

STATE-OF-THE-ART IN PERFORMANCE MODELING AND SIMULATION:
Theory, Techniques, and Tutorials. Edited by K. Bagchi, G. Zobrist, and K. Trivedi.
Gordon and Breach Publishers, 1996.

CHAPTER 12

EVALUATION AND DESIGN OF BENCHMARK SUITES

Jozo J. Dujmović

12.1. INTRODUCTION

Benchmark suites are most frequently designed for industrial evaluation of competitive computer systems and networks. Examples of such benchmark suites include SPEC¹, TPC², GPC³, PERFECT Club Benchmarks⁴, AIM benchmarks⁵, and others. In addition to benchmark suites sponsored by consortia of computer industry there are various collections of benchmarks designed by research organizations, companies, computer magazines, and individuals for benchmarking specific hardware and software systems. Examples of such benchmark suites include database benchmarks⁶, supercomputer benchmarks (Livermore loops⁷, NAS Parallel Benchmarks⁸, Lisp benchmarks⁹, Prolog benchmarks¹⁰, and many others¹¹). In the majority of cases benchmark workloads are selected from a specific set of frequently used real workloads. The selection process is usually aimed at simultaneous satisfaction of two goals: (1) benchmark workloads should be a good functional representative of a given universe of real workloads, and (2) benchmark workloads should yield the same distribution of the utilization of system resources as real workloads. Due to the absence of specific quantitative methods for benchmark suite design, these goals primarily serve as guidelines in an intuitive workload selection process. Such a process cannot include proofs of the extent to which the goals are satisfied,

and frequently yields relatively low reliability of benchmark results and excessive cost of benchmarking.

This chapter will propose quantitative methods for evaluation and design of benchmark suites which can increase the reliability and decrease the cost of benchmarking of computer systems and networks. These methods are based on appropriate white-box and black-box workload difference models. The models are then used to define benchmark suite performance criteria, and to develop techniques for benchmark suite evaluation and design. The proposed design techniques are based on a quantitative optimization of benchmark suite performance criteria.

12.2. WORKLOAD DIFFERENCE CONCEPTS

Let us consider a set of n workloads, denoted $B_1, \dots, B_n, n > 1$. We assume that these workloads are used as benchmark programs for performance measurement and comparison of m competitive computer systems, $S_1, \dots, S_m, m > 1$. In this context we are interested in developing quantitative models of difference and similarity between each pair of workloads. The difference between B_j and B_k will be denoted $d(B_j, B_k)$, and the similarity will be denoted $s(B_j, B_k)$. The first step in the development of workload difference models consists of introducing the concepts of minimum and maximum difference between workloads.

A *program space* is a space where points represent programs (or some other more complex computer workloads). Differences between computer workloads can be expressed as distances between points in the program space. The proximity of two points in the program space denotes the similarity of corresponding computer workloads. Therefore, similar workloads are represented as clusters of points in the program space.

The difference between workloads $d_{jk} = d(B_j, B_k)$ always satisfies the mathematical conditions of non-negativity ($(\forall j, k) \ d_{jk} \geq 0$), non-degeneracy ($d_{jk} = 0$, for $j = k$, and $d_{jk} > 0$, for $j \neq k$), and symmetry ($d_{jk} = d_{kj}$). These conditions are known as *semimetric*.

In benchmarking it is important to note that a given computer workload can be expressed using several equivalent source program forms, and all such forms must be interpreted as the same workload. In other words, $d_{jk} = 0$ simply implies that B_j and B_k are two equivalent forms

of the same workload. For example, in C, the *for* loop

```
for(expr_1; expr_2; expr_3) statement;
```

is equivalent to the *while* loop

```
expr_1; while (expr_2) { statement; expr_3; }
```

After compiling and linking, both loops should generate the same executable code. So these are equivalent source forms of the same workload. Similarly, many benchmark workloads have a loop form

```
for(i=0; i<K; i++) benchmark_workload();
```

where K denotes a scaling factor used to adjust the total run time according to the desired accuracy and convenience of measurement. We assume that the nature of workload does not depend on the value of K , and that the above *for* loop, for any $K > 0$, is equivalent to the loop body, `benchmark_workload()`.

Let us now address the concept of maximum difference between benchmark workloads. If the difference is not limited, then some workloads might have an “infinite difference.” Unfortunately, it is not easy to find a reasonable intuitive interpretation of such a difference. In addition, it is not possible to use fractions of the maximum difference, and it is not possible to define similarity as a simple linear complement of difference. On the other hand, if we assume that the difference between benchmark workloads is limited, and that the maximum difference is d_{max} , then any specific model of difference between workloads will include the concept of “completely different workloads,” where $d(B_j, B_k) = d_{max}$. For each specific difference model the completely different workloads must have an easily acceptable intuitive interpretation which supports the credibility of the model. All differences can now be conveniently interpreted as fractions of d_{max} , and the similarity between workloads can be defined as a complement of difference:

$$s(B_j, B_k) = d_{max} - d(B_j, B_k) .$$

Therefore, it is reasonable to make the following fundamental assumption:

$$0 \leq d(B_k, B_k) \leq d_{max} < +\infty ,$$

which we call *the concept of limited difference*. For most applications it is convenient to adopt $d_{max} = 1$, or $d_{max} = 100\%$.

Let us now show that all workload difference models fundamentally depend on the purpose of benchmarking. In other words, we cannot expect to have a single workload difference model. Each specific benchmarking problem can yield a corresponding workload difference model. This is consistent with the traditional classification of workload models which differentiates physical models (which are system dependent and based on utilization of hardware and software), virtual models (reflecting a programmer's viewpoint of the use of logical resources), and functional models (which are system independent and application oriented).

Benchmarking is most frequently performed to generate data which can be used for comparison and ranking of competitive systems. Comparison is usually based either on measured run times (t_1, \dots, t_m) , or throughputs (x_1, \dots, x_m) . If ε_t and ε_x denote maximum expected measurement errors for times and throughputs respectively, then the notation used for ranking competitive systems can be defined as follows:

$S_j \succ S_k$: S_j is better than S_k , i.e. $t_j < t_k - \varepsilon_t$, or $x_j > x_k + \varepsilon_x$

$S_j \succeq S_k$: S_j is better than or equivalent to S_k , i.e. $t_j \leq t_k - \varepsilon_t$,
or $x_j \geq x_k + \varepsilon_x$

$S_j \sim S_k$: S_j is equivalent to S_k , i.e. $|t_j - t_k| < \varepsilon_t$, or $|x_j - x_k| < \varepsilon_x$.

If the only purpose of benchmarking is to provide a ranking $(S_1 \succeq S_2 \succeq \dots \succeq S_m)$ of m competitive systems, then all benchmarks that yield the same ranking are equivalent. This standpoint can be used when developing difference models for workloads belonging to industrial benchmark suites such as SPEC, TPC, GPC, etc.

If the purpose of benchmarking is to measure the average utilizations of N resources of a computer system, U_1, \dots, U_N , then all benchmarks yielding the same utilization distribution are equivalent, and corresponding workload difference models must be based on this equivalence. This approach is suitable if the workloads are characterized by frequency distributions of machine instructions and/or addressing modes, or various Kiviat graphs.

In the traditional "lowest bid" computer acquisition process the purpose of benchmarking is to show that the processing time (t) for performing a selected set of data processing functions is less than a given threshold value t_{max} . Consequently, all implementations of selected functions satisfying $t < t_{max}$ are equivalent, and this equivalence should affect the workload difference model. This kind of benchmarking is usually based on common system functions (edit, compile, link/load,

debug, profile, etc.), or frequent business applications (sort, bank transactions, word processing, payroll, inventory control, etc.).

These examples support the conclusion that workload difference models $d(B_j, B_k)$ must be derived from the analysis of the purpose of benchmarking. If the purpose of benchmarking is to get an insight into the internal organization and operation of a computer system, then workload characterization will be based on detailed measurements of the use of all individual computer resources for a set of workloads. This approach will be referred to as the *white-box* approach to workload characterization. On the other hand, if the purpose of benchmarking is to get global performance indicators (such as throughput or response time) without any interest in the internals of computer organization and operation, then the corresponding approach to workload characterization will be referred to as the *black-box* approach.

12.3. A WHITE-BOX WORKLOAD DIFFERENCE MODEL

Let N be the number of computer resources used by benchmark programs B_j and B_k , and let U_{ji} and U_{ki} denote utilizations, i.e. the fractions of total time the i^{th} resource is used by programs B_j and B_k respectively. Thus, $0 \leq U_{ji} \leq 1$, $0 \leq U_{ki} \leq 1$, for $i = 1, \dots, N$. The resources can be used either sequentially or simultaneously. For example, if a processor executes machine instructions sequentially, and U_{ji} denotes the relative frequency of the i^{th} opcode, then obviously $\sum_{i=1}^N U_{ji} = 1$. On the other hand, if U_{ji} denotes the utilization of the i^{th} resource (processor, I/O channel, disk, etc.) of a multiprogrammed machine, then we can have simultaneous activity of resources yielding $\sum_{i=1}^N U_{ji} \leq N$. Following is a model of difference between B_j and B_k that is suitable in all cases:

$$d(B_j, B_k) = \left(\frac{\sum_{i=1}^N |U_{ji} - U_{ki}|}{\sum_{i=1}^N (U_{ji} + U_{ki})} \right)^\alpha, \quad \alpha > 0.$$

The parameter α is used to adjust nonlinear features of the difference model. The values $\alpha < 1$ can be used to emphasize the importance of small differences, and the values $\alpha > 1$ to emphasize the values of

larger differences. The nominal case is $\alpha = 1$. Since

$$\sum_{i=1}^N |U_{ji} - U_{ki}| \leq \sum_{i=1}^N U_{ji} + \sum_{i=1}^N U_{ki} ,$$

it follows that $0 \leq d(B_j, B_k) \leq 1$, as expected according to the concept of limited difference.

