MICROARCHITECTURE-INDEPENDENT WORKLOAD CHARACTERIZATION

FOR COMPUTER DESIGNERS, UNDERSTANDING THE CHARACTERISTICS OF WORKLOADS RUNNING ON CURRENT AND FUTURE COMPUTER SYSTEMS IS OF UTMOST IMPORTANCE DURING MICROPROCESSOR DESIGN. A MICROARCHITECTURE-INDEPENDENT METHOD ENSURES AN ACCURATE CHARACTERIZATION OF INHERENT PROGRAM BEHAVIOR AND AVOIDS THE WEAKNESSES OF MICROARCHITECTURE-DEPENDENT METRICS.

Kenneth Hoste
Lieven Eeckhout
Ghent University

The workloads that run on our computer systems are always evolving. Software companies continually come up with new applications, many triggered by the increasing computational power available. It is important that computer designers understand the characteristics of these emerging workloads to optimize systems for their target workloads. Moreover, the need for a solid workload characterization methodology is increasing with the shift to chip multiprocessors—especially heterogeneous CMPs with various cores specialized for particular types of workloads.

Computer architects and performance analysts are well aware of the workload drift phenomenon, and to address it, they typically collect benchmarks to represent emerging workloads. Examples of recently introduced benchmark suites covering emerging workloads are MediaBench for multimedia workloads, MiBench and EEMBC (Embedded Microprocessor Benchmark Consortium, http://www.eembc.org) for embedded workloads, BioMetricsWorkload for biometrics workloads, and BioInfoMark and BioPerf for bioinformatics workloads.[1–5] A key question is how different these workloads are from existing, well-known benchmark suites. Answering this question is important for two reasons. First, it provides insight into whether next-generation microprocessors should be designed differently from today's machines to accommodate the emerging workloads. Second, if the new workload domain is not significantly different from existing benchmark suites, there is no need to include the new benchmarks in the design process—simulating the additional benchmarks would only add to the simulation time without providing additional information.

To find out how different the emerging workloads are from existing benchmark suites, computer designers usually compare the characteristics of the emerging workloads with the characteristics of well-known benchmark suites. A typical approach is to characterize the emerging workload in terms of microarchitecture-dependent metrics. For example, most workload characterization studies run benchmarks representing the emerging workload on a given microprocessor while measuring program characteristics with hardware performance counters. Other studies use simulation to derive similar results. The program characteristics typically measured include instruction mix and microarchitecture-dependent characteristics such as instructions per cycle (IPC), cache miss rates, branch misprediction rates,

0272-1732/07/$20.00 © 2007 IEEE Published by the IEEE Computer Society.

Table 1. Microarchitecture-independent characteristics collected to characterize workload behavior.

Characteristic category	Measurement	Description
Instruction mix	6 percentages	Percentage of loads, stores, branches, arithmetic operations, multiplies, and floating-point operations.
Instruction-level parallelism (ILP)	4 values	IPC achievable for an idealized out-of-order processor (with perfect caches and branch predictor) for window sizes of 32, 64, 128, and 256 in-flight instructions.
Register traffic	2 values and 7 probabilities	Average number of register input operands per instruction, average number of register reads per register write, distribution (measured in buckets) of register dependency distance, or number of instructions between production and consumption of a register instance.
Working-set size	4 numbers	Number of unique 32-byte blocks and 4-Kbyte memory pages touched for both instruction and data streams.
Data stream strides	20 probabilities	Distribution, measured in buckets, of global and local strides. Global stride is the difference in memory addresses between two consecutive memory accesses; local stride is restricted to two consecutive memory accesses by the same static instruction. Strides are measured separately for memory reads and writes.
Branch predictability	4 percentages	Branch prediction accuracy for the theoretical prediction-by-partial-matching (PPM) predictor.[8] We considered global and local history predictors, and per-address and global predictors.

and translation look-aside buffer (TLB) miss rates. These studies conclude either that two workloads are dissimilar if their hardware performance counter characteristics are dissimilar, or that two workloads are similar if their hardware performance counter characteristics are similar.

A major pitfall of a microarchitecture-dependent workload characterization methodology is that it can hide underlying, inherent program behavior. To avoid this pitfall, we advocate characterizing workloads in a microarchitecture-independent manner to capture their true inherent program behavior. This article presents a microarchitecture-independent workload characterization methodology and demonstrates its usefulness in characterizing benchmark suites for emerging workloads.

