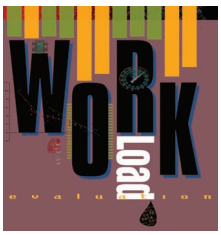
Designing Computer Architecture Research Workloads



MinneSPEC proposes reduced input sets that microprocessor designers can use to model representative short-running workloads. A four-step methodology verifies the program behavior similarity of these input sets to reference sets.

Lieven Eeckhout
Hans Vandierendonck
Koen De Bosschere
Ghent University

Designers of microarchitectures for general-purpose microprocessors once based their design decisions on experts' intuition and rules of thumb. Since the mid-1980s, however, microarchitecture research has become a systematic process that uses simulation tools extensively.[1] Although architectural simulators model microarchitectures at a high abstraction level, the increasing complexity of both the microarchitectures and the applications that run on them make these simulators very time-consuming.

Simulators must execute huge numbers of instructions to create a workload representative of real applications. The Standard Performance Evaluation Corporation's (SPEC) CPU2000 benchmark suite,[2] for example, has many more dynamic instructions than CPU95, which it replaced. Although real hardware evaluations benefit from this increase, using architectural simulators for such large numbers of instructions becomes infeasible. The dynamic instruction count of the SPEC2000 benchmark parser with reference input is about 500 billion instructions, or three weeks of simulation at 300,000 instructions per second.[3] Including the benchmarks that must be run for a huge number of design points creates an unreasonably long simulation time, stretching the time to market. Running the simulations in parallel results in a huge equipment cost.

To solve this problem, we can use reduced input sets instead of reference input sets. The ideal reduced input set has a limited dynamic instruction count but produces program behavior comparable to the reference input set behavior. MinneSPEC collects a number of reduced input sets for some CPU2000 benchmarks.[4] It proposes three reduced inputs: *smred* for short simulations, *mdred* for medium-length simulations, and *lgred* for full-length, reportable simulations. Although a number of techniques—such as truncating or modifying the inputs—can derive these reduced input sets from the reference inputs, it is unclear whether these reduced input sets will produce behavior similar to a program using a reference input set.

We have developed a methodology that reliably quantifies program behavior similarity.[5] As such, we can validate MinneSPEC—that is, we can verify whether the reduced input sets result in program behavior similar to the reference inputs. To overcome the shortcomings of previous work, our methodology uses metrics that are closely related to performance. We also use statistical data analysis techniques to calculate the similarity in program behavior based on uncorrelated workload characteristics.

MEASURING PROGRAM BEHAVIOR SIMILARITY

To validate the reduced input sets they propose in MinneSPEC, A.J. KleinOsowski and David Lilja[4] performed a chi-square analysis of each set's function-level execution profiles. A resemblance of these profiles does not necessarily imply a resemblance of other workload characteristics that are probably more closely related to performance, such as instruction mix, cache behavior, and branch predictability. If we scale down the number of times a function executes by a factor *S*, for example, we still

0018-9162/03/$17.00 © 2003 IEEE Published by the IEEE Computer Society

Principal Components Analysis

Principal components analysis (PCA) is a statistical data analysis technique that builds on the assumption that many variables are correlated and hence measure the same or similar properties of the program-input pairs.[1] PCA computes *principal components*—new variables that are linear combinations of the original variables such that all principal components are uncorrelated.

PCA transforms 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

- $Var[Z_1] > Var[Z_2] > \ldots > Var[Z_p]$, which means that Z_1 contains the most information and Z_p the least; and
- $Cov[Z_i, Z_j] = 0, \forall i \neq j$, which means that there is no information overlap between the principal components.

