Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics

Kenneth Hoste Lieven Eeckhout
ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
Email: {kehoste,leeckhou}@elis.UGent.be

Abstract— **Understanding the behavior of emerging workloads is important for designing next generation microprocessors. For addressing this issue, computer architects and performance analysts build benchmark suites of new application domains and compare the behavioral characteristics of these benchmark suites against well-known benchmark suites. Current practice typically compares workloads based on microarchitecture-dependent characteristics generated from running these workloads on real hardware. There is one pitfall though with comparing benchmarks using microarchitecture-dependent characteristics, namely that completely different inherent program behavior may yield similar microarchitecture-dependent behavior.**

This paper proposes a methodology for characterizing benchmarks based on microarchitecture-independent characteristics. This methodology minimizes the number of inherent program characteristics that need to be measured by exploiting correlation between program characteristics. In fact, we reduce our 47-dimensional space to an 8-dimensional space without compromising the methodology's ability to compare benchmarks. The important benefits of this methodology are that (i) only a limited number of microarchitecture-independent characteristics need to be measured, and (ii) the resulting workload characterization is easy to interpret. Using this methodology we compare 122 benchmarks from 6 recently proposed benchmark suites. We conclude that some benchmarks in emerging benchmark suites are indeed similar to benchmarks from well-known benchmark suites as suggested through a microarchitecture-dependent characterization. However, other benchmarks are dissimilar based on a microarchitecture-independent characterization although a microarchitecture-dependent characterization suggests the opposite to be true.

I. Introduction

The types of applications that are being run on our computer systems is constantly evolving. The reason is twofold. First, computer users constantly come up with new desires which drives software companies into the development of new applications. Second, advances in technology constantly improve compute power which drives the development of applications with increased capabilities.

Computer architects and performance analysts are well aware of this phenomenon and are therefore constantly looking for new emerging workloads so that their newly designed microprocessor performs well on these new application areas. Whenever an emerging workload is identified, computer architects and performance analysts typically collect a number of benchmarks that represents this emerging workload. Examples of recently introduced benchmark suites covering emerging workloads are MediaBench [1] for multimedia workloads, MiBench [2] and EEMBC[1] for embedded workloads, BioMetricsWorkload [3] for biometrics workloads, BioInfoMark [4] and BioPerf [5] for bioinformatics workloads, *etc.* The key question however is how different these workloads are compared to already existing, and well known benchmark suites. Answering this question is important for a number of reasons. First, it provides insight into whether the next generation microprocessors need to be designed differently compared to today's machines because of the emerging workloads. Second, if the new workload domain is not significantly different from well-known benchmark suites, then there is no need for including those benchmarks in the design process. Simulating those additional benchmarks would only add to the overall simulation time without providing additional insight.

In order to answer the question of how different these emerging workloads are from already existing benchmark suites, researchers typically characterize the emerging workload benchmark suite by comparing the characteristics of these benchmarks versus the characteristics of well-known benchmark suites. Current practice in comparing benchmark suites is to characterize the suites in terms of a number of microarchitecture-dependent metrics. Most workload characterization papers run the benchmark suite that represents the emerging workload on a given microprocessor while measuring program characteristics using hardware performance counters; others use simulation for deriving similar results. The program characteristics typically being measured are instruction mix along with a number of microarchitecture-dependent characteristics such as IPC, cache miss rates, branch misprediction rates, TLB miss rates, *etc.* These studies then conclude by saying that two workloads are dissimilar if the hardware performance counter characteristics are dissimilar to each other; and reverse, two workloads are similar if the hardware performance counter characteristics are similar to each other.

There is one major pitfall though with this approach. Program characteristics measured using hardware performance counters may hide the underlying inherent program behavior, *i.e.*, although the hardware performance counter metrics may be similar to each other, the inherent program behavior can be different. And this can be misleading for driving microprocessor design, especially when today's processors are used for characterizing emerging workloads that will eventually run

[1] http://www.eembc.org

on future processors [6]. As a solution to this problem, we propose to characterize benchmarks using microarchitecture-independent characteristics in order to capture the true inherent program behavior. The downside of using microarchitecture-independent characteristics however is that simulation or instrumentation is needed for collecting these characteristics; this is substantially slower than collecting hardware performance counter values. In order to address this issue, we show how to exploit the correlation that exists between microarchitecture-independent characteristics for reducing the number of microarchitecture-independent characteristics that need to be measured. In fact, we reduce the 47-dimensional workload space to an 8-dimensional space with the eight dimensions being eight key inherent program characteristics. As such, only 8 program characteristics need to be measured which is an approximate 3X speedup compared to measuring all 47 program characteristics.

This paper makes the following contributions:

- We show that measuring benchmark similarity based on program characteristics obtained from hardware performance counters can be misleading. In fact, we present a case study showing benchmarks that exhibit similar behavior in terms of the hardware performance counter metrics, however, the underlying inherent program behavior is quite different.
- We present a workload characterization methodology based on microarchitecture-independent characteristics that is more efficient than previously proposed methodologies using principal components analysis (PCA) [7], [8], [9]. There are two major advances over this prior work: (i) the characterization itself is done faster because fewer characteristics need to be measured, and (ii) the dimensions in the workload space have a more intuitive meaning.
- Using this methodology we compare 122 benchmarks from 6 recently introduced benchmark suites based on their inherent behavior. We conclude that many benchmarks from the BioInfoMark, BioMetricsWorkload and CommBench benchmark suites exhibit dissimilar behavior from SPEC CPU. Benchmarks from MediaBench and MiBench on the other hand, tend to exhibit similar behavior compared to SPEC CPU2000.