Pitfall of microarchitecture-dependent workload characterization

Before presenting our methodology, we present a case study that illustrates the problem with microarchitecture-dependent workload characterization. We considered 118 benchmarks from six benchmark suites: SPECcpu2000, MediaBench, MiBench, BioInfoMark, BioMetricsWorkload, and CommBench. Throughout the article, we refer to a benchmark-input pair as a benchmark. For each benchmark, we collected microarchitecture-dependent characteristics with hardware performance counters using the Digital Continuous Profiling Infrastructure (DCPI) tool on two hardware platforms: an Alpha 21164A (EV56) and an Alpha 21264A (EV67) machine. The characteristics we collected are cycles per instruction (CPI) on both the EV56 and the EV67; and the L1 D-cache, L1 I-cache, and L2 cache miss rates on the EV56.

We also measured a set of microarchitecture-independent characteristics in six categories. Table 1 summarizes these characteristics; we present a more detailed description elsewhere.[6] These microarchitecture-independent characteristics can be collected through binary instrumentation using tools such as ATOM (which we use), Pin, Valgrind, and DynamoRIO. These characteristics are microarchitecture-independent but not independent of the instruction set architecture (ISA) and the compiler. In previous work, however, we observed that these characteristics provide a fairly accurate characterization picture, even across platforms.[7]

Of the 118 benchmarks in our data set, we compare two for our case study: SPEC-cpu2000's gzip-graphic and BioInfoMark's fasta. Many pairs of benchmarks could serve as a case study, but we limit ourselves to a single example. Table 2 shows microarchitecture-dependent and microarchitecture-independent characteristics for the two benchmarks, as well as the maximum value observed across all 118 benchmarks (to put the values for gzip and fasta in perspective). The microarchitecture-dependent characteristics, CPI and cache miss rates, are fairly similar for gzip and fasta (especially compared with the maximum values observed across the entire data set).

The microarchitecture-independent characteristics, on the other hand, are quite different. The fasta benchmark's data working set is one order of magnitude smaller than that of gzip (although fasta's dynamic instruction count is about 6.7 times larger). Memory access patterns are also very different for the two benchmarks. For example, the probability of the same static load addressing the same memory location at consecutive executions—a local load stride equal to zero—is more than twice as large for gzip as for fasta. The probability of the difference in memory addresses of consecutive memory writes—the global store stride—being smaller than 64 is more than 2.5 times as large for fasta as for gzip.

We conclude that although microarchitecture-dependent workload behavior is fairly similar, inherent program behavior can be very different, which can be misleading for microprocessor design. Although two workloads behave the same on one microarchitecture, they might exhibit different behavior and performance on other microarchitectures. We therefore advocate using microarchitecture-independent program characteristics to characterize workloads. Similar microarchitecture-independent behavior implies similar microarchitecture-dependent behavior; dissimilar microarchitecture-dependent behavior implies dissimilar inherent behavior.

Methodology

Although collecting microarchitecture-independent characteristics is straightforward with binary instrumentation tools, analyzing the collected data is far from trivial. For example, our data set is a 118×47 data matrix. That is, there are 118 benchmarks, for which we measure $p = 47$ characteristics. Obviously, gaining insight into a large data set is difficult without an appropriate data analysis technique. Here, we discuss two possible techniques. One is a statistical

Table 2. Case study comparing microarchitecture-dependent and -independent characteristics for benchmarks gzip-graphic and fasta.

Characteristic	gzip-graphic	fasta	All 118 benchmarks (maximum)
Microarchitecture-dependent			
CPI on Alpha 21164	1.01	0.92	14.04
CPI on Alpha 21264	0.63	0.66	5.22
L1 D-cache misses per instruction (%)	1.61	1.90	22.58
L1 I-cache misses per instruction (%)	0.15	0.18	6.44
L2 cache misses per instruction (%)	0.78	0.25	17.59
Microarchitecture-independent			
Data working set (32-byte blocks)	3,857,693	438,726	31,709,065
Data working set (4-Kbyte pages)	46,199	4,058	248,108
Instruction working set (32-byte blocks)	1,394	3,801	24,377
Instruction working set (4-Kbyte pages)	33	79	341
Probability of local load stride = 0	0.67	0.30	0.91
Probability of local store stride = 0	0.64	0.05	0.99
Probability of global load stride ≤ 64	0.26	0.18	0.86
Probability of global store stride ≤ 64	0.35	0.93	0.99