The total variance in the data remains the same before and after the transformation, namely

$$\sum_{i=1}^{p} Var[X_i] = \sum_{i=1}^{p} Var[Z_i]$$

Some principal components have a large variance while others have a small variance. Eliminating the components with the smallest variance reduces the number of variables while controlling the amount of information that is thrown away. Retaining q principal components significantly reduces the information, as q is usually less than p. To measure the fraction of information retained in this q-dimensional space, we use the amount of variance (shown below) explained by the q principal components computed as

$$\left(\sum_{i=1}^{q} Var[Z_i]\right) / \left(\sum_{i=1}^{p} Var[X_i]\right)$$

We can interpret the most important q principal components, which in our study are linear combinations of the original workload characteristics, in terms of these characteristics. To facilitate our interpretation, we apply the *varimax rotation*.[1] This rotation makes the coefficient a_{ij} either close to ±1 or zero, such that the original variables either strongly impact a principal component or don't impact it. Although varimax rotation is an orthogonal transformation, implying that the rotated principal components are still uncorrelated, the first component might not explain the largest variance.

Next we display the various benchmarks as points in the q-dimensional space built by the q principal components. To do this, we compute the values of the q principal components for each program-input pair. This representation lets us measure the impact of input data sets on program behavior.

During principal components analysis, we can use either normalized or nonnormalized data. The data is normalized when the mean of each variable is zero and its variance is one. For nonnormalized data, variables with a larger variance get a higher weight. We used normalized data in our experiments because of our heterogeneous data—for example, the instruction-level parallelism variance is orders of magnitude larger than the data cache miss-rate variance.

Reference

1. B.F.J. Manly, *Multivariate Statistical Methods: A Primer*, 2nd ed., Chapman & Hall, 1994.

have the same function-level execution profile.

However, a similar function-level execution profile doesn't guarantee a similar behavior concerning, for example, the data memory. Indeed, reducing the input set often reduces the size of the data the program is working on while leaving the function-level execution profile untouched. As a result, data cache behavior can differ significantly.

Rafael Saavedra and Alan Jay Smith took another approach to quantifying program behavior similarity.[6] They measured several dynamic workload characteristics—instruction mix, number of function calls, number of address computations, and so on. To measure similarities among computer programs, they used the squared Euclidean distance in the space built using these workload characteristics. Small distances imply similar behavior; large distances imply dissimilar behavior. Because they measured the distance between programs on correlated workload characteristics, Saavedra and Smith gave a higher weight to correlated workload characteristics, which might give a distorted view of the workload space.

Wei Chung Hsu and colleagues[7] studied the impact of input sets on program behavior using high-level metrics, such as procedure-level profiles and instructions executed per cycle (IPC), and low-level metrics, such as the execution paths leading to data cache misses.

OUR METHODOLOGY

Our methodology consists of four steps:

- collecting program-input pairs,
- choosing workload characteristics,
- performing principal components analysis (PCA), and
- performing cluster analysis.

For more detailed discussions of the statistical analysis techniques, see the "Principal Components Analysis" and "Cluster Analysis" sidebars.

Collect program-input pairs

In the first step of our methodology, we collected a large number of program-input pairs. For this study, we used the program-input pairs provided by the reduced SPEC CPU2000 benchmark input data set distribution (www.spec.org/cpu2000/research/umn/), limiting ourselves to the integer benchmarks. We also used a database workload consisting of TPC-D queries from the Transaction Processing Performance Council (www.tpc.org). These decision support queries ask complex questions of complex data structures.

Cluster Analysis

Cluster analysis groups n cases (*program-input pairs* in our study) based on the measurements of p variables (or *workload characteristics*). The final goal is a number of clusters containing program-input pairs with similar behavior.

Researchers often use a hierarchical clustering algorithm to perform cluster analysis, starting with a matrix of distances between the n cases or program-input pairs. The algorithm starts by considering each program-input pair as a cluster. In each iteration, the algorithm combines the two clusters with the shortest linkage distance to form a new cluster. Clusters are gradually merged until all cases are in a single cluster.

A *dendrogram*, which graphically represents the linkage distance at each iteration of the algorithm, can represent the cluster analysis. The user then must decide how many clusters to use, a decision that can be based on the linkage distance.