II. Experimental setup

We use the benchmarks as shown in Table I. There are 122 benchmarks in total from 6 benchmark suites: BioInfomark [4] covering a bioinformatic workload, BioMetricsWorkload [3] covering a biometric workload, CommBench [10] covering a telecommunication workload, MediaBench [1] covering a multimedia workload, MiBench [2] covering embedded workloads and SPEC CPU2000[2] covering general-purpose workloads. For all benchmarks we use the largest available input. All benchmarks are compiled for the Alpha ISA using the Compaq `cc` compiler v6.3-025 for the C benchmarks, the GCC `g++` v2.95.2 for the C++ benchmarks and the Compaq `f77` compiler X5.3-1155 for the Fortran benchmarks. The optimization flags were set to `-O3 -ifo`.

[2] `http://www.spec.org`

III. Program characterization methodologies

We use two data sets in this paper, namely a microarchitecture-independent data set and a data set obtained from hardware performance counter profiling. These data sets are detailed in the following two subsections.

A. Microarchitecture-independent characterization

Table II 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*. We use ATOM [11] for collecting these characteristics. 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 program characteristics are measured by running the instrumented binary.

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 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 [12]. 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 [13]. 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

TABLE I

BENCHMARKS USED IN THIS PAPER ALONG WITH THEIR INPUTS AND DYNAMIC INSTRUCTION COUNT (IN MILLIONS).

suite	program	input	I-cnt (M)
BioInfoMark	blast	protein	81,092
	ce	ce	4,816
	clustalw	clustalw	884,859
	fasta	fasta34	759,654
	glimmer	004663	26,610
	hmmer	build	321
		calibrate	43,048
		search (artemia)	47
		search (sprot)	1,785,862
	phylip	dnapenny	184,557
		promlk	557,514
	predator	predator	804,859
BioMetricsWorkload	csu	Bayesian (project)	403,313
		Bayesian (train)	28,158
		PreprocessNormalize	4,059
		SubspaceProject (LDA)	6,054
		SubspaceProject (PCA)	6,098
		SubspaceTrain (LDA)	51,297
		SubspaceTrain (PCA)	41,729
	speak	decode	46,648
CommBench	cast	decode	130
		encode	130
	drr	drr	235
	frag	frag	49
	jpeg	decode	238
		encode	339
	reed	decode	1,298
		encode	912
	rtr	rtr	1,137
	tcp	tcp	58
	zip	decode	50
		encode	322

suite	program	input	I-cnt (M)
MediaBench	epic	test1	205
		test2	2,296
	unepic	test1	35
		test2	876
	g721	decode	323
		encode	343
	ghostscript	gs	868
	mesa	mipmap	32
		osdemo	10
		texgen	86
	mpeg2	decode	149
		encode	1,528
MiBench	CRC32	large	612
	FFT	fft (large)	237
		fftinv (large)	217
	adpcm	rawcaudio	758
		rawdaudio	639
	basicmath	large	1,523
	bitcount	large	681
	blowfish	decode	495
		encode	498
	dijkstra	large	252
	ghostscript	large	868
	ispell	large	1,027
	jpeg	cjpeg	121
		djpeg	24
	lame	large	1,199
	mad	large	345
	patricia	large	399
	pgp	decode	111
		encode	48
	qsort	large	512
	rsynth	say (large)	775
	sha	large	114
	susan	corners (large)	29
		edges (large)	73
		smoothing (large)	300
	tiff	2bw	143
		2rgba	268
		dither	1,228
		median	763
	typeset	lout	609

suite	program	input	I-cnt (M)
SPEC2000	ammp	ref	388,534
	applu	ref	336,798
	apsi	ref	361,955
	art	ref-110	77,067
		ref-470	84,660
	bzip2	graphic	157,003
		program	136,389
		source	122,267
	crafty	ref	194,311
	eon	cook	100,552
		kajiya	131,268
		rush	73,139
	equake	ref	158,071
	facerec	ref	249,735
	fma3d	ref	312,960
	galgel	ref	326,916
	gap	ref	310,323
	gcc	166	46,614
		200	106,339
		expr	11,847
		integrate	13,019
		scilab	60,784
	gzip	graphic	113,400
		log	42,506
		program	161,726
		random	91,961
		source	84,366
	lucas	ref	134,753
	mcf	ref	59,800
	mesa	ref	314,449
	mgrid	ref	440,934
	parser	ref	530,784
	perlbmk	splitmail.535	69,857
		splitmail.704	73,966
		splitmail.850	142,509
		splitmail.957	122,893
		diffmail	43,327
		makerand	2,055
		perfect	29,791
	sixtrack	ref	452,446
	swim	ref	221,868
	twolf	ref	397,222
	vortex	ref1	129,793
		ref2	151,475
		ref3	145,113
	vpr	place	117,001
		route	82,351
	wupwise	ref	337,770

TABLE II

MICROARCHITECTURE-INDEPENDENT CHARACTERISTICS.

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	D-stream at the 32B block level
	21	D-stream at the 4KB-page level
	22	I-stream at the 32B block level
	23	I-stream at the 4KB page level

category	no.	characteristic
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. global load stride $= 0$
	30	prob. global load stride ≤ 8
	31	prob. global load stride ≤ 64
	32	prob. global load stride ≤ 512
	33	prob. global load stride ≤ 4096
	34	prob. local store stride $= 0$
	35	prob. local store stride ≤ 8
	36	prob. local store stride ≤ 64
	37	prob. local store stride ≤ 512
	38	prob. local store 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

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.* [14], which is a universal compression/prediction technique. 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. We want to emphasize that these