The limit value $d_{max} = 1$ can occur if and only if B_j and B_k use two mutually disjoint subsets of resources. In the case of monoresource benchmarks¹³ (workloads that predominantly use a single resource, e.g. processor, or disk) suppose that benchmark B_j predominantly uses the j^{th} resource. Then, the maximum difference is obtained in an ideal case where

$$\begin{aligned} U_{ji} &= 1, & j = i , \\ &= 0, & j \neq i , \quad (i = 1, \dots, N, \quad j = 1, \dots, N) \end{aligned}$$

yielding $d(B_j, B_k) = 1$. This ideal situation is rather unlikely in practice, particularly in cases where U_{j1}, \dots, U_{jN} denote the relative frequencies of machine instructions. Pure processor workloads are usually characterized as computational (integer-intensive, or floating point intensive), combinatorial (using predominantly tests, branches, and jumps), I/O and data transfer intensive, or a combination of these basic types. However, it is almost impossible to find workloads which use ideally mutually disjoint subsets of machine instructions so that the difference $d(B_j, B_k) = (0.5 \sum_{i=1}^N |U_{ji} - U_{ki}|)^\alpha$ is close to 1. The white-box workload difference model based on frequencies of machine instructions usually yields small values of $d(B_j, B_k)$, while larger differences occur very infrequently. This can be modified using $\alpha < 1$.

12.4. BLACK-BOX WORKLOAD DIFFERENCE MODELS

The most frequent use of benchmark suites is the measurement of execution times and throughputs for n benchmark programs executed on m different computer systems. Let $t[i, j]$ be the execution time of program B_j using the system S_i . The results of measurements are organized as a table of selected performance indicators $z[i, j]$, shown in Table 12.1. The performance indicators in this table are usually defined in one of the following ways:

Table 12.1. A typical organization of the performance indicator table

| Performance Indicators | | Benchmark Suite Members | | | | | | |
|-----------------------------|----------|-------------------------|-----|-----------|-----|-----------|-----|-----------|
| | | B_1 | ... | B_j | ... | B_k | ... | B_n |
| Com- petitive Systems | S_1 | $z[1, 1]$ | ... | $z[1, j]$ | ... | $z[1, k]$ | ... | $z[1, n]$ |
| | \vdots | \vdots | | \vdots | | \vdots | | \vdots |
| | S_i | $z[i, 1]$ | ... | $z[i, j]$ | ... | $z[i, k]$ | ... | $z[i, n]$ |
| | \vdots | \vdots | | \vdots | | \vdots | | \vdots |
| | S_m | $z[m, 1]$ | ... | $z[m, j]$ | ... | $z[m, k]$ | ... | $z[m, n]$ |

execution (or response) time: $z[i, j] = t[i, j]$,
 throughput: $z[i, j] = x[i, j] = 1/t[i, j]$,
 relative throughput: $z[i, j] = r[i, j] = t[0, j]/t[i, j]$.

The time $t[0, j]$ corresponds to a reference system S_0 (e.g. in the case of SPEC92 and SPEC95 benchmarks, the reference systems are VAX 11/780 and SUN SPARCstation 10/40, respectively).

The measured performance indicators can have very different ranges of values. Hence, the first step in each analysis is to perform the normalization of original data. Let us first consider the case where $z[i, j] = t[i, j]$. We assume that the nature of workload does not change for loop-structured equivalent workloads. Consequently, for each benchmark program B_j the measured elapsed times in the j^{th} column of the $t[i, j]$ table can be multiplied by any positive constant. The most useful case of such a transformation is the following:

$$T[i, j] = \frac{t[i, j]}{\max_{1 \leq i \leq m} t[i, j]} , \quad 0 < T[i, j] \leq 1 , \quad 1 \leq i \leq m .$$

This form of transformation is also valid for throughput tables:

$$X[i, j] = \frac{x[i, j]}{\max_{1 \leq i \leq m} x[i, j]} = \frac{\min_{1 \leq i \leq m} t[i, j]}{t[i, j]} , \quad 0 < X[i, j] \leq 1 .$$

In the case of the relative throughput tables the division of columns by the maximum value is also possible, but the result of such a normalization is the normalized throughput table (i.e. the normalization eliminates the effect of the reference system S_0):

$$R[i, j] = \frac{r[i, j]}{\max_{1 \leq i \leq m} r[i, j]} = \frac{\min_{1 \leq i \leq m} t[i, j]}{t[i, j]} = X[i, j] .$$

Our next step is to organize the model of difference between benchmark workloads. This difference can be expressed as a distance between column vectors in Table 12.1. The main concept of black-box models is that the proximity of workloads causes the stochastic dependence of column vectors¹⁴. Accordingly, the stochastic independence corresponds to the maximum distance. Our first model is called *the uniform random number difference*. It is based on the idea that the maximum difference between benchmark programs B_j and B_k is obtained when their normalized elapsed times $T[1..m, j]$ and $T[1..m, k]$ behave as two sequences of independent standard uniform random numbers. This approach yields the following workload difference model:

$$d_{jk} = \frac{3}{m} \sum_{i=1}^m |T[i, j] - T[i, k]|, \quad 0 < T[i, j], T[i, k] \leq 1, \quad 1 \leq i \leq m.$$

Indeed, if $T[1..m, j]$ and $T[1..m, k]$ are sequences of standard uniform random numbers, then

$$\lim_{m \rightarrow +\infty} \frac{1}{m} \sum_{i=1}^m |T[i, j] - T[i, k]| = \int_0^1 dy \int_0^1 |y - x| dx = \frac{1}{3}$$

and therefore $\lim_{m \rightarrow +\infty} d_{jk} = 1 = d_{max}$. This value of d_{max} should be interpreted as a “soft limit” in the sense that d_{max} denotes the maximum difference that can occur in “regular cases” (i.e. under normal circumstances). In accidental cases, however, the difference between programs can be greater than d_{max} . These cases would occur if the columns in Table 12.1 were negatively correlated. In all normal cases the columns are positively correlated because faster machines should simultaneously reduce the run times of all benchmark programs. Thus, independent uncorrelated columns represent the limit case, and yield the maximum regular difference d_{max} . The differences greater than d_{max} are unlikely but not impossible: from $0 < T[i, j] \leq 1$ and $0 \leq |T[i, j] - T[i, k]| < 1$, it follows that $0 < \frac{3}{m} \sum_{i=1}^m |T[i, j] - T[i, k]| < 3 = d_{MAX}$. The differences greater than d_{MAX} are not possible. So, d_{MAX} denotes a “hard limit” of workload difference: $d(B_j, B_k) < d_{MAX}$ in all cases, both regular and accidental.

This and other black-box models yield an expanded (stochastic) version of the concept of limited difference, whose general form is

$$\begin{aligned} 0 \leq d(B_j, B_k) \leq d_{max} \leq d_{MAX} < +\infty, & \quad \text{in regular cases} \\ |d(B_j, B_k) - d_{max}| \leq \varepsilon < 0.05d_{max} & \quad , \quad \text{in regular limit cases of} \end{aligned}$$

$$d_{max} + \varepsilon < d(B_j, B_k) \leq d_{MAX} < +\infty, \begin{array}{l} \text{maximum difference} \\ \text{in accidental cases} \\ \text{of overflow} \end{array}$$

Since the maximum regular difference d_{max} is related to the stochastic independence of the columns of performance indicators, and the size of columns is limited by the number of available systems, it follows that d_{max} is computed with inevitable small random fluctuations whose range is denoted ε (normally $\varepsilon < 0.05d_{max}$). For some difference models the concept of limited difference yields a unique limit value, i.e. $d_{max} = d_{MAX}$, as for the white-box model. For other models we have two limit values (soft and hard): $d_{max} < d_{MAX}$, as for black-box models. The soft limit d_{max} can be considered a regular limit value of the maximum difference between workloads, and we assume $d_{max} = 1$. In unlikely cases where $d(B_j, B_k) > d_{max} + \varepsilon$, an additional analysis can be performed to determine the cause of overflow and to suggest an appropriate corrective action.

The black-box benchmark difference models can also be organized using various versions of the coefficient of correlation. The *correlation difference model* is the following semimetric:

$$d_{jk} = 1 - r_{jk}^\alpha, \quad \alpha > 0, \quad (r_{jk} \geq 0).$$

Here r_{jk} denotes the coefficient of correlation between measured performance indicators of benchmarks B_j and B_k , and α is an adjustable parameter whose role is similar to the case of white-box difference. The nominal value is $\alpha = 1$ and we may use $\alpha \neq 1$ only if $r_{jk} \geq 0$. The similarity between benchmarks is defined as $s_{jk} = r_{jk}^\alpha$. The fundamental advantage of this approach is that it is invariant with respect to linear transformations. Accordingly, it automatically includes the normalization of data.

In most cases r_{jk} can be interpreted as the coefficient of linear correlation ($r(X, Y) = (\overline{XY} - \overline{X} \cdot \overline{Y}) / \sigma_X \sigma_Y$). In cases where the nonparametric approach is more suitable, r_{jk} can be interpreted as Spearman's rank correlation coefficient:

$$r_s(B_j, B_k) = 1 - \frac{6}{m(m^2 - 1)} \sum_{i=1}^m (L_{ji} - L_{ki})^2, \quad -1 \leq r_s(B_j, B_k) \leq +1.$$

where L_{ji} and L_{ki} denote the rank of the i^{th} computer obtained using B_j and B_k respectively. The ranks are positive integers: the rank of the fastest computer is 1, and the rank of the slowest is m .

Both linear correlation and rank correlation reflect the similarity between B_j and B_k : if $0.5 < r(B_j, B_k) \leq 1$ then B_j and B_k are (very) similar, and if $r(B_j, B_k)$ is close to 0 then there is no similarity between B_j and B_k . Let us note that it is possible to have small negative values of $r(B_j, B_k)$ as a consequence of random fluctuations in cases where B_j and B_k are sufficiently different and r is close to 0. However, it is highly unlikely to have large negative values of $r(B_j, B_k)$ because computers (as well as all other engineering products) are designed to have positively correlated performances of individual resources. For example, computer systems with high processor speed typically have large memory and fast disks; it would be counterproductive to combine an increase in processor performance with a decrease in memory capacity and/or file I/O performance. Therefore, it is very unlikely that B_j could produce a ranking that is opposite to the ranking obtained from B_k . In all regular cases the correlation model of difference will generate values in the interval $0 \leq d(B_j, B_k) \leq 1$, yielding the soft limit $d_{max} = 1$. Consequently, the differences in the interval $1 < d(B_j, B_k) \leq 2$ are unlikely but not impossible, yielding for any value of α the hard limit $d_{MAX} = 2$. The soft limit $d_{max} = 1$ will rarely be exceeded, and the hard limit $d_{MAX} = 2$ plays the role of a theoretical upper bound.

