Analysis of Benchmark Characteristics and Benchmark Performance Prediction†§

Rafael H. Saavedra‡
Alan Jay Smith♯

ABSTRACT

Standard benchmarking provides the run times for given programs on given machines, but fails to provide insight as to why those results were obtained (either in terms of machine or program characteristics), and fails to provide run times for that program on some other machine, or some other programs on that machine. We have developed a machine-independent model of program execution to characterize both machine performance and program execution. By merging these machine and program characterizations, we can estimate execution time for arbitrary machine/program combinations. Our technique allows us to identify those operations, either on the machine or in the programs, which dominate the benchmark results. This information helps designers in improving the performance of future machines, and users in tuning their applications to better utilize the performance of existing machines.

Here we apply our methodology to characterize benchmarks and predict their execution times. We present extensive run-time statistics for a large set of benchmarks including the SPEC and Perfect Club suites. We show how these statistics can be used to identify important shortcomings in the programs. In addition, we give execution time estimates for a large sample of programs and machines and compare these against benchmark results. Finally, we develop a metric for program similarity that makes it possible to classify benchmarks with respect to a large set of characteristics.

---

† The material presented here is based on research supported principally by NASA under grant NCC2-550, and also in part by the National Science Foundation under grants MIP-8713274, MIP-9116578 and CCR-9117028, by the State of California under the MICRO program, and by the International Business Machines Corporation, Philips Laboratories/Signetics, Apple Computer Corporation, Intel Corporation, Mitsubishi Electric, Sun Microsystems, and Digital Equipment Corporation.
§ This paper is available as Computer Science Technical Report USC-CS-92-524, University of Southern California, and Computer Science Technical Report UCB/CSD 92/715, UC Berkeley.
‡ Computer Science Department, Henry Salvatori Computer Science Center, University of Southern California, Los Angeles, California 90089-0781 (e-mail: saavedra@palenque.usc.edu).
♯ Computer Science Division, EECS Department, University of California, Berkeley, California 94720.

1. Introduction

Benchmarking is the process of running a specific program or workload on a specific machine or system, and measuring the resulting performance. This technique clearly provides an accurate evaluation of the performance of that machine for that workload. These benchmarks can either be complete applications [UCB87, Dong88, MIPS89], the most executed parts of a program (kernels) [Bail85, McMa86, Dodu89], or synthetic programs [Curn76, Weic88]. Unfortunately, benchmarking fails to provide insight as to why those results were obtained (either in terms of machine or program characteristics), and fails to provide run times for that program on some other machine, or some other program on that machine [Worl84, Dong87]. This is because benchmarking fails to characterize either the program or machine. In this paper we show that these limitations can be overcome with the help of a performance model based on the concept of a high-level abstract machine.

Our machine model consists of a set of abstract operations representing, for some particular programming language, the basic operators and language constructs present in programs. A special benchmark called a *machine characterizer* is used to measure experimentally the time it takes to execute each abstract operation (*AbOp*). Frequency counts of AbOps are obtained by instrumenting and running benchmarks. The machine and program characterizations are then combined to obtain execution time predictions. Our results show that we can predict with good accuracy the execution time of arbitrary programs on a large spectrum of machines, thereby demonstrating the validity of our model. As a result of our methodology, we are able to individually evaluate the machine and the benchmark, and we can explain the results of individual benchmarking experiments. Further, we can describe a machine which doesn't actually exist, and predict with good accuracy its performance for a given workload.

In a previous paper we discussed our methodology and gave an in-depth presentation on machine characterization [Saav89]. In this paper we focus on program characterization and execution time prediction; note that this paper overlaps with [Saav89] to only a small extent, and only with regard to the discussion of the necessary background and methodology. Here, we explain how programs are characterized and present extensive statistics for a large set of programs including the Perfect Club and SPEC benchmarks. We discuss what these benchmarks measure and evaluate their effectiveness; in some cases, the results are surprising.

We also use the dynamic statistics of the benchmarks to define a metric of similarity between the programs; similar programs exhibit similar relative performance across many machines.

The structure of the paper is as follows. In Section 2 we present an overview of our methodology, explain the main concepts, and discuss how we do program analysis and execution time prediction. We proceed in Section 3 by describing the set of benchmarks used in this study. Section 4 deals with execution time prediction. Here, we present predictions for a large set of machine-program combinations and compare these against real execution times. In Section 5 we present an extensive analysis of the benchmarks. The concept of program similarity is presented in Section 6. Section 7 ends the paper with a summary and some of our conclusions. The presentation is self-contained and does not assume familiarity with the previous paper.

2. Abstract Model and System Description

In this section we present an overview of our abstract model and briefly describe the components of the system. The machine characterizer is described in detail in [Saav89]; this paper is principally concerned with the execution predictor and program analyzer.

2.1. The Abstract Machine Model

The abstract model we use is based on the Fortran language, but it equally applies to other algorithmic languages. Fortran was chosen because it is relatively simple, because the majority of standard benchmarks are written in Fortran, and because the principal agency funding this work (NASA) is most interested in that language. We consider each computer to be a Fortran machine, where the run time of a program is the (linear) sum of the execution times of the Fortran abstraction operations (AbOps) executed. Thus, the total execution time of program A on machine M ($T_{A,M}$) is just the linear combination of the number of times each abstract operation is executed (C_i), which depends only on the program, multiplied by the time it takes to execute each operation (P_i), which depends only on the machine:

$$T_{A,M} = \sum_{i=1}^{n} C_{A,i}\, P_{M,i} = \mathbf{C_A} \cdot \mathbf{P_M} \tag{1}$$

$\mathbf{P_M}$ and $\mathbf{C_A}$ represent the machine performance vector and program characterization vector respectively.

Equation (1) decomposes naturally into three components: the *machine characterizer*, *program analyzer*, and *execution predictor*. The machine characterizer runs experiments to obtain vector $\mathbf{P_M}$. The dynamic statistics of a program, represented by vector $\mathbf{C_A}$ are obtained using the program analyzer. Using these two vectors, the execution predictor computes the total execution time for program A on machine M.

We assume in the rest of this paper that all programs are written in Fortran, are compiled with optimization turn off, and executed in scalar mode. All our statistics reflect these assumptions. In [Saav92a] we show how our model can be extended (very successfully) to include the effects of compiler optimization and cache misses.

2.2. Linear Models

As noted above, our execution prediction is the linear sum of the execution times of the AbOps executed; equation (1) shows this linear model. Although linear models have been used in the past to fit a k-parametric "model" to a set of benchmark results, our approach is entirely different; we *never* use curve fitting. All parameter values are the result of direct measurement, and none are inferred as the solution of some fitted model. We make a specific point of this because this aspect of our methodology has been misunderstood in the past.

2.3. Machine Characterizer

The machine characterizer is a program which uses *narrow spectrum benchmarking* or *microbenchmarking* to measure the execution time of each abstract operation. It does this by, in most cases, timing a loop both with and without the AbOp of interest; the change in the run time is due to that operation. Some AbOps cannot be so easily isolated and more complicated methods are used. There are 109 operations in the abstract model, up from 102 in [Saav89]; the benchmark set has been expanded since that time, and additional AbOps were found to be needed.

The number and type of operations is directly related to the kind of language constructs present in Fortran. Most of these are associated with arithmetic operations and trigonometric functions. In addition, there are parameters for procedure call, array index calculation, logical operations, branches, and do loops. In appendix A (tables 14 and 15), we present the set of 109 parameters with a small description of what each operation measures.

We note that obtaining accurate measurements of the AbOps is very tricky because the operations take nanoseconds and the clocks on most machines run at 60 or 100 hertz. To get accurate measurements, we run our loops large numbers of times and then repeat each such loop measurement several times. There are residual errors, however, due to clock resolution, external events like interrupts, multiprogramming and I/O activity, and unreproducible variations in the hit ratio of the cache, and paging [Clap86]. These issues are discussed in more detail in [Saav89].

2.4. The Program Analyzer

The analysis of programs consists of two phases: the *static analysis* and the *dynamic analysis*. In the static phase, we count the number of occurrences of each AbOp in each line of source code. In the dynamic phase, we instrument the source code to give us counts for the number of executions of each line of source code, and then compile and run the instrumented version. The instrumented version tends to run about 15% slower than the uninstrumented version.

Let A be a program with input data I. Let us number each of the basic blocks of the program $j = 1, 2, \cdots, m$, and let $s_{i,j}$ $(i = 1, 2, \cdots, n)$ designate the number of static occurrences of operation P_i in block B_j. Matrix $\mathbf{S_A} = [s_{i,j}]$ of size $n \times m$ represents the complete static statistics of the program. Let $\boldsymbol{\mu}_\mathbf{A} = <\mu_1, \mu_2, \cdots, \mu_j>$ be the number of times each basic block is executed, then matrix $\mathbf{D_A} = [d_{i,j}] = [\mu_j \cdot s_{i,j}]$ gives us the dynamic statistics by basic block. Vector $\mathbf{C_A}$ and matrix $\mathbf{D_A}$ are related by the following equation

$$C_i = \sum_{j=1}^{m} d_{i,j}. \tag{2}$$

Obtaining the dynamic statistics in this way makes it possible to compute execution time predictions for each of the basic blocks, not only for the whole program.

The methodology described above permits us to measure M machines and N programs and then compute run time predictions for $N \cdot M$ combinations. Note that our methodology will not apply in two cases. First, if the execution history of a program is precision dependent (as is the case with some numerical analysis programs), then the number of AbOps will vary from machine to machine. Second, the number of AbOps may vary if the execution history is real-time dependent; the machine characterizer is an example of a real-time dependent program, since the number of times a loop is executed is a function of the machine speed and the clock resolution. All programs that we consider in this paper have execution histories that are precision and time independent[1].

---

[1] The original version of TRACK found in the Perfect Club benchmarks exhibited several execution histories due to an inconsistency in the passing of constant parameters. The version that we used in this paper does not have this problem.

2.5. Execution Prediction

The execution predictor is a program that computes the expected execution time of program A on machine M from its corresponding program and machine characterizations. In addition, it can produce detailed information about the execution time of sets of basic blocks or how individual abstract operations contribute to the total time.

```
        PROGRAM STATISTICS FOR THE TRFD BENCHMARK ON THE IBM RS/6000 530:
                Lines processed             -> from 1 to 485 [485]

 mnem   operation      times-executed  fraction   execution-time   fraction
[arsl] add    (002)  exec:          7 (0.0000)  time:    0.000001 (0.0000)
[sisl] store  (015)  exec:    6583752 (0.0043)  time:    0.000000 (0.0000)
[aisl] add    (016)  exec:    9497124 (0.0062)  time:    1.292559 (0.0036)
[misl] mult   (017)  exec:        196 (0.0000)  time:    0.000031 (0.0000)
[disl] divide (018)  exec:        210 (0.0000)  time:    0.000198 (0.0000)
[tisl] trans  (021)  exec:     101949 (0.0001)  time:    0.012071 (0.0000)
[srdl] store  (022)  exec:  216205010 (0.1416)  time:    2.832286 (0.0079)
[ardl] add    (023)  exec:  215396153 (0.1411)  time:   23.090467 (0.0642)
[mrdl] mult   (024)  exec:  214742010 (0.1406)  time:   22.504963 (0.0626)
[drdl] divide (025)  exec:     735371 (0.0005)  time:    0.563588 (0.0016)
[erdl] exp-i  (026)  exec:         28 (0.0000)  time:    0.000002 (0.0000)
[trdl] trans  (028)  exec:   18545814 (0.0121)  time:    1.743307 (0.0048)
[sisg] store  (043)  exec:        175 (0.0000)  time:    0.000000 (0.0000)
[aisg] add    (044)  exec:     730303 (0.0005)  time:    0.110495 (0.0003)
[misg] mult   (045)  exec:         35 (0.0000)  time:    0.000005 (0.0000)
[tisg] trans  (049)  exec:          9 (0.0000)  time:    0.000003 (0.0000)
[andl] and-or (057)  exec:          1 (0.0000)  time:    0.000000 (0.0000)
[cisl] i-sin  (060)  exec:    1514464 (0.0010)  time:    0.426170 (0.0012)
[crdl] r-dou  (061)  exec:    6723500 (0.0044)  time:    2.989268 (0.0083)
[crdg] r-dou  (066)  exec:          2 (0.0000)  time:    0.000001 (0.0000)
[proc] proc   (067)  exec:       5289 (0.0000)  time:    0.001074 (0.0000)
[argl] argums (068)  exec:       5394 (0.0000)  time:    0.001101 (0.0000)
[arr1] in:1-s (071)  exec:  166300304 (0.1089)  time:   33.060501 (0.0919)
[arr2] in:2-s (072)  exec:  499858800 (0.3274)  time:  204.792156 (0.5696)
[loin] do-ini (076)  exec:    7474649 (0.0049)  time:    1.456062 (0.0040)
[loov] do-lop (077)  exec:  162509732 (0.1064)  time:   64.678873 (0.1799)
[loix] do-ini (078)  exec:          1 (0.0000)  time:    0.000002 (0.0000)
[loox] do-lop (079)  exec:          7 (0.0000)  time:    0.000004 (0.0000)

                  Predicted execution time = 359.555187 secs
```

Figure 1: Execution time estimate for the *TRFD* benchmark program run on an IBM RS/6000 530.

Figure 1 shows a sample of the output produced by the execution predictor. Each line gives the number of times that a particular AbOp is executed, and the fraction of the total that it represents. Next to it is the expected execution time contributed by the AbOp and also the fraction of the total. The last line reports the expected execution time for the whole program.

The statistics from the execution predictor provide information about what factors contribute to the execution time, either at the level of the abstract operations or individual basic blocks. For example, figure 1 shows that 57% of the time is spent computing the address of a two-dimensional array element (`arr2`). This operation, however, represents only 33% of all operations in the program (column six). By comparing the execution predictor outputs of different machines for the same program, we can see if there is some kind of imbalance in any of the machines that makes its overall execution time larger than expected [Saav90].

2.6. Related Work

Several papers have proposed different approaches to execution time prediction, with significant differences in their degrees of accuracy and applicability. These attempts have ranged from using simple Markov Chain models [Rama65, Beiz70] to more complex approaches that involve solving a set of recursive performance equations [Hick88]. Here we mention three proposals that are somewhat related to our concept of an abstract machine model and the use of static and dynamic program statistics.

One way to compare machines is to do an analysis similar to ours, but at the level of the machine instruction set [Peut77]. This approach only permits comparisons between machines which implement the same instruction set.

In the context of the PTRAN project [Alle87], execution time prediction has been proposed as a technique to help in the automatic partitioning of parallel programs into tasks. In [Sark89], execution profiles are obtained indirectly by collecting statistics on all the loops of a possible unstructured program, and then combining that with analysis of the control dependence graph.

In [Bala91] a prototype of a static performance estimator which could be used by a parallel compiler to guide data partitioning decisions is presented. These performance estimates are computed from machine measurements obtained using a set of routines called the *training set*. The training set is similar to our machine characterizer. In addition to the basic CPU measurements, the training set also contains tests to measure the performance of communication primitives in a loosely synchronous distributed memory machine. The compiler then makes a static analysis of the program and combines this information with data produced by the training set. A prototype of the performance estimator has been implemented in the ParaScope interactive parallel programming environment [Bala89]. In contrast to our execution time predictions, the compiler does not incorporate dynamic program information; the user must supply the lower and upper bounds of symbolic variables used for do loops, and branching probabilities for if-then statements (or use the default probabilities provided by the compiler.)

3. The Benchmark Programs

For this study, we have assembled and analyzed a large number of scientific programs, all written in Fortran, representing different application domains. These programs can be classified in the following three groups: SPEC benchmarks, Perfect Club benchmarks, and small or generic benchmarks. Table 1 gives a short description of each program. In the list for the Perfect benchmarks we have omitted the program *SPICE*, because it is included in the SPEC benchmarks as *SPICE2G6*. For each benchmark except *SPICE2G6*, we use only one input data set. In the case of *SPICE2G6*, the Perfect Club and SPEC versions use different data sets and we have characterized both executions and also include other relevant examples.

SPEC Benchmarks		
DODUC	double	A Monte-Carlo simulation for a nuclear reactor's component [Dodu89]
FPPPP	8 bytes	A computation of a two electron integral derivate
TOMCATV	8 bytes	Mesh generation with Thompson solver
MATRIX300	8 bytes	Matrix operations using LINPACK routines
NASA7	double	A collection of seven kernels typical of NASA Ames applications.
SPICE2G6	double	Analog circuit simulation an analysis program
BENCHMARK	double	MOS amplifier, Schmitt circuit, tunnel diode, etc
BIPOLE	double	Schottky TTL edge-triggered register
DIGSR	double	CMOS digital shift register
GREYCODE	double	Grey code counter
MOSAMP2	double	MOS amplifier (transient phase)
PERFECT	double	PLA circuit
TORONTO	double	Differential comparator

Perfect Club Benchmarks		
ADM	single	Pseudospectral air pollution simulation
ARC2D	double	Two-dimensional fluid solver of Euler equations
FLO52	single	Transonic inviscid flow past an airfoil
OCEAN	single	Two dimension ocean simulation
SPEC77	single	Weather simulation
BDNA	double	Molecular dynamic package for the simulation of nucleic acids
MDG	double	Molecular dynamics for the simulation of liquid water
QCD	single	Quantum chromodynamics
TRFD	double	A kernal simulating a two-electron integral transformation
DYFESM	single	Structural dynamics benchmark (finite element)
MG3D	single	Depth migration code
TRACK	double	Missile tracking

Various Applications and Synthetic Benchmarks		
ALAMOS	single	A set of loops which measure the execution rates of basic vector operations
BASKETT	single	A backtrack algorithm to solve the Conway-Baskett puzzle [Beel84]
ERATHOSTENES	single	Uses a sieve algorithm to obtain all the primes less than 60000
LINPACK	single	Standard benchmark which solves a systems of linear equations [Dong88]
LIVERMORE	8 bytes	The twenty four Livermore loops [McMa86]
MANDELBROT	single	Computes the mapping $Z_n \leftarrow Z_{n-1}^2 + C$ on a 200x100 grid
SHELL	single	A sort of ten thousand numbers using the Shell algorithm
SMITH	2, 4, 8 bytes	Seventy-seven loops which measure different aspects of machine performance
WHETSTONE	single	A synthetic benchmark based on Algol 60 statistics [Curn76]

Table 1: Description of the SPEC, Perfect Club, and small benchmarks. For program *SPICE2G6* we include seven different models. The second column indicates whether the floating point declarations use absolute or relative precision. For those programs that use absolute declarations, we include the number of bytes used.

3.1. Floating-Point Precision

In Fortran, the precision of a floating point variable can be specified either absolutely (by the number of bytes used, e.g. real*4), or relatively, by using the words "single" and "double." The interpretation of the latter terms is compiler and machine dependent, Most of the benchmarks we consider (see table 1) use relative declarations; this means that the measurements taken on the Cray machines (see table 2) are not directly comparable with those taken on the other machines. We chose not to modify any of the source code to avoid this problem.

3.2. The SPEC Benchmark Suite

The Systems Performance Evaluation Cooperative (SPEC) was formed in 1989 by several machine manufacturers to make available believable industry standard benchmark results. The main efforts of SPEC have been in the following areas: 1) selecting a set of non trivial applications to be used as benchmarks; 2) formulating the rules for the execution of the benchmarks; and 3) making public performance results obtained using the SPEC suite.

The 1989 SPEC suite consists of six Fortran and four C programs taken from the scientific and systems domains [SPEC89, SPEC90]. (There is a second set of SPEC benchmarks, available in 1992, which we do not consider.) For each benchmark, the SPECratio is the ratio between the execution time on the machine being measured to that on a VAX-11/780. The SPECmark is the overall performance measure, and is defined as the geometric mean of all SPECratios. In this study, when we mention the SPEC benchmarks we refer only to the Fortran programs in the suite, plus six additional input models for *SPICE2G6*. We now give a brief explanation of what these programs do:

DODUC is a Monte Carlo simulation of the time evolution of a thermohydraulical modelization ("hydrocode") for a nuclear reactor's component. It has very little vectorizable code, but has an abundance of short branches and loops.

FPPPP is a quantum chemistry benchmark which measures performance on one style of computation (two electron integral derivative) which occurs in the Gaussian series of programs.

TOMCATV is a very small (less than 140 lines) highly vectorizable mesh generation program. It is a double precision floating-point benchmark.

MATRIX300 is a code that performs various matrix multiplications, including transposes using Linpack routines SGEMV, SGEMM, and SAXPY, on matrices of order 300. More than 99 percent of the execution is in a single basic block inside SAXPY.

NASA7 is a collection of seven kernels representing the kind of algorithms used in fluid flow problems at NASA Ames Research Center. All the kernels are highly vectorizable.

SPICE2G6 is a general-purpose circuit simulation program for nonlinear DC, nonlinear transient, and linear AC analysis. This program is a very popular CAD tool widely used in industry. We use seven models on this programs: *BENCHMARK*, *BIPOLE*, *DIGSR*, *GREYCODE*, *MOSAMP2*, *PERFECT*, and *TORONTO*. *GREYCODE* and *PERFECT* are the examples included in the SPEC and Perfect Club benchmarks.