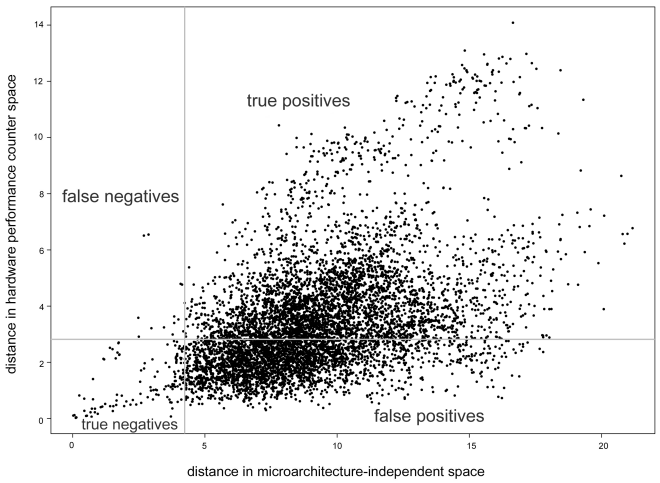

Fig. 1. Distance in the hardware performance counter space versus the distance in the microarchitecture-independent space.

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 rather than an actual predictor that is to be built in hardware.

B. Hardware performance counter characterization

The hardware performance counter metrics that we use in this paper are typical for what is being observed in many workload characterization papers. We collect hardware performance counter values for IPC, branch misprediction rate, L1 D-cache miss rate, L1 I-cache miss rate, L2 cache miss rate and D-TLB miss rate. The machine on which we collect these hardware performance counter values is the Alpha 21164A processor. The Alpha 21164A (EV56) processor in an in-order dual-pipeline superscalar processor. We use the `dcpi`-tool [15] for collecting these hardware performance counter values. Next to the above hardware counter values we also collect the IPC on the Alpha 21264A (EV67) which is an out-order four-wide superscalar processor. The reason we do not have cache miss rates and branch misprediction rates on the Alpha 21264A is that the `dcpi`-tool does not allow for directly measuring cache miss rates and branch misprediction rates on the Alpha 21264A processor.

IV. Pitfall in hardware performance counter based program characterization

As mentioned in the introduction, comparing benchmarks based on microarchitecture-dependent characteristcs can be misleading. The fundamental reason is that different inherent (microarchitecture-independent) program behavior can yield similar microarchitecture behavior. The pitfall of microarchitecture-dependent characterization is that the conclusions taken based on this microarchitecture-dependent characterization may not be generalized to other microarchitectures.

In order to quantify this we have done the following experiment. We build two workload spaces, a microarchitecture-dependent workload space using the hardware performance counter metrics from section III-B and a microarchitecture-independent workload space using the metrics from section III-A. These spaces are built up by normalizing both the hardware performance counter data set as well as the microarchitecture-independent data set, *i.e.*, the mean is zero and the standard deviation is one for all characteristics across all benchmarks. The goal of this normalization step is to put all characteristics on a common scale. In both spaces, we then compute the Euclidean distance between all benchmark tuples. Figure 1 shows the distance in the hardware performance counter space on the vertical axis versus the distance in the microarchitecture-independent space on the horizontal axis. Each dot represents a benchmark tuple. We observe that the correlation between the distance in the hardware performance counter space and the microarchitecture-independent space is modest with a correlation coefficient of 0.46.

As a subsequent step in the analysis, we classify all benchmark tuples into four categories according to the distances in the hardware performance counter space versus the distance in the microarchitecture-independent workload space, see also Figure 1. The threshold distance in the hardware performance counter space and microarchitecture-independent space are 20% of the maximum distance observed in both spaces. Obviously, these thresholds are subjective, however, we believe these thresholds are reasonable for the purpose of identifying program similarity in terms of microarchitecture-dependent and microarchitecture-independent behavior. A distance in the hardware performance counter space is called to be large in case the distance is larger than 20% of the maximum distance observed; likewise for the microarchitecture-independent space, a distance is called to be large if the distance is larger than 20% of the maximum distance observed. A benchmark tuple is categorized as a true positive in case a large distance in the hardware performance counter space occurs along with a large distance in the microarchitecture-independent space; a benchmark tuple is categorized as a true negative in case a small distance in the hardware performance counter space occurs along with a small distance in the microarchitecture-independent space. Likewise we can define a false positive as well as a false negative.

Table III summarizes the result of this experiment. The fraction benchmark tuples is shown for all four categories. There are a number of interesting observations to be made from this analysis. First, the number of benchmark tuples categorized as a false negative is small, less than 1%. This shows that the microarchitecture-independent workload analysis is capable of identifying program similarity: very few benchmark tuples that are found to behave similarly in terms of microarchitecture-independent characteristics show dissimilar behavior in terms of their hardware performance counter behavior. Second, we observe that there is a large fraction of benchmark tuples categorized as a false positive (41% of all benchmark tuples). This corresponds to similarly behav-

TABLE III

CLASSIFYING BENCHMARK TUPLES ACCORDING TO THEIR HARDWARE PERFORMANCE COUNTER BEHAVIOR VERSUS THEIR MICROARCHITECTURE-INDEPENDENT BEHAVIOR.

	small distance in uarch-indep space	large distance in uarch-indep space
large distance in hardware performance counter space	false negative: 0.2%	true positive: 56.9%
small distance in hardware performance counter space	true negative: 1.8%	false positive: 41.1%

ing benchmarks in the hardware performance counter space while exhibiting dissimilar behavior in the microarchitecture-independent space. The fact that this fraction is fairly large explains the modest correlation coefficient in Figure 1 and illustrates that program characterization based on hardware performance counter values can be misleading in a fair amount of cases.