From the standpoint of benchmark comparison we can frequently consider B_j and B_k identical if they produce identical rankings of a representative set of m computers. Therefore, the difference metrics can also be based on the difference between rankings. Let R_j and R_k be rankings produced by B_j and B_k . Following is the ranking distance proposed by Kemeny and Snell¹⁵:

$$\begin{aligned}
 J_{pq} &= 1 && \text{if } S_p \succ S_q \text{ for } B_j \\
 &= -1 && \text{if } S_p \prec S_q \text{ for } B_j \\
 &= 0 && \text{if } S_p \sim S_q \text{ for } B_j \\
 K_{pq} &= 1 && \text{if } S_p \succ S_q \text{ for } B_k \\
 &= -1 && \text{if } S_p \prec S_q \text{ for } B_k \\
 &= 0 && \text{if } S_p \sim S_q \text{ for } B_k \\
 d(R_j, R_k) &= \frac{1}{2} \sum_{p=1}^m \sum_{q=1}^m |J_{pq} - K_{pq}|
 \end{aligned}$$

The elements of the ordering matrices J and K have the property $J_{pq} = -J_{qp}$ and $K_{pq} = -K_{qp}$. A complete ranking is one containing no ties; e.g. for $m=2$ such a ranking is $R = (S_1 \succ S_2)$. An opposite ranking

is symbolically denoted $-R = (S_1 \prec S_2)$. The “O-ranking” assumes all ties: $O = (S_1 \sim S_2)$. So, for $m = 2$, $d(R, O) = d(-R, O) = 1$ and $d(R, -R) = 2$. Generally, $d(R, O) = d(-R, O) = m(m - 1)/2$ and $d(R, -R) = m(m - 1)$. Using this type of metric, and the concept of limited difference, we can derive the following *ranking difference model*:

$$d(B_j, B_k) = \left(\frac{1}{m(m - 1)} \sum_{p=1}^m \sum_{q=1}^m |J_{pq} - K_{pq}| \right)^\alpha, \quad \alpha > 0.$$

For this model $d(R, O) = d(-R, O) = 1$ and $d(R, -R) = 2^\alpha$. So, the soft limit is $d_{max} = 1$ and the hard limit is $d_{MAX} = 2^\alpha$.

12.5. CRITERIA FOR BENCHMARK SUITE EVALUATION AND DESIGN

Evaluation and design of benchmark suites is based on a specific set of benchmark suite performance criteria. Two main groups of criteria can be identified: qualitative, and quantitative criteria. We propose the use of ten basic criteria. Qualitative criteria specify global features of benchmark suites and include the following five compliance requirements:

- compliance with the goal of benchmarking
- application area compliance
- workload model compliance
- hardware platform compliance
- software environment compliance

Quantitative criteria are related to the distribution of component benchmark programs in the program space, and include the following five characteristics of benchmark suites:

- size
- completeness
- density
- granularity
- redundancy

Following is a brief description of these ten criteria.

Each benchmark suite supports a specific *goal of benchmarking*. Typical goals are: performance evaluation of a given system, performance comparison of several competitive systems, standardized comparison and ranking of all commercially available systems in a given class, selection of the best system according to specific user requirements, resource consumption measurement and analysis, and performance tuning. A clear and complete specification of the goal of benchmarking is the initial step in all benchmarking efforts. The next step is a convincing justification that a given benchmark suite supports the specified goal. Consequently, the first qualitative criterion benchmark suites must satisfy is the support of a clearly defined goal of benchmarking.

The criterion of *application area compliance* specifies the requirement for a desired application type of benchmark workloads. The benchmark suite members are always assumed to be good representatives of a desired application area. Application areas are sometimes related to the activity of typical users (e.g. scientific, business, educational, and home applications). Another approach is to define application areas according to characteristics of computer workload (e.g. numerical, nonnumerical (combinatorial), seminumerical, graphic, database, systems programming, and networking applications). Each benchmarking effort should be related to a given application area, and so should the members of benchmark suites.

The design of benchmark suites regularly includes the selection of the most appropriate *workload model*. The workload models can be physical, virtual, and functional¹². The selection of workload model must be justified by specific requirements of a given benchmarking problem. A desired workload model can also be specified during the evaluation of benchmark suites. Once the desired workload model is selected, the benchmark suite must satisfy the workload model compliance criterion.

Functional and virtual workload models are simpler and more frequent than physical models. A recent example of a functional model is a suite of typical business applications for personal computers running under Microsoft Windows, developed by Business Application Performance Corporation⁶ (BAPCo). This suite consists of ten popular software products performing typical business functions in the areas of word processing, spreadsheets, database, desktop graphics, desktop presen-

tation, and desktop publishing. As an example of benchmark suites based on a virtual model we can use SPEC benchmarks^{1,6}. SPEC92 and SPEC95 are focused on integer and floating point performance as two fundamental components of processor performance. These components can be interpreted as “logical resources” based on performance of several physical resources (processor(s), cache(s), memory, and compilers).

Different functional or virtual workloads can yield very similar usage of hardware and software resources and in such cases their functional/virtual difference becomes insignificant. In such cases we need physical workload models, because they are quantitative and provide measures of redundancy between component programs in a benchmark suite. Generally, there is no justification for using redundant workloads because the cost of benchmarking increases without a corresponding increase in benefits. Therefore, the physical level is fundamental for workload modeling.

The *hardware platform compliance* criterion specifies a target hardware category, and a computer architecture for which a benchmark suite is designed. The most frequent hardware platforms are: personal computers, workstations, mainframes, network servers, local/wide area networks, vector machines, parallel machines, database machines, and communication systems. For each platform it is necessary to identify the main set of resources and to prove that the members of the benchmark suite sufficiently use all identified resources.

The criterion of *software environment compliance* is analogous to the hardware platform compliance criterion. It specifies the desired operating system, windowed environments, programming languages, database systems, communication software, and program development tools that benchmark workloads must use. This criterion identifies all relevant software resources and takes care that they are properly used. In many cases, benchmarking can be limited to the performance analysis of software products (e.g. compilers for a specific language, database systems, operating system overhead, etc.).

Qualitative criteria help to specify a general framework and guidelines for benchmarking efforts. However, all more specific requirements for a set of benchmark workloads must be defined using quantitative criteria. The main advantage of quantitative criteria is the possibility to easily provide a quantitative proof that a benchmark suite satisfies some specific requirements.

The fundamental quantitative criterion is the *size of benchmark suite*, D . It is defined as the diameter of the smallest circumscribed hypersphere containing all benchmark workloads, B_1, \dots, B_n , $n > 1$. The central point of this hypersphere is a hypothetical benchmark B_0 whose coordinates will be denoted z_{10}, \dots, z_{m0} . We can compute z_{10}, \dots, z_{m0} from the condition that the difference between B_0 and the furthestmost of n benchmarks in the suite has the minimum value. Consequently, the size of benchmark suite is

$$D = 2 \min_{z_{10}, \dots, z_{m0}} \max_{1 \leq k \leq n} d_{0k}, \quad d_{0k} = d(B_0, B_k), \quad k = 1, \dots, n.$$

In the case of n benchmark programs, the differences between programs form a symmetric square matrix $[d_{jk}]_{n,n}$, where $d_{jk} = d_{kj}$, and $d_{jj} = 0$. The same properties hold for the similarity matrix $[s_{jk}]_{n,n}$, where $s_{jk} = d_{max} - d_{jk}$. For any group of benchmarks the *central* benchmark, B_c , and the *most peripheral* benchmark, B_p , can be determined from

$$d_{cp} = \min_{1 \leq j \leq n} \max_{1 \leq k \leq n} d_{jk}, \quad s_{cp} = \max_{1 \leq j \leq n} \min_{1 \leq k \leq n} s_{jk}.$$

The fundamental property of central benchmark B_c is that its furthestmost neighbor, B_p , is located closer than in the case of other benchmarks in the group. So, B_c has the maximum similarity with other benchmarks in the group, and it is usually close to B_0 . The central position justifies the use of B_c as the *best representative* of the group. Let us note that the above definition yields B_c and B_p for *any* distribution of workloads, but in some cases the central region of a group of benchmarks can be empty. For example, if we have only two benchmarks, one of them must play the role of B_c , and it is not centrally located. The situation is similar for three benchmarks forming an equilateral triangle. However, real benchmark suites always include centrally located benchmarks which can be used as approximations of B_0 . For example, if B_c is centrally located then the difference d_{cp} can be used to compute an approximation of the size of benchmark suite: $D \approx 2d_{cp}$. Of course, this is not a good approximation if B_c is not centrally located.

The maximum value of D corresponds to the simplex distribution of benchmark workloads, where $d(B_j, B_k) = d_{max}$, $j = 1, \dots, n$, $k = 1, \dots, n$, $j \neq k$, $n > 1$. Geometrically, such a set of n points in the program space is an equilateral simplex (and no benchmark is centrally located). Since the distance between individual benchmarks has the maximum value d_{max} , the diameter of the circumscribed hypersphere

also has the maximum value. The maximum value of D obtained for the simplex distribution depends on n and will be denoted $D_{max}(n)$. This value is the maximum size of a benchmark suite in the case of n benchmarks. Consequently, we can define the following program space *coverage* (or *utilization*) indicator:

$$U(n) = D/D_{max}(n) , \quad 0 < U(n) \leq 1 , \quad n > 1 .$$

The *completeness* of benchmark suites can be evaluated using the coverage indicator $U(n)$. Small values of $U(n)$ indicate incomplete benchmark suites whose members can be excessively redundant, and the values close to 1 indicate benchmark suites that include a wide spectrum of workloads. So, $U \approx 1$ can be considered a necessary condition for a good quality of benchmark suite. Another necessary condition is, of course, a proper distribution of workloads within the program space.