3.3. The Perfect Club Suite

The Perfect Club Benchmark Suite is a set of thirteen scientific programs, intended to represent supercomputer scientific workloads [Cybe90]. Performance in the Perfect Club approach is defined as the harmonic mean of the MFLOPS (Millions of FLoating-point Operations per Second) rate for each program on the given machine. The number of FLOPS in a program is determined by the number of floating-point instructions executed on the CRAY X-MP, using the CRAY X-MP performance monitor.

The Perfect programs can be classified into four different groups depending on the type of the problem solved: fluid flow, chemical & physical, engineering design, and signal processing.

Programs in the fluid flow group are: *ADM*, *ARC2D*, *FLO52*, *OCEAN*, and *SPEC77*.

ADM simulates pollutant concentration and deposition patterns in lakeshore environments by solving the complete system of hydrodynamic equations.

ARC2D is an implicit finite-difference code for analyzing two-dimensional fluid flow problems by solving the Euler equations.

FLO52 performs an analysis of a transonic inviscid flow past an airfoil by solving the unsteady Euler equations in a two-dimensional domain. A multigrid strategy is used and the code vectorizes well.

OCEAN is a two-dimensional ocean simulation.

SPEC77 provides a global spectral model to simulate atmospheric flow. Weather simulation codes normally consists of four modules: preprocessing, computing normal mode coefficients, forecasting, and postprocessing. *SPEC77* only includes the forecasting part.

Programs in the chemical and physical group are: *BDNA*, *MDG*, *QCD*, and *TRFD*.

BDNA is a molecular dynamics package for the simulations of the hydration structure and dynamics of nucleic acids. Several algorithms are used in solving the translational and rotational equations of motion. The input for this benchmark is a simulation of the hydration structure of 20 potassium counter-ions and 1500 water molecules in B-DNA.

MDG is another molecular dynamic simulation of 343 water molecules. Intra and intermolecular interactions are considered. The Newtonian equations of motion are solved using Gera's sixth-order predictor-corrector method.

QCD was original developed at Caltech for the MARK I Hypercube and represents a gauge theory simulation of the strong interactions which binds quarks and gluons into hadrons which, in turn, make up the constituents of nuclear matter.

TRFD represents a kernel which simulates the computational aspects of two electron integral transformation. The integral transformation are formulated as a series of matrix multiplications, so the program vectorizes well. Given the size of the matrices, these are not kept completely in main memory.

The engineering design programs are: *DYFESM* and *SPICE* (described with the SPEC benchmarks).

DYFESM is a finite element structural dynamics code.

Finally, the signal processing programs are: *MG3D* and *TRACK*.

MD3G is a seismic migration code used to investigate the geological structure of the Earth. Signals of different frequencies measured at the Earth's surface are extrapolated backwards in time to get a three-dimensional image of the structure below the surface.

TRACK is used to determine the course of a set of an unknown number of targets, such as rocket boosters, from observations of the targets taken by sensors at regular time intervals. Several algorithms are used to estimate the position, velocity, and acceleration components.

3.4. Small Programs and Synthetic Benchmarks

Our last group of programs consists of small applications and some popular synthetic benchmarks. The small applications are: *BASKETT*, *ERATHOSTENES*, *MANDELBROT*, and *SHELL*. The synthetic benchmarks are: *ALAMOS*, *LINPACK*, *LIVERMORE*, *SMITH*, and *WHETSTONE*. A description of these programs can be found in [Saav88].

4. Predicting Execution Times

We have used the execution predictor to obtain estimates for the programs in table 1, and for the machines shown in table 2. These results are presented in figure 2. In addition, in tables 33 through 35 in Appendix D we report the actual execution time, the predicted execution, and the error $((pred - real)/real)$ in percent. The minus (plus) sign in the error corresponds to a prediction which is smaller (greater) than the real time. We also show the arithmetic mean and root mean square errors across all machines and programs. From the results in Appendix D we see that the average error for all programs is less than 2% with a root mean square of less than 20%.

A subset of programs did not execute correctly on all machines at the time of this research; some of these problems may have been corrected since that time. Some of the

Table 2: **Characteristics of the machines**							
Machine	Name/Location	Operating System	Compiler version	Memory	Integer	Real	
					single	single	double
CRAY Y-MP/8128	reynolds.nas.nasa.gov	UNICOS 5.0.13	CFT77 3.1.2.6	128 Mw	46	64	128
CRAY-2	navier.nas.nasa.com	UNICOS 6.1	CFT 5.0.3.5	256 Mw	46	64	128
CRAY X-MP/48	NASA Ames	COS 1.16	CFT 1.14	8 Mw	46	64	128
NEX SX-2	harc.edu	VM/CMS	FORT77SX	32 Mw	64	64	128
Convex C-1	convex.riacs.edu	UNIX C-1 v6	FC v2.2	100 MB	32	32	64
IBM 3090/200	cmsa.berkeley.edu	VM/CMS r.4	FORTRAN v2.3	32 MB	32	32	64
IBM RS/6000 530	coyote.berkeley.edu	AIX V.3	XL Fortran v1.1	16 MB	32	32	64
IBM RT-PC/125	loki.berkeley.edu	ACIS 4.3	F77 v1	4 MB	32	32	64
MIPS M/2000	mammoth.berkeley.edu	RISC/os 4.50B1	F77 v2.0	128 MB	32	32	64
MIPS M/1000	cassatt.berkeley.edu	UMIPS-BSD 2.1	F77 v1.21	16 MB	32	32	64
Decstation 3100	ylem.berkeley.edu	Ultrix 2.1	F77 v2.1	16 MB	32	32	64
Sparcstation I	genesis.berkeley.edu	SunOS R4.1	F77 v1.3	8 MB	32	32	64
Sun 3/50 (68881)	venus.berkeley.edu	UNIX 4.2 r.3.2	F77 v1	4 MB	32	32	64
Sun 3/50	baal.berkeley.edu	UNIX 4.2 r.3.2	F77 v1	4 MB	32	32	64
VAX 8600	vangogh.berkeley.edu	UNIX 4.3 BSD	F77 v1.1	28 MB	32	32	64
VAX 3200	atlas.berkeley.edu	Ultrix 2.3	F77 v1.1	8 MB	32	32	64
VAX-11/785	pioneer.arc.nasa.gov	Ultrix 3.0	F77 v1.1	16 MB	32	32	64
VAX-11/780	wilbur.arc.nasa.gov	UNIX 4.3 BSD	F77 v2	4 MB	32	32	64
Motorola M88K	rumble.berkeley.edu	UNIX R32.V1.1	F77 v2.0b3	32 MB	32	32	64
Amdahl 5840	prandtl.nas.nasa.gov	UTS V	F77 v2.0	32 MB	32	32	64

Table 2: Characteristics of the machines. The size of the data type implementations are in number of bits.

reasons for this were internal compiler errors, run time errors, or invalid results. Livermore Loops is an example of a program which executed in all machines except in the IBM RS/6000 530 where it gave a run time error. A careful analysis of the program reveals that the compiler is generating incorrect code. For three programs in the Perfect suite, the problems were mainly shortcomings in the programs. For example, *TRACK* gave invalid results in most of the workstations even after fixing a bug involving passing of a parameter; *MG3D* needed 95MB of disk space for a temporary file that few of the workstations had; *SPEC77* gave an internal compiler error on machines using MIPS Co. processors, and on the Motorola 88000 the program never terminated.

Our results show not only accurate predictions in general but also reproduce apparent ‘anomalies’, such as the fact that the CRAY Y-MP is 35% faster than the IBM RS/6000 for *QCD* but is slower for *MDG*. Note that because of the relative declarations used for precision, the Cray is actually computing results at twice the precision of the RS/6000. On CRAYs, the performance of double precision floating-point arithmetic is about ten times slower than single precision, because the former are emulated in software. Conversely, some workstations do all arithmetic in double (64-bit) precision. Therefore, the observed difference in relative performance between *QCD* and *MDG* can be easily explained by looking at their respective dynamic statistics. *QCD* executes in single precision, while *MDG* is a double precision benchmark.

In table 3 we summarize the accuracy of our run time predictions. The results show that 51% of all predictions fall within the 10% of the real execution times, and almost 79% are within 20%. Only 15 out of 244 predictions (6.15%) have an error of more than 30%. The results represent 244 program-machine combinations encompassing 18 machines and 28 programs. These results are very good if we consider that the characterization of machines and programs is done using a high level model.

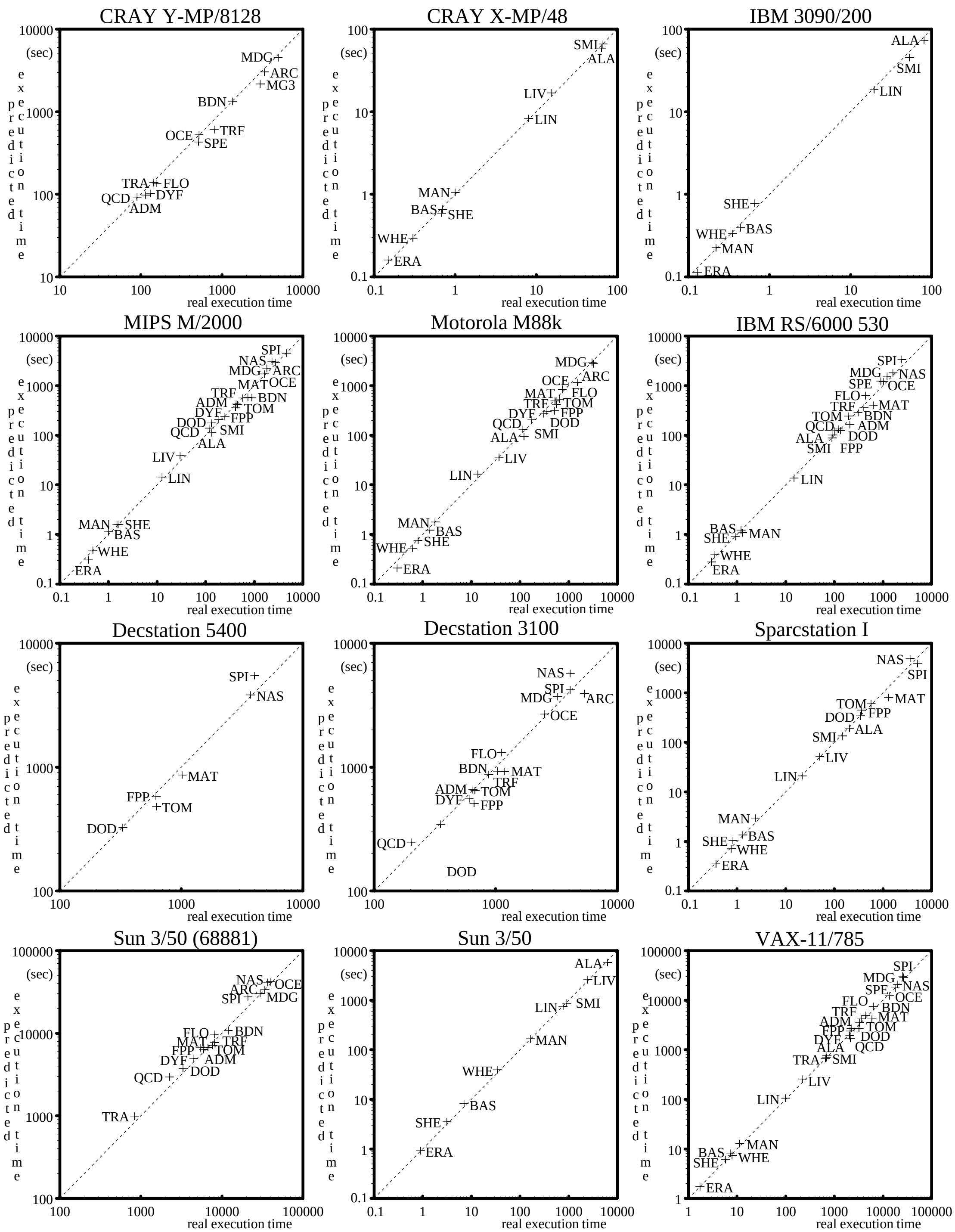


Figure 2: Comparison between real and predicted execution times. The predictions were computed using the program dynamic distributions and the machine characterizations. The vertical distance to the diagonal represents the predicted error.

Table 3: **Error distribution for execution time predictions**					
< 5 %	< 10 %	< 15 %	< 20 %	< 30 %	> 30 %
68 (27.9)	124 (55.5)	171 (70.1)	192 (78.7)	229 (93.9)	15 (6.15)

Table 3: Error distribution for the predicted execution times. For each error interval, we indicate the number of programs, from a total of 244, having errors that fall inside the interval (percentages inside parenthesis). The error is computed as the relative distance to the real execution time.

The maximum discrepancy in the predictions occurs for *MATRIX300*, which has an average error of −24.51% and a root mean square error of 26.36%. Our predictions for this program consistently underestimate the execution time on all machines because for this program the number of cache and TLB misses is significant; the model used for this paper does not consider this factor. In [Saav92a,c] we extend our model to include the effects of locality, and show that for programs with high miss ratios, run time predictions improve significantly. Because most of the benchmarks in the SPEC and Perfect suite tend to have low cache and TLB miss ratios [GeeJ91, GeeJ93], our other prediction errors do not have the same problem as for *MATRIX300*.

4.1. Single Number Performance

Although it may be misleading, it is frequently necessary or desirable to describe the performance of a given machine by a single number. In table 4 we present both the actual and predicted geometric means of the normalized execution times, and the percentage of error between them. We can clearly see from the results that our estimates are very accurate; in all cases the difference is less than 8%. In those cases for which they are available, we also show the SPECmark numbers; note that our results are for unoptimized code and the SPEC figures are for the best optimized results.

	Cray X–MP/48	IBM 3090/200	Amdahl 5840	Convex C-1	IBM RS/6000 530	Sparcstation I	Motorola 88k
SPECmark	N.A.	N.A.	N.A.	N.A.	28.90	11.80	15.80
actual mean	26.25	33.79	6.47	7.36	16.29	11.13	14.24
prediction	26.07	32.27	6.71	6.99	15.69	10.58	15.34
difference	+0.69%	−4.50%	+3.71%	−5.03%	−3.68%	−4.94	+7.72%

	MIPS M/2000	Dec 3100	VAX 8600	VAX−11/785	VAX−11/780	Sun 3/50	Average
SPECmark	17.60	11.30	N.A.	N.A.	1.00	N.A.	N.A.
actual mean	13.88	9.01	5.87	2.01	1.00	0.69	12.25
prediction	13.70	8.43	5.63	2.12	1.00	0.72	12.02
difference	−1.30%	−6.44	−4.09%	+5.47%	N.A.	+4.35%	−1.88%

Table 4: Real and predicted geometric means of normalized benchmark results. Execution times are normalized with respect to the VAX-11/780. For some machines we also show their published SPEC ratios. The reason why some of the SPECmark numbers are higher than either the real or predicted geometric means is because in contrast to our measurements the SPEC results are for optimized codes.

5. Program Characterization

There are several reasons why it is important to know in what way a given benchmark ‘uses’ a machine; i.e. which abstract operations the benchmark performs most frequently. That information allows us to understand the extent to which the benchmark may be considered representative, it shows how the program may be tuned, and indicates the goodness of the fit between the program and the machine. With our methodology, this information is provided by the dynamic statistics of the program.

5.1. Normalized Dynamic Distributions

The complete normalized dynamic statistics for all benchmarks, including the seven data sets for *SPICE2G6*, are presented in tables 16-25 in Appendix B. For each program[2] we give the fraction, with respect to the total, that each abstract operation is executed. Those AbOps that are executed less frequently than .01% are indicated by the entry < 0.0001. We also identify the five most executed operations of the program with a number in a smaller point size on the left of the corresponding entry.

The detailed counts of AbOps are too voluminous to provide an easy grasp of the results, so in figures 3-8 and 10-11, we summarize the results; the numbers on which those graphs are based are given in tables 26-32 of Appendix C.

5.2. Basic Block and Statement Statistics

Figure 3 shows the distribution of statements, classified into assignments, procedure calls, IF statements, branches, and DO loop iterations; also see tables 26-28 of Appendix C. On this and similar figures we cluster the benchmarks according to the similarity of their distributions. The cluster to which each benchmark belongs is indicated by a roman numeral at the top of the bar.

The results show that there are several programs in the Perfect suite whose distributions differ significantly from those of other benchmarks in the suite. In particular, programs *QCD*, *MDG*, and *BDNA* execute an unusually large fraction of procedure calls. A similar observation can be made in the case of IF statements for programs *QCD*, *MDG*, and *TRACK*. *TRACK* executes an unusually large number of branches.

The SPEC and Perfect suites have similar distributions. *SPICE2G6* using model *GREYCODE* and *DODUC* are two programs which execute a large fraction of IF statements and branches. In *GREYCODE*, 35% of all its statements are branches, and *DODUC* has a large number of IF statements. The distribution of statements also provides additional data. The distributions for programs *FPPPP* and *BDNA* are similar in the sense that both show a large fraction of assignments and a small fraction of DO loops. Consistent with this is the observation that the most important basic block in *FPPPP* contains more than 500 assignments.

In table 5 we give the average distributions of statements for the SPEC, Perfect Club, and small benchmarks. We also indicate the average over all programs. These numbers correspond to the average dynamic distributions shown in figure 3. It is worth observing from this data that although the Perfect Club methodology counts only FLOPS, not all of the benchmarks are dominated by floating point operations.

---

[2] In the rest of the paper, the term ''program'' refers to both the code and a particular set of data. Hence the same source code with a different input data is considered a different program.