Related work

A fairly large body of work exists on microarchitecture-independent workload characterization. The first thread of research on the subject has shown that there is a strong correlation between executed code and performance.[1–5] The SimPoint tool builds on this notion by selecting sampling units for use during sampled simulation. SimPoint's key insight is that execution intervals that execute similar code behave similarly in terms of various microarchitecture-dependent program characteristics such as cache miss rates, branch misprediction rates, and IPC. Code signatures thus allow identification of microarchitecture-independent program phases.

But code signatures cannot be used for identifying program similarity across programs. Researchers instead use a collection of program characteristics to compare benchmarks. For example, Weicker, and Saavedra and Smith characterize benchmarks at the programming-language level by counting the numbers of assignments, if-then-else statements, function calls, loops, and so forth.[6,7] More recent work on program similarity applies statistical data analysis techniques to binary-level program characteristics. Some researchers use microarchitecture-dependent characteristics only;[8] others use a mixture of microarchitecture-dependent and -independent characteristics;[9] yet others use microarchitecture-independent characteristics only.[10,11] We use genetic algorithms to learn how microarchitecture-independent characteristics relate to overall performance.[12] This lets us rank machines for an application of interest according to its inherent program similarity with a set of previously profiled benchmarks. Yi, Lilja, and Hawkins present a completely different approach to finding benchmark similarity based on a Placket-Burman design.[13] They classify benchmarks according to the degree to which they stress various processor structures.

Recent work in workload characterization focuses on better understanding of how benchmarks evolve over time. For example, Joshi et al. study how the SPECcpu suites evolved over four generations (cpu89, cpu92, cpu95, and cpu2000), concluding that none of the inherent program characteristics changed as dramatically as the dynamic instruction count.[14] They also conclude that temporal data locality has become increasingly poor over time, while other characteristics have remained more or less the same. Yi et al. go a step further, studying how benchmark drift affects processor design.[15] They conclude that benchmark drift can have significant negative impact on the performance of next-generation processors running future workloads if their design is driven solely by yesterday's benchmarks. In other words, to ensure that next-generation processors perform well, designers need an accurate workload characterization methodology to compare emerging workloads with existing workloads.

References

1. J. Lau, S. Schoenmackers, and B. Calder, ''Structures for Phase Classification,'' *Proc. Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 04), IEEE CS Press, 2004, pp. 57-67.
2. T. Sherwood et al., ''Automatically Characterizing Large Scale Program Behavior,'' *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 45-57.
3. M. Annavaram et al., ''The Fuzzy Correlation between Code and Performance Predictability,'' *Proc. 37th Ann. Int'l Symp. Microarchitecture* (MICRO 04), IEEE CS Press, 2004, pp. 93-104.
4. J. Lau et al., ''The Strong Correlation between Code Signatures and Performance,'' *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 05), IEEE Press, 2005, pp. 236-247.
5. H. Patil et al., ''Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation,'' *Proc. 37th Ann. Int'l Symp. Microarchitecture* (MICRO 04), IEEE CS Press, 2004, pp. 81-93.
6. R.P. Weicker, ''An Overview of Common Benchmarks,'' *Computer*, vol. 23, no. 12, Dec. 1990, pp. 65-75.
7. R.H. Saavedra and A.J. Smith, ''Analysis of Benchmark Characteristics and Benchmark Performance Prediction,'' *ACM Trans. Computer Systems*, vol. 14, no. 4, Nov. 1996, pp. 344-384.
8. H. Vandierendonck and K. De Bosschere, ''Experiments with Subsetting Benchmark Suites,'' *Proc. 7th Ann. IEEE Int'l Workshop Workload Characterization* (WWC 04), IEEE Press, 2004, pp. 55-62.
9. L. Eeckhout, H. Vandierendonck, and K. De Bosschere, ''Quantifying the Impact of Input Data Sets on Program Behavior and Its Applications,'' *J. Instruction-Level Parallelism*, vol. 5, Feb. 2003, http://www.jilp.org/vol5.
10. A. Phansalkar et al., ''Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites,'' *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 05), IEEE Press, 2005, pp. 10-20.
11. L. Eeckhout, J. Sampson, and B. Calder, ''Exploiting Program Microarchitecture Independent Characteristics and Phase Behavior for Reduced Benchmark Suite Simulation,'' *Proc. IEEE Int'l Symp. Workload Characterization* (IISWC 05), IEEE Press, 2005, pp. 2-12.
12. K. Hoste et al., ''Performance Prediction Based on Inherent Program Similarity,'' *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 06), IEEE CS Press, 2006, pp. 114-122.
13. J.J. Yi, D.J. Lilja, and D.M. Hawkins, ''A Statistically Rigorous Approach for Improving Simulation Methodology,'' *Proc. 9th Int'l Symp. High-Performance Computer Architecture* (HPCA 03), IEEE CS Press, 2003, pp. 281-291.
14. A. Joshi et al., ''Measuring Benchmark Similarity Using Inherent Program Characteristics,'' *IEEE Trans. Computers*, vol. 55, no. 6, June 2006, pp. 769-782.
15. J.J. Yi et al., ''The Exigency of Benchmark and Compiler Drift: Designing Tomorrow's Processors with Yesterday's Tools,'' *Proc. 20th Ann. Int'l Conf. Supercomputing* (ICS 06), ACM Press, 2006, pp. 75-86.