Generally, by increasing the number of benchmark workloads we create an opportunity to cover a spectrum of workload features; this supports increasing the value of n . On the other hand, the cost of benchmarking is proportional to n and this is a reason for reducing the value of n . Hence, the resulting value of n is a compromise between two opposite criteria. To help in selecting the value of n it is useful to define the *density* of a benchmark suite as the number of benchmark workloads per unit of covered program space. Since the coverage is expressed using $U(n)$, the density indicator can be defined as follows:

$$H(n) = n/U(n) \approx (3n - 2)/2D .$$

The concept of granularity is closely related to the concept of density of benchmark suites. We define the benchmark suite *granularity*, G , as the ratio of the number of benchmark programs in the suite, n , and the minimum necessary number of such programs, N , i.e. $G = n/N$. The most reasonable value of N is the number of different computer resources that are used by the benchmark suite. The most frequent resources are: processors, memory, cache memories, disk channels, disks, and software resources. Generally, a computer resource is defined as any major hardware/software component or feature which contributes to the performance of analyzed computer systems.

An obvious reason for designing and using benchmark suites is that a spectrum of performance features of complex hardware/software systems cannot be properly and sufficiently evaluated using a single benchmark workload. If the computer performance is defined as an array of

N performance indicators of individual hardware/software resources, then N benchmark workloads in a benchmark suite play the role of N equations necessary to compute the performance indicators.

In real situations which involve complex and nonlinear phenomena caused by multi-level caching, advanced memory management techniques, parallelism, networking, advanced compiler techniques, etc., it is frequently difficult to clearly identify all relevant hardware and software resources. For example, a control program that is executed by an I/O processor that controls an array of disks can be an important (but not easily visible) resource which in the case of a constrained degree of multiprogramming can limit the global throughput of the file I/O subsystem. Unfortunately, a number of nontrivial experiments is needed to identify the existence and the actual role of such a resource. Consequently, N usually denotes the number of “obvious resources,” and the actual number of resources can be greater than N . In such cases the number of benchmarks in the benchmark suite should also be greater than N . Thus, the desirable level of granularity is $G > 1$. This yields a simple guideline for selecting the number of benchmarks in the suite: $n > N$. In the context of the white-box approach N could be interpreted as the dimensionality of the program space. The values of H and G should be as big as possible. However, their maximum values are obviously limited by financial constraints.

Two (or more) benchmark programs are considered *redundant* if their difference is relatively small. Assuming $d_{max} = 100\%$, differences that are less than 15% can usually be considered small differences. In many practical cases it is easy to encounter differences that are less than 5%. In such cases it is difficult to justify why so similar workloads must simultaneously be used because their contribution to the cost of benchmarking is much larger than their contribution to the comparison of competitive systems. The redundancy of benchmark workloads can be visualized using cluster analysis and by presenting benchmark suites in the form of dendrograms¹⁶.

12.6. SIMPLEX BENCHMARK SUITES

A simplex benchmark suite (SBS) is defined as a suite where each pair of benchmarks has the maximum difference d_{max} . This concept is related to the concept of *monoresource benchmarks*¹³, where the goal of

each component benchmark is to provide maximum utilization of a selected computer resource, and the minimum possible usage of all other resources. Therefore, the simplex suite contains the minimum number of component benchmarks, $n = N$. The main SBS features are:

- a maximum difference between component benchmarks,
- a uniform coverage of the program space,
- the absence of a central benchmark,
- a minimum redundancy between component benchmarks (no benchmark can be removed),
- no benchmark can be added to SBS without introducing redundancy,
- for each number of component benchmarks, SBS has the maximum size and a unit granularity,
- the component benchmarks form an equilateral simplex in the program space.

The goal of SBS is to include component benchmarks which are mutually exclusive and collectively exhaustive. In other words, they should completely cover a spectrum of desired performance characteristics, and should not be redundant. Such programs are *necessary*, and theoretically, *sufficient* for a given performance evaluation task. They are necessary because removing any of programs would result in the total absence of effects caused by some relevant computer resources. They can be sufficient because they contain minimum but complete information about all identified resources. In practice, however, it is not possible to completely isolate the activity of a single resource (e.g. processor activity cannot be eliminated during the measurements of other resources) and it is difficult to prove that all relevant resources are included in SBS.

The concept of SBS is useful both as a theoretical model and as a guideline for practical implementations. The SBS model should be used as a reference point in the design of benchmark suites. Real benchmark suites can be designed as “expanded SBS’s” which include SBS component benchmarks plus additional redundant benchmarks which cover the central region of the program space, and provide activity of less important (and initially omitted) computer resources.

Let us now estimate the maximum size of a benchmark suite containing n workloads. The mean absolute difference between a uniformly distributed random variable and the arithmetic mean of n such variables, proved by Milan Merkle¹⁷, is:

$$E|Z_i - (Z_1 + \dots + Z_n)/n| = \frac{1}{4} - \frac{1}{6n}, \quad Z_i \sim \text{Uniform}(0,1), \quad i = 1, \dots, n.$$

In the case of the uniform random number difference model, uniformly distributed execution times, and approximating B_0 by the arithmetic mean of all benchmarks, we can use the Merkle formula to estimate the maximum size of a benchmark suite and the program space coverage indicator:

$$D_{max}(n) \approx 2 \lim_{m \rightarrow +\infty} \frac{3}{m} \sum_{i=1}^m |T[i, j] - \frac{1}{n} \sum_{k=1}^n T[i, k]| = \frac{3}{2} - \frac{1}{n},$$

$$U(n) \approx \frac{2nD}{3n-2} \approx \frac{4nd_{cp}}{3n-2} = \frac{\min_{1 \leq j \leq n} \max_{1 \leq k \leq n} d_{jk}}{0.75 - 1/2n}, \quad 2 \leq n \leq +\infty.$$

This result can be compared to the case of Euclidean space where for $n = 2$ we start with two points having the distance $D_{max}(2) = d_{max}$. For $n = 3$ the three points form an equilateral triangle inscribed in a circle whose diameter is $D_{max}(3) = 1.15d_{max}$. For $n = 4$ the four points form an equilateral tetrahedron inscribed in a sphere having the diameter $D_{max}(4) = 1.22d_{max}$. Eventually, $D_{max}(+\infty) = \sqrt{2} \cdot d_{max}$. This analysis also shows that in all cases $D_{max}(n)$ is a strictly increasing function of n : $(\forall i > 1) D_{max}(i) < D_{max}(i+1)$, but the range is limited: $D_{max}(+\infty) - D_{max}(2) \approx 0.5d_{max}$.

12.7. EVALUATION OF BENCHMARK SUITES

The goal of benchmarking using benchmark suites is to evaluate and compare a given set of hardware/software systems. In the case of standard industrial benchmarking the comparison includes all commercially available computer systems in a given performance range. To evaluate an existing benchmark suite means to analyze the suite using the set of ten basic criteria introduced in the preceding Section.

In order to exemplify the use of basic quantitative criteria for evaluation of benchmark suites let us consider the case of the first SPEC suite

(SPEC89¹). This example is suitable for analysis because it includes only 10 processor bound benchmarks. These include 4 integer-intensive C programs (*gcc*, *eqntott*, *espresso*, and *li*), and 6 floating point intensive FORTRAN programs (*spice2g6*, *doduc*, *nasa7*, *tomcatv*, *fppppp*, and *matrix300*).

The primary goal of SPEC benchmarks (the first generation, SPEC89, the second generation, SPEC92, and the third generation, SPEC95) is to evaluate integer performance and floating point performance of a processor (or processors) as well as hierarchically organized memory, and to use the resulting compound performance indicators for standard comparison and ranking of commercially available computers with the emphasis on ranking workstations and servers. This type of goal justifies the criterion that two benchmark programs can be considered different if and only if they yield different ranking of evaluated systems. If two benchmark programs produce identical rankings of a set of computers, then, from the standpoint of standardized comparison and ranking, such benchmarks may be considered identical, and their difference (distance) must be zero. The black box difference models are most suitable for the evaluation of benchmark suites of this type.

The goal of the analyzed benchmark suite is restricted to the evaluation of performance of the central processing unit. This goal also restricts the number of performance related computer resources and the corresponding number of necessary component benchmarks. The estimated number of performance related resources is $N = 7$ (the central processor, floating point coprocessor, instruction and data caches, main memory, C compiler and FORTRAN compiler). This yields the acceptable granularity $G = 1.43$, and shows that the numbers of SPEC92 benchmarks (20) and SPEC95 benchmarks (18) can easily be reduced.