Figure 3: Distribution of statements

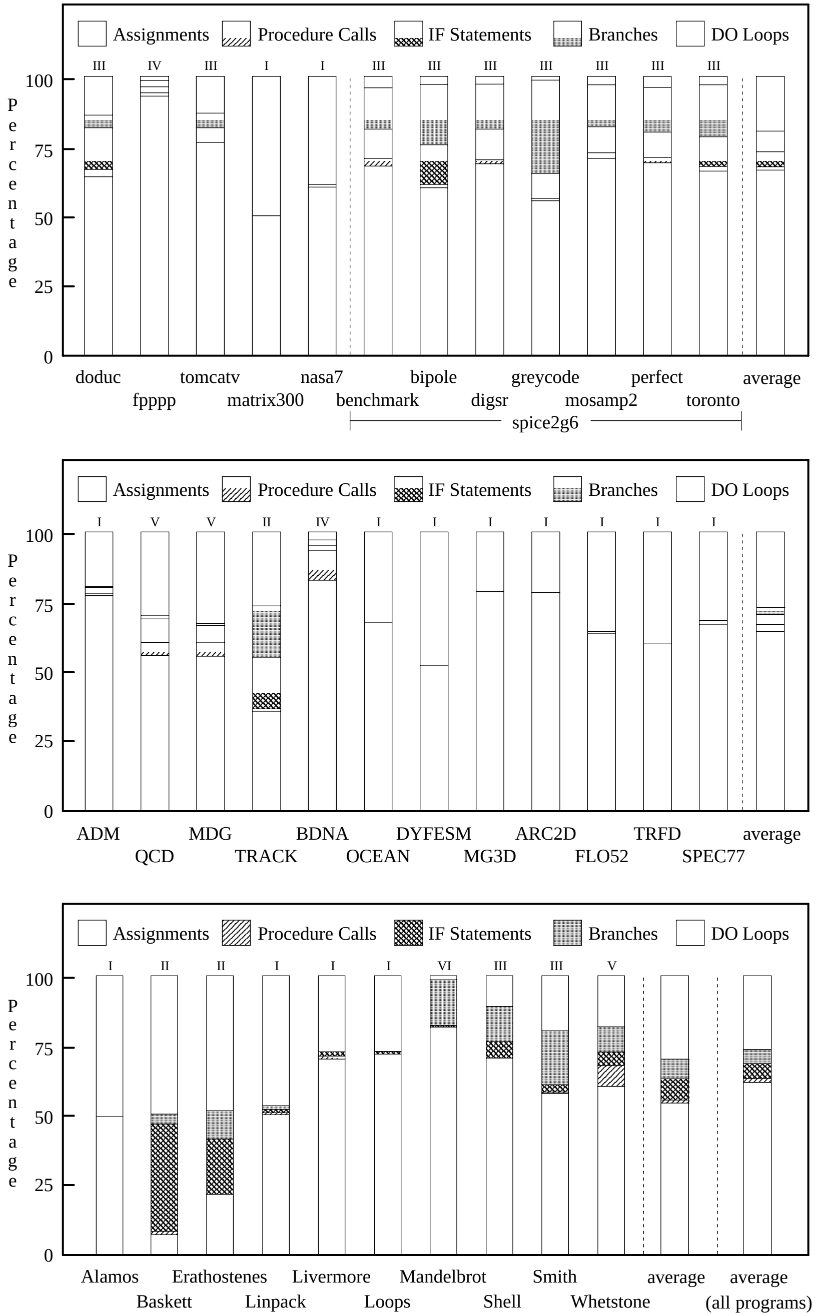
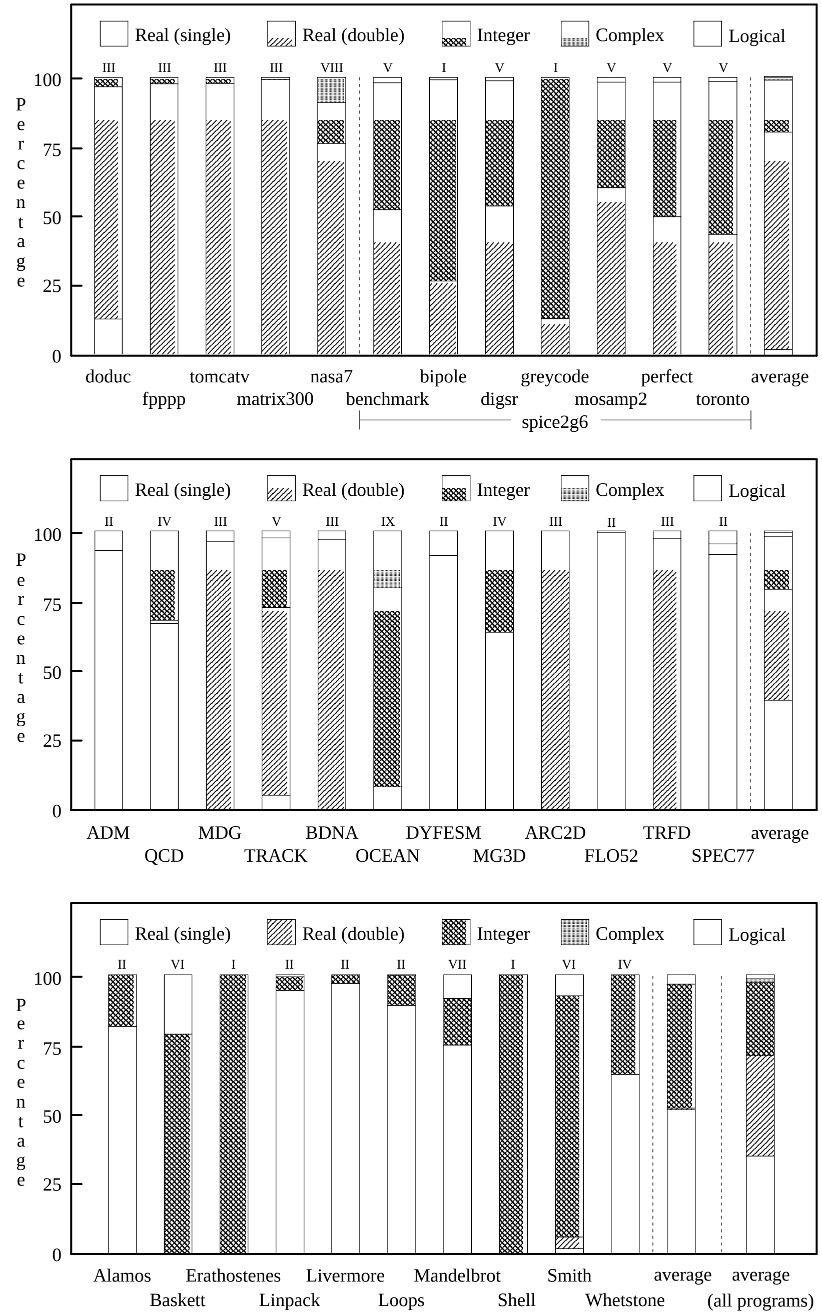


Figure 4: Distribution of operations



Figures 3 and 4: Distribution of statement types, and distribution of arithmetic and logical operations according to data type and precision. Bar *Loops* represents only the 24 computational kernels of benchmark *Livermore*, while ignoring the rest of the computation. Each bar is labeled with a roman numeral identifying those benchmarks with similar distributions. We give average distributions for each suite and for all programs. Of the seven models for *spice2g6*, only *greycode* and *perfect* are considered in the computation of the averages.

Distribution of Statements (average)

	SPEC	Perfect	Various	All Progs
Assignments	66.4 %	64.5 %	53.9 %	61.4 %
Procedure Calls	1.1 %	2.7 %	1.2 %	1.8 %
IF Statements	5.5 %	2.9 %	7.6 %	5.3 %
Branches	7.2 %	2.8 %	7.3 %	5.0 %
DO Loops	19.8 %	27.1 %	30.0 %	26.4 %

Table 5: Average dynamic distributions of statements for each of the suites and for all benchmarks.

5.3. Arithmetic and Logical Operations

Figures 4 and 5 depict the distribution of operations according to their type and what they compute; see also tables 29-31 (Appendix C). As it is clear from the graphs, for each program, operations on one or two data types are dominant. In this respect the Perfect benchmarks can be classified in the following way: *ADM*, *DYFESM*, *FLO52*, and *SPEC77* execute mainly floating-point single precision operators; *MDG*, *BDNA*, *ARC2D*, and *TRFD* floating-point double precision operators; *QCD* and *MG3D* floating-point single precision and integer operators; *TRACK* floating-point double precision and integer; and *OCEAN* integer and complex operators. These results further suggest the inadequacy of counting FLOPS as a performance measure. A similar classification can be obtained for the SPEC and the other benchmarks.

With respect to the distribution of arithmetic operators, figure 5 shows that the largest fraction correspond to addition and subtraction, followed by multiplication. Other operations like division, exponentiation and comparison are relatively infrequent.

Distribution of Operations (average)

	SPEC	Perfect	Various	All Progs
Real (single)	2.0 %	39.5 %	51.4 %	35.1 %
Real (double)	78.0 %	40.1 %	0.9 %	35.5 %
Integer	17.9 %	18.2 %	44.8 %	27.0 %
Complex	1.7 %	1.8 %	0.1 %	1.2 %
Logical	0.4 %	0.4 %	2.8 %	1.2 %

Distribution of Arithmetic Operators (average)

	SPEC	Perfect	Various	All Progs
Add/Subtract	52.6 %	52.4 %	50.0 %	51.7 %
Multiply	38.7 %	38.4 %	22.4 %	33.1 %
Quotient	1.9 %	2.4 %	1.3 %	1.9 %
Exponentiation	0.1 %	0.6 %	0.2 %	0.3 %
Comparison	6.7 %	6.2 %	25.9 %	12.9 %

Table 6: Average dynamic distributions of arithmetic and logical operations for each of the suites and for all benchmarks.

Figure 5: Distribution of operators

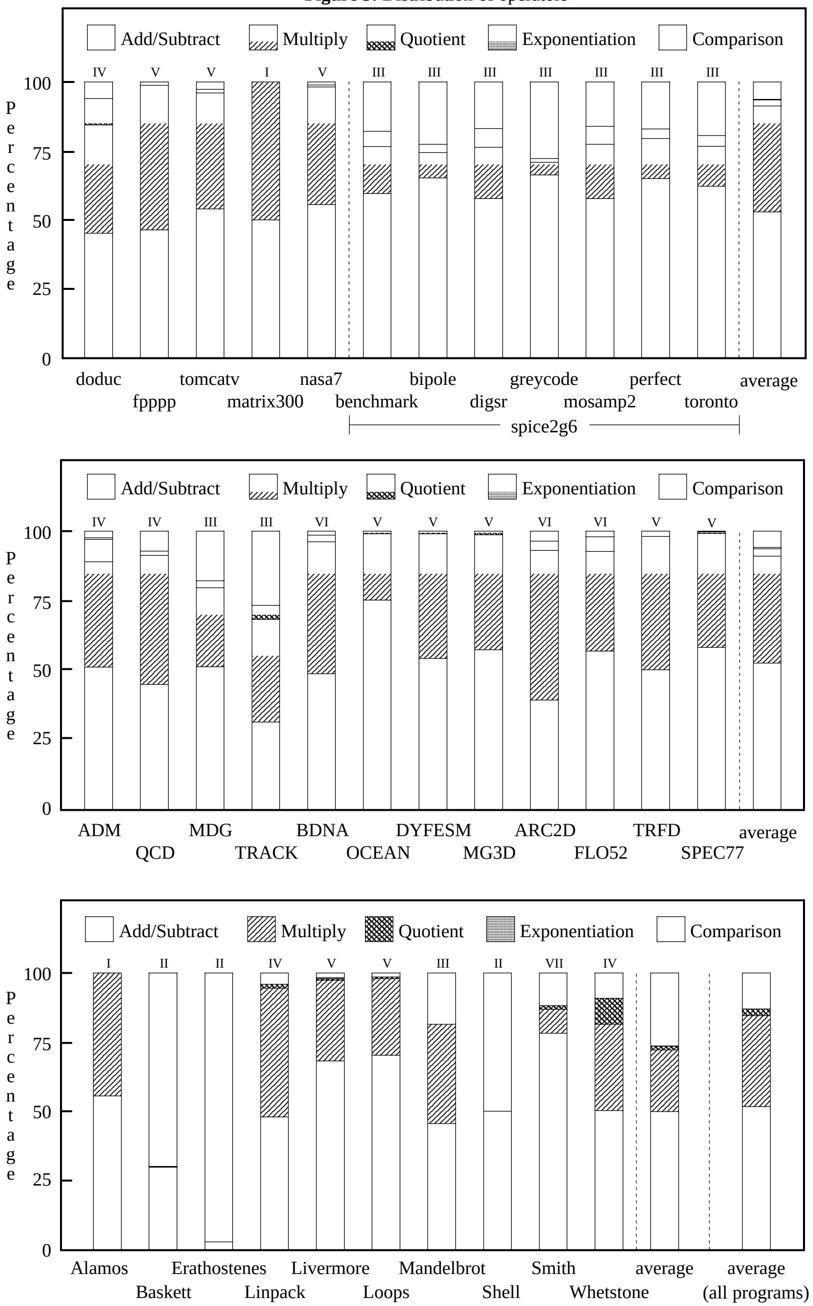
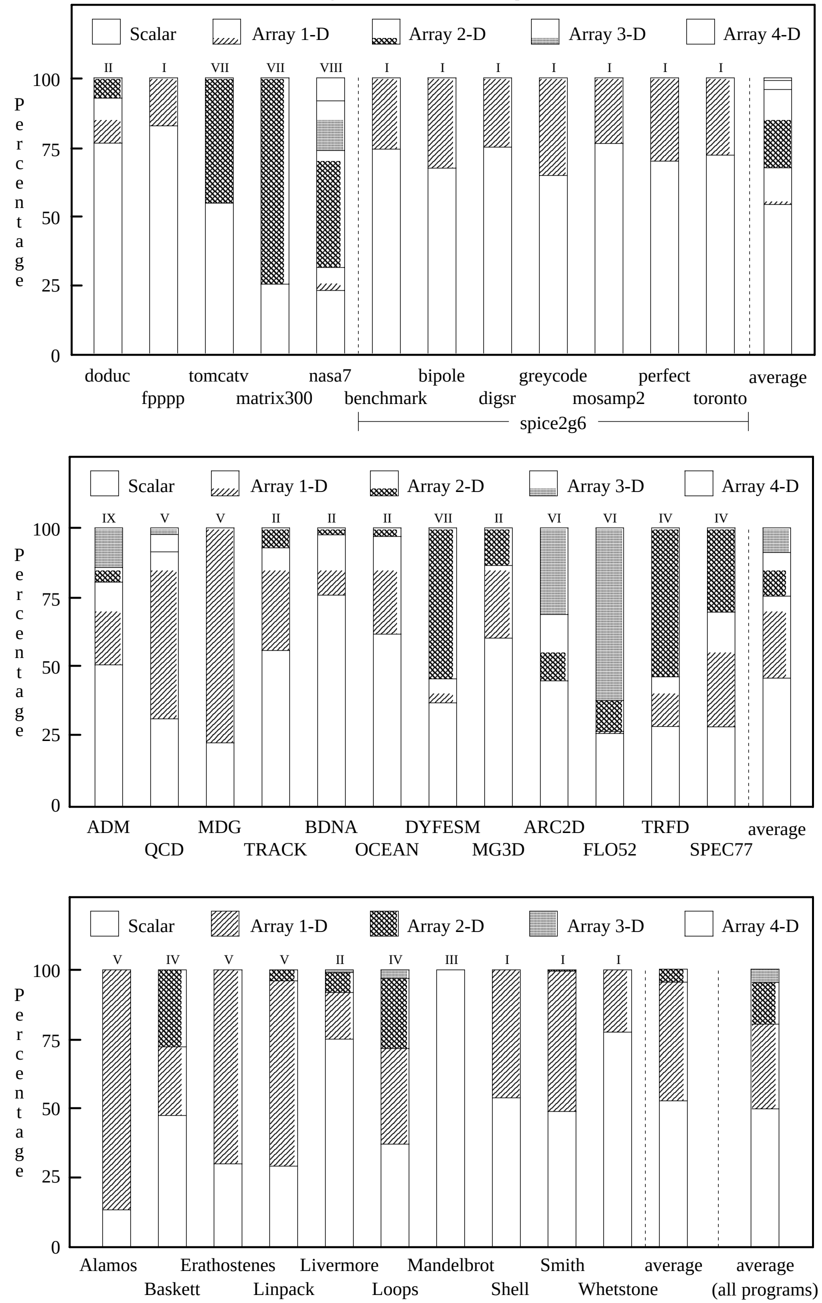


Figure 6: Distribution of operands



Figures 5 and 6: Distribution of operators and distribution of operands.

5.4. References to Array and Scalar Variables

Run time is affected by the need to compute the addresses of array data; no extra time is needed to reference scalar data. The frequencies of references to scalar and N-dimensional arrays are shown in figure 6. We can see that for most of the Perfect benchmarks, the proportion of array references is larger than for scalar references. The Perfect benchmark with the highest fraction of scalar operands is *BDNA*, and on the SPEC benchmarks, *DODUC*, *FPPPP*, and all models of *SPICE2G6* lean towards scalar processing. The distribution of the number of dimensions shows that on most programs a large portion of the references are to 1-dimensional arrays with a smaller fraction in the case of two dimensions. However, programs *ADM*, *ARC2D*, and *FLO52* contain a large number of references to arrays with 3 dimensions. *NASA7* is the only program which contains 4-dimensional array references.

Most compilers compute array addresses by calculating, from the indices, the offset relative to a base element; the base element (such as X(0,0,...0)) may not actually be a member of the array. If $X(i_1, i_2, \cdots, i_n)$ is an n-dimensional array reference, then its address ($ADDR$) is

$$ADDR[X(i_1, i_2, \cdots, i_n)] = ADDR[X(0, 0, \cdots, 0)] + Offset[X(i_1, i_2, \cdots, i_n)], \tag{3}$$

where

$$Offset[X(i_1, i_2, \cdots, i_n)] = B_{elem}((\cdots((i_n \cdot d_{n-1} + i_{n-1})\, d_{n-2} + i_{n-3}) \cdots)\, d_1 + i_1), \tag{4}$$

where $\{d_1, d_2, \cdots, d_n\}$ represents the set of dimensions and B_{elem} the number of bytes per element. Most compilers use the above equation when optimization is disabled, and this requires $n-1$ adds and $n-1$ multiplies. In scientific programs, array address computation can be a significant fraction of the total execution time. For example, in benchmark *MATRIX300* this can account, on some machines, for more than 60% of the unoptimized execution time. When using optimization, most array address computations are strength-reduced to simple additions; see [Saav92a] for how we handle that case.

The results in figure 6 show that the average number of dimensions in an array reference for the Perfect and SPEC benchmarks are 1.616 and 1.842 respectively. However, the probability that an operand is an array reference is greater in the Perfect benchmarks (.5437 vs. .4568).

Distribution of Operands (average)

	SPEC	Perfect	Various	All Progs
Scalar	54.0 %	45.7 %	52.5 %	49.8 %
Array 1-D	13.4 %	29.6 %	42.6 %	30.3 %
Array 2-D	28.1 %	15.5 %	4.8 %	14.7 %
Array 3-D	3.3 %	9.2 %	0.1 %	4.9 %
Array 4-D	1.2 %	0.0 %	0.0 %	0.2 %

Table 7: Average dynamic distributions of operands in arithmetic expressions for each of the suites and for all benchmarks.

Figure 7: Distribution of execution time (IBM RS/6000 530)

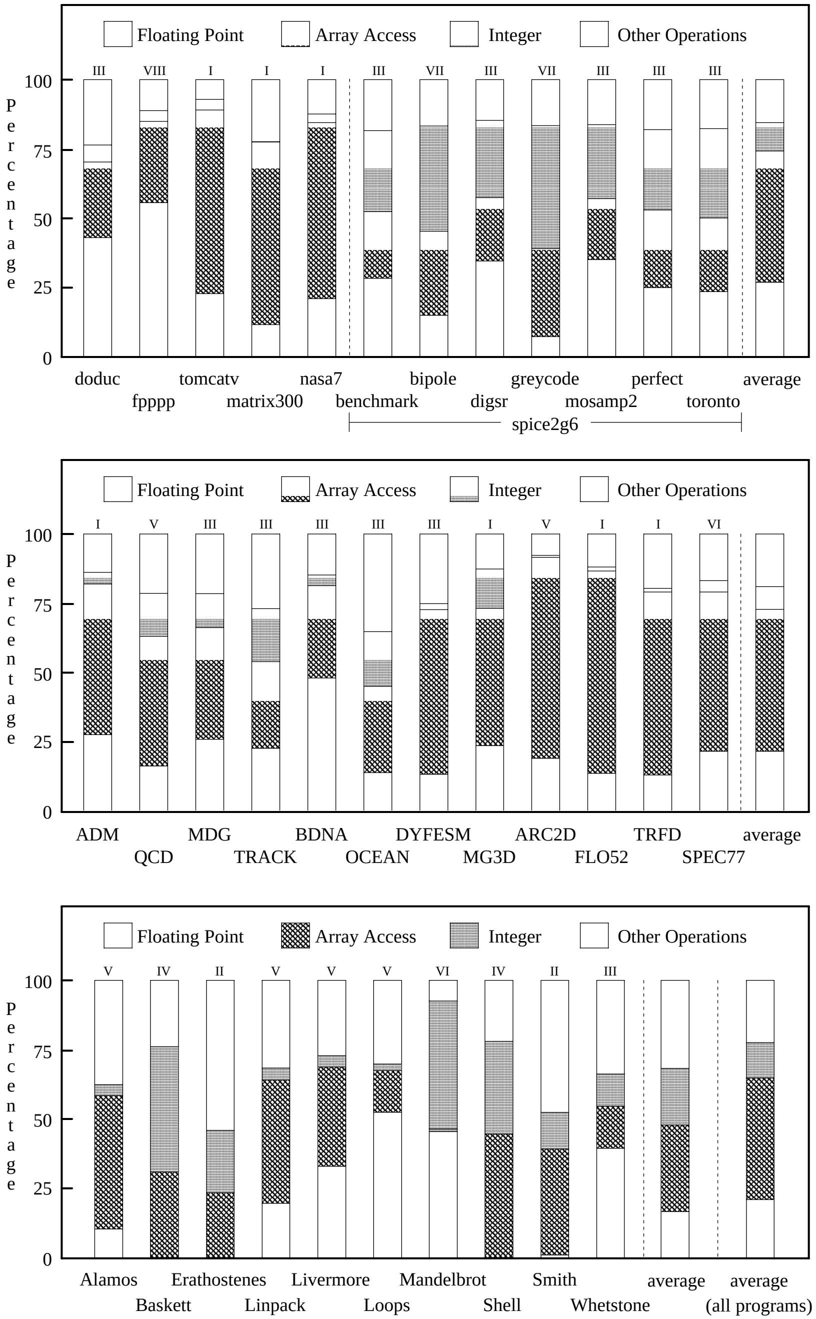
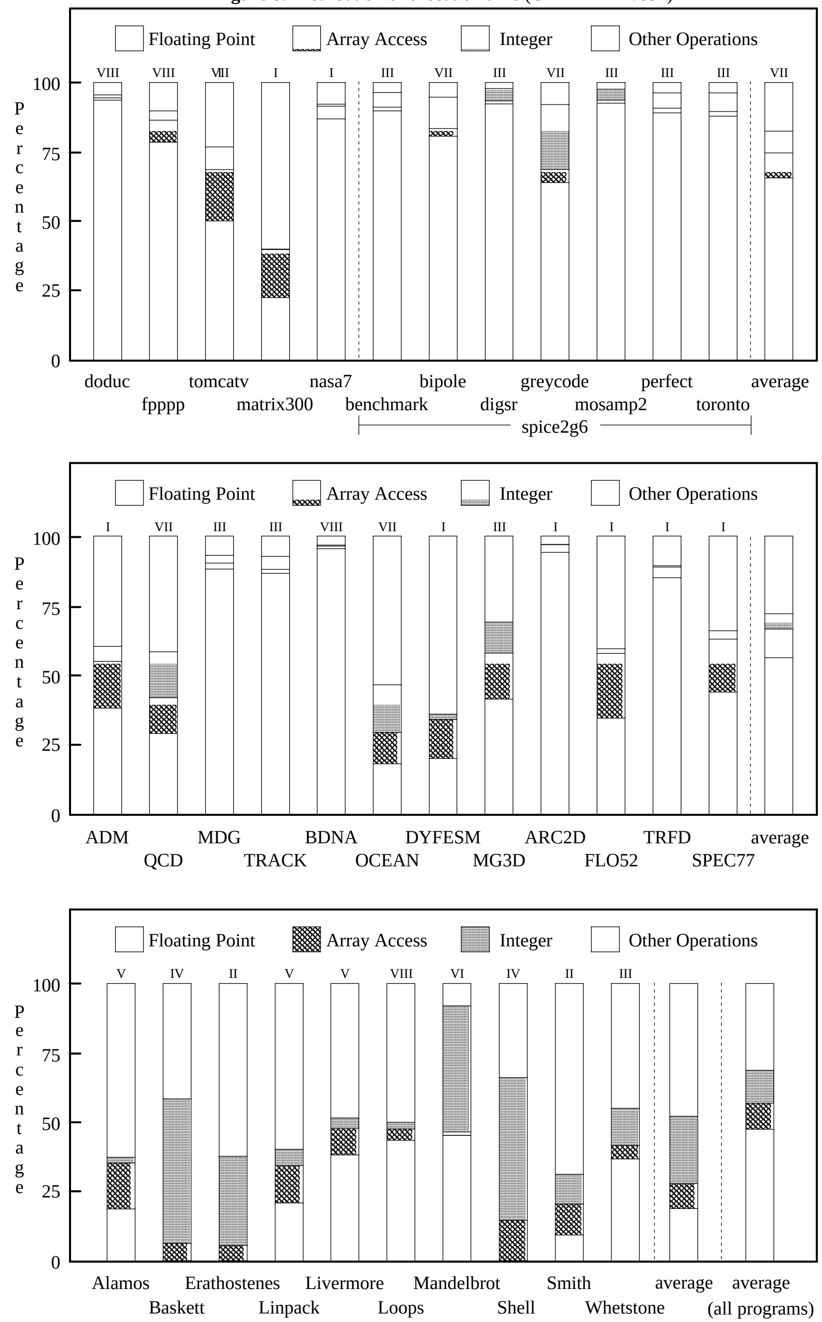


Figure 8: Distribution of execution time (CRAY Y-MP/832)



Figures 7 and 8: Distribution of execution time for the IBM RS/6000 530 and the CRAY Y-MP/832.