Cluster analysis depends on an appropriate distance measure. In our analysis, we compute the distance between two program-input pairs as the Euclidean distance in the workload space obtained using PCA. Principal components, which are uncorrelated by construction, determine the space's axes values, meaning the values are uncorrelated.

The absence of correlation is important when calculating Euclidean distances because two correlated variables—variables that essentially measure the same thing—will contribute a similar amount to the overall distance as an independent variable, and as such would be counted twice, which is undesirable. This is also why we apply cluster analysis in the transformed q-dimensional space rather than the original p-dimensional space. Moreover, the variance along the q principal components measures the diversity along each principal component by construction.

In addition to defining the distance between two program-input pairs, we must define the distance between clusters of two program-input pairs. We can compute the distance between two clusters by using the furthest-neighbor strategy or complete linkage. This strategy defines the distance between two clusters as the largest distance between two members of each cluster.

We downloaded the SPEC benchmarks, optimized for the Alpha 21264 architecture, from the SimpleScalar Web site (www.simplescalar.com). In addition, we used single-threaded Postgres v6.3 (www.postgresql.org), an advanced open source database, running the decision support TPC-D queries over a 10-Mbyte balanced tree (Btree) indexed database. The Digital C compiler fully optimizes postgres (–04) and links it statically. We used a total of 101 program-input pairs, including ref, train, and test from SPEC and smred, mdred, and lgred from MinneSPEC.

Choose workload characteristics

It's important to choose the workload characteristics for your analysis carefully. A workload characteristic that doesn't affect a computer program's behavior, such as dynamic instruction count, might discriminate program-input pairs on that characteristic without giving any information about the application's behavior.

Obviously, the workload characteristics you choose will depend on your application domain. If you're using the workload to design a low-power microprocessor, you'll need to include power-related characteristics (or characteristics that significantly affect power consumption). On the other hand, if you're designing a high-performance microprocessor, you should select characteristics more directly related to performance, such as the amount of parallelism.

Our analysis included the 18 workload characteristics listed in Table 1. We commonly use these characteristics to describe computer programs for general-purpose microprocessors.

We measure workload characteristics using ATOM,[8] a binary instrumentation tool for the Alpha architecture. ATOM can transform statically linked binaries to instrumented binaries. Executing this instrumented binary yields the workload characteristics we used throughout our analysis.

Table 1. Workload characteristics included in the analysis.

Benchmark	No.	Workload characteristic
Instruction mix	1	Percentage integer arithmetic operations
	2	Percentage logical operations
	3	Percentage shift and byte manipulation operations
	4	Percentage load/store operations
	5	Percentage control operations
Branch prediction (BP)	6	Branch prediction accuracy for a hybrid branch predictor selecting between 16 Kbits of 8,192 entries bimodal predictor and 16 Kbits of 8,192 entries gshare predictor (history = 12 branches); the metapredictor contains 16 Kbits of 8,192 entries.
Sequential flow breaks	7	Number of instructions between two sequential flow breaks, or the number of instructions between two taken branches
Data cache miss rates	8	Miss rate of a 8-Kbyte direct-mapped data cache
	9	Miss rate of a 16-Kbyte direct-mapped data cache
	10	Miss rate of a 32-Kbyte two-way set-associative data cache
	11	Miss rate of a 64-Kbyte two-way set-associative data cache
	12	Miss rate of a 128-Kbyte four-way set-associative data cache
Instruction cache miss rates	13	Miss rate of a 8-Kbyte direct-mapped instruction cache
	14	Miss rate of a 16-Kbyte direct-mapped instruction cache
	15	Miss rate of a 32-Kbyte two-way set-associative instruction cache
	16	Miss rate of a 64-Kbyte two-way set-associative instruction cache
	17	Miss rate of a 128-Kbyte four-way set-associative instruction cache
Instruction-level parallelism (ILP)	18	The amount of ILP on an infinite-resource processor (assuming an infinite number of functional units, perfect caches, perfect branch prediction, and unit instruction execution latency)—that is, an infinite-resource processor only considers read-after-write dependencies through registers and memory.

