and the application of interest is fairly uniform. There are a few exceptions though, see for example art and galgel. These two benchmarks are substantially closer to each other than any of the other benchmarks. Another interesting note is that some benchmarks do not appear as a proxy in this table, such as gcc, mcf, lucas and swim. This is because these benchmarks are isolated in the benchmark space, or in other words, these benchmarks exhibit a unique inherent behavior. Other benchmarks are very popular proxies (bzip2 being the most notable example) and hence are similar to several other benchmarks.

benchmark	first proxy		second proxy		third proxy	
	benchmark	weight	benchmark	weight	benchmark	weight
ammp	sixtrack	0.3450023	facerec	0.3298283	equake	0.3251694
applu	apsi	0.3529558	mgrid	0.3321383	sixtrack	0.3149060
apsi	facerec	0.3632251	applu	0.3310163	mgrid	0.3057585
art	galgel	0.5262046	equake	0.2486268	applu	0.2251686
bzip2	crafty	0.3678218	vpr	0.3247506	gzip	0.3074277
crafty	bzip2	0.3457154	gzip	0.3291154	mesa	0.3251692
eon	vortex	0.3432195	perlbmk	0.3363128	wupwise	0.3204677
equake	mgrid	0.3698197	facerec	0.3155218	wupwise	0.3146585
facerec	apsi	0.3708346	wupwise	0.3234905	mesa	0.3056749
fma3d	perlbmk	0.3660400	mesa	0.3522770	eon	0.2816830
galgel	art	0.5048660	equake	0.2628316	applu	0.2323024
gap	parser	0.3844569	perlbmk	0.3277766	bzip2	0.2877665
gcc	gap	0.3390179	vortex	0.3317401	perlbmk	0.3292420
gzip	bzip2	0.3723797	vpr	0.3230235	parser	0.3045968
lucas	sixtrack	0.3547355	ammp	0.3331614	apsi	0.3121031
mcf	twolf	0.3655187	vpr	0.3456542	bzip2	0.2888271
mesa	crafty	0.3360154	fma3d	0.3327870	perlbmk	0.3311976
mgrid	applu	0.3867341	equake	0.3145360	apsi	0.2987299
parser	gap	0.3594303	bzip2	0.3560188	crafty	0.2845508
perlbmk	mesa	0.3450405	bzip2	0.3330650	fma3d	0.3218945
sixtrack	fma3d	0.3439120	applu	0.3375917	apsi	0.3184963
swim	lucas	0.3569074	mgrid	0.3227215	sixtrack	0.3203710
twolf	vpr	0.4386195	bzip2	0.3045816	parser	0.2567989
vortex	eon	0.3465275	perlbmk	0.3282399	parser	0.3252326
vpr	twolf	0.3855639	bzip2	0.3689636	parser	0.2454725
wupwise	facerec	0.3516994	eon	0.3451375	equake	0.3031630

Table 4: The three proxies along with their weights for each of the benchmarks.

5. RELATED WORK

The fundamental facilitator for our performance prediction approach is a good quantitative measure for program similarity. Several researchers have proposed methods for quantifying program similarity. Saavedra and Smith [13] use the squared Euclidean distance computed in a benchmark space built up using dynamic program characteristics at the Fortran programming language level such as operation mix, number of function calls, number of address computations, etc. Conte [3] uses kiviat views to qualitatively compare program behavior based on microarchitecture-dependent characteristics such as cache miss rates, branch mispredict rates, etc. Yi *et al.* [17] use a Plackett-Burman design for classifying benchmarks based on how the benchmarks stress the same processor components to similar degrees. Vandierendonck and De Bosschere [16] rank benchmarks based on their uniqueness in the standard benchmark suite using the SPEC performance rating, *i.e.*, the benchmarks that exhibit different speedups on most of the machines are given a higher rank. All of these studies reveal interesting insights into how benchmarks behave and into how (dis)similar benchmarks are from each other.

Based on this prior work, researchers have proposed benchmark suite composition techniques [4, 5, 12]. These techniques first measure a number of program characteristics, then apply principal components analysis, and finally apply cluster analysis in order to find distinct groups of program behavior. A representative is then chosen from each cluster for inclusion in the benchmark suite. The key idea is to select benchmarks so that all major program behaviors are represented in the benchmark suite. This technique can be used for building a benchmark suite that covers the benchmark space well, or it could be used to build a reduced benchmark suite from an existing benchmark suite. This reduced benchmark suite yields accurate performance predictions compared to the original benchmark suite.