5.5. Execution Time Distribution

One of our most interesting measurements is the fraction of run time consumed by the various types of operations; this figure is a function of the program and the machine. As examples, in figures 7 and 8 we show the distribution of execution time for the IBM RS/6000 530 and CRAY Y-MP/832. We decompose the execution time in four classes: floating-point arithmetic, array access computation, integer and logical arithmetic, and other operations. All distributions were obtained using our abstract execution model, the dynamic statistics of the programs, and the machine characterizations.

Our previous assertion that scientific programs do more than floating-point computation is evident from figures 7 and 8. For example, programs *QCD*, *OCEAN*, and *DYFESM* spend more than 60% of their time executing operations that are not floating-point arithmetic or array address computation. This is even more evident for *GREYCODE*. Here less than 10% of the total time on the RS/6000 530 is spent doing floating-point arithmetic. The numerical values for each benchmark suite are given in table 8.

Distribution of Execution Time: IBM RS/6000 530 (average)

	SPEC	Perfect	Various	All Progs
Floating Point	26.64 %	21.33 %	16.61 %	20.94 %
Array Access	47.50 %	51.40 %	31.19 %	43.80 %
Integer	10.30 %	8.26 %	20.51 %	12.80 %
Other Operations	15.55 %	19.01 %	31.69 %	22.47 %

Distribution of Execution Time: CRAY Y-MP/832 (average)

	SPEC	Perfect	Various	All Progs
Floating Point	65.59 %	56.15 %	18.77 %	45.79 %
Array Access	9.36 %	10.42 %	9.10 %	9.75 %
Integer	5.98 %	5.45 %	24.26 %	11.84 %
Other Operations	19.07 %	27.98 %	47.87 %	32.63 %

Table 8: Average dynamic distributions of execution time for each of the suites and for all benchmarks on the IBM RS/6000 530 and the CRAY Y-MP/832.

From the figures, it is evident that the time distributions for the RS/6000-530 and the CRAY Y-MP are very different even when all programs are executed in scalar mode on both machines. On the average, the fraction of time that the CRAY Y-MP spends executing floating-point operations is 46%, which is significantly more than the 21% on the RS/6000. These results are very surprising, as the CRAY Y-MP has been designed for high performance floating point. As noted above, however, most of the benchmarks are double precision, which on the CRAY is 128-bits, and double precision on the CRAY is about 10 times slower than 64-bit single precision. This effect is seen clearly in programs: *DODUC*, *SPICE2G6*, *MDG*, *TRACK*, *BDNA*, *ARC2D*, and *TRFD*. Using our program statistics, however, we can easily compute the performance when all programs execute using 64-bit quantities on all machines. In this case, we compute that the fraction of time represented by floating-point operations on the CRAY Y-MP decreases to 29%, still higher than for the RS/6000. Note that this is an example of the power of our methodology- we are able to compute the performance of something which doesn’t exist.

The results also show the large fraction of time spent by the IBM RS/6000 in array address computation. One example is program *FLO52*, which makes extensive use of 3-dimensional arrays. In contrast, the distributions of *MANDELBROT* and *WHETSTONE* clearly show that these are a scalar codes completely dominated by floating-point computation. Remember, however, that our statistics correspond to unoptimized programs. With optimization, the fraction of time spent computing array references is smaller, as optimizers in most cases replace most array address computations with a simple add by precomputing the offset between two consecutive element of the array. This corresponds to applying strength reduction and backward code motion.

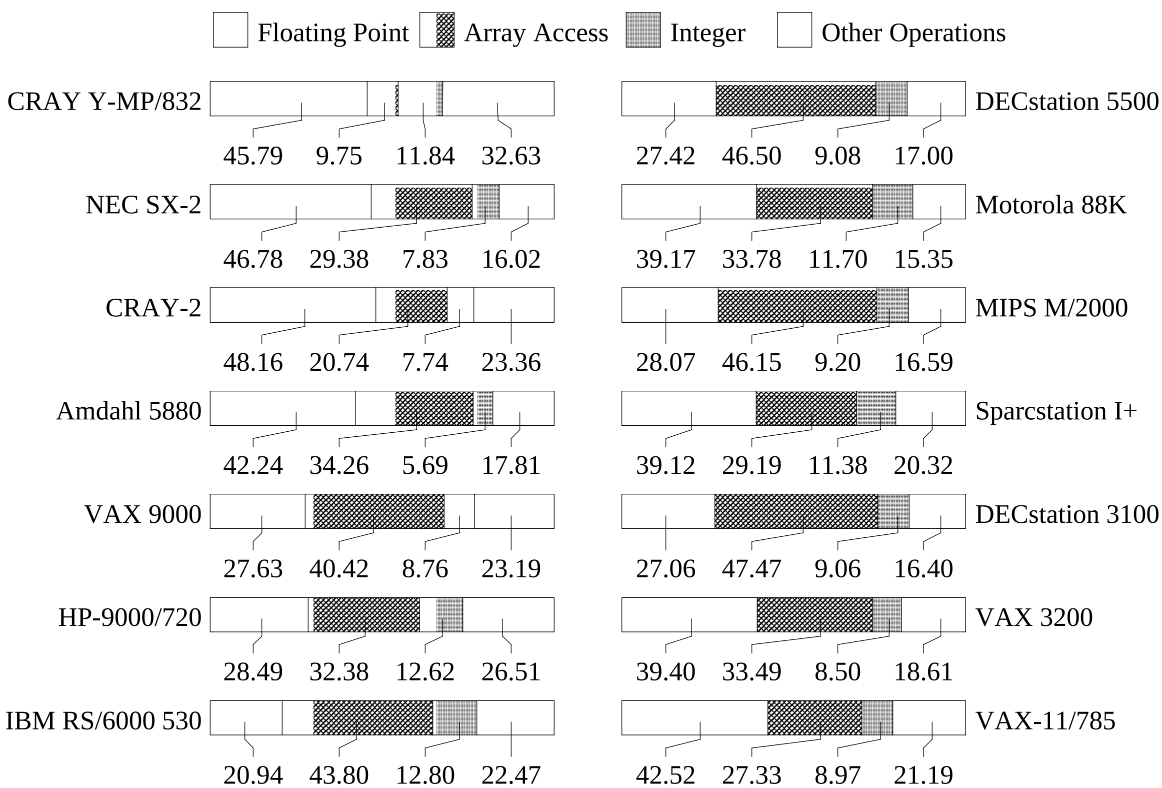


Figure 9: Average time distributions. The distributions are computed over all programs. Of the seven models for *spice2g6*, only *greycode* and *perfect* are considered in the computation of the averages.

In figure 9 we show the overall average time distribution for several of the machines. In the case of the supercomputers (CRAY Y-MP, NEC SX-2, and CRAY-2), single and double precision correspond to 64 and 128 bits. The results show that on the VAX 9000, HP-9000/720, RS/6000 530, and machines based on the R3000/R3010 processors, the floating-point contribution is less than 30%. The contribution of address array computation varies from 8% on the CRAY Y-MP to 47% on the DECstation 3100, DECstation 5500, and MIPS M/2000. The contribution of integer operations exhibit less variation, ranging from 6 to 13%.

Figure 10: Distribution of basic blocks

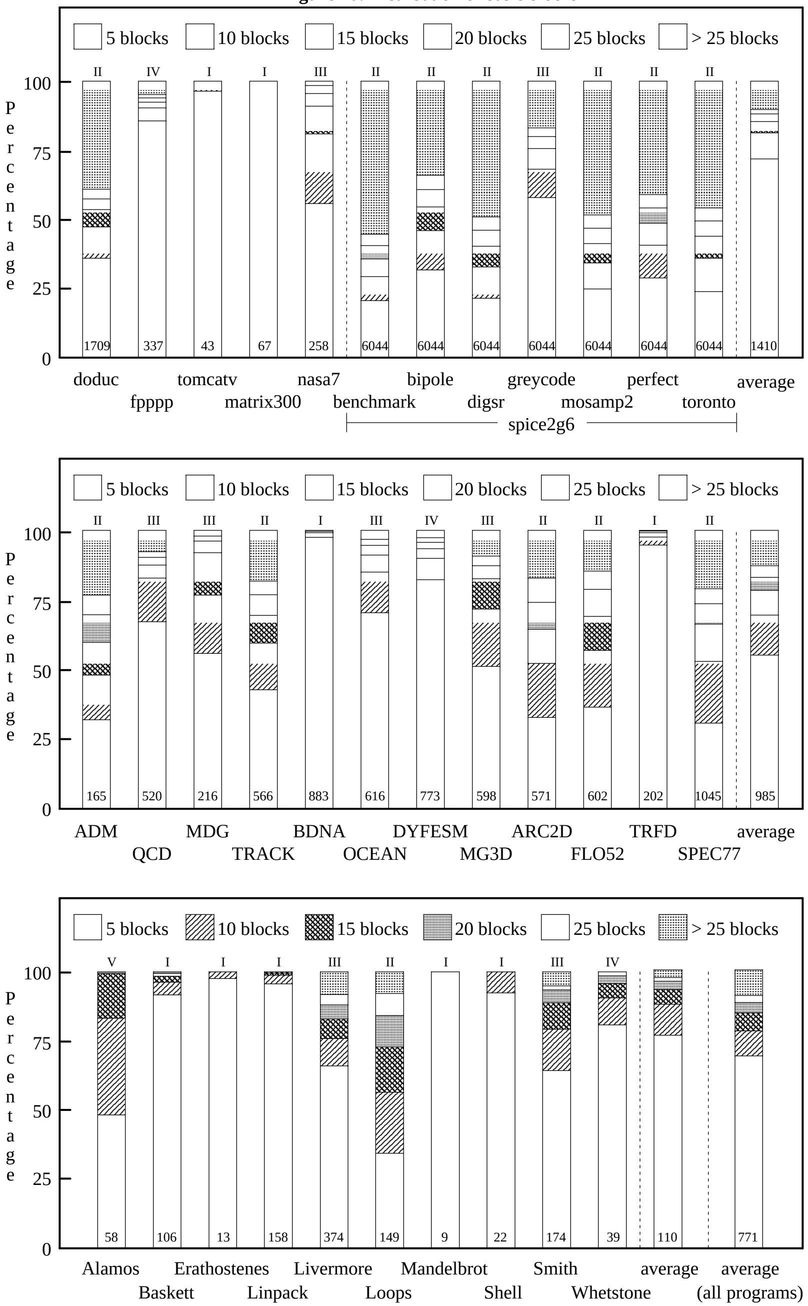
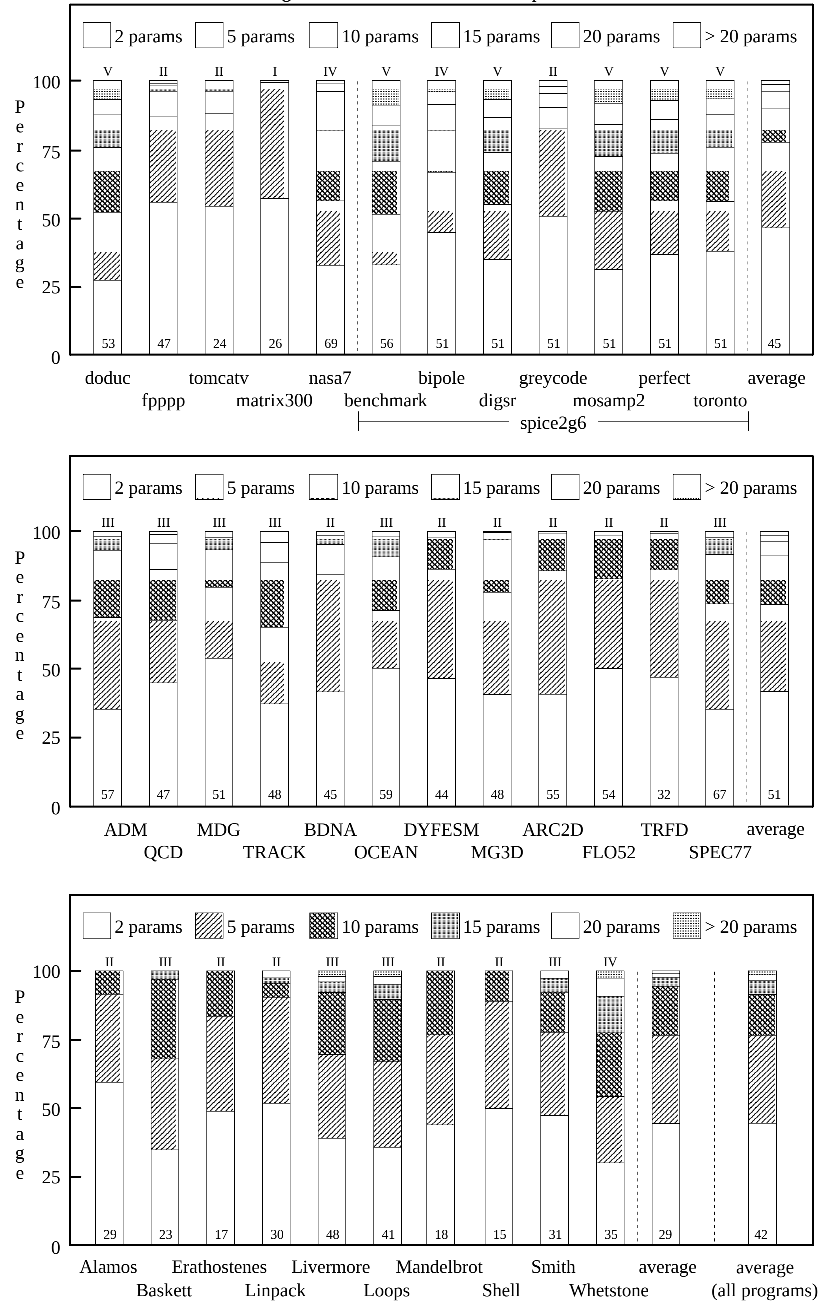


Figure 11: Distribution of abstract parameters



Figures 10 and 11: Portion of all basic block executions accounted for by 5 most frequent, 10 most frequent, etc. Also portion of all AbOp (parm) executions accounted for by 2 most frequent, 5 most frequent, etc.

Above, we noted that we could compute the running time for a machine that didn't exist - a CRAY which did double precision in 64 bits. This is a very simple example of an extremely powerful application of our evaluation methodology. We can define an arbitrary synthetic machine, i.e. a "what if" machine, by setting the AbOps to whatever values we desire, and then determine the performance of that machine for a given workload. For example, we could estimate the effect of very fast floating point, or slow loads and stores.

5.6. Dynamic Distribution of Basic Blocks

Figure 10 shows the fraction of basic block executions accounted for by the 5, 10, 15, 20, and 25 most frequently executed basic blocks. (A basic block is a segment of code executed sequentially with only one entry and one exit point.) There is an implicit assumption among benchmark users that a large program with a long execution time represents a more difficult and 'interesting' benchmark. This argument has been used to criticize the use of synthetic and kernel-based benchmarks and has been one of the motivations for using real applications in the Perfect and SPEC suites. However, as the results of figure 10 show, many of the programs in the Perfect and SPEC suites have very simple execution patterns, where only a small number of basic blocks determine the total execution time. The Perfect benchmark results show that on programs *BDNA* and *TRFD* the 5 most important blocks account for 95% of all operations, from a total of 883 and 202 blocks respectively. Moreover, on seven of the Perfect benchmarks, more than 50% of all operations are found in only 5 blocks. The same observation can be made for the SPEC benchmarks. In fact, *MATRIX300* has one basic block containing a single statement that amounts for 99.9% of all operations executed. On the average, five blocks account for 55.45% and 71.85% of the total time on the Perfect and SPEC benchmarks.

Distribution of Basic Blocks (average)

	SPEC	Perfect	Various	All Progs
1–5 blocks	72.1 %	55.0 %	76.8 %	66.1 %
6–10 blocks	9.1 %	14.6 %	10.8 %	12.1 %
11–15 blocks	3.9 %	8.3 %	5.1 %	6.3 %
16–20 blocks	2.7 %	4.9 %	2.9 %	3.7 %
21–25 blocks	1.9 %	4.5 %	1.8 %	3.0 %
> 25 blocks	10.3 %	12.7 %	2.6 %	8.8 %

Table 9: Portion of basic block executions accounted for by 5 most frequent, 6-10'th most frequent, etc, for each of the suites and for all benchmarks.

5.6.1. Quantifying Benchmark Instability Using Skewness

When a large fraction of the execution time of a benchmark is accounted for by a small amount of code, the relative running time of that benchmark may vary widely between machines depending on the execution time of the relevant AbOps on each machine; i.e. the benchmark results may be 'unstable.' We describe the extent to which the execution time is concentrated among a small number of basic blocks or AbOps as the degree of *skewness* of the benchmark. (This is not the same as the statistical coefficient of skewness, but the concept is the same.) We define our skewness metric for basic blocks as $1/\bar{X}$, where $\bar{X} = \sum_{i=1}^{\infty} j \cdot p(j)$, where $p(j)$ is the frequency of the j'th most frequently executed basic block.

program	Skewness	program	Skewness
01 Matrix300	0.983	15 Nasa7	0.155
02 Mandelbrot	0.790	16 MDG	0.145
03 Linpack	0.637	17 Smith	0.136
04 BDNA	0.567	18 QCD	0.133
05 Tomcatv	0.535	19 Livermore	0.132
06 Baskett	0.466	20 MG3D	0.108
07 Erathostenes	0.452	21 Spice2g6	0.084
08 TRFD	0.405	22 FLO52	0.078
09 Shell	0.385	23 ARC2D	0.073
10 DYFESM	0.250	24 TRACK	0.073
11 Whetstone	0.229	25 SPEC77	0.065
12 Fpppp	0.201	26 ADM	0.060
13 OCEAN	0.171	27 Doduc	0.049
14 Alamos	0.162		

Table 10: Skewness of ordered basic block distribution for the SPEC, Perfect and Small benchmarks. The skewness is defined to be the inverse of the mean of the distribution.

Table 10 gives the amount of skewness of the basic blocks for all programs. The results show that *MATRIX300*, *MANDELBROT*, and *LINPACK* are the ones with the largest skewness.