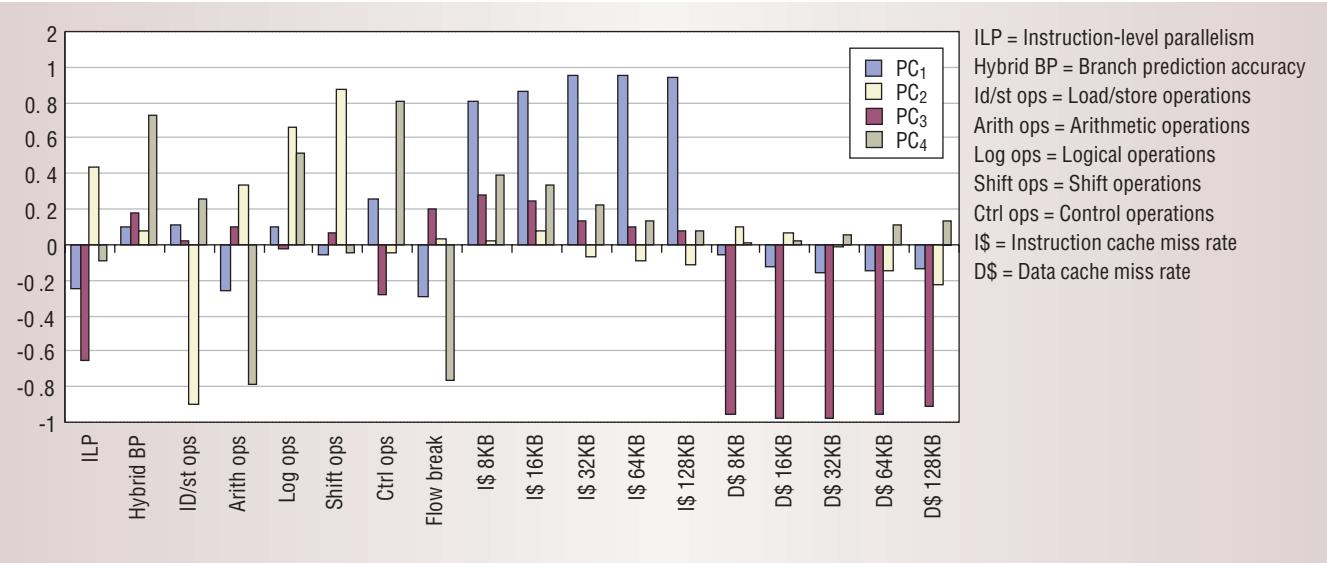


Figure 1. Factor loadings for four principal components, PC_1–PC_4. A factor loading close to ±1 means a high impact in the principal component; a factor loading close to 0 means no impact.

Because this binary runs on native hardware, we can obtain these characteristics quickly.

Perform PCA

Next, we normalize the 101 (number of program-input pairs) by 18 (or p, number of workload characteristics) data points so that for each workload characteristic, the average equals zero and the variance equals one. Using Statistica (www.statsoft.com), a statistical computation package, we performed the statistical analysis described in the "Principal Components Analysis" sidebar on these normalized data points.

First we input a 2D matrix with $p = 18$ columns representing the original workload characteristics and 101 rows representing the program-input pairs. PCA transforms the p original variables into new variables, or *principal components*, which are linear combinations of the original variables. These principal components are uncorrelated—that is, unlike the original variables, they contain no information overlap.

Based on the amount of information contained in the components, the user must now determine how many principal components to retain, say q. Typically, a user must keep two to four principal components to retain most—for example, more than 85 percent—of the total information.

The user can analyze and interpret the q principal components with the coefficients a_{ij}, or *factor loadings*. A positive coefficient a_{ij} means that workload characteristic X_j positively impacts principal component Z_i; a negative coefficient a_{ij} implies a negative impact. If a coefficient a_{ij} is close to zero, X_j has almost no impact on Z_i.