technique called principal components analysis (PCA). The other is a machine-learning algorithm called a genetic algorithm (GA). The goal of both approaches is to make the data set more understandable by reducing its dimensionality.

Principal components analysis

PCA has two main properties: It reduces the data set's dimensionality, and it removes correlation from the data set.[9] Both features are important to increasing the data set's understandability. First, analyzing a lower-dimensional space is easier than analyzing a higher-dimensional space. Second, analyzing correlated data results in a distorted view—a distance measure in a correlated space places too much weight on correlated variables. For example, suppose that two correlated program characteristics are a consequence of the same underlying program characteristic. Then consider two benchmarks that show different behavior in terms of this underlying characteristic. Measuring this difference in terms of the correlated program characteristics will magnify it. Removing the correlation from the data set will give equal weight to all underlying program characteristics.

Before applying PCA, we first normalize the data set. We do this in two steps:

1. computing the mean $\bar{x}$ and standard deviation s per microarchitecture-independent characteristic X_i, $1 \leq i \leq p$ across all benchmarks, and
2. subtracting the mean and dividing by the standard deviation: $Y_i = (X_i - \bar{x})/s$.

The result is that the transformed characteristics Y_i, $1 \leq i \leq p$ have a zero mean and a unit standard deviation. The goal of the normalization is to put all characteristics on a common scale.

The input to PCA is a matrix in which the rows are the benchmarks and the columns are the normalized microarchitecture-independent characteristics Y_i. PCA computes new variables, called principal components, which are linear combinations of the microarchitecture-independent characteristics, such that all principal components are uncorrelated. In other words, PCA transforms the p normalized microarchitecture-independent characteristics $Y_1, Y_2, \ldots, Y_p$ into p principal components $Z_1, Z_2, \ldots, Z_p$ with $Z_i = \sum_{j=1}^{p} a_{ij} Y_j$. This transformation has two main properties. First, the first principal component exhibits the largest variance, followed by the second, followed by the third, and so on. That is, $Var[Z_1] \geq Var[Z_2] \geq \ldots \geq Var[Z_p]$. Intuitively, this means that Z_1 contains the most information, and Z_p the least. Second, the dimensions along which the principal components are identified are orthogonal to each other. That is, the covariance between principal components is zero, that is, $Cov[Z_i, Z_j] = 0, \forall i \neq j$. This means there is no information overlap between the principal components.

By removing the principal components with the lowest variance, we can reduce the data set's dimensionality while controlling the amount of information lost. Determining the number of principal components to retain is also important. Too few principal components won't capture important trends in the data set, and too many can lead to a curse of dimensionality problem. To measure the fraction of information retained in this q-dimensional space, we use the amount of variance $\left(\sum_{i=1}^{q} Var[Z_i]\right) / \left(\sum_{i=1}^{p} Var[Z_i]\right)$ accounted for by these q principal components. For example, we can use criteria for data reduction such as "the retained principal components should explain 70 or 80 percent of the total variance." For our data set, we retain eight principal components, which explain 78 percent of the original data set's total variance.