In order to generate relative throughput indicators defined in Table 12.1 we used measurements performed by manufacturers of 49 computer systems (the official reports are published in the SPEC Newsletter). Hence, $n = 10$, $m = 49$, and each benchmark is characterized by a 49-component column-vector. These vectors (columns in Table 12.1) are used for computing differences between individual benchmarks. After the normalization of all columns we applied the uniform random number difference model to compute the matrix of differences between benchmarks presented in Table 12.2. The two closest benchmarks are *nasa7* and *tomcatv*: they merely differ by 6.72%. The maximum individual difference is between *matrix300* and *li* (84%). The central

Table 12.2. An analysis of differences between workloads

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|--|-------|-----|-----|-----|-----|-----|-----|--------------|--------|-------|
| 1 gcc | .00 | .18 | .25 | .24 | .55 | .21 | .12 | .68 | .48 | .54 |
| 2 espresso | .18 | .00 | .32 | .35 | .66 | .13 | .15 | .78 | .64 | .64 |
| 3 spice2g6 | .25 | .32 | .00 | .11 | .35 | .38 | .23 | .45 | .33 | .32 |
| 4 doduc | .24 | .35 | .11 | .00 | .32 | .40 | .24 | .44 | .29 | .30 |
| 5 nasa7 | .55 | .66 | .35 | .32 | .00 | .72 | .55 | .15 | .10 | .07 |
| 6 li | .21 | .13 | .38 | .40 | .72 | .00 | .19 | .84 | .68 | .70 |
| 7 eqntott | .12 | .15 | .23 | .24 | .55 | .19 | .00 | .67 | .53 | .53 |
| 8 matrix300 | .68 | .78 | .45 | .44 | .15 | .84 | .67 | .00 | .20 | .14 |
| 9 fpppp | .48 | .64 | .33 | .29 | .10 | .68 | .53 | .20 | .00 | .10 |
| 10 tomcatv | .54 | .64 | .32 | .30 | .07 | .70 | .53 | .14 | .10 | .00 |
| Minimum difference between programs | | | | | | | | Dmin = | .07 | |
| Average difference between programs | | | | | | | | Dave = | .38 | |
| Maximum difference between programs | | | | | | | | Dmax = | .84 | |
| Central program having the min max difference = | doduc | | | | | | | | | (# 4) |
| Radius of the group (max difference from the CO) | | | | | | | | R = | .44 | |
| Central object location indicator (<=100%) | | | | | | | | 100Dmax/2R = | 95.00% | |

SPEC89 benchmark is $B_c = doduc$. Its furthestmost neighbor $B_p = matrix300$ differs only by 44%, and the size of SPEC89 can be approximated by $2d_{cp} = 0.88$. A more precise result is $D_{SPEC89} = 0.8625$. The maximum size of a 10-benchmark suite is $D_{max}(10) = 3/2 - 1/10 = 1.4$. Thus, the program space coverage (utilization) for SPEC89 is rather modest: $U_{SPEC89} = 0.8625/1.4 = 0.616$ (i.e. less than 62%). The corresponding density is $H_{SPEC89} = 16.2$ workloads per unit of covered program space.

More detailed results of the redundancy and coverage analysis are presented in Fig. 12.1. They include a dendrogram and a covergram generated using a hierarchical clustering technique¹⁶. The dendrogram is a binary tree whose nodes denote clusters of objects. The nodes are merging points of two clusters (subtrees). The intercluster difference is used to place the node on the workload difference scale (i.e. the distance between the merging point and the left margin equals the difference between the two merging clusters). Each merging point is denoted by the name of the best representative of the cluster.

The most centrally located workload is *doduc* (another centrally located benchmark is *spice2g6*). There are four pairs of closely related benchmarks: $\{gcc, eqntott\}$, $\{espresso, li\}$, $\{spice2g6, doduc\}$, and $\{nasa7, tomcatv\}$. They differ for less than 13%. This is a high redundancy and indicates both the possibility of reducing the num-

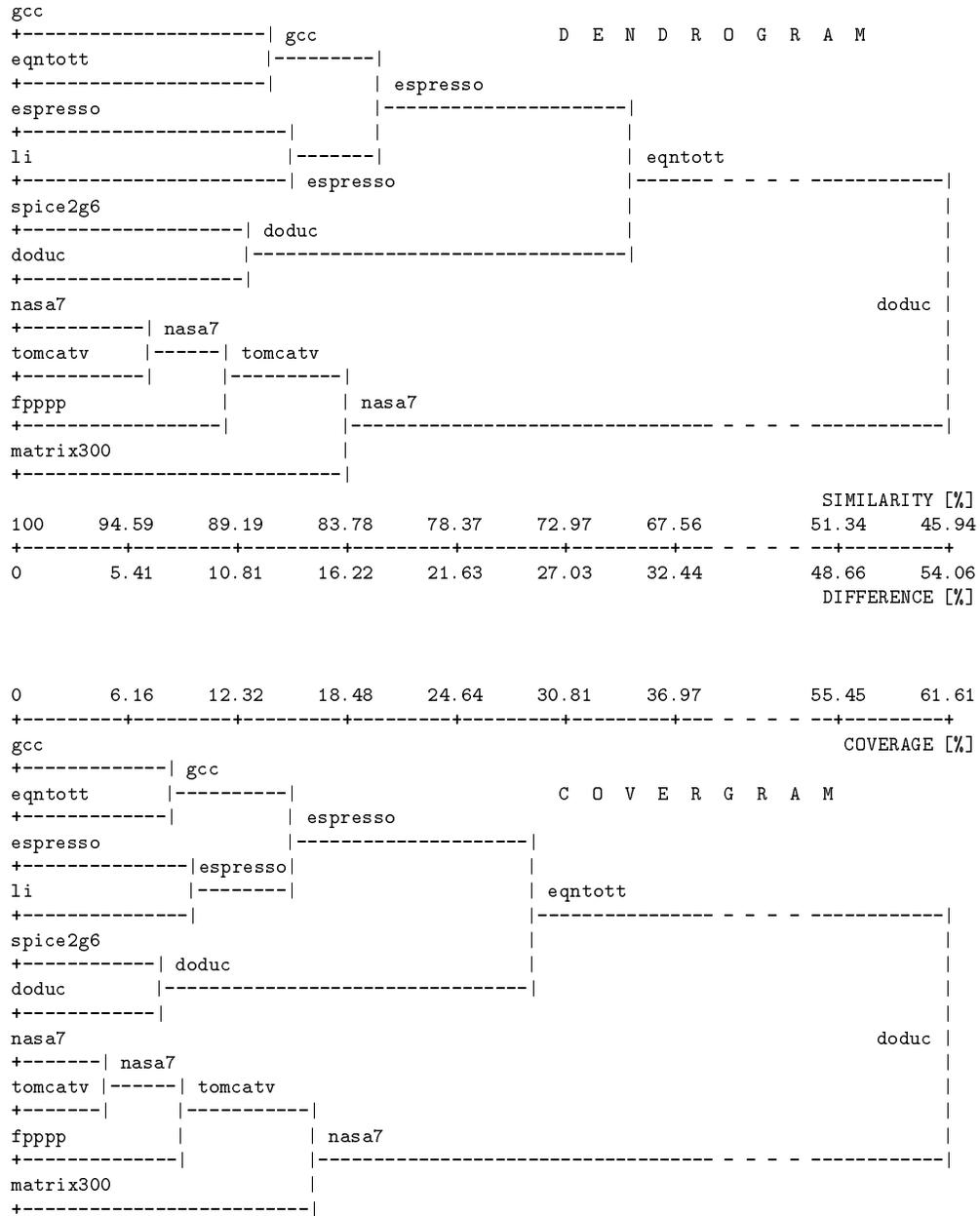


Figure 12.1. SPEC89 redundancy and coverage analysis

ber of benchmarks (without reducing the quality and completeness of global performance indicators) and the need to replace some component benchmarks by less redundant workloads. Fig. 12.1 also shows a relatively high redundancy of the group $\{nasa7, tomcatv, fppppp\}$, and the possibility of replacing it by the best representative, *tomcatv*. Integer workloads form a compact cluster whose best representative is *espresso*. Floating point benchmarks form two groups: the redundant pair $\{spice2g6, doduc\}$, and the remaining group $\{nasa7, tomcatv, fppppp, matrix300\}$ which is located very far (54.06%) from other benchmarks. The presented dendrogram also suggests the attractive possibility of reducing the whole suite to three best representatives: *espresso* (for integer part), and *doduc* plus *nasa7* (for the floating point part). Such a decision would yield insufficient granularity, but the cost of benchmarking would be reduced by 70%.

As a new tool for the analysis of the completeness of benchmark suites, we propose the use of *covergrams*. A covergram is a dendrogram whose cluster merging points are placed using the program space coverage scale. Using covergrams we can directly read the percent coverage of the program space (U) for each individual cluster. In order to be reasonably complete, a benchmark suite should cover at least 50% of the program space. The covergram in Fig. 12.1 shows that 15.7% of the program space is covered by the group $\{gcc, eqntott, espresso, li\}$, 8.1% by $\{spice2g6, doduc\}$, and 16.5% by $\{nasa7, tomcatv, fppppp, matrix300\}$. The total coverage of these three groups is 61.61%. This coverage is low, and it is achieved by three clusters which contain closely located benchmarks. Thus, this example shows that the analyzed suite can be substantially improved in two directions: (1) redundant benchmarks should be removed, and (2) new benchmarks which increase the coverage should be introduced. The candidates for removal are one benchmark from each of the groups $\{gcc, eqntott\}$, $\{espresso, li\}$, and $\{spice2g6, doduc\}$, as well as two benchmarks from the group $\{nasa7, tomcatv, fppppp\}$.

Basic redundancy analysis is frequently sufficient to detect possible problems and to point at corrective actions. This is illustrated by published redundancy analysis examples¹⁴ which show a way to improve Livermore FORTRAN Kernels⁷, Lisp benchmarks⁹, and Prolog benchmarks¹⁰. One conclusion (derived from the analysis of 16 Prolog benchmarks, which all have a maximum intercluster difference of only 9.3%) is that it is useless to apply large benchmark suites in cases where

component benchmarks are executed by interpreters. In such cases the only executable in memory is the interpreter itself, and workloads differ only to the extent of variations resulting from executing different parts of the same interpreter. Such variations are typically insignificant (less than 10%), and the whole benchmarking effort can easily be reduced to the use of one to three best representatives of the initial benchmark suite. This result might be a useful hint for designers of Java benchmarks.

The presented evaluation examples illustrate the benefits of the proposed evaluation technique for reducing the redundancy and increasing the coverage of benchmark suites. Consistent use of ten basic evaluation criteria during the design and updates of benchmark suites can improve the majority of their features.

12.8. COVERAGE FUNCTIONS

The most frequent approach to the design of benchmark suites is based on selecting benchmark workloads from a set of existing programs which properly represent the standard activity of an analyzed class of users. A more complex approach is to use (natural or synthetic) workloads with adjustable parameters which affect the utilization of computer resources. In such cases, in addition to selecting a suitable set of workloads, it is also necessary to properly adjust their parameters. This process can always be abstracted as the problem of optimum distribution of points in the program space.