Analyzing the workload space. For our program-input pairs, PCA extracts four principal components, accounting for 85.5 percent of the total variance. Figure 1 presents the factor loadings for these four components—for example, $PC_1 = -0.25(ILP) + 0.10(branch\ prediction) + 0.11(load/store\ operations) + \ldots$. As the figure shows,

- PC_1, which accounts for 25 percent of the total variance, is positively dominated by the instruction cache miss rate. Thus, program-input pairs with a high value for the first principal component in the q-dimensional space have a high instruction cache miss rate.
- PC_2, which accounts for 29.7 percent of the total variance, is negatively dominated by the data cache miss rate. Thus, program-input pairs with a high value for the second principal component have a low data cache miss rate.
- PC_3, which accounts for 13.5 percent of the total variance, is positively dominated by the percentage of logical and shift operations and negatively dominated by the percentage of load/store operations.
- PC_4, which accounts for 17.3 percent of the total variance, is positively dominated by the branch prediction accuracy and the percentage of control operations and negatively dominated by the percentage of arithmetic operations and the number of instructions between two sequential flow breaks.

The variances in the four components mean that the variability in program behavior due to the input set results from, in decreasing order, the data cache behavior, the instruction cache behavior, the branch behavior, and the instruction mix.

Visualizing the workload space. We can now display the program-input pairs in the workload space shaped by the q principal components by computing the following for each pair:

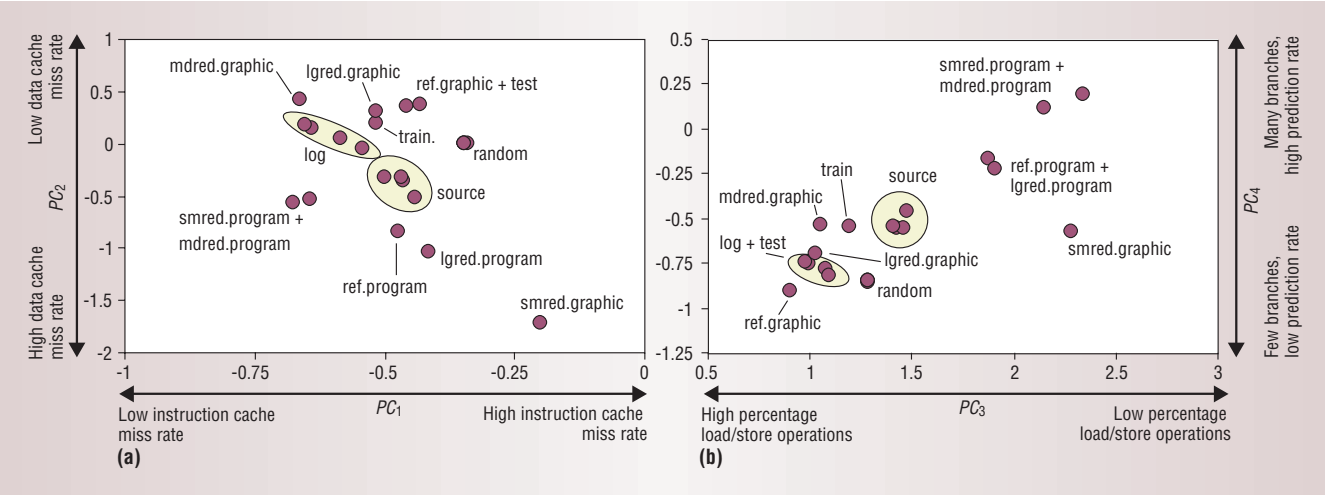


Figure 2. Excerpt from the 4D space shaped by the four principal components for gzip: (a) instruction cache behavior versus data cache behavior, and (b) percentage load/store operations versus branch behavior.

$$Z_i = \sum_{j=1}^{p} a_{ij} X_j$$

Figure 2 shows an excerpt of this 4D space containing the program-input pairs associated with SPEC benchmark gzip. Figure 2a compares the first principal component PC_1 (instruction cache behavior) to principal component PC_2 (data cache behavior), and Figure 2b compares PC_3 (percentage of load/store operations) to PC_4 (branch behavior). Program-input pairs with a high data cache miss rate and a high instruction cache miss rate appear in the bottom right-hand corner of Figure 2a as smred.graphic.