After PCA, it is important to normalize the principal components to give equal weight to all the retained principal components. Our intuition is that by doing so, we give equal weight to the underlying program behaviors extracted by PCA.

By examining the most important q principal components $Z_i = \sum_{j=1}^{p} a_{ij} Y_j$, $i = 1, \ldots, q$, we can interpret these principal components in terms of the original microarchitecture-independent characteristics. A coefficient a_{ij} that is close to +1 or −1 implies a strong impact of the original characteristic X_j on principal component Z_i.

A coefficient a_{ij} close to 0, on the other hand, implies no impact.

Genetic algorithm

Although PCA reduces the data set's dimensionality effectively, the fact that each principal component is a linear combination of the original workload characteristics complicates the understandability of the lower-dimensional workload space. Our second workload analysis methodology, which uses a genetic algorithm (GA), also reduces the data set's dimensionality, but the retained dimensions are easier to understand because each dimension is a single workload characteristic.

A GA is an evolutionary optimization method that starts from a set of populations of random solutions. The algorithm computes a fitness score for each solution and selects the solutions with the highest fitness scores to construct the next generation of solutions. It constructs the next generation by applying mutation, crossover, and migration to the selected solutions. Mutation randomly changes a single solution, crossover generates new solutions by mixing existing solutions, and migration allows solutions to switch populations. The GA repeats this process—that is, it constructs new generations—until the fitness score shows no further improvement.

A solution is a series of N zeros and ones, with N being the number of microarchitecture-independent characteristics. A one selects a program characteristic, and a zero excludes a program characteristic.

The GA's fitness score evaluates the correlation coefficient of the distances between benchmark pairs in the original data set versus the distances between benchmark pairs in the reduced data set, which includes only characteristics with a one assigned. We use PCA to compute the distance in the original data set as well as in the reduced data set. That is, we first apply PCA on both data sets, retain the principal components with a variance greater than one, normalize the principal components, and finally compute the Euclidean distances between benchmarks in terms of their normalized principal components. We take this additional PCA step to discount the correlation in the data set from the distance measure while accounting for the most important underlying program characteristics. The GA's end result is a limited number of program characteristics that accurately characterize a program's behavior. In selecting eight program characteristics (to yield the same dimensionality that we obtained with PCA), the GA achieved a 0.86 correlation coefficient between the distances in the original data set compared with the distances in the reduced data set.

Evaluation

To gain confidence about the reduced data set's validity with respect to the original data set, we use the reduced data set to compose a subset of benchmarks. Previous work proposed an approach for composing representative benchmark suites based on inherent program behavior.[10–13] The goal of benchmark suite composition is to select a small set of benchmarks representative of a larger set, so that all major program behaviors are represented in the composed benchmark suite. This benchmark suite composition method consists of three steps. The first step measures a number of microarchitecture-independent characteristics for all benchmarks. The second step reduces the data set's dimensionality. We can do this with PCA, or we can use the GA to select a limited number of program characteristics. The third step clusters the various benchmarks according to their inherent behavior. The goal of cluster analysis is to group benchmarks with similar program behavior in a single cluster and benchmarks with dissimilar program behavior in different clusters.[9] The benchmark suite subset composer then chooses a representative from each cluster; this is the benchmark closest to the cluster's centroid. The representative's weight is the number of benchmarks it represents in the cluster.

This methodology lets us find subsets of benchmarks that are representative across different microarchitectures.[11,13] Figure 1 illustrates this by showing the CPI prediction error for the selected subset with respect to all benchmarks for the Alpha 21264 machine. The CPI prediction error