The problem of achieving some desired distribution of workloads in the program space is the fundamental problem of benchmark suite design. In other words, the individual benchmarks are expected to cover the program space in a given way. The coverage of the program space can be analyzed using the concept of *coverage functions*. We assume that each benchmark B_j , $j = 1, \dots, n$, provides a nonuniform “coverage field” of the program space within a hypersphere Θ_j , with a central point B_j and radius d_{max} . The coverage field denotes the zone of influence created by an individual benchmark. The strongest coverage field is in the central parts of Θ_j , while the coverage field at the circumference and outside of Θ_j must be zero. The total intensity of the coverage field in a given point of the program space is defined as the superposition of intensities of the coverage fields of all individual

benchmarks. For simplicity we will assume that the term *coverage* denotes the intensity of the compound coverage field in a given point of the program space.

It may be useful to visualize each benchmark as a source of light that illuminates the interior of the hypersphere Θ_j ; of course, central parts are illuminated stronger than the peripheral parts. We can also visualize a set of benchmark workloads as a set of sources of light that are nonuniformly distributed in the program space. The intensity of illumination at each point is obtained by superposition of illumination coming from all sources. The objective of benchmark suite design is to realize a given (uniform or nonuniform) distribution of “illumination” of the program space.

The analogy of the coverage of a program space and the illumination of a physical space is useful to identify the following three ways to achieve the desired distribution:

- by moving (repositioning) workloads in the program space,
- by adding and/or removing selected workloads, and
- by adjusting the “intensity” of each individual workload.

The positioning of individual benchmarks requires adjustable workload parameters, and can be realized only if we use the white-box approach; in such cases we must modify the available benchmark parameters in order to move a program in the direction of the desired destination in the program space. The remaining two techniques can be realized within the black-box approach, and they are discussed in subsequent sections.

Let us now introduce the individual coverage function of the j^{th} benchmark, $\delta_j \mapsto c_j(\delta_j)$, where δ_j denotes the distance from an arbitrary point within Θ_j to the B_j benchmark. We define $c_j(\delta_j)$ as a strictly decreasing function having the following properties:

$$c_j(\delta_j) = \begin{cases} 1, & \delta_j = 0 \\ 0, & \delta_j \geq d_{max}, \end{cases}$$

$$\frac{dc_j(\delta_j)}{d\delta_j} < 0, \quad 0 < \delta_j < d_{max}, \quad j = 1, \dots, n.$$

Let Ω be a given region of the program space. We assume that the global coverage of the program space is obtained by the linear superposition of coverages of all individual benchmarks. Thus, for each point

in the region Ω we can define the *compound coverage function*

$$C(\delta_1, \dots, \delta_n) = \sum_{j=1}^n c_j(\delta_j) ,$$

and the *normalized coverage function*

$$C_{norm}(\delta_1, \dots, \delta_n) = \frac{1}{n} \sum_{j=1}^n c_j(\delta_j) .$$

The compound coverage function shows the global intensity of the coverage field resulting from all benchmarks in a benchmark suite. If V denotes the volume of the region Ω then the average compound coverage is

$$C_{ave} = \frac{1}{V} \int_{\Omega} C(\delta_1, \dots, \delta_n) dV , \quad V = \int_{\Omega} dV .$$

The average compound coverage C_{ave} increases when the density and granularity of a benchmark suite increase, and therefore it can serve as one of the benchmark suite density indicators.

Individual coverage functions can be organized in various ways. First, we define an auxiliary threshold function that has the value 1 inside and the value 0 outside the hypersphere Θ_j :

$$I(\delta_j) = \begin{cases} 1 , & \delta_j < d_{max} \\ 0 , & \delta_j \geq d_{max} . \end{cases}$$

Now we can introduce the following three characteristic forms of the $c_j(\delta_j)$ function:

$$\begin{aligned} \Lambda(\delta_j) &= \left(1 - \frac{\delta_j}{d_{max}}\right) I(\delta_j) = \max\left(0 , 1 - \frac{\delta_j}{d_{max}}\right) , \\ \Lambda_1(\delta_j) &= \frac{1}{2} \left(1 + \cos \frac{\pi \delta_j}{d_{max}}\right) I(\delta_j) , \\ \Lambda_2(\delta_j) &= \frac{1}{2} \left(1 + \cos \frac{\pi(1 - \cos(\pi \delta_j / d_{max}))}{2}\right) I(\delta_j) . \end{aligned}$$

These formulas are valid for all values of δ_j , including the rare cases where random fluctuations might cause $\delta_j > d_{max}$. The Λ function yields a linearly decreasing intensity of the coverage field. In the case of Λ_1 and Λ_2 the decrease of the coverage field is nonlinear. Both

Λ_1 and Λ_2 have an intensified coverage in the central part of Θ_j ; this effect is particularly strong in the case of Λ_2 . Therefore, Λ , Λ_1 , and Λ_2 offer three increasing levels of coverage of the central region of Θ_j ; their common properties are $c_j(0) = 1$, $c_j(d_{max}/2) = 1/2$, and $c_j(d_{max}) = 0$.

In a general case of n benchmarks and known distances $d_{jk} = d(B_j, B_k)$ we can easily compute the value of the compound coverage function at all points of the program space where benchmarks are located (these points will be called “*benchmark points*”). At the point B_j the distance from B_k is $\delta_k = d_{jk}$ and the compound coverage is

$$C_j = C(d_{j1}, \dots, d_{jn}) = \sum_{k=1}^n c_j(d_{jk}), \quad j = 1, \dots, n.$$

In the case of the Λ function the compound coverages at benchmark points are

$$C_j = \sum_{k=1}^n \max\left(0, 1 - \frac{d_{jk}}{d_{max}}\right) = n - \frac{1}{d_{max}} \sum_{k=1}^n d_{jk}, \quad j = 1, \dots, n.$$

Assuming $d_{max} = 1$ it follows that for all benchmark points, $0 \leq d_{jk} \leq 1$, $1 - d_{jk} = s_{jk}$, and the values of compound coverages reduce to the sum of similarities:

$$C_j = \sum_{k=1}^n s_{jk}, \quad j = 1, \dots, n.$$

The distribution of values C_j , $j = 1, \dots, n$ is a readily available and easily understandable indicator of the uniformity of coverage of the program space.

12.9. OPTIMUM SUBSETS OF BENCHMARK PROGRAMS

A desired coverage of the program space can frequently be achieved using a suitable subset of the available benchmark programs. The simplest way to create appropriate subsets of benchmark programs is to use the subsets of best representatives created by cluster analysis. The best representatives are the most centrally located benchmarks in selected clusters, and accordingly they cover the program

space in an approximately uniform way. For each subset of J benchmarks ($J = n, n - 1, \dots, 1$) we can define a binary selector vector $(b_1 \dots b_n)$, $b_k \in \{0, 1\}$, $k = 1, \dots, n$ that denotes subsets of benchmark programs: if $b_k = 1$ then B_k is included in a given subset, and if $b_k = 0$ then B_k is excluded. The compound coverage vector at *all* benchmark points, $(C_1 \dots C_n)$, can now be computed as follows:

$$C_j = \sum_{k=1}^n b_k c_j(d_{jk}) , \quad j = 1, \dots, n .$$

This vector can be compared with the desired distribution of compound coverage at benchmark points. Suppose the desired distribution is uniform. We can determine the average coverage at the benchmark points $\bar{C} = \frac{1}{n} \sum_{j=1}^n C_j$, and define the average error of the achieved distribution as the coefficient of variation

$$v_c(C_1, \dots, C_n) = 100 \left(\frac{n \sum_{j=1}^n C_j^2}{\left(\sum_{j=1}^n C_j\right)^2} - 1 \right)^{\frac{1}{2}} .$$

The distributions obtained using the subsets of best representatives generated by the cluster analysis can be rather good, but they are *not* the best possible distributions. Indeed, we can find better solutions if we systematically investigate all possible subsets.

Let us again use the binary selector vector $(b_1 \dots b_n)$ denoting subsets of benchmark programs, and let us introduce the index of the subset, I , the total number of benchmarks in the subset, J , and the uniform distribution error, v_c , as follows:

$$C_j[I; J] = \sum_{k=1}^n b_k c_j(d_{jk}) , \quad j = 1, \dots, n ,$$

$$I = \sum_{k=1}^n 2^{k-1} b_k, \quad 0 \leq I \leq I_{max}, \quad I_{max} = 2^n - 1; \quad J = \sum_{k=1}^n b_k, \quad 0 \leq J \leq n,$$

$$v_c[I; J] = v_c(C_1[I; J], C_2[I; J], \dots, C_n[I; J]) .$$

For each J we can now find the optimum subset of benchmarks:

$$v_c[I^*(J); J] = \min_{1 \leq I \leq I_{max}} v_c[I; J] , \quad J = 1, \dots, n .$$

The optimum subsets that contain $1, \dots, n$ benchmarks and yield the minimum distribution error are denoted by indices $I^*(1), \dots, I^*(n)$.

In the majority of practical cases the number of benchmarks in a benchmark suite is relatively small (it rarely exceeds 20). Consequently, the minimization of $v_c[I; J]$ can be performed by simply investigating all possible subsets. In the case of larger groups of benchmarks that would not be possible and more sophisticated combinatorial optimization algorithms would be needed.

It is important to emphasize that the quality of distribution is not the only criterion for the design of benchmark suites. Any decrease of the number of individual benchmarks usually decreases both the utilization of program space and the granularity and density of a benchmark suite. Therefore, the selection of optimum subsets of benchmarks can be performed only if we can satisfy the following additional criteria:

- sufficient granularity condition: $G > 1$,
- sufficient density condition: $H > H_{min}$,
- sufficient coverage condition: $U > U_{min}$,

where H_{min} and U_{min} denote the desired threshold values. The design of benchmark suites based on optimum subsets of benchmarks $I^*(1), \dots, I^*(n)$ consists of selecting a subset that sufficiently satisfies the additional criteria of granularity, density, and coverage.