In Figure 2, input sets associated with random are very close to each other in the workload space, indicating that they have a minor impact on the program behavior. Other input sets—graphic and program, for example—lead to increased diversity in program behavior.

We can use the transformed workload space in Figure 2 to measure the impact of input data sets on program behavior. Weak clustering (for various inputs and a single benchmark) indicates that the input set significantly impacts program behavior, whereas strong clustering indicates a small impact.

The representation also suggests which input sets we should select when composing a workload. Strong clustering suggests that one or a few input sets could represent the cluster. Using fewer input sets reduces the total simulation time significantly because it reduces the total number of benchmark-input pairs, and it lets us select the benchmark-input pair with the smallest dynamic instruction count.

Perform cluster analysis

The "Cluster Analysis" sidebar describes how we use the distance between the various program-input pairs in the 4D space as a subsequent step in workload analysis.

Figure 3 shows the dendrogram displaying the linkage distance obtained through cluster analysis. Program-input pairs connected by small linkage distances are clustered in early iterations of the analysis and thus exhibit similar behavior—similar cache miss rates, branch predictability, instruction mix, and so on. Program-input pairs connected by large linkage distances, on the other hand, exhibit dissimilar behavior.

In other words, we can now use the dendrogram to evaluate program behavior similarity due to input set. The analysis provides three possible classifications:

- *S*, or small linkage distance ($d < 2$), reveals similar behavior;
- *M*, or medium linkage distance ($2 < d < 5$), reveals more or less similar behavior; and
- *D*, or large linkage distance ($d > 5$), reveals dissimilar behavior.

Input smred.graphic, for example, is far from its reference counterpart ref.graphic (linkage distance ±17). Input smred.log, on the other hand, is very close to ref.log (linkage distance ±1). The linkage distance from inputs smred.program and mdred.program to ref.program is ±3.5, a medium distance.

We can use the information that PCA provides—in particular, the workload space visualization (see Figure 2)—to explain why two program-input pairs differ. For example, smred.graphic differs from ref.graphic because of variations along the second and the third principal component—that is, the data cache miss rate is too high and the percentage of load/store operations is too low for smred.graphic.

VALIDATING MINNESPEC

To validate MinneSPEC, we use the dendrogram in Figure 3 to verify each reduced input's similar-