5.6.2. Optimization and *MATRIX300*

One of the reasons to detect unstable, or highly skewed, programs, is that optimization efforts may easily be concentrated on the relevant code. Such focussed optimization efforts may make a given program unsuitable for benchmarking purposes. Benchmark *MATRIX300* is a clear example of this situation; not only is its amount of skewness very high, but recent SPEC results on this program put in question its effectiveness as a benchmark. For example, in [SPEC91a], the SPECratio of the CDC 4330 (a machine based on the MIPS 3000 microprocessor) on *MATRIX300* was reported as 15.7 with an overall SPECmark of 18.5, but in [SPEC91b] the SPECratio and SPECmark jumped to 63.9 and 22.4. A similar situation exists for the new HP-9000 series 700. On the HP-9000/720, the SPECratio of *MATRIX300* has been reported at 323.2, which is more than 4 times larger than the second largest SPECratio [SPEC91b]! Furthermore, if the SPECratio for *MATRIX300* is ignored in the computation of the SPECmark, the overall performance of the machine decreases 21%, from 59.5 to 49.3.

The reason behind these dramatic performance improvements is that these machines use a pre-processor to inline three levels of routines and in this way expose the matrix multiply algorithm, which is the core of the computation in *MATRIX300*. The same pre-processor then replaces the algorithm by a library function call which implements matrix multiply using a blocking (tiling) algorithm. A *blocking algorithm* is one in which the algorithm is performed on sub-blocks of the matrices which are smaller than the cache, thus significantly reducing the number of cache and TLB misses. *MATRIX300* uses matrices of size 300x300, which are much larger than current cache sizes. Non-blocking matrix multiply algorithms generate $O(N^3)$ misses, when the order of the matrices is larger than the data cache size, while a blocking algorithm generates only $O(N^2)$ misses.

5.6.3. How Effective Are Benchmarks?

There are two aspects to consider when evaluating the effectiveness of a CPU benchmark. The first has to do with how well the program exercises the various functional units and the pipeline, while the other refers to how the program behaves with respect to the memory system. A program which executes many different sequences of instructions may be a good test of the pipeline and functional units, but not necessarily of the memory system [Koba83, Koba84]. The Livermore Loops is one example. It consists of 24 small kernels. Each kernel is executed many times in order to obtain a meaningful observation. Since each kernel does not touch more than 2000 floating-point numbers, all of its data sits comfortably in most caches. Thus, after the first iteration the memory system is not tested. Furthermore, the kernels consist of few instructions, so they even fit in very small instruction caches.

SPEC results for the IBM RS/6000 530 clearly show how performance is affected by the demands of the benchmark on the memory system. For example, benchmark *MATRIX300* is dominated by a single statement that the IBM Fortran compiler can optimize, by decomposing it into a single multiply-add instruction. The SPECratio of the IBM RS/6000 530 on this program, however, is lower than the overall SPECmark. In contrast, the SPECratio on program *TOMCATV* is 2.6 times larger than the SPECmark, although the principal basic blocks are more complex than on *MATRIX300*. The main difference between the main basic blocks of these two programs is the number of memory requests per floating-point operation executed. On *MATRIX300* on average there is one read for every floating-point operation and there is very little re-use of registers; the machine is thus memory speed limited for this benchmark. Studies on the SPEC benchmarks [Pnev90, GeeJ91] show that most of these programs have low miss ratios for cache configurations which are normal on existing workstations. The effect of the memory system on run times is considered further in [Saav92b].

Distribution of Abstract Parameters (average)

	SPEC	Perfect	Various	All Progs
2 params	46.3 %	41.3 %	44.0 %	43.3 %
5 params	31.2 %	31.7 %	32.5 %	31.9 %
10 params	12.3 %	17.7 %	18.1 %	16.7 %
15 params	6.5 %	5.7 %	3.0 %	5.0 %
20 params	2.5 %	2.3 %	1.9 %	2.0 %
> 20 params	1.2 %	1.3 %	0.5 %	1.0 %

Table 11: Portion of AbOp executions accounted for by 2 most frequent, 5 most frequent, etc, for each of the suites and for all benchmarks.

5.7. Distribution of AbOps

Figure 11 shows the cumulative distribution of abstract operations (AbOps) for the different benchmark suites. Each bar indicates at the bottom the number of different AbOps operations executed by the benchmark. The results show that most programs execute only a small number of different operations, with *MATRIX300* as an extreme example. The averages for the three suites and for all programs are presented in table 11. We can also compute the skewness of the ordered distribution of AbOps in the same way as we did with basic blocks, i.e. as the inverse of the expected value of the distribution; the results are shown in table 12. The programs with the largest values of skewness are *MATRIX300*, *ALAMOS*, and *ERATHOSTENES*. The results also show that *DODUC* is the SPEC benchmark with the lowest amount of skewness both in the distribution of basic blocks and AbOps.

program	Skewness
01 Matrix300	0.405
02 Alamos	0.400
03 Erathostenes	0.367
04 Shell	0.353
05 Tomcatv	0.341
06 TRFD	0.325
07 Fpppp	0.315
08 Linpack	0.309
09 DYFESM	0.296
10 ARC2D	0.286
11 Mandelbrot	0.279
12 Baskett	0.263
13 MDG	0.256
14 MG3D	0.255
15 Smith	0.254
16 BDNA	0.251
17 Spice2g6	0.248
18 FLO52	0.243
19 OCEAN	0.217
20 SPEC77	0.215
21 Livermore	0.213
22 ADM	0.210
23 QCD	0.200
24 TRACK	0.180
25 Nasa7	0.169
26 Whetstone	0.155
27 Doduc	0.139

Table 12: Skewness of ordered abstract operation distribution for the SPEC, Perfect and Small benchmarks. The skewness is defined to be the inverse of the mean of the distribution.

5.7.1. Characterizing the Ordered Distribution of Abstract Operations

It has been argued that for an average program the distribution of the most executed operations (blocks) is geometric [Knut71]. What this means is that the most executed operation of the program accounts for an α fraction of the total, the second for α of the residual, that is, $\alpha \cdot (1-\alpha)$, and so on. Therefore, the cumulative distribution can be approximated by $f(n) = 1 - K(1-\alpha)^n$, where n represents the n-th most executed operations, and K and α are constants. The n-th residual is given by $(1-\alpha)^n$. Thus, the cumulative distribution at point n is one minus the n-th residual.

In figure 12 we show the fitted and actual average distributions for each suite and for all programs; as may be seen, the geometric distribution is a good fit. Figure 12 clearly shows that, on the average, three operations account for 55-60% of all operations and five operations for almost 75%. Thus, most programs consist of a small number of different operations, each executed many times. These operations, however, are not the same in all benchmarks.

5.8. The *SPICE2G6* Benchmark

In this section, we discuss in more detail the differences between the seven data sets used for the *SPICE2G6* benchmark. *SPICE2G6* is normally considered, for performance purposes, to be a good example of a large CPU-bound scalar double precision floating-point benchmark, with a small fraction of complex arithmetic and negligible vectorization. Given its large size (its code and data sizes on a VAX-11/785 running ULTRIX are 325 Kbytes and 8 Mbytes respectively), it might be expected to be a good test for the instruction and data caches. The SPEC suite uses, as input, a time consuming bipolar circuit model called *GREYCODE*, while the Perfect Club uses a PLA circuit called *PERFECT*. *GREYCODE* was

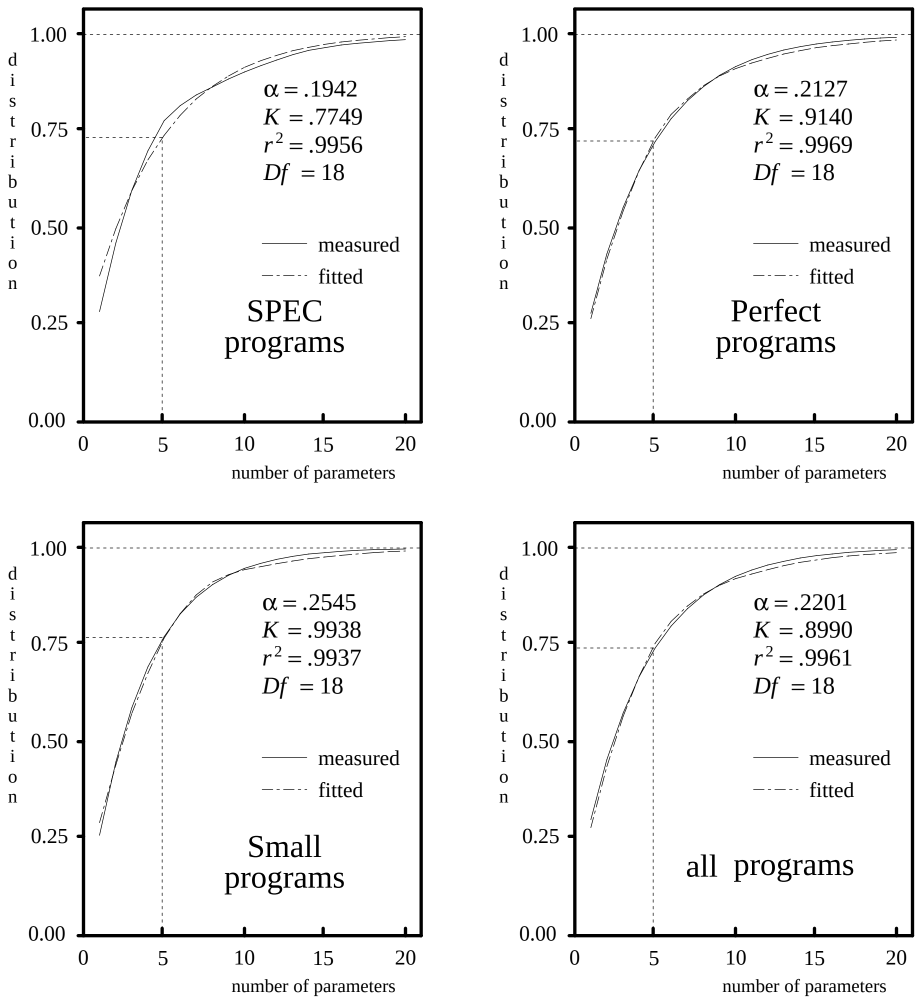


Figure 12: Fitted and actual cumulative distributions as a function of the n most important abstract operations executed by each benchmark. Equation $1 - K(1-\alpha)^n$ is used to fit the actual distributions. In addition to α and K, each graph indicates the values of the coefficient of correlation and the number of degrees of freedom. All coefficients of correlation are significant at the 0.9995 level.

selected mainly because of its long execution time, but we shall see that its execution behavior is not typical, nor does it measure what *SPICE2G6* is believed to measure.

Table 26 (see Appendix C) gives the general statistics for the seven data models of *SPICE2G6*. The results show that the number of abstract operations executed by *GREY-CODE* (2.005×10^{10}) is almost two orders of magnitude larger than the maximum on any of the other models (3.184×10^{8}). For *GREYCODE*, however, only 33% of all basic blocks are executed. In contrast, the number of basic blocks touched by *BENCHMARK* is 52%. Another abnormal feature of *GREYCODE* is that it has the lowest fraction of assignments executed (60%), and of these only 19% are arithmetic expressions; the rest represent simple memory-to-memory operations. In the other models, assignments amount, on the average, to 70% of all statements, with arithmetic expressions being more than 35% of the total. Another distinctive feature of *GREYCODE* is the small fraction of procedure calls (2.8%)

and the very large number of branches (36%) that it executes.

More significant are the results in figure 4. The distribution of arithmetic and logical operations shows that *GREYCODE* is mainly an integer benchmark; almost 87% of the operations involve addition and comparison between integers. On the other models the percentage of floating-point operations is never less than 26% and it reaches 60% for *MOSAMP2*.

The reason why *GREYCODE* executes so many integer operations and so few basic blocks can be found in the following basic block.

```
140   LOCIJ = NODPLC (IRPT + LOCIJ)
      IF (NODPLC (IROWNO + LOCIJ) .EQ. I) GO TO 155
      GO TO 140
```

This and two other similar integer basic blocks account for 50% of all operations. The data structures used in *SPICE2G6* were not designed to handle large circuits, so most of the execution time is spent traversing them. In contrast, in the case of *BENCHMARK*, *DIGSR*, and *PERFECT*, the ten most executed blocks account for less than 35% of all operations and most of these consist of floating-point operations. The three integer blocks on *GREYCODE* represent more than 41% of the execution time on a VAX 3200 and 26% on a CRAY Y-MP/8128. These statistics suggest that *GREYCODE* is not an adequate benchmark for testing scalar double precision arithmetic. Much better input models for *SPICE2G6* are *BENCHMARK*, *DIGSR*, or *PERFECT*.

6. Measuring Similarity Between Benchmarks

A good benchmark suite is representative of the ‘real’ workload, but there is little point to filling a benchmark suite with several programs which provide similar loads on the machine. In this section we address the problem of measuring benchmark similarity by presenting two different metrics for program similarity and comparing them. One is based on the dynamic statistics that we presented earlier. The rationale behind this metric is that we expect that programs which execute similar operations will tend to produce similar runtime results. The other metric works from the other end; benchmarks which yield proportional performance on a variety of machines should be considered to be similar.

Our results show that the two metrics are highly correlated; what is similar by one measure is generally similar by the other. Note that the first metric is easier to compute (we only have to measure each benchmark, rather than run it on each machine), and would thus be preferred.

6.1. Program Similarity Metric Based on Dynamic Statistics

To simplify the benchmark characterization and clustering, we have grouped the 109 AbOps into 13 ‘reduced parameters’, each of which represents some aspect of machine implementation; these parameters are listed in table 13. Note that the reduced parameters presented here are not the same as those used in [Saav89]; the ones presented here better represent the various aspects of machine architecture. As we would expect for a language like Fortran, most of the parameters correspond to floating-point operations. Others are integer arithmetic, logical arithmetic, procedure calls, memory bandwidth, and intrinsic functions. Integer and floating-point division are assigned to a single parameter. AbOps that change the flow of execution, branches and DO loop instructions, are also assigned to a single parameter.

Reduced Parameters

1	memory bandwidth	8	division
2	integer addition	9	logical operations
3	integer multiplication	10	intrinsic functions
4	single precision addition	11	procedure calls
5	single precision multiplication	12	address computation
6	double precision addition	13	branches and iteration
7	double precision multiplication		

Table 13: The thirteen reduced parameters used in the definition of program similarity. Each parameter represents a subset of basic operations, and its value is obtained by adding all contributions to the dynamic distribution. Integer and floating point division are merged in a single parameter.

The formula we use as metric for program similarity is the squared Euclidean distance, where every dimension is weighted according to the average run time accounted for by that parameter, averaged over the set of all programs. Let $\mathbf{A} = < A_1, \cdots, A_n >$ and $\mathbf{B} = < B_1, \cdots, B_n >$ be two vectors containing the reduced statistics for programs A and B, then the distance between the two programs ($d(\mathbf{A}, \mathbf{B})$) is given by

$$d(\mathbf{A}, \mathbf{B}) = \sum_{i=1}^{n} W_i \, (A_i - B_i)^2 \tag{5}$$

where W_i is the value of parameter i averaged over all machines.

We computed the similarity distance between all program pairs; see table 36 of the Appendix E for the 50 pairs with the largest and smallest differences. We included all programs, but only the *GREYCODE* and *PERFECT* input data sets for *SPICE2G6*. The average distance between all programs is 1.1990 with a standard deviation of 0.8169. Figure 13 shows the clustering of programs according to their distances. Pairs of programs having distance less than 0.4500 are joined by a bidirectional arrow. The thickness of the arrow is related to the magnitude of the distance. The most similar programs are *TRFD* and *MATRIX300* with a distance of only 0.0172. In the next five distances we find the pairwise relations between programs *DYFESM*, *LINPACK*, and *ALAMOS*. Programs *TRFD*, *MATRIX300*, *DYFESM*, and *LINPACK* have similarities that go beyond their dynamic distributions. These four programs have the property that their most executed basic blocks are syntactic variations of the same code (SAXPY), which consists in adding a vector to the product between a constant and a vector, as shown in the following statement:

```
X(I,J) = X(I,J) + A * Y(K,I) .
```

Note that IBM RS/6000 has a special instruction to speed up the execution of these types of statements. In that machine, a multiply-add instruction takes four arguments and performs a multiply on two of them, adds that product to the third argument, and leaves the result in the fourth argument. By eliminating the normalization and round operations between the multiply and add, the execution time of this operation is significantly reduced compared to a multiply followed by an add [Olss90].

Three clusters are present in figure 13. One, with eight programs and containing *LINPACK* as a member, includes those programs that are dominated by single precision floating-point arithmetic. Another cluster, also having eight programs, contains those benchmarks dominated by double precision floating-point arithmetic. There is a subset of programs in this cluster containing programs *TRFD*, MATRIX300, *NASA7*, *ARC2D*, and *TOMCATV*, which form a 5-node complete subgraph. All distances between pairs of elements are smaller than 0.4500. The smallest cluster, with three elements, contains those programs with significant integer and floating-point arithmetic. We also include in the diagram those programs whose smallest distance to any other program is larger than 0.4500. These are represented as isolated nodes with the value of the smallest distance indicated below the name.

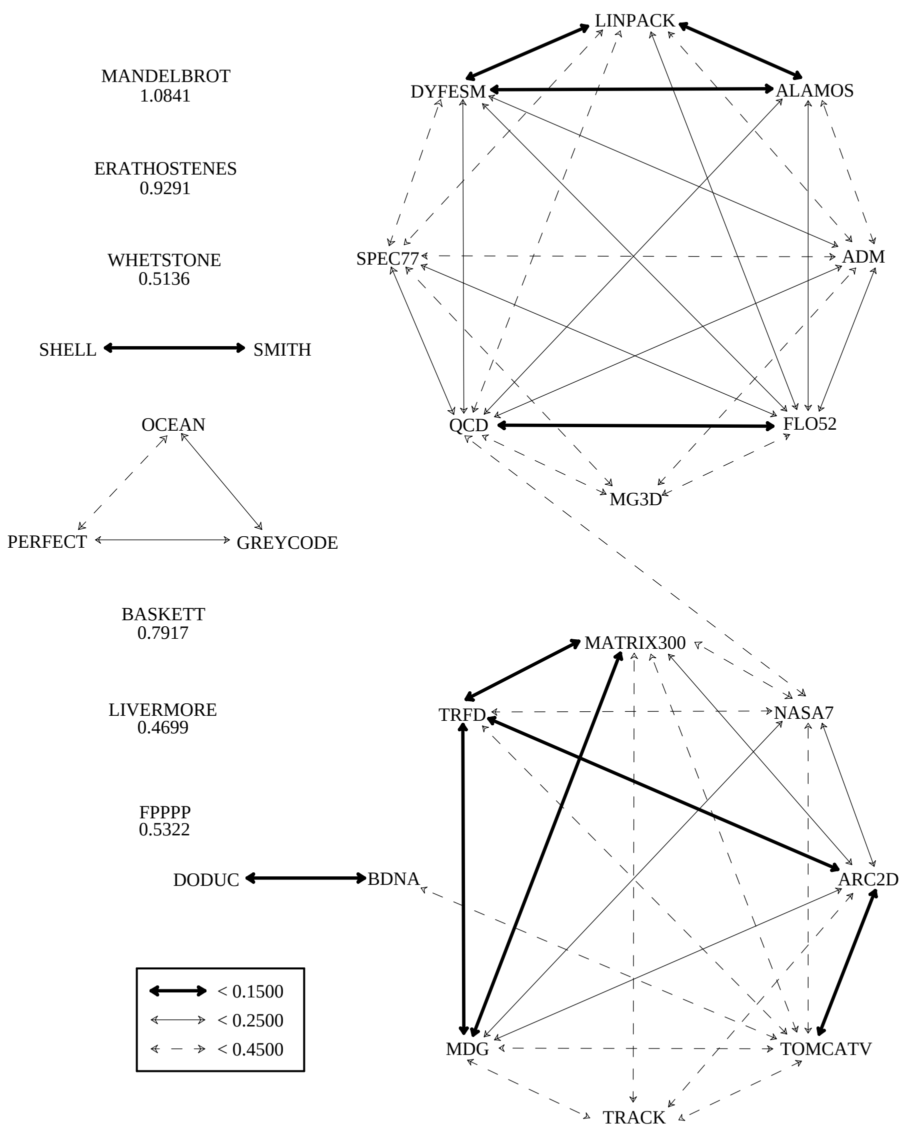


Figure 13: Principal clusters found in the Perfect, SPEC, and Small benchmarks. Distance is represented by the thickness of the arrow. Programs whose smallest distance to any other program is greater than 0.45 show under their name the magnitude of their smallest distance.

6.1.1. Minimizing the Benchmark Set

The purpose of a suite of benchmarks is to represent the target workload. Within that constraint, we would like to minimize the number of actual benchmarks. Our results thus far show: (a) Most individual benchmarks are highly skewed with respect to their generation of abstract operations, (b) but the clusters shown in figure 13 suggest that subsets of the suites test essentially the same aspects of performance. Thus, an acceptable variety of benchmark measurements could be obtained with only a subset of the programs analyzed earlier. A still better approach would be to run only one benchmark, our machine characterizer. Note that since the machine characterizer measures the run time for all AbOps, it is possible to accurately estimate the performance of any characterized machine for any AbOp distribution, without having to run any benchmarks. Such an AbOp distribution can be chosen as the weighted sum of some set of existing benchmarks, as an estimate of some target or existing workload or in any other manner.