12.10. OPTIMUM WEIGHTS OF BENCHMARK PROGRAMS

In cases where a benchmark suite already exists it is possible to achieve a desired coverage of the program space by adjusting the individual “intensity” of each benchmark. Let the intensity of the benchmark B_j be expressed using a non-negative real-valued weight W_j . If the Λ coverage model is used, the weighted compound coverage at benchmark points can be defined as follows:

$$C_j = \sum_{k=1}^n W_k s_{jk} , \quad W_j \geq 0, \quad j = 1, \dots, n .$$

Our problem is to determine the weights W_1, \dots, W_n that will achieve a desired distribution of the compound coverage at points B_1, \dots, B_n .

Let C_j^* , $j = 1, \dots, n$ be the desired distribution of the compound coverage at the benchmark points. The mean square error function

$$e(W_1, \dots, W_n) = \sum_{j=1}^n \left(C_j^* - \sum_{k=1}^n W_k s_{jk} \right)^2$$

in the case of non-singular similarity matrices attains the zero minimum value for weights that satisfy

$$\begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_n \end{bmatrix} = \begin{bmatrix} s_{11} & s_{12} & \dots & s_{1n} \\ s_{21} & s_{22} & \dots & s_{2n} \\ \dots & \dots & \dots & \dots \\ s_{n1} & s_{n2} & \dots & s_{nn} \end{bmatrix}^{-1} \begin{bmatrix} C_1^* \\ C_2^* \\ \dots \\ C_n^* \end{bmatrix}$$

Unfortunately, the similarity matrix can be singular, or some of the resulting weights can be negative. That is unacceptable and indicates the need to omit the corresponding benchmarks. A more general result can be obtained if we introduce an auxiliary threshold function

$$\mu(W_j) = \begin{cases} 0, & W_j \geq 0 \\ \beta, & W_j < 0, \end{cases} \quad \beta = \text{constant} \quad (\beta \gg 1)$$

and then define the following compound error function:

$$E(W_1, \dots, W_n) = \sum_{j=1}^n \left(C_j^* - \sum_{k=1}^n W_k s_{jk} \right)^2 + \sum_{j=1}^n \mu(W_j) .$$

This function can be efficiently minimized using the Nelder-Mead simplex method¹⁷. The minimization yields non-negative weights that are coordinates of the error function minimum:

$$E(W_1^*, \dots, W_n^*) = \min E(W_1, \dots, W_n) , \quad W_j^* \geq 0, \quad j = 1, \dots, n.$$

The shape of function $E(W_1, \dots, W_n)$ can be rather irregular and it is important to start minimization from a suitable initial position. The most suitable initial values of weights are $W_j = 1$, $j = 1, \dots, n$, (i.e. all benchmarks are initially included) and in the important special case of uniform distribution it is suitable to use the constant initial values $C_i^* = \bar{C} = \frac{1}{n} \sum_{j=1}^n \sum_{k=1}^n s_{jk}$, $i = 1, \dots, n$.

The resulting weights of component benchmarks W_1^*, \dots, W_n^* are not normalized (i.e. $W_1^* + \dots + W_n^* \neq 1$). The normalized (relative) weights are

$$w_j = W_j^* / \sum_{j=1}^n W_j^* , \quad j = 1, \dots, n$$

and can be used to express the relative importance (or “intensity”) of each particular benchmark, reflecting the desired distribution of compound coverage at the benchmark points of the program space. For some subset of component benchmarks this method usually yields zero weights ($w_j = 0$, $j \in Z \subset \{1, \dots, n\}$); such benchmarks are omitted, reducing the cost of benchmarking in all cases which include actual measurements. The results that can be obtained in this way are always better than or equal to the results than can be achieved using the optimum subset method.

The normalized weights can be used for computing the global performance indicators of competitive computer systems. For example, let $t[0, j]/t[i, j]$ denote the relative throughput of the system S_i with respect to the reference system S_0 , in the case of the benchmark B_j . Then the average global relative throughput of system S_i can be computed as follows:

$$\bar{R}_i = \prod_{j=1}^n \left(\frac{t[0, j]}{t[i, j]} \right)^{w_j}, \quad i = 1, \dots, m.$$

This performance indicator shows how many times, on the average, system S_i outperforms the reference system S_0 in the case where the comparison is based on the desired distribution of benchmarks in the program space, and the distribution is expressed in terms of the compound coverage at benchmark points.

The described method enables the use of any (sufficiently diversified) set of benchmark programs to achieve an optimized approximation of a desired distribution of the compound coverage. This yields the possibility to design *universal benchmark suites* which uniformly cover the program space, and then to select suitable subsets of these benchmarks in order to express some desired workload properties. Design, maintenance, and updating of such universal suites should be the primary activity of industrial consortia such as SPEC, GPC, and others.

The subsets of benchmarks selected by the optimum weight method must satisfy additional requirements related to granularity, density, and utilization of program space. These requirements can be satisfied either by additional testing of the generated subsets, or by extending the compound error function with additional terms that cause the simplex method to select those optimum weights that simultaneously satisfy all the conditions:

Table 12.3 Compound coverage at benchmark points

COMPOUND COVERAGE (C) AND NORMALIZED COVERAGE (Cn=100*C/N) FOR N = 10 OBJECTS

| Object | C | Cn % | 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100% |
|----------------------------|------|------|---|----|----|----|----|----|----|----|----|----|------|
| matrix300 | 5.67 | 56.7 | | | | | | | * | | | | |
| li | 5.76 | 57.6 | | | | | | | * | | | | |
| espresso | 6.15 | 61.5 | | | | | | | * | | | | |
| nasa7 | 6.53 | 65.3 | | | | | | | * | | | | |
| fpppp | 6.65 | 66.5 | | | | | | | * | | | | |
| tomcatv | 6.67 | 66.7 | | | | | | | * | | | | |
| gcc | 6.76 | 67.6 | | | | | | | * | | | | |
| eqntott | 6.80 | 68.0 | | | | | | | * | | | | |
| spice2g6 | 7.26 | 72.6 | | | | | | | | * | | | |
| doduc | 7.31 | 73.1 | | | | | | | | * | | | |
| Average: | 6.55 | 65.5 | 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100% |
| Sigma(compound coverage) = | | | .53 Coefficient of variation = 8.0 % | | | | | | | | | | |

$$E(W_1, \dots, W_n) = \sum_{j=1}^n \left(C_j^* - \sum_{k=1}^n W_k S_{jk} \right)^2 + \sum_{j=1}^n \mu(W_j) + \mu(G - 1) + \mu(H - H_{min}) + \mu(U - U_{min}) .$$

12.11. A BENCHMARK SUITE DESIGN EXAMPLE

Let us now design a benchmark suite using the proposed method. Suppose that the suite is primarily intended for the comparison of processing power of Unix workstations. To apply the black-box approach we must initially have a set of candidate workloads representing the activity of a class of users. Suppose that the set of candidate workloads consists of 10 workloads adopted by SPEC in 1989 for their first benchmark suite.

Suppose now that our goal is a uniform coverage of the program space. The average compound coverage at 10 benchmark points using the Λ coverage model is shown in Table 12.3. The distribution is not uniform: the maximum/minimum coverage ratio is 1.29.

The design of a benchmark suite having a uniform distribution of compound coverage at benchmark points yields results shown in Table

12.4. We first present all subsets of best representatives and the corresponding average coverage. The best representatives, however, generate distributions that are not uniform, and the average error with respect to the uniform distribution is frequently greater than 10%.

The distribution of coverage can be improved using the method of optimum subsets. It is interesting to note that a good uniformity of coverage at 10 benchmark points (the error of only 2.3%) can be achieved with only two optimally selected benchmarks (*li* and *matrix300*). When a third benchmark is added the error quickly increases from 2.3% to 7%, and then again decreases when the fourth benchmark is added to the suite. On the other hand, if a small subset of benchmarks can cause a very good coverage at the benchmark points, this might indicate that the benchmark points are located rather closely, and that the redundancy level of benchmarks is too high.

The obtained results can be further improved by using the method of optimum weights. Suppose that we want to achieve the same coverage at all the benchmark points, and that it must be equal to the initial average coverage $\bar{C} = 6.55$. We can achieve this goal with the average error of only 1.563%. This is substantially better than in the case of optimum subsets. The optimum weight method automatically selects the appropriate subset of benchmarks and adjusts their relative importance: the results in Table 12.4 are achieved using only four of the available 10 benchmarks. This reduces the granularity of the suite, but in the case of measuring only the processing power of workstations the granularity for $n = 4$ can still be sufficient. However, the cost of benchmarking is now 2.5 times less than the cost of benchmarking with all candidate workloads.