We now further illustrate this pitfall in hardware performance counter program characterization by looking into an example, namely SPEC CPU's bzip2 versus BioInfoMark's blast. Figures 2 and 3 show normalized metrics in the hardware performance counter space and the microarchitecture-independent space, respectively; normalization is done per individual characteristic by dividing the measured value by the maximum value observed across the various benchmark tuples. Note that we use the instruction mix here as part of the hardware performance counter characterization as is done in many workload characterization papers. Figure 2 shows that the hardware performance counter metrics are similar between bzip2 and blast. However, the other microarchitecture-independent program characteristics are fairly different, as illustrated in Figure 3. The most strikingly different characteristics are the working set sizes for both the instruction stream and the data stream. Also the branch predictability based on global history seems to be fairly dissimilar; another example of dissimilar inherent program behavior is the amount of global store strides.

V. TOWARDS AN EFFICIENT MICROARCHITECTURE-INDEPENDENT WORKLOAD CHARACTERIZATION METHODOLOGY

Since characterizing benchmarks based on program metrics obtained from hardware performance counters can be misleading, we propose the use of microarchitecture-independent program characteristics. However, the downside of microarchitecture-independent program characterization compared to using hardware performance counters is that it requires extensive instrumentation or simulation runs which can be time-consuming. Instrumentation and simulation are several orders of magnitude slower than native hardware execution. For example, in our setup, measuring all the microarchitecture-independent characteristics from Table II through instrumentation takes about 110 machine-days on a 600MHz Alpha 21264A machine. Measuring the hardware performance counter values on the Alpha 21164A machine takes only 4 machine-days. Note that we had to run the benchmarks multiple times in order to collect all hardware performance counter values and microarchitecture-independent characteristics.

In order to reduce the time required during microarchitecture-independent program characterization, we propose to reduce the number of microarchitecture-independent characteristics that are to be measured. We propose and evaluate two approaches for achieving this.

A. Correlation elimination

The first approach, called correlation elimination, works as follows. For each program characteristic, we compute the correlation coefficient with all the other program characteristics and subsequently take the average over these correlation coefficients. All program characteristics are then ranked by their average correlation coefficient. The program characteristic that shows the highest average correlation coefficient thus contains the least additional information compared to all the other program characteristics. This motivates us to remove this program characteristic from the data set which reduces the N-dimensional data set to an $(N-1)$-dimensional space. This process is iterated by progressively removing program characteristics. By eliminating highly correlated program characteristics, we reduce the dimensionality of the data set without loosing the insight that the workload characterization provides.

B. Genetic algorithm

The second approach that we evaluate for reducing the number of microarchitecture-independent characteristics uses a genetic algorithm. 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.

A solution in our case, is a series of N 0's and 1's, with N being the number of microarchitecture-independent characteristics. A '1' selects a program characteristic and a '0' excludes a program characteristic. The fitness score $f = \rho \cdot (1 - n/N)$ that we use here consists of two factors. The first factor is the correlation coefficient ρ of the distance between the benchmark tuples in the original data set versus the distance between the benchmark tuples in the reduced data set. In other words, the genetic algorithm tries to find a subset of the program characteristics in the original data set that correlates well with the original data set. The second

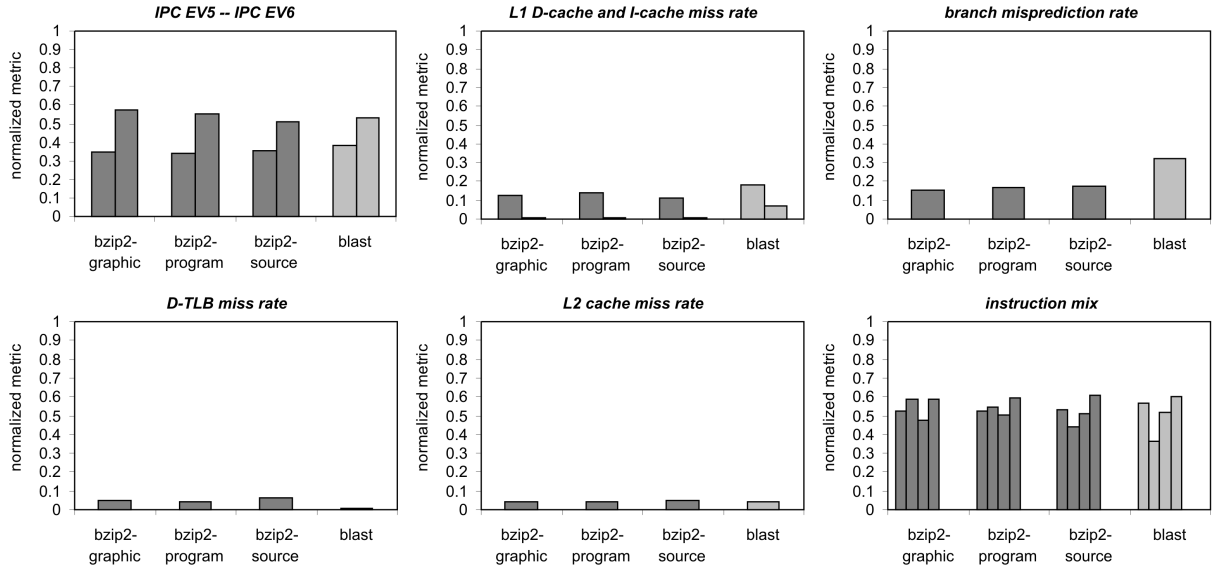

Fig. 2. Comparing the hardware performance counter characteristics for bzip2 versus blast.