6.2. The Amount of Skewness in Programs and the Distribution of Errors

Earlier, as discussed in sections §5.6 and §5.7, we noted that many of the benchmarks concentrate their execution on a small number of AbOps. We would expect that our predictions of running time for benchmarks with highly skewed distributions of AbOp execution would show greater errors than those with less skewed distributions. This follows directly from the assumption that our errors in measuring AbOp times are random; there will be less cancellation of errors when summing over a small number of large values than a larger number of small values. (This can be explained more rigorously by considering the formula for the variance of a sum of random variables.)

We tested the hypothesis that prediction errors for programs with a skewed distribution of either basic blocks or abstract operations will tend to be larger than for those with less skewed distributions. The scattergrams for both distributions are shown in figure 17 (Appendix E). An examination of that figure shows that there is no correlation between prediction error and the skewness of the frequency of basic block execution. There is a small amount of correlation between the skewness of the AbOp execution distribution and the prediction error. This lack of correlation seems to be due to two factors: (a) those programs with the most highly skewed distributions emphasize AbOps such as floating point, for which measurement errors are small. (b) prediction errors are mostly due to other factors (e.g. cache misses), rather than errors in the measurement of AbOp execution times.

6.3. Program Similarity and Benchmark Results

Our motivation in proposing a metric for program similarity in §6.1 was to identify groups of programs having similar characteristics; such similar programs should show proportional run times on a number of different machines. In this section, we examine this hypothesis.

First, we introduce the concept of benchmark equivalence.

Definition: If t_{A,M_i} is the execution time of program A on machine M_i, then two programs are *benchmark equivalent* if, for any pair of machines M_i and M_j, the following condition is true

$$\frac{t_{A,M_i}}{t_{A,M_j}} = \frac{t_{B,M_i}}{t_{B,M_j}}, \tag{6}$$

i.e. the execution times obtained using program A differ from the execution times using program B, on all machines, by a multiplicative factor k

$$\frac{t_{A,M_i}}{t_{B,M_i}} = k \qquad \text{for any machine } M_i. \tag{7}$$

It is unlikely that two different programs will exactly satisfy our definition of benchmark equivalence. Therefore, we define a weaker concept, that of *execution time similarity*, to measure how far two programs are from full equivalence. Given two sets of benchmark results, we define the execution time similarity of two benchmarks by computing the coefficient of variation of the variable $z_{A,B,i} = t_{A,M_i}/t_{B,M_i}$[3]. The coefficient of variation measures how well the execution times of one program can be inferred from the execution times of the other program.

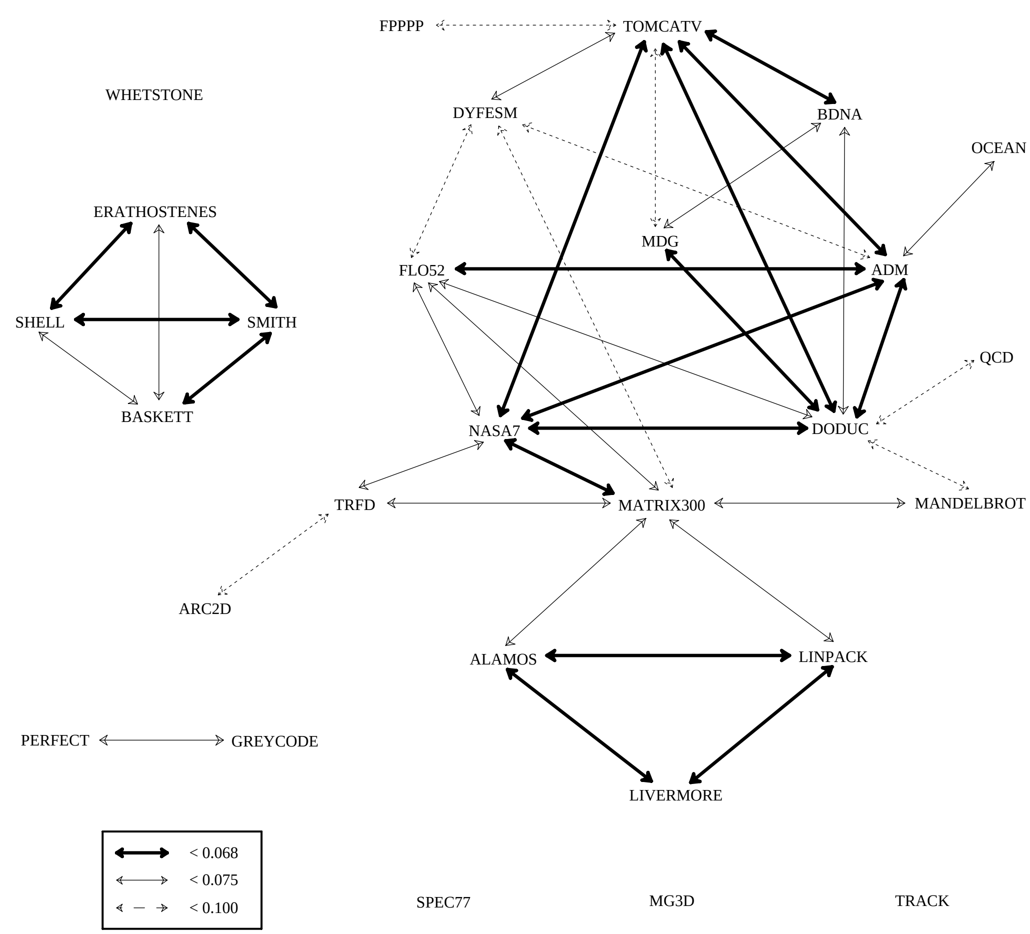


Figure 14: Principal clusters found in the Perfect, SPEC, and Small benchmarks using the run time similarity metric. Distance is represented by the thickness of the arrow.

[3] Programs that are benchmark equivalent will have zero as their coefficient of variation.

As we did in §6.1, we present in table 37 (Appendix E) the 50 most and least similar programs, using here the coefficient of variation as metric computed from the execution times (see figure 17, Appendix E). In figure 14 we show a clustering diagram similar to the one presented in figure 13. The diagram shows three well-defined clusters. One contains basically the integer programs: *SHELL*, *ERATHOSTENES*, *BASKETT*, and *SMITH*. Another cluster is formed by *MATRIX300*, ALAMOS, *LIVERMORE*, and *LINPACK*. The largest cluster is centered around programs *TOMCATV*, ADM, *DODUC*, *FLO57*, and *NASA7*, with most of the other programs connected to these clusters in an unstructured way.

Now that we have defined two different metrics for benchmark similarity, one based on program characteristics (see §6.1), and the other based on execution time results, we can compare the two metrics to see if there exists a good correlation in the way they rank pairs of programs. We measure the level of significance using the Spearman's rank correlation coefficient ($\hat{\rho}_s$), which is defined as

$$\hat{\rho}_s = 1 - \frac{6 \sum_{i=1}^{n} {d_i}^2}{n^3 - n}, \tag{8}$$

where d_i is the difference of ranking of a particular pair on the two metrics. For our two similarity metrics the coefficient $\hat{\rho}_s$ indicates that there is a correlation at a level of significance which is better than 0.00001.[4]

A scattergram of the two metrics is given in figure 15; each point. The horizontal axis corresponds to the metric based on the dispersion of the execution time results while the vertical axis correspond to the metric based on dynamic program statistics. Each "+" on the graph represents a pair of benchmark programs. The results indicate that there is a significant positive correlation between the two metrics at the level of 0.0001. Visually, we can see that the two metrics correlate reasonable well. What this means is that if two benchmarks differ widely in the AbOps that they use most frequently, the chances are that they will give inconsistent performance comparisons between pairs of machines (relative to other benchmarks), and conversely. That is, if benchmarks A and B are quite different, benchmark A may rate machine X faster than Y, and benchmark B may rate Y faster than X. This suggests that our measure of program similarity is sufficiently valid that we can use it to eliminate redundant benchmarks from a large set.

6.4. Limitations Of Our Model

There are some limitations in our linear high-level model and in using software experiments to characterize machine performance. Here we briefly mention the most important of them. For a more in-depth discussion see [Saav88,89,92a,b,c].

The main sources of error in the results from our model can be grouped in two classes. The first corresponds to elements of the machine architecture which have not been captured by our model. The model described here does not account for cache or TLB misses; an extension to our model is presented in [Saav92a,c] which adds this factor. We do not

---

[4] In computing the rank correlation coefficient we use the same set of program pairs for both metrics. The number of pairs for which there was enough benchmark results to compute the coefficient of variation is only half the total number of pairs.

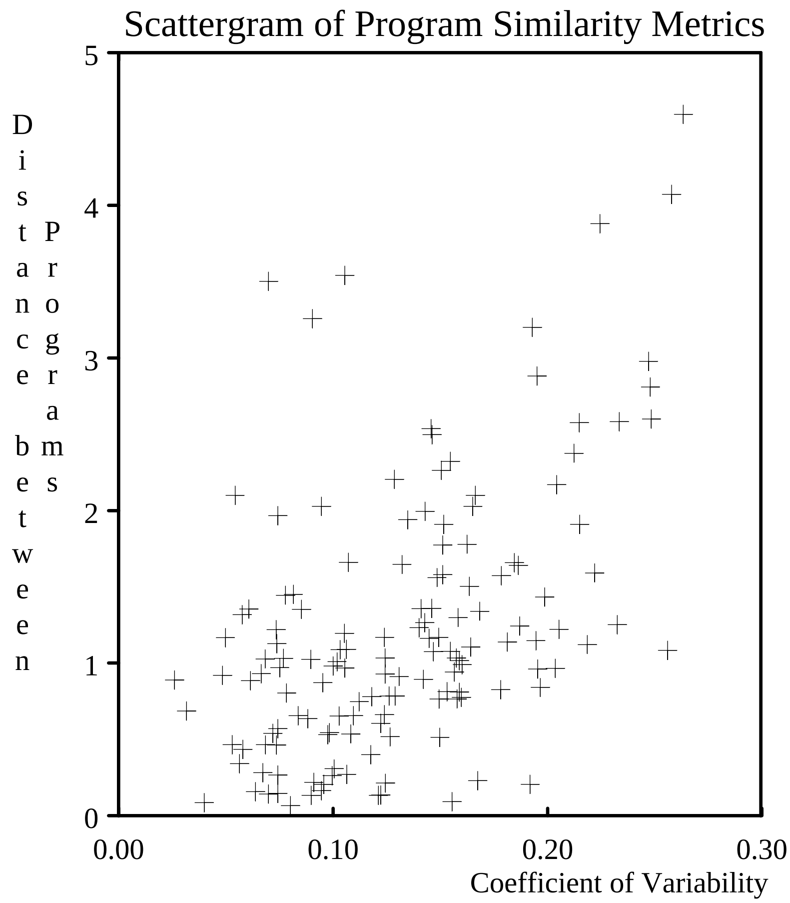


Figure 15: Scattergram of the two program similarity metrics. The horizontal axis corresponds to the metric computed from benchmark execution times, while the one on the vertical axis is computed from dynamic program statistics. The results exhibit a significant positive correlation.

successfully capture aspects of machine architecture which are manifested only by the performance of certain sequences of AbOps, and not by a single AbOp in isolation - e.g. the IBM RS/6000 multiply-add instruction; we discuss this further below. We are not able to account for hardware or software interlocks, non-linear interactions between consecutive machine instructions [Clap86], the effectiveness of branch prediction [Lee84], and the effect on timing of branch distance and direction. We have also not accounted for specialized architectural features such as vector operations and vector registers.

Another source of errors corresponds to limitations in our measuring tools and factors independent from the programs measured: resolution and intrusiveness of the clock, random noise, and external events (interrupts, page faults, and multiprogramming) [Curr75].

It is also important to mention that the model and the results presented here reflect only unoptimized code. As shown in [Saav92b], our model can be extended with surprising success to the prediction of the running times of optimized codes.

It is worth making specific mention of recent trends in high performance microprocessor computer architecture. The newest machines, such as the IBM RS/6000 [Grov90], can issue more than one instruction per cycle; such machines are called either Superscalar or VLIW (very long instruction word), depending on their design. The observed level of performance of such machines is a function of the actual amount of concurrency that is achieved. The level of concurrency is itself a function of which operations are available to be executed in parallel, and whether those operations conflict in their use of operands or functional units. Our model considers abstract operations individually, and is not currently able to determine the achieved level of concurrency. Much of this concurrency will also be manifested in the execution of our machine characterizer; i.e. on a machine with concurrency, we will measure faster AbOp times. Thus on the average we should be able to predict the overall level of speedup. Unfortunately, this accuracy on the average need not apply to predictions for the running times of individual programs. In fact this is what we observed in the case on the IBM RS/6000 530. In this machine the standard deviation of the errors is 21 percent, which is the largest for all machines. Furthermore, the results on the RS/6000 also gives the maximum positive and negative errors (–35.9% and 44.0%). Note that although these errors are larger than for the other machines, our overall predictions are still quite accurate.

The other ‘‘new’’ technique, superpipelining, doesn’t introduce any new difficulties. Superpipelining is a specific type of pipelining in which one or more individual functional units are pipelined; for example, more than one multiply can be in execution at the same time. Superpipelining introduces the same problems as ordinary pipelining, in terms of pipeline interlocks, and functional unit and operand conflicts. Such interlocks and conflicts can only be analyzed accurately at the level of a model of the CPU pipeline.

7. Summary and Conclusions

In this paper we have discussed program characterization and execution time prediction in the context of our abstract machine model. These two aspects of our methodology allows us to investigate the characteristics of benchmarks and compute accurate execution time estimates for arbitrary Fortran programs. The same approach could be used for other algebraic languages with different characteristics than Fortran. In most cases, however, a larger number of parameters will be needed and some special care should be taken in the characterization of library functions whose execution is input-dependent, e.g., string library functions in C.

There are a number of results from and applications of our research: (1) Our methodology allows us to analyze the behavior of individual machines, and identify their strong and weak points. (2) We can analyze individual benchmark programs, determine what operations they execute most frequently, and accurately predict their running time on those machines which we have characterized. (3) We can determine "where the time goes", which aids greatly in tuning programs to run faster on specific machines. (4) We can evaluate the suitability of individual benchmarks, and of sets of benchmarks, as tools for evaluation. We can identify redundant benchmarks in a set. (5) We can estimate the performance of proposed workloads on real machines, of real workloads on proposed machines, and of proposed workloads on proposed machines.

As part of our research, we have presented extensive statistics on the SPEC and Perfect Club benchmark suites, and have illustrated how these can be used to identify deficiencies in the benchmarks.

Related work appears in [Saav92b], in which we extend our methodology to the analysis of optimized code, and in [Saav92c], in which we extend our methodology to consider cache and TLB misses. See also [Saav89], which concentrates on machine characterization.

Acknowledgements

We would like to thank K. Stevens, Jr. and E. Miya for providing access to facilities at NASA Ames, as well as David E. Culler and Luis Miguel who let us run our programs in their machines. We also thank Vicki Scott from MIPS Co. who assisted us with the SPEC benchmarks, and Oscar Loureiro and Barbara Tockey who made useful suggestions.

Bibliography

[Alle87] Allen, F., Burke, M., Charles, P., Cytron, R., and Ferrante J., ‘‘An Overview of the PTRAN Analysis System for Multiprocessing.’’, *Proc. of the Supercomputing ’87 Conf.*, 1987.

[Bala89] Balasundaram, V., Kennedy, K, Kremer, U., McKinley, K., and Subhlok, J., ‘‘The ParaScope Editor: an Interactive Parallel Programming Tool’’, *Proc. of the Supercomputing ’89 Conf.*, Reno, Nevada, November 1989.

[Bail85] Bailey, D.H., Barton, J.T., ‘‘The NAS Kernel Benchmark Program’’, NASA Technical Memorandum 86711, August 1985.

[Bala91] Balasundaram, V., Fox, G., Kennedy, K, and Kremer, U., ‘‘A Static Performance Estimator to Guide Data Partitioning Decisions’’, *Third ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog.*, Williamsburg, Virginia, April 21-24 1991, pp. 213-223.

[Beiz78] Beizer, B., *Micro Analysis of Computer System Performance*, Van Nostrand, New York, 1978.

[Clap86] Clapp, R.M., Duchesneau, L., Volz, R.A., Mudge, T.N., and Schultze, T., ‘‘ Toward Real-Time Performance Benchmarks for ADA’’, *Comm. of the ACM*, Vol.29, No.8, August 1986, pp. 760-778.

[Curn76] Curnow H.J., and Wichmann, B.A., ‘‘A Synthetic Benchmark’’, *The Computer Journal*, Vol.19, No.1, February 1976, pp. 43-49.

[Curr75] Currah B., ‘‘Some Causes of Variability in CPU Time’’, *Computer Measurement and Evaluation*, SHARE project, Vol. 3, 1975, pp. 389-392.

[Cybe90] Cybenko, G., Kipp, L., Pointer, L., and Kuck, D., *Supercomputer Performance Evaluation and the Perfect Benchmarks*, University of Illinois Center for Supercomputing R&D Tech. Rept. 965, March 1990.

[Dodu89] Doduc, N., ‘‘Fortran Execution Time Benchmark’’, *paper in preparation*, Version 29, March 1989.

[Dong87] Dongarra, J.J., Martin, J., and Worlton, J., ‘‘Computer Benchmarking: paths and pitfalls’’, *Computer*, Vol.24, No.7, July 1987, pp. 38-43.

[Dong88] Dongarra, J.J., ‘‘Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment’’, *Comp. Arch. News*, Vol.16, No.1, March 1988, pp. 47-69.

[GeeJ91] Gee, J., Hill, M.D., Pnevmatikatos, D.N., and Smith A.J., ‘‘Cache Performance of the SPEC Benchmark Suite’’, *submitted for publication*, also UC Berkeley, Tech. Rept. No. UCB/CSD 91/648, October 1991.

[GeeJ93] Gee, J. and Smith, A.J., ‘‘TLB Performance of the SPEC Benchmark Suite’’, *paper in preparation*, 1993.

[Grov90] Groves, R.D. and Oehler, R., ‘‘RISC System/6000 Processor Architecture’’, *IBM RISC System/6000 Technology*, SA23-2619, IBM Corp., 1990, pp. 16-23.

[Hick88] Hickey, T., and Cohen, J., ‘‘Automating Program Analysis’’, *J. of the ACM*, Vol. 35, No. 1, January 1988, pp. 185-220.

[Koba83] Kobayashi, M., ‘‘Dynamic Profile of Instruction Sequences for the IBM System/370’’, *IEEE Trans. on Computers*, Vol. **C-32**, No. 9, September 1983, pp. 859-861.

[Koba84] Kobayashi, M., ‘‘Dynamic Characteristics of Loops’’, *IEEE Trans. on Computers*, Vol. **C-33**, No. 2, February 1984, pp. 125-132.

[Knut71] Knuth, D.E., ‘‘An Empirical Study of Fortran Programs’’, *Software-Practice and Experience*, Vol. 1, pp. 105-133 (1971).

[McMa86] McMahon, F.H., ‘‘The Livermore Fortran Kernels: A Computer Test of the Floating-Point Performance Range’’, LLNL, UCRL-53745, December 1986.

[MIPS89] MIPS Computer Systems, Inc., ‘‘MIPS UNIX Benchmarks’’ *Performance Brief: CPU Benchmarks*, Issue 3.8, June 1989.

[Olss90] Olsson, B., Montoye, R., Markstein, P., and NguyenPhu, M., ‘‘RISC System/6000 Floating-Point Unit’’, *IBM RISC System/6000 Technology*, SA23-2619, IBM Corp., 1990, pp. 34-43.

[Peut77] Peuto, B.L., and Shustek, L.J., ‘‘An Instruction Timing Model of CPU Performance’’, *The fourth Annual Symp. on Computer Arch.*, Vol.5, No.7, March 1977, pp. 165-178.

[Pnev90] Pnevmatikatos, D.N. and Hill, M.D., ‘‘Cache Performance of the Integer SPEC Benchmarks on a RISC, *Comp. Arch. News*, Vol. 18, No. 2, June 1990, pp. 53-68.

[Pond90] Ponder, C.G., ‘‘An Analytical Look at Linear Performance Models’’, LLNL, Tech. Rept. UCRL-JC-106105, September 1990.

[Rama65] Ramamoorthy, C.V., ‘‘Discrete Markov Analysis of Computer Programs’’, *Proc. ACM Nat. Conf.*, pp. 386-392, 1965.

[Saav88] Saavedra-Barrera, R.H., ‘‘Machine Characterization and Benchmark Performance Prediction’’, UC Berkeley, Tech. Rept. No. UCB/CSD 88/437, June 1988.

[Saav89] Saavedra-Barrera, R.H., Smith, A.J., and Miya, E. ‘‘Machine Characterization Based on an Abstract High-Level Language Machine’’, *IEEE Trans. on Comp.* Vol.38, No.12, December 1989, pp. 1659-1679.