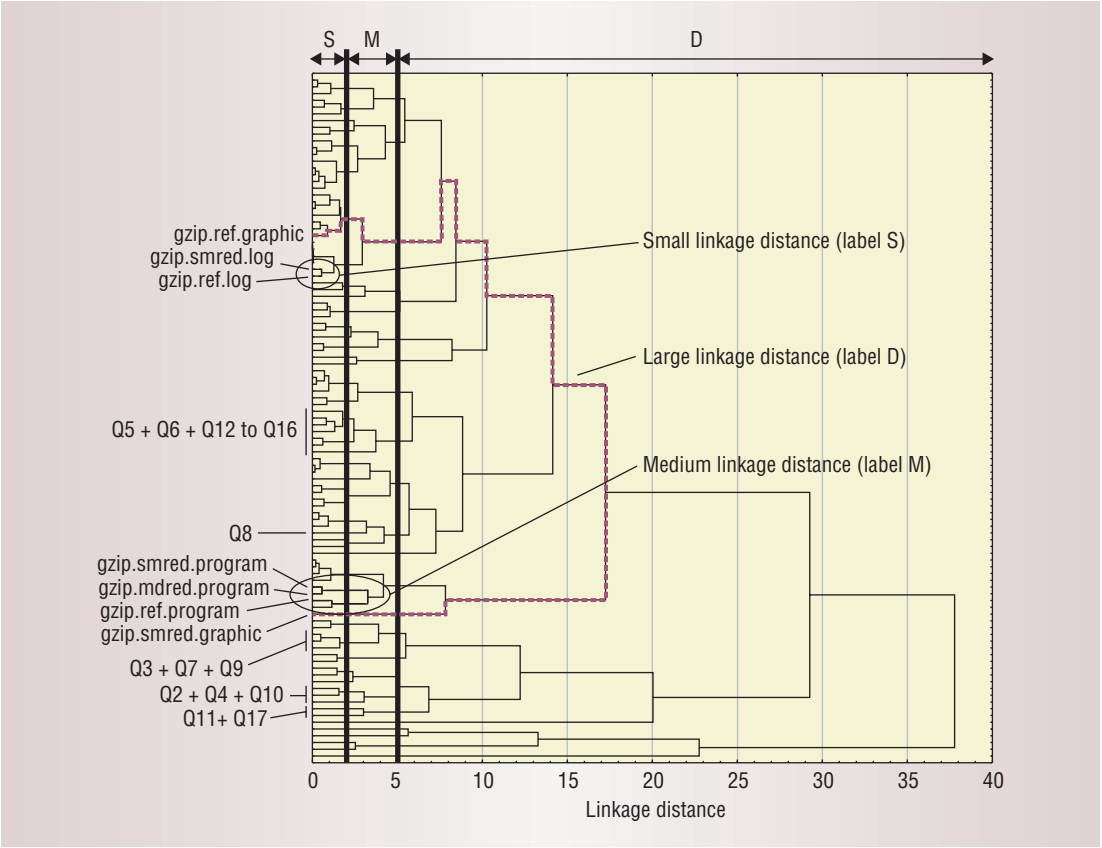


Figure 3. Linkage distances obtained through cluster analysis for several program-input pairs. For gzip, smred.log is very similar to ref.log, smred.program and mdred.program are more or less similar to ref.program, and smred.graphic is very dissimilar to ref.graphic.

ity or dissimilarity to the corresponding reference input. We can make some general conclusions based on the results:

- The lgred input is generally more similar to the reference input than the other inputs proposed in MinneSPEC.
- The smallest inputs, smred and mdred, generally lead to dissimilar behavior.

KleinOsowski and Lilja[4] compared input sets based on *function-level execution profiles*—distributions measuring the fraction of time a program spends in each function—measured using the Unix utility gprof. Although their results generally agree with ours, some results differ. KleinOsowski and Lilja found, for example, the lgred input of benchmark vpr.place to be similar to the ref input; our results show dissimilar behavior. The reverse also occurs: Whereas the MinneSPEC results suggest a dissimilar behavior for the train input of bzip2 and the reduced inputs for perlbmk, we show medium similarity. For more information about MinneSPEC, see www-mount.ee.umn.edu/~lilja/spec2000/.

Generally, comparing input sets based on function-level execution profiles is accurate. In some cases, however, similar function-level execution profiles might result in dissimilar program behavior, and vice versa, so we must be careful when comparing input sets using these profiles only. KleinOsowski and Lilja also recognize this problem and validate their results with instruction mixes and data cache miss rates.

We include additional characteristics, such as instruction cache miss rate and branch prediction accuracy, and integrate these measurements in a single analysis. Our analysis revealed several interesting results:

- The reference inputs of vortex result in very similar behavior, raising the question of whether simulating all reference inputs during architectural simulations is useful.
- For vpr, place and route reference inputs result in dissimilar behavior. Thus, we should consider both inputs in architectural simulations.
- For vortex, mdred is closer to the ref input than the lgred input, although it has a smaller dynamic instruction count. As such, mdred is a better candidate for architectural simulations.
- The reduced inputs for perlbmk are very similar to each other, but differ slightly from the reference inputs. The test input, on the other hand, resembles the ref.perfect input.