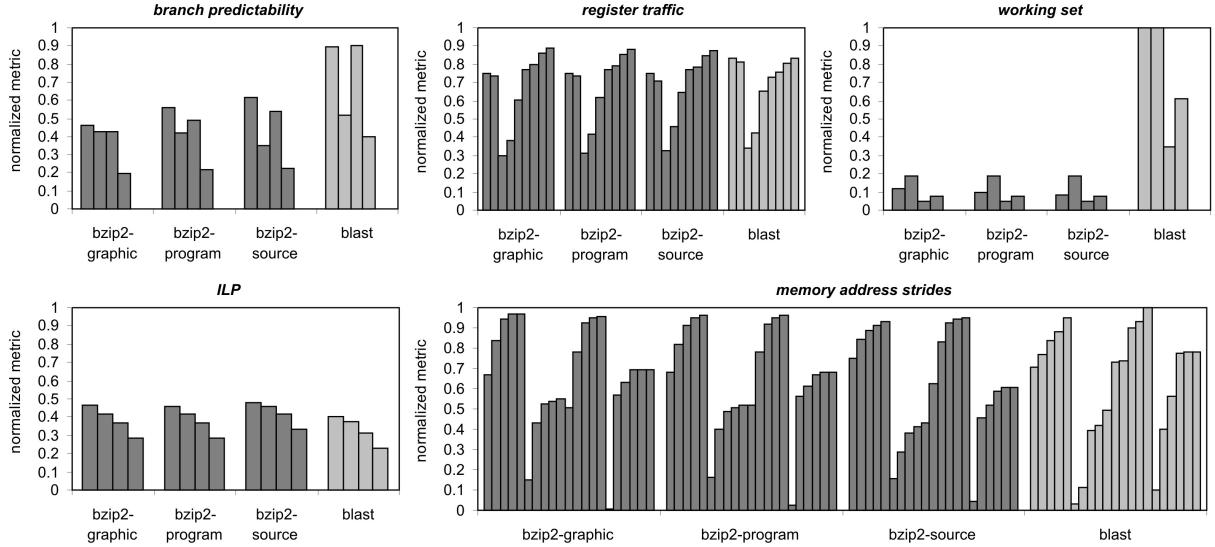

Fig. 3. Comparing the microarchitecture-independent characteristics for bzip2 versus blast. Note that the various microarchitecture-independent characteristics appear in the same order as they appear in Table II.

factor is $(1 - n/N)$ with n being the number of selected program characteristics, or in other words, the number of 1's. The purpose of this second factor is to minimize the number of program characteristics to be selected in the reduced space. The end result of the genetic algorithm is a limited number of program characteristics that allow for an accurate microarchitecture-independent program characterization.

C. Discussion

The goal of these two methods, correlation elimination and the genetic algorithm, is to select a number of key microarchitecture-independent program characteristics. This is an advance over prior work done on characterizing workloads through principal components analysis (PCA) [7], [8], [9], [16], [17]. The goal of PCA is to remove correlation from a data set so that the original data set can be faithfully represented by a lower-dimensional data set. The methods presented here are different in two ways. First, although all three methods reduce the dimensionality of the data set, PCA still requires that all original program characteristics be measured. As such, collecting all program characteristics as required by PCA takes longer than collecting the program characteristics determined by correlation elimination and the genetic algorithm. The second advantage of the methods presented here in this paper is that the dimensions of the lower-dimensional data set are easier to interpret because the dimensions are microarchitecture-independent program characteristics. For PCA on the other hand, the dimensions are linear combinations of program characteristics which is more difficult to get an intuitive understanding.

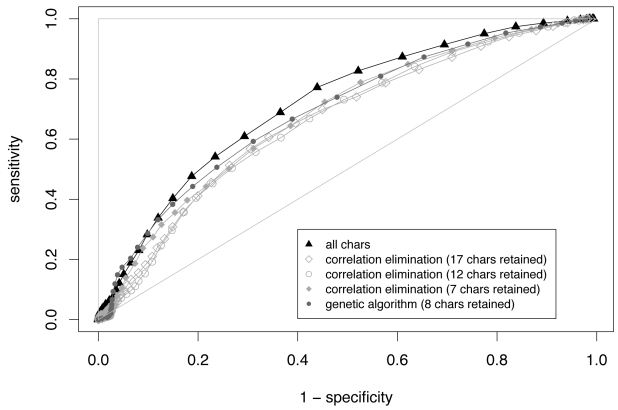

Fig. 4. ROC curves for the all characteristics methods, the correlation elimination method and the genetic algorithm.

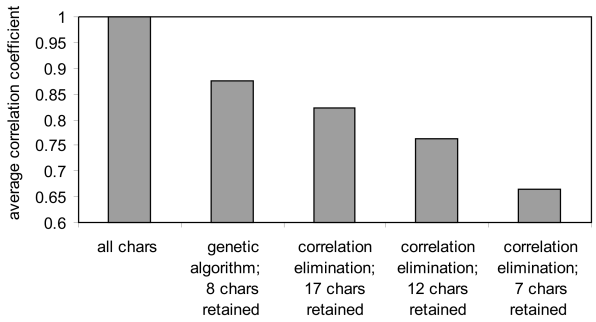

Fig. 5. The correlation coefficient of the distances between all benchmark tuples for the 'all characteristics' method versus the distances for the correlation elimination method and the genetic algorithm.

TABLE IV

THE MICROARCHITECTURE-INDEPENDENT CHARACTERISTICS SELECTED BY THE GENETIC ALGORITHM.