[Saav90] Saavedra-Barrera, R.H. and Smith, A.J., *Benchmarking and The Abstract Machine Characterization Model*, UC Berkeley, Tech. Rept. No. UCB/CSD 90/607, November 1990.

[Saav92a] Saavedra-Barrera, R.H., *CPU Performance Evaluation and Execution Time Time Prediction Using Narrow Spectrum Benchmarking*, Ph.D. Thesis, UC Berkeley, Tech. Rept. No. UCB/CSD 92/684, February 1992.

[Saav92b] Saavedra, R.H. and Smith, A.J., ‘‘Benchmarking Optimizing Compilers’’, submitted for publication, USC Tech. Rept. No. USC-CS-92-525, also UC Berkeley, Tech. Rept. No. UCB/CSD 92/699, August 1992.

[Saav92c] Saavedra, R.H., and Smith, A.J., ‘‘Measuring Cache and TLB Performance’’, in preparation, 1992.

[Sark89] Sarkar, V., ‘‘Determining Average Program Execution Times and their Variance’’, *Proc. of the SIGPLAN’89 Conf. on Prog. Lang. Design and Impl.*, Portland, June 21-23, 1989, pp. 298-312.

[SPEC89] SPEC, ‘‘*SPEC Newsletter: Benchmark Results*’’, Vol.2, Issue 1, Winter 1990.

[SPEC89, 90a,b] SPEC, ‘‘*SPEC Newsletter*’’, a: Vol.2, Issue 2, Spring 1989. b: Vol.3, Issue 1, Winter 1990. c: Vol.3, Issue 2, Spring 1990.

[UCB87] U.C. Berkeley, CAD/IC group. ‘‘SPICE2G.6’’, EECS/ERL Industrial Liason Program, UC Berkeley, March, 1987.

[Weic88] Weicker, R.P., ‘‘Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules’’, *SIGPLAN Notices*, Vol.23, No.8, August 1988.

[Worl84] Worlton, J., ‘‘Understanding Supercomputer Benchmarks’’, *Datamation*, September 1, 1984, pp. 121-130.

Abstract operations in the system characterizer (part 1 of 2)

1 **real operations (single, local)**	
01 SRSL	store
02 ARSL	addition
03 MRSL	multiplication
04 DRSL	division
05 ERSL	exponential (X^I)
06 XRSL	exponential (X^Y)
07 TRSL	memory transfer

5 **real operations (single, global)**	
29 SRSG	store
30 ARSG	addition
31 MRSG	multiplication
32 DRSG	division
33 ERSG	exponential (X^I)
34 XRSG	exponential (X^Y)
35 TRSG	memory transfer

2 **complex operations, local operands**	
08 SCSL	store
09 ACSL	addition
10 MCSL	multiplication
11 DCSL	division
12 ECSL	exponential (X^I)
13 XCSL	exponential (X^Y)
14 TCSL	memory transfer

6 **complex operations, global operands**	
36 SCSG	store
37 ACSG	addition
38 MCSG	multiplication
39 DCSG	division
40 ECSG	exponential (X^I)
41 XCSG	exponential (X^Y)
42 TCSG	memory transfer

3 **integer operations, local operands**	
15 SISL	store
16 AISL	addition
17 MISL	multiplication
18 DISL	division
19 EISL	exponential (I^2)
20 XISL	exponential (I^J)
21 TISL	memory transfer

7 **integer operations, global operands**	
43 SISG	store
44 AISG	addition
45 MISG	multiplication
46 DISG	division
47 EISG	exponential (I^2)
48 XISG	exponential (I^J)
49 TISG	memory transfer

4 **real operations (double, local)**	
22 SRDL	store
23 ARDL	addition
24 MRDL	multiplication
25 DRDL	division
26 ERDL	exponential (X^I)
27 XRDL	exponential (X^Y)
28 TRDL	memory transfer

8 **real operations (double, global)**	
50 SRDG	store
51 ARDG	addition
52 MRDG	multiplication
53 DRDG	division
54 ERDG	exponential (X^I)
55 XRDG	exponential (X^Y)
56 TRDG	memory transfer

Table 14: Abstract operations in the System Characterizer (part 1 of 2)

Abstract operations in the system characterizer (part 2 of 2)

9 **logical operations (local)**	
57 ANDL	AND & OR
58 CRSL	compare, real, single
59 CCSL	compare, complex
60 CISL	compare, integer, single
61 CRDL	compare, real, double

10 **logical operations (global)**	
62 ANDG	AND & OR
63 CRSG	compare, real, single
64 CCSG	compare, real, double
65 CISG	compare, integer, single
66 CRDG	compare, real, double

11 **function call and arguments**	
67 PROC	procedure call
68 ARGL	argument load

13 **branching operations**	
69 GOTO	simple goto
70 GCOM	computed goto

12 **references to array elements**	
71 ARR1	array 1 dimension
72 ARR2	array 2 dimensions
73 ARR3	array 3 dimensions
74 ARR4	array 4 dimensions
75 IADD	array index addition

14 **DO loop operations**	
76 LOIN	loop initialization (step 1)
77 LOOV	loop overhead (step 1)
78 LOIX	loop initialization (step n)
79 LOOX	loop overhead (step n)

15 **intrinsic functions (real)**	
80 LOGS	logarithm
81 EXPS	exponential
82 SINS	sine
83 TANS	tangent
84 SQRS	square root
85 ABSS	absolute value
86 MODS	module
87 MAXS	max. and min.

16 **intrinsic functions (double)**	
88 LOGD	logarithm
89 EXPD	exponential
90 SIND	sine
91 TAND	tangent
92 SQRD	square root
93 ABSD	absolute value
94 MODD	module
95 MAXD	max. and min.

17 **intrinsic functions (integer)**	
96 SQRI	square root
97 ABSI	absolute value
98 MODI	module
99 MAXI	max. and min.

18 **intrinsic functions (complex)**	
100 LOGC	logarithm
101 EXPC	exponential
102 SINC	sine
103 SQRC	square root
104 ABSC	absolute value
105 MAXC	max. and min.

19 **coercion functions (complex)**	
106 CLPX	real to complex
107 REAL	select real
108 IMAG	select imaginary
109 CONJ	conjugate function

Table 15: Abstract operations in the System Characterizer (part 2 of 2)

Appendix B

operation	DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	GREYCODE	Average
001 SRSL	0.0027	–	–	–	–	–	0.0005
002 ARSL	0.0494	–	–	–	–	–	0.0082
003 MRSL	0.0040	–	–	–	< 0.0001	–	0.0007
004 DRSL	0.0029	–	< 0.0001	–	0.0001	–	0.0005
005 ERSL	–	–	–	–	–	–	0.0000
006 XRSL	–	–	–	–	–	–	0.0000
007 TRSL	0.0161	–	–	–	–	–	0.0027
008 SCDL	–	–	–	0.0268	–	–	0.0045
009 ACDL	–	–	–	–	0.0190	< 0.0001	0.0032
010 MCDL	–	–	–	–	0.0086	–	0.0014
011 DCDL	–	–	–	–	–	–	0.0000
012 ECDL	–	–	–	–	0.0019	–	0.0003
013 XCDL	–	–	–	–	–	–	0.0000
014 TCDL	–	–	–	–	0.0031	–	0.0005
015 SISL	0.0047	0.0052	0.0110	0.0005	< 0.0001	0.0070	0.0047
016 AISL	0.0035	0.0069	0.0110	0.0009	0.0437	0.0118	0.0130
017 MISL	< 0.0001	0.0008	–	0.0005	< 0.0001	0.0010	0.0004
018 DISL	< 0.0001	0.0006	< 0.0001	< 0.0001	< 0.0001	0.0001	0.0002
019 EISL	–	–	–	–	< 0.0001	–	0.0000
020 XISL	–	–	–	–	< 0.0001	< 0.0001	0.0000
021 TISL	0.0429	0.0015	0.0001	< 0.0001	0.0002	$_3$ 0.1247	0.0283
022 SRDL	$_4$ 0.0697	0.0489	$_4$ 0.1366	$_2$ 0.1414	0.0367	0.0116	0.0741
023 ARDL	$_3$ 0.1285	$_2$ 0.2367	$_2$ 0.2130	$_3$ 0.1414	$_4$ 0.0792	0.0143	0.1355
024 MRDL	$_2$ 0.1397	0.0271	$_3$ 0.1748	$_4$ 0.1414	$_5$ 0.0705	0.0076	0.0935
025 DRDL	0.0335	0.0006	0.0055	< 0.0001	0.0020	0.0034	0.0075
026 ERDL	0.0003	0.0005	–	–	0.0014	–	0.0004
027 XRDL	0.0007	< 0.0001	–	–	–	–	0.0001
028 TRDL	0.0593	0.0069	0.0110	0.0007	0.0114	0.0077	0.0162
029 SRSG	–	–	–	–	–	–	0.0000
030 ARSG	–	–	–	–	0.0009	–	0.0001
031 MRSG	< 0.0001	–	–	–	0.0001	–	0.0000
032 DRSG	–	–	–	–	< 0.0001	–	0.0000
033 ERSG	–	–	–	–	–	–	0.0000
034 XRSG	–	–	–	–	–	–	0.0000
035 TRSG	–	–	–	–	–	–	0.0000
036 SCDG	–	–	–	–	0.0006	–	0.0001
037 ACDG	–	–	–	–	0.0001	–	0.0000
038 MCDG	–	–	–	–	0.0018	–	0.0003
039 DCDG	–	–	–	–	–	–	0.0000
040 ECDG	–	–	–	–	–	–	0.0000
041 XCDG	–	–	–	–	–	–	0.0000
042 TCDG	–	–	–	–	< 0.0001	–	0.0000
043 SISG	< 0.0001	< 0.0001	–	–	–	0.0009	0.0002
044 AISG	< 0.0001	0.0001	–	–	< 0.0001	$_2$ 0.2471	0.0412
045 MISG	< 0.0001	< 0.0001	–	–	–	< 0.0001	0.0001
046 DISG	0.0003	–	–	–	–	< 0.0001	0.0001
047 EISG	–	–	–	–	–	–	0.0000
048 XISG	–	–	–	–	–	–	0.0000
049 TISG	0.0004	< 0.0001	–	–	< 0.0001	< 0.0001	0.0001
050 SRDG	0.0105	$_4$ 0.0796	–	–	0.0298	0.0125	0.0221
051 ARDG	0.0199	$_5$ 0.0658	–	–	0.0321	0.0124	0.0217
052 MRDG	0.0310	$_1$ 0.3208	–	–	0.0533	0.0111	0.0694
053 DRDG	0.0046	0.0008	–	–	0.0001	0.0024	0.0013
054 ERDG	–	0.0001	–	–	–	–	0.0000
055 XRDG	–	–	–	–	–	–	0.0000
056 TRDG	0.0076	0.0044	< 0.0001	–	0.0006	0.0010	0.0023

Table 16: Dynamic distributions for the SPEC benchmarks (part 1 of 2).

operation	DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	GREYCODE	Average
057 ANDL	0.0017	0.0014	–	–	–	0.0015	0.0008
058 CRSL	–	–	–	–	< 0.0001	–	0.0000
059 CCSL	–	–	–	–	< 0.0001	–	0.0000
060 CISL	0.0028	0.0054	< 0.0001	0.0005	< 0.0001	$_4$ 0.1040	0.0188
061 CRDL	0.0156	0.0018	0.0109	< 0.0001	–	0.0034	0.0053
062 ANDG	–	–	–	–	–	–	0.0000
063 CRSG	–	–	–	–	–	–	0.0000
064 CCSG	–	–	–	–	–	–	0.0000
065 CISG	0.0067	< 0.0001	–	–	–	0.0115	0.0031
066 CRDG	0.0018	0.0003	–	–	–	0.0005	0.0004
067 PROC	0.0090	0.0020	< 0.0001	0.0005	0.0019	0.0024	0.0027
068 ARGL	0.0383	0.0025	0.0001	0.0028	0.0021	0.0063	0.0087
069 GOTO	0.0139	0.0030	0.0109	< 0.0001	< 0.0001	0.0994	0.0212
070 GCOM	0.0012	0.0006	–	< 0.0001	–	$_5$ 0.0003	0.0004
071 ARR1	$_1$ 0.1421	$_3$ 0.1672	–	–	0.0453	$_1$ 0.2706	0.1042
072 ARR2	$_5$ 0.0648	< 0.0001	$_1$ 0.3332	$_1$ 0.4264	$_1$ 0.2274	< 0.0001	0.1753
073 ARR3	–	–	–	–	$_2$ 0.0964	–	0.0161
074 ARR4	–	–	–	–	0.0431	–	0.0072
075 ADDI	0.0108	0.0033	$_5$ 0.0326	–	$_3$ 0.0871	0.0135	0.0246
076 LOIN	0.0061	0.0007	0.0001	0.0005	0.0004	$_5$ 0.0005	0.0014
077 LOOV	0.0467	0.0023	0.0275	$_2$ 0.1425	0.0704	0.0042	0.0489
078 LOIX	–	–	–	–	< 0.0001	–	0.0000
079 LOOX	–	–	–	–	< 0.0001	–	0.0000
080 LOGS	–	–	–	–	–	–	0.0000
081 EXPS	–	0.0002	–	–	–	–	0.0000
082 SINS	–	–	–	–	–	–	0.0000
083 TANS	–	< 0.0001	–	–	–	–	0.0000
084 SQRS	–	0.0004	–	–	–	–	0.0001
085 ABSS	–	0.0013	–	< 0.0001	–	–	0.0002
086 MODS	–	–	–	–	–	–	0.0000
087 MAXS	–	–	–	–	–	–	0.0000
088 LOGD	< 0.0001	–	–	–	0.0001	0.0003	0.0001
089 EXPD	0.0013	–	–	–	–	0.0004	0.0003
090 SIND	–	–	–	–	< 0.0001	< 0.0001	0.0000
091 TAND	–	–	–	–	–	< 0.0001	0.0000
092 SQRD	0.0008	–	–	–	0.0004	0.0001	0.0002
093 ABSD	0.0030	–	0.0218	–	0.0004	0.0036	0.0048
094 MODD	–	–	–	–	0.0001	–	0.0000
095 MAXD	0.0011	–	< 0.0001	–	< 0.0001	0.0010	0.0004
096 LOGC	–	–	–	–	0.0009	–	0.0001
097 EXPC	–	–	–	–	0.0001	–	0.0000
098 SINC	–	–	–	–	–	–	0.0000
099 SQRC	–	–	–	–	–	–	0.0000
100 ABSC	–	–	–	–	< 0.0001	–	0.0000
101 MAXC	–	–	–	–	< 0.0001	–	0.0000
102 SQRI	–	–	–	–	–	–	0.0000
103 ABSI	–	–	–	< 0.0001	–	< 0.0001	0.0000
104 MODI	–	< 0.0001	–	< 0.0001	< 0.0001	< 0.0001	0.0001
105 MAXI	< 0.0001	–	–	–	< 0.0001	< 0.0001	0.0001
106 CMPX	–	–	–	–	–	–	0.0000
107 REAL	–	–	–	–	–	–	0.0000
108 IMAG	–	–	–	–	–	–	0.0000
109 CONJ	–	–	–	–	< 0.0001	–	0.0000

Table 17: Dynamic distributions for the SPEC benchmarks (part 2 of 2).

operation	BENCHMARK	BIPOLE	DIGSR	MOSAMP2	PERFECT	TORONTO	Average
001 SRSL	–	–	–	–	–	–	0.0000
002 ARSL	–	–	–	–	–	–	0.0000
003 MRSL	–	–	–	–	–	–	0.0000
004 DRSL	–	–	–	–	–	–	0.0000
005 ERSL	–	–	–	–	–	–	0.0000
006 XRSL	–	–	–	–	–	–	0.0000
007 TRSL	–	–	–	–	–	–	0.0000
008 SCDL	< 0.0001	–	–	–	–	–	0.0000
009 ACDL	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.0001
010 MCDL	–	–	–	–	–	–	0.0000
011 DCDL	–	–	–	–	–	–	0.0000
012 ECDL	–	–	–	–	–	–	0.0000
013 XCDL	–	–	–	–	–	–	0.0000
014 TCDL	0.0001	–	–	–	–	–	0.0000
015 SISL	0.0192	0.0130	0.0119	0.0145	0.0177	0.0122	0.0148
016 AISL	0.0121	0.0129	0.0111	0.0083	0.0087	0.0097	0.0105
017 MISL	0.0019	0.0011	0.0005	0.0008	0.0006	0.0007	0.0009
018 DISL	0.0015	0.0005	0.0002	0.0006	0.0005	0.0004	0.0006
019 EISL	–	–	–	–	–	–	0.0000
020 XISL	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.0001
021 TISL	0.0473	$_{3}$ 0.0993	$_{5}$ 0.0591	0.0357	0.0389	$_{4}$ 0.0643	0.0574
022 SRDL	$_{4}$ 0.0577	0.0254	$_{4}$ 0.0660	$_{4}$ 0.0738	$_{5}$ 0.0593	$_{5}$ 0.0564	0.0564
023 ARDL	$_{3}$ 0.0766	0.0304	$_{3}$ 0.0830	$_{3}$ 0.0954	$_{3}$ 0.0821	$_{3}$ 0.0681	0.0726
024 MRDL	0.0384	0.0159	0.0512	0.0460	0.0287	0.0333	0.0356
025 DRDL	0.0137	0.0075	0.0223	0.0186	0.0080	0.0095	0.0133
026 ERDL	–	–	–	–	–	–	0.0000
027 XRDL	–	–	–	–	–	–	0.0000
028 TRDL	0.0393	0.0166	0.0318	0.0444	0.0402	0.0325	0.0341
029 SRSG	–	–	–	–	–	–	0.0000
030 ARSG	–	–	–	–	–	–	0.0000
031 MRSG	–	–	–	–	–	–	0.0000
032 DRSG	–	–	–	–	–	–	0.0000
033 ERSG	–	–	–	–	–	–	0.0000
034 XRSG	–	–	–	–	–	–	0.0000
035 TRSG	–	–	–	–	–	–	0.0000
036 SCDG	–	–	–	–	–	–	0.0000
037 ACDG	–	–	–	–	–	–	0.0000
038 MCDG	–	–	–	–	–	–	0.0000
039 DCDG	–	–	–	–	–	–	0.0000
040 ECDG	–	–	–	–	–	–	0.0000
041 XCDG	–	–	–	–	–	–	0.0000
042 TCDG	–	–	–	–	–	–	0.0000
043 SISG	0.0056	0.0034	0.0012	0.0026	0.0032	0.0023	0.0031
044 AISG	$_{2}$ 0.1232	$_{2}$ 0.2048	$_{2}$ 0.1406	$_{2}$ 0.1147	$_{2}$ 0.1338	$_{2}$ 0.1570	0.1457
045 MISG	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.0001
046 DISG	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.0001
047 EISG	–	–	–	–	–	–	0.0000
048 XISG	–	–	–	–	–	–	0.0000
049 TISG	0.0011	0.0001	0.0004	0.0007	0.0006	0.0005	0.0006
050 SRDG	0.0251	0.0244	0.0216	0.0228	0.0237	0.0220	0.0233
051 ARDG	0.0284	0.0250	0.0242	0.0276	0.0283	0.0232	0.0261
052 MRDG	0.0285	0.0213	0.0314	0.0372	0.0268	0.0263	0.0286
053 DRDG	0.0072	0.0046	0.0080	0.0088	0.0051	0.0063	0.0067
054 ERDG	–	–	–	–	–	–	0.0000
055 XRDG	–	–	–	–	–	–	0.0000
056 TRDG	0.0102	0.0018	0.0083	0.0113	0.0128	0.0090	0.0089

Table 18: Dynamic distributions for the Spice2g6 benchmarks (part 1 of 2).