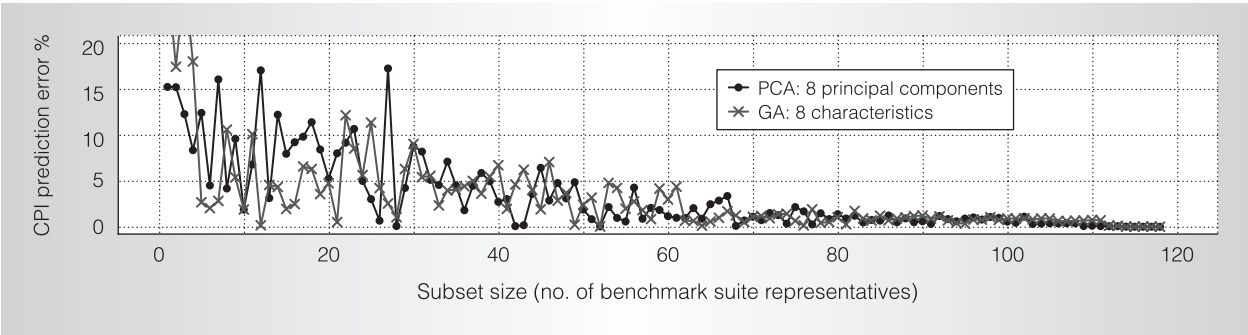

Figure 1. CPI prediction error as a function of subset size for the PCA and GA data reduction approaches.

is the relative error between the average CPI of all benchmarks and the CPI computed as a weighted average over the CPI values of the representatives. This graph shows curves for both the PCA and GA approaches. In both cases, eight dimensions are retained. CPI prediction error typically decreases with increasing subset sizes. Once beyond a subset size of 50 of the 118 benchmarks, the maximum CPI prediction error is consistently below 5 percent.

Comparing benchmark suites

Because the workload characterization obtained with the GA is easier to understand, we use that methodology to characterize the 118 benchmarks. The following are the eight microarchitecture-independent characteristics retained by the GA:

- probability of a register dependence distance ≤ 16
- branch predictability of per-address, global history table (PAg) prediction-by-partial-matching (PPM) predictor
- percentage of multiply instructions
- data stream working-set size at 32-byte block level
- probability of a local load stride $= 0$
- probability of a global load stride ≤ 8
- probability of a local store stride ≤ 8
- probability of a local store stride $\leq$ 4,096

These key characteristics relate to register dependence distance distribution, branch predictability, instruction mix, data stream working-set size, and data working-set access patterns.

We use Kiviat plots to visualize a benchmark's inherent behavior in terms of the eight key microarchitecture-independent characteristics. Figure 2 shows Kiviat plots for the 118 benchmarks. Each axis represents a microarchitecture-independent characteristic. The various rings within the plot represent the mean value minus one standard deviation, the mean value, and the mean value plus one standard deviation along each dimension. The center point and the outer ring represent values outside the range defined by the mean minus and plus the standard deviation. We characterize the prominent program behaviors by connecting their key characteristics to form an area shown in dark gray, which visualizes a benchmark's inherent behavior. By comparing the dark areas across the various benchmarks, we can see how different their behaviors are. We also cluster the benchmarks into 50 clusters, each bounded by a box, with the cluster representative indicated by an asterisk.

These plots provide valuable insight. For example, some benchmarks seem to be isolated. These benchmarks exhibit program characteristics very dissimilar to any of the other benchmarks, so they appear as singleton clusters. Examples are blast, mcf, and swim. We can derive the reason for their particular program behavior from the plots. For example, the reason for blast's being isolated is its large working set, and mcf exhibits long local store strides (that is, large address differences between consecutive memory accesses by the same static store instruction).

Another observation is that some benchmarks are susceptible to their input, whereas