1	percentage loads
2	average number of input operands
3	probability for a register dependence distance ≤ 8
4	probability for a local load stride ≤ 64
5	probability for a global load stride ≤ 512
6	probability for a local store stride ≤ 4096
7	D-stream working set size at the 4KB page level
8	ILP for a 256-entry window

D. Evaluation

We now evaluate the correlation elimination and genetic algorithm methods for identifying key microarchitecture-independent program characteristics. This evaluation is done using the receiver operating characteristic (ROC) curve because the ROC curve quantifies how well a workload characterization method is capable of identifying similar program behavior. A ROC curve is a well known method originating from signal detection theory that plots the sensitivity versus one minus the specificity. In our context, the sensitivity or the true positive rate is defined as the fraction benchmark tuples for which a large distance is observed in the microarchitecture-independent space in case a large distance is observed in the hardware performance counter space. The specificity is the fraction of benchmark tuples where a small distance is observed in the microarchitecture-independent space in case the distance is also small in the hardware performance counter space. Ideally, we want both the sensitivity and the specificity to be close to 1. However, for the purpose of finding program similarity, the key point is to minimize the fraction of false negatives, or in other words, to increase the probability that similar microarchitecture-independent behavior corresponds to similar microarchitecture-dependent behavior. In practice, this means that the sensitivity should be high along with a low specificity.

Figure 4 shows the ROC curves for various workload characterization methods. The more a ROC curve approaches the point $(0, 1)$ (the left upper corner) in this plot, the better. The classification threshold in the hardware performance counter space is fixed and is set to 20% of the maximum distance observed over all benchmark tuples. The various points on each ROC curve represent different classification thresholds in the microarchitecture-independent space. We observe that the ROC curve for the genetic algorithm outperforms the ROC curves for the correlation elimination method; the ROC curve for the genetic algorithm approaches the 'all characteristics' ROC curve more closely than the correlation elimination ROC curves do. This can be quantified using the area under the ROC curve. The area under the 'all characteristics' ROC curve is 0.72; the area under the genetic algorithm ROC curve is 0.69 and the area under the correlation elimination ROC curve is 0.67 and 0.64 when 17 and 12/7 metrics are retained, respectively.

To further evaluate the benefit of the genetic algorithm over the correlation elimination method for identifying key microarchitecture-independent characteristics, we also compare the correlation coefficient of the distances between all benchmark tuples in the original space versus the distances in the reduced spaces. This is shown in Figure 5. The correlation coefficient for the genetic algorithm equals 0.876 whereas the correlation coefficient for the correlation elimination method quickly drops when multiple program characteristics are removed from the data set. For example, with 17 metrics retained from the data set through correlation elimination, the correlation coefficient equals 0.823 which is smaller than 0.876 for the 8 retained metrics using the genetic algorithm. As such, we conclude that the genetic algorithm outperforms the correlation elimination method.

Table IV lists the eight microarchitecture-independent characteristics retained by the genetic algorithm. The above analyses show that these eight characteristics include almost the same information as the original data set including all 47 characteristics. The important benefit though is that collecting these eight characteristics takes approximately 37 machine-days on a single Alpha 21264A machine which is an approximate 3X speed improvement over collecting all program characteristics.