Therefore, the final result of designing a benchmark suite which uniformly covers the program space and serves for the comparison of processing power of Unix workstations is the suite consisting of four benchmark workloads: *espresso*, *li*, *matrix300* and *fpppp*. Their relative weights are respectively 0.126 , 0.38 , 0.452 , and 0.042 . The comparison of m workstations should be performed using the average relative throughputs $\bar{R}_1, \dots, \bar{R}_m$ computed from the following formula:

$$\bar{R}_i = \left(\frac{t[0, 1]}{t[i, 1]} \right)^{0.126} \left(\frac{t[0, 2]}{t[i, 2]} \right)^{0.38} \left(\frac{t[0, 3]}{t[i, 3]} \right)^{0.452} \left(\frac{t[0, 4]}{t[i, 4]} \right)^{0.042} .$$

The run times of individual workloads for the reference system are denoted $t[0, 1]$, $t[0, 2]$, $t[0, 3]$, and $t[0, 4]$. The reference system can be

Table 12.4. Results of the benchmark suite design

SUBSETS OF BEST REPRESENTATIVES AND THEIR COMPOUND COVERAGE

| No. | Average coverage | Average error | Subsets of best representatives | | | | | | | | | |
|-----|------------------|---------------|---------------------------------|---|---|---|----|----|----|----|---|----|
| 10 | 6.55 | 8.0% | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 9 | 5.89 | 9.1% | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |
| 8 | 5.24 | 12.0% | 1 | 2 | 3 | 4 | 6 | 7 | 8 | 10 | | |
| 7 | 4.51 | 11.8% | 1 | 2 | 4 | 6 | 7 | 8 | 10 | | | |
| 6 | 3.83 | 8.8% | 1 | 2 | 4 | 6 | 8 | 10 | | | | |
| 5 | 3.26 | 8.0% | 1 | 2 | 4 | 8 | 10 | | | | | |
| 4 | 2.67 | 12.5% | 1 | 2 | 4 | 5 | | | | | | |
| 3 | 2.00 | 9.4% | 2 | 4 | 5 | | | | | | | |
| 2 | 1.33 | 9.2% | 5 | 7 | | | | | | | | |
| 1 | .73 | 17.0% | 4 | | | | | | | | | |

OPTIMUM SUBSETS OF BENCHMARKS AND THEIR COMPOUND COVERAGE

| No. | Average coverage | Average error | Optimum subsets of benchmarks | | | | | | | | | |
|-----|------------------|---------------|-------------------------------|---|---|---|---|----|----|----|----|----|
| 10 | 6.55 | 8.0% | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 9 | 5.82 | 7.1% | 1 | 2 | 3 | 5 | 6 | 7 | 8 | 9 | 10 | |
| 8 | 5.10 | 6.2% | 1 | 2 | 5 | 6 | 7 | 8 | 9 | 10 | | |
| 7 | 4.48 | 6.6% | 1 | 2 | 3 | 5 | 6 | 8 | 10 | | | |
| 6 | 3.76 | 5.1% | 2 | 5 | 6 | 7 | 8 | 10 | | | | |
| 5 | 3.14 | 6.0% | 2 | 4 | 5 | 6 | 8 | | | | | |
| 4 | 2.41 | 3.2% | 2 | 5 | 6 | 8 | | | | | | |
| 3 | 1.87 | 7.0% | 4 | 6 | 8 | | | | | | | |
| 2 | 1.14 | 2.3% | 6 | 8 | | | | | | | | |
| 1 | .73 | 17.0% | 4 | | | | | | | | | |

OPTIMUM INTENSITIES OF BENCHMARK PROGRAMS

| No. | BENCHMARK | COMPOUND COVERAGE | | | | WEIGHT | |
|-----|-----------|-------------------|---------|----------|-------|----------|----------|
| | | Initial | Desired | Achieved | Error | Absolute | Relative |
| 1 | gcc | 6.76 | 6.55 | 6.47 | -1.3% | .000 | .0% |
| 2 | espresso | 6.15 | 6.55 | 6.45 | -1.6% | 1.423 | 12.6% |
| 3 | spice2g6 | 7.26 | 6.55 | 6.72 | 2.5% | .000 | .0% |
| 4 | doduc | 7.31 | 6.55 | 6.71 | 2.3% | .000 | .0% |
| 5 | nasa7 | 6.53 | 6.55 | 6.48 | -1.1% | .000 | .0% |
| 6 | li | 5.76 | 6.55 | 6.51 | -.6% | 4.291 | 38.0% |
| 7 | eqntott | 6.80 | 6.55 | 6.61 | .9% | .000 | .0% |
| 8 | matrix300 | 5.67 | 6.55 | 6.51 | -.7% | 5.105 | 45.2% |
| 9 | fpppp | 6.65 | 6.55 | 6.43 | -1.9% | .473 | 4.2% |
| 10 | tomcatv | 6.67 | 6.55 | 6.64 | 1.3% | .000 | .0% |

fmin = .105 AVERAGE ERROR = 1.563%

selected as one of the competitive systems. Then \overline{R}_i denotes how many times the i^{th} system is faster than the reference system.

12.12. SUMMARY AND CONCLUSIONS

Any program becomes a benchmark program whenever we focus our interest on its resource consumption instead of its results. However, benchmark workloads and benchmark suites must not be randomly selected. Quantitative methods for evaluation and design of benchmark suites are necessary both for those who create benchmark suites and for those interested in a proper interpretation of performance measurement results. The basic approaches to workload characterization are the white-box and the black-box approach. In cases where it is possible to monitor internal operations and individual resource activities of a computer system, we can use the white-box approach. In cases where the available information is reduced to global performance indicators, such as system response times or throughputs, we use the black-box approach. The theory of program space is applicable in both cases.

The black-box approach can be realized without special measurement tools, and generally it enables easier collection of measurement data than the white-box approach. Therefore, our methodology for evaluation and design of benchmark suites is primarily oriented towards the black-box approach. The proposed evaluation method is based on five qualitative and five quantitative evaluation criteria. The quantitative criteria are used to evaluate the size, redundancy, completeness, granularity, and density of benchmark suites. These criteria are used for both evaluation and design (or upgrading) of benchmark suites. The presented benchmark suite evaluation examples illustrate a general approach which can be used to reduce the redundancy and to improve the completeness of all benchmark suites.

The design of benchmark suites is based on a desired distribution of benchmarks in the program space. In the case of benchmark workloads with adjustable parameters it is possible to move workloads through the program space towards a desired destination. However, this requires the white-box approach, and can differ for various hardware/software architectures. To provide a similar effect within the black-box approach we introduced coverage functions and developed two methods, based on optimum subsets and optimum weights of benchmark programs. The

interesting result is that a desired distribution of compound coverage of the program space can be achieved without moving benchmark programs through the program space. If we aggregate the performance measurement results of individual benchmark programs using properly selected weights, then the effects on the aggregate performance indicators are similar to the effects of changing the distribution of benchmarks in the program space.

Thus, it is possible to design universal benchmark suites that uniformly cover the program space. Instead of designing various specific benchmark suites, and performing costly measurements, the desired benchmarking features can be expressed through appropriate weights of individual benchmarks of the universal suite. This technique can reduce the cost of industrial benchmarking and should be widely used for computing aggregate performance indicators, and for comparison of computer systems.

REFERENCES

1. Standard Performance Evaluation Corporation (SPEC), "SPEC Benchmark Suite Release 1.0," SPEC Newsletter **1**, 1, pp. 5-9 (Fall 1989).
This is the first presentation of 10 SPEC89 benchmarks, and the SPECmark performance indicator. This benchmark suite became an industry standard and has been upgraded twice (SPEC92 and SPEC95).
2. W. KOHLER, "How to Improve OLTP Performance and Price/Performance," TPC Quarterly Report, Oct. 15, pp. 1-9 (1992).
3. GPC (Graphics Performance Characterization Committee), "The Effort to Standardize Performance Measurement," GPC Quarterly Report, **2**, 4, pp. 10-12 (1992).
4. THE PERFECT CLUB, "The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers," The International J. of Supercomputer Applications, **3**, 3, pp. 5-40 (1989).
5. AIM TECHNOLOGY, "The AIM Performance Report: Technical Description," AIM Technology UNIX System Price Performance Guide, pp. 127-129 (Summer 1993).

6. J. GRAY, Ed., The Benchmarking Handbook for Database and Transaction Processing Systems, Morgan Kaufmann, 1991.

This book summarizes all important research results in the area of benchmarking database systems. It is written in a similar style as the first book on benchmarking, N. Benwell (Ed.), Benchmarking - Computer Evaluation and Measurement, J. Wiley, 1975. A "must read" for performance analysts and researchers in this area.

7. F. McMAHON, The Livermore FORTRAN Kernels: a Computer Test of the Numerical Performance Range, Lawrence Livermore National Laboratory, Berkeley, Calif., UCRL-537415, 1986.

8. D.H. BAILEY et al., "The NAS Parallel Benchmarks," The International J. of Supercomputer Applications, **5**, 3, pp. 63-73 (1991).

9. R.P. GABRIEL, Performance and Evaluation of Lisp Systems, The MIT Press, Cambridge, Massachusetts, 1986.

A systematic and detailed presentation of a comprehensive suite of Lisp benchmarks. Included are both source programs and the results of performance measurements.

10. L. BURKHOLDER et al., "PROLOG for the People," AI Expert, pp. 63-84 (June 1987).

11. R. GRACE, The Benchmark Book, Prentice-Hall, 1996.

A practitioner's market-oriented survey of currently popular benchmark suites. It includes basic data about SPEC benchmarks, TPC benchmarks, Neal Nelson benchmarks, AIM benchmarks, networking benchmarks, personal computer benchmarks, GPC benchmarks, computer magazine benchmarks, and selected small scientific benchmarks.

12. D. FERRARI, G. SERAZZI, and A. ZEIGNER, Measurement and Tuning of Computer Systems, Prentice-Hall, 1983.

13. J.J. DUJMOVIĆ, "Workload Characterization, Benchmarking, and the Concept of Total Resources Consumption," Proceedings of the Second International Conference on Computer Capacity Management (H.L. Bording and D.J. Schumacher, Ed.), San Francisco, April 8-10, pp. 151-163 (1980).

14. J.J. DUJMOVIĆ, "Clustering, Comparison and Selection of Standard Synthetic Benchmark Programs," Computer Systems Science and

Engineering, **6**, 4, pp. 195-210 (October 1991).

This is the first journal presentation of the black-box approach to workload characterization (the first conference presentation was in June 1988). It includes examples of evaluation of Livermore loops, instruction mixes, Lisp, and Prolog benchmark suites.

15. J.G. KEMENY and J.L. SNELL, Mathematical Models in the Social Sciences, Chapter II (Preference Ranking - An Axiomatic Approach). The MIT Press, 1978.

16. A.K. JAIN and R.C. DUBES, Algorithms for Clustering Data, Prentice Hall, 1988.

17. M. MERKLE, Personal communication, University of Belgrade, 1991.

18. J.A. NELDER and R. MEAD, "A Simplex Method for Function Minimization," The Computer Journal, **7**, pp. 308-313 (1965).

This is a simple, robust, and efficient algorithm for finding a local maximum/minimum of a function of 2 or more variables using only the values of the function (no need for derivatives). The practical value of this method is substantially greater than its popularity. In the Algorithms Hall of Fame the simplex algorithm should be in the first row, together with Quicksort, binary search, and the Runge-Kutta method.