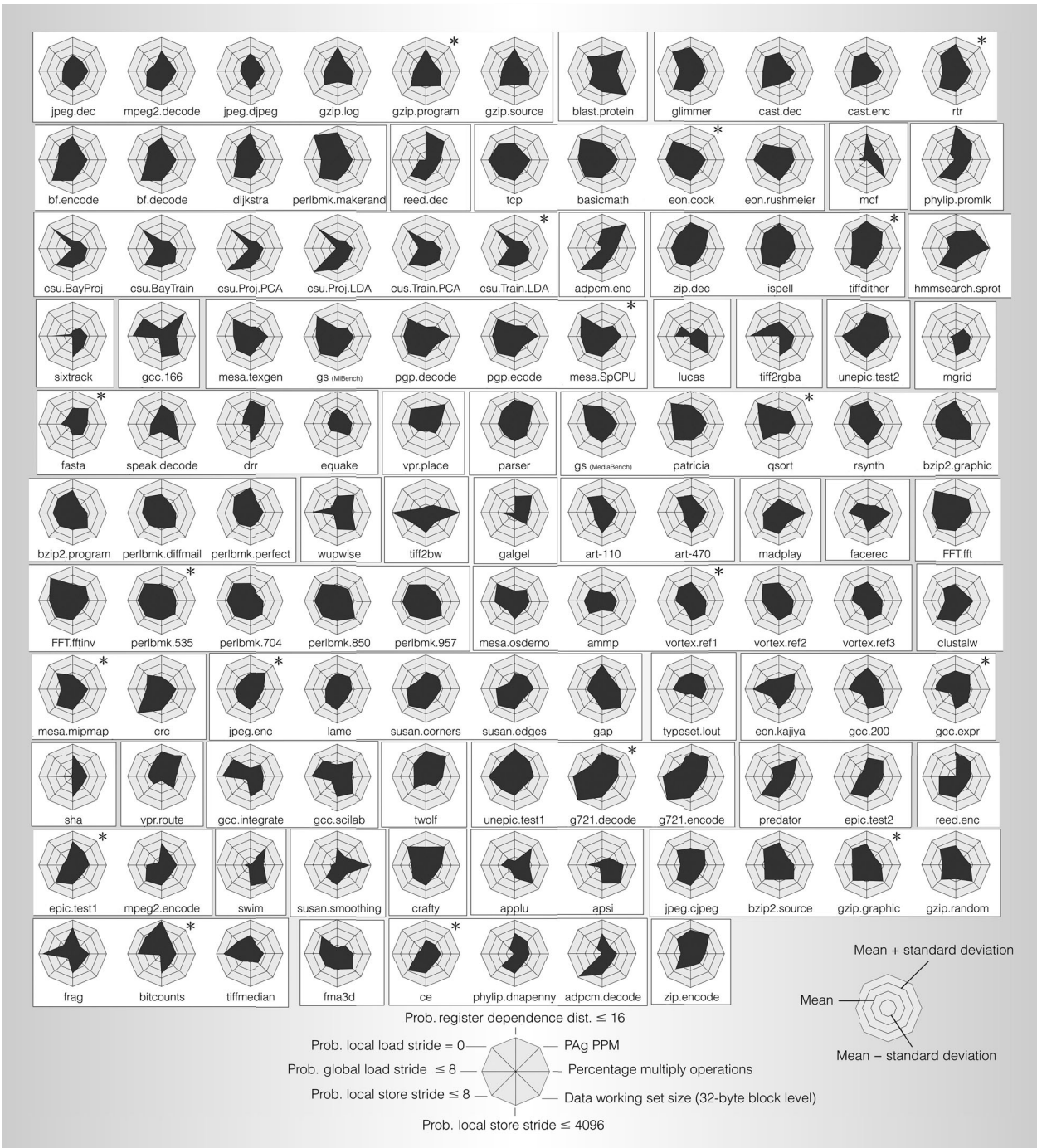

Figure 2. Kiviat diagrams representing the eight key microarchitecture-independent characteristics of the 118 benchmarks. Darkest areas represent inherent behavior patterns. Boxes represent clusters, and asterisks indicate cluster representatives.

others are not. For example, the input to gcc and perl causes quite different behavioral characteristics; for vortex, gzip, and csu, on the other hand, the input doesn't seem to affect the behavioral characteristics as greatly. Yet another interesting observation is that many benchmarks from recently introduced benchmark suites exhibit dissimilar inherent behavior compared to SPECcpu2000 benchmarks. More particularly, about 40 percent of the clusters don't contain any of the SPECcpu benchmarks. This suggests that

SPECcpu doesn't adequately cover the entire workload space, and that a more complete benchmark suite should incorporate additional benchmarks for the microprocessor design cycle. The methodology presented here lets designers select a diverse and representative benchmark set.

Several potential avenues toward an even more effective workload characterization methodology are worth addressing. First, speeding up the time-consuming profiling step of collecting microarchitecture-independent characteristics would greatly improve our methodology's usability. Possible ways to accomplish this include sampling, hardware acceleration, and multithreaded profiling. Second, as we have demonstrated, a microarchitecture-independent workload characterization is more accurate and informative than a microarchitecture-dependent characterization. Ideally, however, the characterization should also be independent of the ISA and the compiler, to capture a program's true inherent behavior. Making the characterization both ISA and compiler independent is thus another interesting path to explore. Third, an important question is what program characteristics to include in the analysis. It is important that the characterization includes a sufficiently diverse set of characteristics. For example, extending the current set of characteristics to characterize multithreaded workloads in a microarchitecture-independent manner is an area to focus on. Also, as architectures evolve, it is important to revisit these program characteristics. Finally, other statistical, machine-learning, or data-mining techniques might be useful for analyzing workload behavior and extracting key program behavior characteristics. MICRO

Acknowledgments

We thank the anonymous reviewers for their valuable comments. Kenneth Hoste receives support through a doctoral student fellowship from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT-Flanders). Lieven Eeckhout receives support through a postdoctoral fellowship of the Fund for Scientific Research, Flanders, Belgium (FWO-Vlaanderen). We also thank Kjell Andresen from the University of Oslo for offering Alpha machine compute cycles.