For the database workload used in this analysis (postgres running the TPC-D queries), we observe five clusters of queries, as Figure 3 shows: queries 2, 4, and 10; queries 5, 6, and 12 to 16; query 8; queries 3, 7, and 9; and queries 11 and 17. This clustering reveals that queries can lead to quite dis-

similar database behavior. An analysis of the cause of this difference reveals that the second cluster contains queries resulting in relatively low data cache miss rates, and the fourth and the fifth clusters contain queries with relatively high instruction cache miss rates.

We could also use our methodology in profile-driven compiler optimizations. During this process, the compiler uses information from previous program runs—obtained through profiling—to guide compiler optimizations. Obviously, for effective optimizations, the input set used to obtain this profiling information should be similar to a large set of possible input sets.

Joseph Fisher and Stefan Freudenberger[9] show that branches generally take the same direction independent of the input data. They warn, however, that some program runs exercise entirely different parts of the program. Hence, we can't use these runs to predict each other. In a study of several types of profiles, such as basic block counts and the number of references to global variables, David Wall[10] obtained the largest speedup when using the same input for profiling and measuring the speedup.

Designing a computer architecture research workload should be done with care. Our methodology lets us select a limited set of representative program-input pairs with small dynamic instruction counts. As such, we can spend less time simulating microprocessor architectures while still producing reliable research results. ■

References

1. P. Bose and T. M. Conte, "Performance Analysis and Its Impact on Design," *Computer*, May 1998, pp. 41-49.
2. J.L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *Computer*, July 2000, pp. 28-35.
3. T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, Feb. 2002, pp. 59-67.
4. A.J. KleinOsowski and D.J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters*, June 2002, pp. 10-13.
5. L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Workload Design: Selecting Representative Program-Input Pairs," *Proc. 2002 Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 02), IEEE CS Press, 2002, pp. 83-94.
6. R.H. Saavedra and A.J. Smith, "Analysis of Benchmark Characteristics and Benchmark Performance Prediction," *ACM Trans. Computer Systems*, vol. 14, no. 4, 1996, pp. 344-384.
7. W.C. Hsu et al., "On the Predictability of Program Behavior Using Different Input Data Sets," *Proc. 6th Workshop on Interaction between Compilers and Computer Architectures* (INTERACT 02), held in conjunction with the *8th Int'l Symp. High-Performance Computer Architecture* (HPCA-8), IEEE CS Press, 2002, pp. 45-53.
8. A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," tech. report 94/2, Western Research Lab, Compaq, 1994.
9. J. Fisher and S. Freudenberger, "Predicting Conditional Branch Directions from Previous Runs of a Program," *Proc. 5th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS-V), ACM Press, 1992, pp. 85-95.
10. D.W. Wall, "Predicting Program Behavior Using Real or Estimated Profiles," *Proc. 1991 Int'l Conf. Programming Language Design and Implementation* (PLDI 91), ACM Press, 1991, pp. 59-70.

Lieven Eeckhout *is a PhD candidate in computer science and engineering at Ghent University, Belgium. His research interests include computer architecture, performance analysis, and workload characterization. Eeckhout received an engineering degree in computer science from Ghent University. Contact him at leeckhou@elis.rug.ac.be.*

Hans Vandierendonck *is a PhD candidate in computer science and engineering at Ghent University. His research interests include computer architecture, performance modeling, and workload characterization. Vandierendonck received an engineering degree in computer science from Ghent University. Contact him at hvdieren@elis.rug.ac.be.*

Koen De Bosschere *is a professor at the Faculty of Applied Sciences of Ghent University, where he teaches courses in computer architecture, operating systems, and declarative programming languages. His research interests include logic programming, system programming, parallelism, and debugging. De Bosschere received a PhD in applied sciences from Ghent University. Contact him at kdb@ elis.rug.ac.be.*