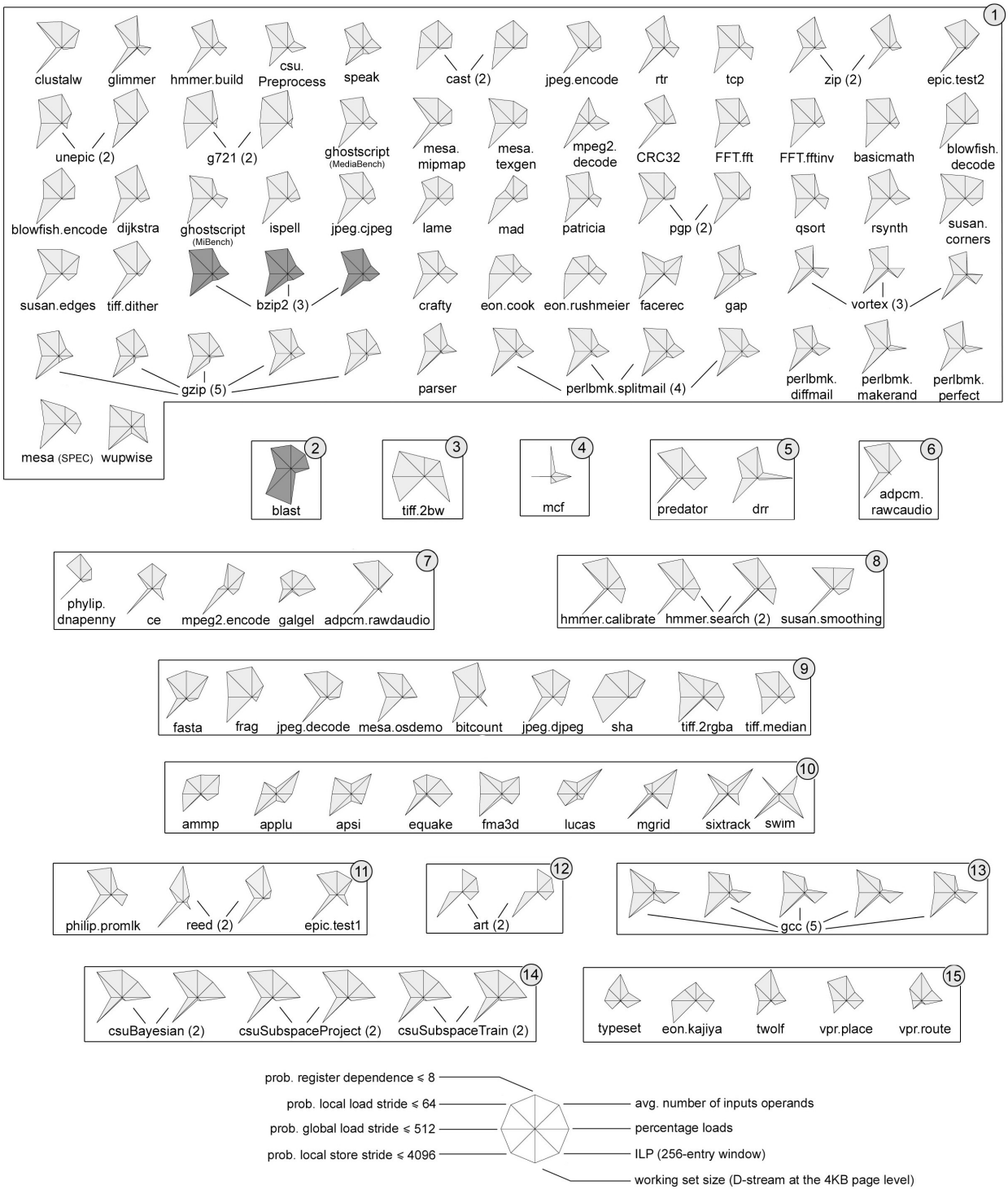

Fig. 6. Kiviat diagrams for the various benchmarks categorized into 15 clusters based on key microarchitecture-independent characteristics.

VI. Comparing existing benchmark suites

We now compare the 122 benchmarks from the 6 benchmark suites, see Table I, using the microarchitecture-independent methodology from the previous section. We use clustering in the 8-dimensional workload space obtained from the genetic algorithm for grouping benchmarks into similarly behaving benchmark clusters. More in particular, we use k-means clustering for a number of K values (from 1 to 70) and determine the K value that yields a Bayesian Information Criterion (BIC) score [18] within 90% of the maximum score. We then generate kiviat plots [19] for all benchmarks, see Figure 6, in which the axes are the microarchitecture-independent characteristics. These kiviat plots are ordered per cluster of similarly behaving benchmarks; there are 15 clusters in total.

There are several interesting insights to be obtained from these kiviat plots. First, a number of benchmarks seem to be isolated. These benchmarks exhibit program characteristics that are very dissimilar to any of the other benchmarks which makes them appear as singleton clusters; note that for some of these singleton clusters (clusters 3 and 6) there is only isolated behavior for particular inputs. Examples are blast (cluster 2), tiff (cluster 3), mcf (cluster 4), adpcm (cluster 6), art (cluster 12), gcc (cluster 13) and csu (cluster 14). The reason for their particular program behavior can be derived from looking into the various dimensions of the kiviat plots. For example, the reason for blast to be isolated is its large working set. A second interesting note to be made is that a majority of the floating-point SPEC CPU2000 benchmarks, namely 9 out of the 14, appear in a single cluster (cluster 10), without any benchmarks from other benchmark suites. A third interesting observation to be made is that some benchmarks from recently introduced benchmark suites exhibit dissimilar inherent behavior compared to SPEC CPU2000, *i.e.*, these benchmarks do not appear in a cluster that also contains SPEC CPU2000 benchmarks. This is especially the case for benchmarks from the BioInfoMark benchmark suite such as blast, fasta, hmmer, phylip (for the promlk input) and predator; this is also the case for the csu benchmark from the BioMetricsWorkload suite, as well as for some of the CommBench benchmarks, see drr, frag, jpeg and reed. For the MediaBench and MiBench benchmark suites there are only a couple benchmarks that exhibit dissimilar behavior compared to SPEC CPU2000.

VII. Related work

We are not the first to study the correlation between microarchitecture-independent program behavior and microarchitecture-dependent program behavior. In fact, there exists a large body of work on the subject. A first thread of research on the subject has shown that there exists a strong correlation between the code that is executed versus performance [13], [18], [20], [21], [22]. The SimPoint tool builds on this notion for selecting sampling units to be used during sampled simulation. They found that execution intervals that are executing similar code behave similarly according to various microarchitecture-dependent program characteristics such as cache miss rates, branch misprediction rates, IPC, *etc*. Code signatures thus allow for identifying microarchitecture-independent program phases for a given benchmark and a given input.

Code signatures cannot be used though for identifying program similarity across programs. Researchers instead use a collection of program characteristics for comparing benchmarks. For example, Weicker [23] characterize benchmarks at the programming language level by counting the number of assignments, the number of if-then-else statements, the number of function calls, the number of loops, *etc*. Saavedra and Smith [24] use various program characteristics at the Fortran programming language level such as operation mix, number of function calls, number of address computations, etc. More recent work on studying program similarity uses statistical data analysis techniques on lower-level program characteristics. Some papers on the subject use microarchitecture-dependent characteristics only [25]; others use a mixture of microarchitecture-dependent and microarchitecture-independent characteristics [7]; yet others advocate the use microarchitecture-independent characteristics solely (as we do in this paper) [9], [16]. A completely different approach was presented by Yi *et al.* [26] for identifying program similarity based on a Placket-Burman design of experiment. Namely, they classify benchmarks based on how the benchmarks stress the same processor components to similar degrees. The important contribution of this paper over this prior work is that we show that microarchitecture-dependent program characterization can be misleading. In addition, we show that an easy-to-understand and efficient-to-measure set of microarchitecture-independent characteristics can be obtained by focusing on non-correlating program characteristics.

In our prior work [8], we also used genetic algorithms for learning how to scale microarchitecture-independent characteristics so that accurate machine rankings can be obtained for an application of interest based on its inherent program similarity with a set of previously profiled benchmarks. The focus of the current paper is different though. We now focus on identifying a limited number of microarchitecture-independent characteristics that provide an accurate picture of the inherent program behavior; in our prior work though, we use a much larger number of program characteristics and learn how to weight each of those.