References

1. C. Lee, M. Potkonjak, and W.H. Mangione-Smith, ''MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems,'' *Proc. 30th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO 97), IEEE CS Press, 1997, pp. 330-335.
2. M.R. Guthaus et al., ''MiBench: A Free, Commercially Representative Embedded Benchmark Suite,'' *Proc. 4th Ann. IEEE Int'l Workshop Workload Characterization* (WWC 01), IEEE CS Press, 2001, pp. 3-14.
3. C.-B. Cho et al., ''Workload Characterization of Biometric Applications on Pentium 4 Microarchitecture,'' *Proc. IEEE Int'l Symp. Workload Characterization* (IISWC 05), IEEE Press, 2005, pp. 76-86.
4. Y. Li and T. Li, *BioInfoMark: A Bioinformatic Benchmark Suite for Computer Architecture Research*, tech. report, Univ. of Florida, Dept. of ECE, 2005.
5. D.A. Bader et al., ''BioPerf: A Benchmark Suite to Evaluate High-Performance Computer Architecture on Bioinformatics Applications,'' *Proc. IEEE Int'l Symp. Workload Characterization* (IISWC 05), IEEE Press, 2005, pp. 163-173.
6. K. Hoste and L. Eeckhout, ''Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics,'' *Proc. IEEE Int'l Symp. Workload Characterization* (IISWC 06), IEEE Press, 2006, pp. 83-92.
7. K. Hoste et al., ''Performance Prediction Based on Inherent Program Similarity,'' *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 06), IEEE CS Press, 2006, pp. 114-122.
8. I.K. Chen, J.T. Coffey, and T.N. Mudge, ''Analysis of Branch Prediction via Data Compression,'' *Proc. 7th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS VII), ACM Press, 1996, pp. 128-137.
9. R.A. Johnson and D.W. Wichern, *Applied Multivariate Statistical Analysis*, 5th ed., Prentice Hall, 2002.
10. L. Eeckhout, H. Vandierendonck, and K. De Bosschere, ''Quantifying the Impact of Input

Data Sets on Program Behavior and Its Applications,'' *J. Instruction-Level Parallelism*, vol. 5, Feb. 2003, http://www.jilp.org/vol5.

11. A. Joshi et al., ''Measuring Benchmark Similarity Using Inherent Program Characteristics,'' *IEEE Trans. Computers*, vol. 55, no. 6, June 2006, pp. 769-782.
12. L. Eeckhout, J. Sampson, and B. Calder, ''Exploiting Program Microarchitecture Independent Characteristics and Phase Behavior for Reduced Benchmark Suite Simulation,'' *Proc. IEEE Int'l Symp. Workload Characterization* (IISWC 05), IEEE Press, 2005, pp. 2-12.
13. J.J. Yi et al., ''Evaluating Benchmark Subsetting Approaches,'' *Proc. IEEE Int'l Symp. Workload Characterization* (IISWC 06), IEEE Press, 2006, pp. 93-104.

Kenneth Hoste is a doctoral student in the Electronics and Information Systems Department of Ghent University, Belgium. His research interests include computer architecture in general and workload characterization in particular. Hoste has an MS in computer science from Ghent University.

Lieven Eeckhout is an assistant professor in the Electronics and Information Systems Department of Ghent University, Belgium. His research interests include computer architecture, virtual machines, performance analysis and modeling, and workload characterization. Eeckhout has a PhD in computer science and engineering from Ghent University. He is a member of the IEEE.

Direct questions and comments about this article to Kenneth Houste and Lieven Eeckhout, ELIS, Ghent University, Sint-Pieternieuwstraat 41, B-9000 Gent, Belgium; kehoste@elis.UGent.be and leeckhou@elis.UGent.be.