The current paper extends [11] which used the above workload characterization methodology consisting of principal components analysis and cluster analysis to predict performance for individual benchmarks. As shown in this paper, an important issue with principal components analysis however is that the distance measure in the benchmark space may not relate well to the performance differences across various platforms. This paper shows that the genetic algorithm for learning how the differences in microarchitecture-independent characteristics relate to performance differences yields better results. The current paper also improves [11] in three other ways: (i) the use of a better set of microarchitecture-independent characteristics, (ii) limiting the number of proxies and (iii) the use of more benchmarks in the evaluation. Doing a head-to-head comparison between the method in [11] and our approach, for the benchmarks considered in [11], shows an improvement of the average rank correlation coefficient from 0.76 to 0.91.

A large body of work has also been done on the correlation between microarchitecture-independent program characteristics and processor performance, see for example [1, 9, 14]. However, these techniques do not predict performance for an application of interest based on cross-program similarity. Instead, these techniques predict performance based on intra-program phase-level similarities. This requires that particular phases of the application need to be executed for making a performance prediction; this is not the case for our method.

Another approach to the benchmarking problem that we address in this paper is analytical modeling. Ideally, an analytical model would consume microarchitecture-independent characteristics as well as microarchitecture parameters and produce accurate performance estimates of the given application on the given microarchitecture. The work that gets close to such an approach is the superscalar processor model presented by Karkhanis and Smith [8] that estimates performance based on microarchitecture-dependent characteristics such as cache miss rates and branch misprediction rates. And various researchers have proposed techniques to predict cache miss rates based on microarchitecture-independent characteristics such as the stack distance, see for example [18]. However, we are unaware of any work that proposes a superscalar processor model based on microarchitecture-independent characteristics solely — the major impediment for achieving this is a good model for estimating branch misprediction rates based on microarchitecture-independent characteristics.

6. SUMMARY

This paper proposed an approach for addressing the ubiquitous problem in benchmarking which is the identification of the platform that yields the best performance for the given application of interest. The key idea is to compare inherent program characteristics of the application of interest against the same characteristics for all programs in the standardized benchmark suite. Based on the inherent similarity of the application of interest with the benchmarks in the benchmark suite, a number of proxies are identified and a performance prediction can be made using the performance numbers of the proxies.

We evaluated three approaches for transforming the raw microarchitecture-independent data matrix into a benchmark space in which the relative distance is an accurate measure for program similarity and hence the relative performance differences across a variety of platforms. Of the three approaches, normalization, principal components analysis and genetic algorithm, the genetic algorithm is the most accurate. Our framework yields an average 0.89 rank correlation coefficient for predicting relative performance ranks of 36 commercial machines with different ISAs, compilers and microprocessors; the worst case correlation coefficient is 0.80. Current practice which uses the average machine ratings for predicting machine ranks achieves an average 0.83 rank correlation coefficient and a worst case correlation coefficient of 0.64.

7. ACKNOWLEDGEMENTS

This research is supported in part by Ghent University, the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research – Flanders (FWO Vlaanderen), the HiPEAC Network of Excellence, the European SARC project No. 27648, the NSF grant 0429806, IBM, Intel and AMD Corporations. Lieven Eeckhout is a Postdoctoral Fellow with the Fund for Scientific Research – Flanders (Belgium) (FWO Vlaanderen).

8. REFERENCES

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

[2] I. K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 128–137, Oct. 1996.

[3] T. Conte. Insight, not (random) numbers. Keynote talk at the 2005 International Symposium on Performance Analysis of Systems and Software (ISPASS), Mar. 2005.

[4] L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization (IISWC)*, pages 2–12, Oct. 2005.

[5] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5, Feb. 2003. `http://www.jilp.org/vol5`.

[6] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 22nd Annual International Symposium on Microarchitecture (MICRO-22)*, pages 236–245, Dec. 1992.

[7] R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, fifth edition, 2002.

[8] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA-31)*, pages 338–349, June 2004.

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

[10] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *Proceedings of the 2004 International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2004.

[11] A. Phansalkar and L. K. John. Performance prediction using program similarity. In *Proceedings of the 2006 SPEC Benchmark Workshop*, Jan. 2006.

[12] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, pages 10–20, Mar. 2005.

[13] R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, Nov. 1996.

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

[15] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report 94/2, Western Research Lab, Compaq, Mar. 1994.

[16] H. Vandierendonck and K. De Bosschere. Many benchmarks stress the same bottlenecks. In *Proceedings of the Seventh Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 57–64, Feb. 2004.

[17] J. J. Yi, D. J. 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)*, pages 281–291, Feb. 2003.

[18] Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *Proceedings of the Twelveth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2003.