Recent work in workload characterization also focused on better understanding how benchmarks evolve over time. For example, Joshi *et al.* [17] have studied how the SPEC CPU benchmark suites evolved over its four generations (CPU89, CPU92, CPU95 and CPU2000). They concluded that none of the inherent program characteristics changed as dramatically as the dynamic instruction count. They also concluded that the temporal data locality has become increasingly poor over time; other characteristics have remained more or less the same. Yi *et al.* [6] went one step further and studied how benchmark drift affects processor design. They conclude that benchmark drift can have a significant impact on the performance of next generation processors if the design of the next generation

processors is driven solely by yesterday's benchmarks. This observation also motivates the subject of the current paper, namely that an accurate workload characterization methodology is required for comparing emerging workloads against existing workloads, so that the next generation processors perform well on future workloads.

VIII. Conclusion

This paper showed that a workload characterization based on microarchitecture-dependent characteristics can be misleading because it does not reveal the true inherent behavior of an application. In fact, two benchmarks may look similar in terms of their microarchitecture-dependent behavior, however, the inherent program behavior may be dissimilar. This motivates us for proposing a workload characterization methodology based on microarchitecture-independent program behavior. The disadvantage of microarchitecture-independent program characterization however is the time needed for profiling an application. For addressing this issue we proposed two approaches for limiting the number of microarchitecture-independent characteristics that need to be measured, namely correlation elimination and a genetic algorithm. The key idea for both approaches is to remove characteristics that show good correlation with other characteristics. Our experiments show that the genetic algorithm outperforms the correlation elimination method. The end result is a limited set of program characteristics that characterizes an application nearly as accurate as using a much broader set of characteristics, yet is faster to collect. Using this efficient and accurate workload characterization method we then compare 122 benchmarks from 6 benchmark suites, BioInfoMark, BioMetricsWorkload, CommBench, MediaBench, MiBench and SPEC CPU2000. This characterization is done in terms of eight key microarchitecture-independent program characteristics. From this analysis we conclude that various benchmarks from the BioInfoMark, BioMetricsWorkload and CommBench suites are dissimilar with respect to SPEC CPU2000. Most of the MediaBench and MiBench benchmarks though show similar behavior with at least some of the SPEC CPU2000 benchmarks.

Acknowledgements

The authors would also like to thank the anonymous reviewers for their valuable comments. Lieven Eeckhout is a Postdoctoral Fellow with the Fund for Scientific Research—Flanders (Belgium) (FWO—Vlaanderen). This research is also supported by Ghent University, FWO, IWT and the European SARC project No. 27648.

References

[1] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *MICRO*, Dec. 1997, pp. 330–335.

[2] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *WWC*, Dec. 2001.

[3] C.-B. Cho, A. V. Chande, Y. Li, and T. Li, "Workload characterization of biometric applications on pentium 4 microarchitecture," in *IISWC*, Oct. 2005, pp. 76–86.

[4] Y. Li and T. Li, "BioInfoMark: A bioinformatic benchmark suite for computer architecture research," ECE, University of Florida, Tech. Rep., Jan. 2005.

[5] D. A. Bader, Y. Li, T. Li, and V. Sachdeva, "BioPerf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications," in *IISWC*, Oct. 2005, pp. 163–173.

[6] J. J. Yi, H. Vandierendonck, L. Eeckhout, and D. J. Lilja, "The exigency of benchmark and compiler drift: Designing tomorrow's processors with yesterday's tools," in *ICS*, June 2006, pp. 75–86.

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

[8] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, "Performance prediction based on inherent program similarity," in *PACT*, Sept. 2006.

[9] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John, "Measuring program similarity: Experiments with SPEC CPU benchmark suites," in *ISPASS*, Mar. 2005, pp. 10–20.

[10] T. Wolf and M. A. Franklin, "CommBench: A telecommunications benchmark for network processors," in *ISPASS*, Apr. 2000, pp. 154–162.

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

[12] M. Franklin and G. S. Sohi, "Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors," in *MICRO*, Dec. 1992, pp. 236–245.

[13] J. Lau, S. Schoenmackers, and B. Calder, "Structures for phase classification," in *ISPASS*, Mar. 2004.

[14] I. K. Chen, J. T. Coffey, and T. N. Mudge, "Analysis of branch prediction via data compression," in *ASPLOS*, Oct. 1996, pp. 128–137.

[15] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?" *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 357–390, Nov. 1997.

[16] L. Eeckhout, J. Sampson, and B. Calder, "Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation," in *IISWC*, Oct. 2005, pp. 2–12.

[17] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John, "Measuring benchmark similarity using inherent porgram characteristics," *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 769–782, June 2006.

[18] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS-X*, Oct. 2002, pp. 45–57.

[19] T. Conte, "Insight, not (random) numbers," Keynote talk at *ISPASS*, Mar. 2005.

[20] M. Annavaram, R. Rakvic, M. Polito, J. Bouguet, R. Hankins, and B. Davies, "The fuzzy correlation between code and performance predictability," in *MICRO*, Dec. 2004, pp. 93–104.

[21] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder, "The strong correlation between code signatures and performance," in *ISPASS*, Mar. 2005, pp.236–247.

[22] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of larhe Intel Itanium programs with dynamic instrumentation," in *MICRO*, Dec. 2004, pp. 81–93.

[23] R. P. Weicker, "An overview of common benchmarks," *IEEE Computer*, vol. 23, no. 12, pp. 65–75, Dec. 1990.

[24] R. H. Saavedra and A. J. Smith, "Analysis of benchmark characteristics and benchmark performance prediction," *ACM Transactions on Computer Systems*, vol. 14, no. 4, pp. 344–384, Nov. 1996.

[25] H. Vandierendonck and K. De Bosschere, "Experiments with subsetting benchmark suites," in *WWC*, Oct. 2004, pp. 55–62.

[26] J. J. Yi, D. J. Lilja, and D. M. Hawkins, "A statistically rigorous approach for improving simulation methodology," in *HPCA*, Feb. 2003, pp. 281–291.