operation	BENCHMARK	BIPOLE	DIGSR	MOSAMP2	PERFECT	TORONTO	Average
057 ANDL	0.0076	0.0036	0.0054	0.0072	0.0067	0.0060	0.0061
058 CRSL	–	–	–	–	–	–	0.0000
059 CCSL	–	–	–	–	–	–	0.0000
060 CISL	0.0320	$_{5}$ 0.0657	0.0346	0.0222	0.0279	0.0438	0.0377
061 CRDL	0.0127	0.0058	0.0135	0.0154	0.0130	0.0099	0.0117
062 ANDG	–	–	–	–	–	–	0.0000
063 CRSG	–	–	–	–	–	–	0.0000
064 CCSG	–	–	–	–	–	–	0.0000
065 CISG	0.0182	0.0221	0.0186	0.0191	0.0209	0.0210	0.0200
066 CRDG	0.0092	0.0010	0.0091	0.0118	0.0044	0.0058	0.0069
067 PROC	0.0082	0.0040	0.0040	0.0058	0.0056	0.0048	0.0054
068 ARGL	0.0267	0.0112	0.0181	0.0259	0.0222	0.0210	0.0208
069 GOTO	0.0431	$_{4}$ 0.0660	0.0457	0.0416	0.0433	0.0543	0.0490
070 GCOM	0.0018	0.0007	0.0014	0.0021	0.0023	0.0018	0.0017
071 ARR1	$_{1}$ 0.2128	$_{1}$ 0.2586	$_{1}$ 0.2153	$_{1}$ 0.2016	$_{1}$ 0.2380	$_{1}$ 0.2302	0.2261
072 ARR2	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.0001
073 ARR3	–	–	–	–	–	–	0.0000
074 ARR4	–	–	–	–	–	–	0.0000
075 ADDI	$_{5}$ 0.0577	0.0329	0.0336	$_{5}$ 0.0515	$_{4}$ 0.0617	0.0414	0.0465
076 LOIN	0.0028	0.0014	0.0019	0.0029	0.0038	0.0027	0.0026
077 LOOV	0.0125	0.0088	0.0083	0.0090	0.0114	0.0093	0.0099
078 LOIX	–	–	–	–	–	–	0.0000
079 LOOX	–	–	–	–	–	–	0.0000
080 LOGS	–	–	–	–	–	–	0.0000
081 EXPS	–	–	–	–	–	–	0.0000
082 SINS	–	–	–	–	–	–	0.0000
083 TANS	–	–	–	–	–	–	0.0000
084 SQRS	–	–	–	–	–	–	0.0000
085 ABSS	–	–	–	–	–	–	0.0000
086 MODS	–	–	–	–	–	–	0.0000
087 MAXS	–	–	–	–	–	–	0.0000
088 LOGD	0.0013	0.0006	0.0015	0.0019	0.0014	0.0013	0.0013
089 EXPD	0.0012	0.0009	0.0014	0.0016	0.0012	0.0011	0.0012
090 SIND	< 0.0001	< 0.0001	0.0002	< 0.0001	< 0.0001	< 0.0001	0.0001
091 TAND	< 0.0001	< 0.0001	0.0002	< 0.0001	< 0.0001	< 0.0001	0.0001
092 SQRD	0.0023	0.0002	0.0045	0.0034	0.0009	0.0010	0.0020
093 ABSD	0.0084	0.0065	0.0064	0.0084	0.0104	0.0068	0.0078
094 MODD	–	–	–	–	–	–	0.0000
095 MAXD	0.0042	0.0024	0.0034	0.0048	0.0060	0.0044	0.0042
096 LOGC	–	–	–	–	–	–	0.0000
097 EXPC	–	–	–	–	–	–	0.0000
098 SINC	–	–	–	–	–	–	0.0000
099 SQRC	–	–	–	–	–	–	0.0000
100 ABSC	–	–	–	–	–	–	0.0000
101 MAXC	–	–	–	–	–	–	0.0000
102 SQRI	–	–	–	–	–	–	0.0000
103 ABSI	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.0001
104 MODI	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.0001
105 MAXI	0.0002	0.0001	< 0.0001	0.0001	< 0.0001	< 0.0001	0.0001
106 CMPX	0.0001	–	–	–	–	–	0.0000
107 REAL	–	–	–	–	–	–	0.0000
108 IMAG	< 0.0001	–	–	–	–	–	0.0000
109 CONJ	0.0001	0.0115	0.0050	0.0111	< 0.0001	< 0.0001	0.0047

Table 19: Dynamic distributions for the Spice2g6 benchmarks (part 2 of 2).

operation	ADM	QCD	MDG	TRACK	BDNA	OCEAN	Average
001 SRSL	$_{3}$ 0.1179	$_{5}$ 0.0349	–	< 0.0001	–	0.0038	0.0261
002 ARSL	$_{2}$ 0.1478	$_{3}$ 0.0961	–	< 0.0001	–	0.0105	0.0424
003 MRSL	$_{4}$ 0.1124	$_{2}$ 0.1185	–	0.0010	–	0.0187	0.0418
004 DRSL	0.0230	0.0010	–	< 0.0001	–	0.0014	0.0042
005 ERSL	0.0015	–	–	–	–	< 0.0001	0.0003
006 XRSL	0.0004	–	–	–	–	–	0.0001
007 TRSL	0.0503	0.0371	–	0.0008	–	0.0289	0.0195
008 SCDL	< 0.0001	–	–	–	–	$_{4}$ 0.0645	0.0108
009 ACDL	< 0.0001	–	–	–	–	$_{5}$ 0.0576	0.0096
010 MCDL	< 0.0001	–	–	–	–	0.0212	0.0035
011 DCDL	< 0.0001	–	–	–	–	0.0018	0.0003
012 ECDL	–	–	–	–	–	–	0.0000
013 XCDL	–	–	–	–	–	–	0.0000
014 TCDL	< 0.0001	–	–	–	–	0.0290	0.0049
015 SISL	0.0136	0.0131	0.0086	0.0088	0.0058	0.0370	0.0145
016 AISL	0.0138	0.0448	0.0050	0.0137	0.0035	$_{2}$ 0.2240	0.0508
017 MISL	0.0011	0.0302	< 0.0001	0.0046	0.0030	< 0.0001	0.0065
018 DISL	0.0005	0.0028	< 0.0001	–	–	0.0002	0.0006
019 EISL	< 0.0001	–	–	–	–	–	0.0000
020 XISL	–	–	< 0.0001	–	–	–	0.0000
021 TISL	0.0017	0.0256	0.0011	0.0039	0.0032	0.0001	0.0059
022 SRDL	–	0.0013	$_{3}$ 0.1062	$_{5}$ 0.0749	$_{1}$ 0.2177	–	0.0667
023 ARDL	–	0.0013	$_{2}$ 0.1277	$_{4}$ 0.0861	$_{3}$ 0.1905	–	0.0676
024 MRDL	–	0.0013	$_{5}$ 0.0639	$_{2}$ 0.1275	$_{4}$ 0.1662	–	0.0598
025 DRDL	–	0.0013	0.0044	0.0150	0.0096	–	0.0050
026 ERDL	–	–	< 0.0001	–	< 0.0001	–	0.0000
027 XRDL	–	–	< 0.0001	–	–	–	0.0000
028 TRDL	–	–	0.0106	0.0277	0.0132	–	0.0086
029 SRSG	< 0.0001	< 0.0001	–	< 0.0001	–	< 0.0001	0.0001
030 ARSG	0.0001	< 0.0001	–	< 0.0001	–	< 0.0001	0.0001
031 MRSG	0.0064	0.0001	–	–	–	0.0015	0.0013
032 DRSG	0.0019	< 0.0001	–	–	–	< 0.0001	0.0003
033 ERSG	–	–	–	–	–	–	0.0000
034 XRSG	–	–	–	–	–	–	0.0000
035 TRSG	< 0.0001	0.0003	–	< 0.0001	–	0.0244	0.0042
036 SCDG	–	–	–	–	–	–	0.0000
037 ACDG	–	–	–	–	–	–	0.0000
038 MCDG	–	–	–	–	–	–	0.0000
039 DCDG	–	–	–	–	–	–	0.0000
040 ECDG	–	–	–	–	–	–	0.0000
041 XCDG	–	–	–	–	–	–	0.0000
042 TCDG	–	–	–	–	–	–	0.0000
043 SISG	–	< 0.0001	< 0.0001	0.0006	< 0.0001	< 0.0001	0.0002
044 AISG	–	0.0032	0.0036	0.0006	0.0028	0.0058	0.0027
045 MISG	–	0.0005	< 0.0001	–	< 0.0001	0.0531	0.0090
046 DISG	–	–	–	–	–	< 0.0001	0.0000
047 EISG	–	–	–	–	–	–	0.0000
048 XISG	–	–	–	–	–	–	0.0000
049 TISG	–	< 0.0001	< 0.0001	0.0034	< 0.0001	< 0.0001	0.0006
050 SRDG	–	–	< 0.0001	0.0023	0.0026	–	0.0008
051 ARDG	–	–	0.0036	0.0156	0.0026	–	0.0036
052 MRDG	–	–	0.0132	0.0036	0.0255	–	0.0070
053 DRDG	–	–	0.0024	0.0035	< 0.0001	–	0.0010
054 ERDG	–	–	–	–	< 0.0001	–	0.0000
055 XRDG	–	–	–	–	–	–	0.0000
056 TRDG	–	–	< 0.0001	0.0121	< 0.0001	–	0.0020

Table 20: Dynamic distributions for the Perfect Club benchmarks (part 1 of 4).

operation	ADM	QCD	MDG	TRACK	BDNA	OCEAN	Average
057 ANDL	0.0011	< 0.0001	< 0.0001	0.0114	< 0.0001	< 0.0001	0.0022
058 CRSL	0.0014	0.0005	< 0.0001	0.0187	< 0.0001	< 0.0001	0.0035
059 CCSL	–	–	–	–	–	–	0.0000
060 CISL	0.0059	0.0224	0.0014	0.0055	0.0028	0.0001	0.0063
061 CRDL	–	–	0.0351	0.0055	0.0028	–	0.0072
062 ANDG	–	–	–	–	< 0.0001	–	0.0000
063 CRSG	< 0.0001	–	–	–	–	–	0.0000
064 CCSG	–	–	–	–	–	–	0.0000
065 CISG	< 0.0001	< 0.0001	< 0.0001	0.0690	< 0.0001	0.0001	0.0116
066 CRDG	–	–	0.0122	< 0.0001	< 0.0001	–	0.0021
067 PROC	0.0021	0.0093	0.0113	0.0029	0.0316	< 0.0001	0.0096
068 ARGL	0.0163	0.0284	0.0352	0.0126	0.0317	0.0002	0.0207
069 GOTO	0.0009	0.0026	0.0017	0.0696	0.0054	0.0001	0.0134
070 GCOM	–	–	< 0.0001	–	–	< 0.0001	0.0000
071 ARR1	$_{1}$ 0.2039	$_{1}$ 0.3299	$_{1}$ 0.4064	$_{1}$ 0.2405	$_{2}$ 0.1944	$_{1}$ 0.2718	0.2745
072 ARR2	0.0357	0.0343	< 0.0001	0.0462	0.0222	0.0233	0.0270
073 ARR3	$_{5}$ 0.0976	0.0128	–	0.0001	–	–	0.0184
074 ARR4	–	–	–	–	–	–	0.0000
075 ADDI	0.0896	$_{4}$ 0.0712	0.0195	–	$_{5}$ 0.0445	0.0112	0.0393
076 LOIN	0.0036	0.0068	0.0064	0.0051	< 0.0001	0.0005	0.0037
077 LOOV	0.0454	0.0605	$_{4}$ 0.0753	$_{3}$ 0.1004	0.0087	$_{3}$ 0.0883	0.0631
078 LOIX	0.0006	–	< 0.0001	–	–	0.0001	0.0001
079 LOOX	0.0014	–	< 0.0001	–	–	0.0017	0.0005
080 LOGS	0.0001	0.0004	–	–	–	< 0.0001	0.0001
081 EXPS	< 0.0001	0.0001	–	–	–	< 0.0001	0.0001
082 SINS	< 0.0001	–	–	–	–	< 0.0001	0.0000
083 TANS	< 0.0001	–	–	–	–	–	0.0000
084 SQRS	0.0006	0.0003	–	–	–	< 0.0001	0.0002
085 ABSS	0.0001	< 0.0001	–	–	–	< 0.0001	0.0001
086 MODS	–	–	–	–	–	< 0.0001	0.0000
087 MAXS	0.0009	–	–	–	–	< 0.0001	0.0002
088 LOGD	–	–	–	–	–	–	0.0000
089 EXPD	–	–	0.0045	–	0.0013	–	0.0010
090 SIND	–	–	< 0.0001	0.0046	< 0.0001	–	0.0008
091 TAND	–	–	–	–	< 0.0001	–	0.0000
092 SQRD	–	–	0.0057	0.0003	0.0087	–	0.0024
093 ABSD	–	–	0.0350	0.0017	< 0.0001	–	0.0061
094 MODD	–	–	–	–	–	–	0.0000
095 MAXD	–	–	< 0.0001	–	–	–	0.0000
096 LOGC	–	–	–	–	–	–	0.0000
097 EXPC	–	–	–	–	–	< 0.0001	0.0000
098 SINC	–	–	–	–	–	–	0.0000
099 SQRC	–	–	–	–	–	–	0.0000
100 ABSC	< 0.0001	–	–	–	–	–	0.0000
101 MAXC	–	–	–	–	–	–	0.0000
102 SQRI	–	–	–	–	–	–	0.0000
103 ABSI	–	–	< 0.0001	–	–	–	0.0000
104 MODI	0.0003	0.0059	< 0.0001	–	< 0.0001	< 0.0001	0.0011
105 MAXI	< 0.0001	< 0.0001	–	–	–	< 0.0001	0.0001
106 CMPX	< 0.0001	–	–	–	–	0.0093	0.0016
107 REAL	< 0.0001	0.0013	< 0.0001	0.0002	–	0.0022	0.0007
108 IMAG	< 0.0001	–	–	–	–	0.0022	0.0004
109 CONJ	–	–	–	–	–	0.0056	0.0009

Table 21: Dynamic distributions for the Perfect Club benchmarks (part 2 of 4).

operation	DYFESM	MG3D	ARC2D	FLO52	TRFD	SPEC77	Average
001 SRSL	$_3$ 0.1335	0.0739	–	0.0374	–	0.0612	0.0510
002 ARSL	$_4$ 0.1327	$_4$ 0.1049	< 0.0001	0.0720	< 0.0001	$_2$ 0.1492	0.0765
003 MRSL	$_5$ 0.1300	$_2$ 0.1920	< 0.0001	$_4$ 0.0822	–	$_3$ 0.1171	0.0869
004 DRSL	< 0.0001	0.0057	< 0.0001	0.0153	–	0.0013	0.0037
005 ERSL	–	–	–	0.0057	–	< 0.0001	0.0010
006 XRSL	–	–	–	< 0.0001	–	< 0.0001	0.0000
007 TRSL	0.0123	0.0440	–	0.0039	–	0.0055	0.0109
008 SCDL	–	< 0.0001	–	–	–	0.0135	0.0023
009 ACDL	–	–	–	–	–	0.0135	0.0022
010 MCDL	–	< 0.0001	–	–	–	–	0.0000
011 DCDL	–	–	–	–	–	–	0.0000
012 ECDL	–	–	–	–	–	–	0.0000
013 XCDL	–	–	–	–	–	–	0.0000
014 TCDL	–	–	–	–	–	0.0028	0.0005
015 SISL	0.0073	0.0243	0.0001	0.0011	0.0043	0.0089	0.0077
016 AISL	0.0221	$_3$ 0.1769	0.0001	0.0011	0.0062	0.0113	0.0363
017 MISL	< 0.0001	0.0010	< 0.0001	< 0.0001	< 0.0001	0.0003	0.0003
018 DISL	–	0.0002	–	< 0.0001	< 0.0001	< 0.0001	0.0001
019 EISL	–	–	–	< 0.0001	–	–	0.0000
020 XISL	–	–	–	–	–	–	0.0000
021 TISL	0.0028	0.0006	0.0003	< 0.0001	0.0001	0.0012	0.0008
022 SRDL	–	–	$_4$ 0.1441	–	$_2$ 0.1416	0.0003	0.0477
023 ARDL	–	–	$_5$ 0.1402	–	$_3$ 0.1411	0.0006	0.0470
024 MRDL	–	–	$_2$ 0.1912	–	$_4$ 0.1406	0.0013	0.0555
025 DRDL	–	–	0.0122	–	0.0005	0.0005	0.0022
026 ERDL	–	–	0.0122	–	< 0.0001	< 0.0001	0.0021
027 XRDL	–	–	< 0.0001	–	–	< 0.0001	0.0000
028 TRDL	–	–	0.0200	–	0.0121	0.0005	0.0054
029 SRSG	0.0009	< 0.0001	–	$_5$ 0.0783	–	0.0002	0.0132
030 ARSG	0.0008	0.0003	< 0.0001	$_3$ 0.0935	–	0.0003	0.0158
031 MRSG	0.0013	0.0099	–	0.0225	–	0.0042	0.0063
032 DRSG	< 0.0001	–	–	0.0002	–	< 0.0001	0.0001
033 ERSG	–	–	–	< 0.0001	–	–	0.0000
034 XRSG	–	–	–	< 0.0001	–	< 0.0001	0.0000
035 TRSG	< 0.0001	< 0.0001	–	0.0027	–	< 0.0001	0.0005
036 SCDG	–	–	–	–	–	0.0001	0.0000
037 ACDG	–	–	–	–	–	0.0001	0.0000
038 MCDG	–	–	–	–	–	–	0.0000
039 DCDG	–	–	–	–	–	–	0.0000
040 ECDG	–	–	–	–	–	–	0.0000
041 XCDG	–	–	–	–	–	–	0.0000
042 TCDG	–	–	–	–	–	0.0002	0.0000
043 SISG	< 0.0001	–	< 0.0001	0.0002	< 0.0001	–	0.0001
044 AISG	0.0033	< 0.0001	< 0.0001	< 0.0001	0.0005	–	0.0007
045 MISG	< 0.0001	< 0.0001	< 0.0001	–	< 0.0001	–	0.0001
046 DISG	–	< 0.0001	–	–	–	–	0.0000
047 EISG	–	–	–	–	–	–	0.0000
048 XISG	–	–	–	–	–	–	0.0000
049 TISG	< 0.0001	–	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.0001
050 SRDG	–	–	0.0005	–	–	–	0.0001
051 ARDG	–	–	0.0005	–	–	–	0.0001
052 MRDG	–	–	0.0015	–	–	–	0.0003
053 DRDG	–	–	< 0.0001	–	–	–	0.0000
054 ERDG	–	–	< 0.0001	–	–	–	0.0000
055 XRDG	–	–	–	–	–	–	0.0000
056 TRDG	–	–	< 0.0001	–	–	–	0.0000

Table 22: Dynamic distributions for the Perfect Club benchmarks (part 3 of 4).

operation	DYFESM	MG3D	ARC2D	FLO52	TRFD	SPEC77	Average
057 ANDL	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.0001
058 CRSL	0.0020	0.0002	0.0005	< 0.0001	–	0.0005	0.0006
059 CCSL	–	–	–	–	–	–	0.0000
060 CISL	< 0.0001	0.0003	< 0.0001	< 0.0001	0.0010	0.0001	0.0003
061 CRDL	–	–	< 0.0001	< 0.0001	0.0044	< 0.0001	0.0008
062 ANDG	–	–	< 0.0001	–	–	–	0.0000
063 CRSG	< 0.0001	–	< 0.0001	< 0.0001	< 0.0001	–	0.0001
064 CCSG	–	–	–	–	–	–	0.0000
065 CISG	0.0005	–	< 0.0001	0.0001	–	< 0.0001	0.0001
066 CRDG	–	–	–	–	< 0.0001	–	0.0000
067 PROC	0.0003	0.0001	< 0.0001	0.0012	< 0.0001	0.0001	0.0003
068 ARGL	0.0013	0.0007	0.0002	0.0025	0.0002	0.0004	0.0009
069 GOTO	0.0001	< 0.0001	< 0.0001	< 0.0001	–	0.0004	0.0001
070 GCOM	–	< 0.0001	< 0.0001	–	–	–	0.0000
071 ARR1	0.0523	$_1$ 0.2035	0.0003	0.0040	$_5$ 0.1089	$_1$ 0.2021	0.0952
072 ARR2	$_1$ 0.3207	$_5$ 0.0906	$_3$ 0.1648	0.0604	$_1$ 0.3274	$_3$ 0.1477	0.1853
073 ARR3	0.0089	0.0028	$_1$ 0.2138	$_1$ 0.3349	–	0.0003	0.0934
074 ARR4	< 0.0001	0.0116	–	–	–	–	0.0019
075 ADDI	0.0156	0.0166	0.0443	$_2$ 0.1065	–	$_5$ 0.1160	0.0498
076 LOIN	0.0071	0.0003	0.0003	0.0012	0.0049	0.0013	0.0025
077 LOOV	$_2$ 0.1430	0.0321	0.0460	0.0687	0.1064	0.0177	0.0690
078 LOIX	< 0.0001	0.0006	–	< 0.0001	< 0.0001	0.0018	0.0005
079 LOOX	0.0008	0.0068	–	0.0008	< 0.0001	0.0272	0.0059
080 LOGS	–	–	–	< 0.0001	–	–	0.0000
081 EXPS	–	–	–	< 0.0001	–	0.0002	0.0001
082 SINS	< 0.0001	< 0.0001	–	< 0.0001	–	< 0.0001	0.0001
083 TANS	–	–	< 0.0001	< 0.0001	–	–	0.0000
084 SQRS	< 0.0001	< 0.0001	< 0.0001	0.0006	–	< 0.0001	0.0002
085 ABSS	< 0.0001	< 0.0001	–	0.0016	–	0.0002	0.0003
086 MODS	–	–	–	–	–	–	0.0000
087 MAXS	–	< 0.0001	–	0.0013	–	< 0.0001	0.0003
088 LOGD	–	–	–	–	–	–	0.0000
089 EXPD	–	–	–	–	–	< 0.0001	0.0000
090 SIND	–	–	< 0.0001	–	–	< 0.0001	0.0000
091 TAND	–	–	< 0.0001	–	–	< 0.0001	0.0000
092 SQRD	–	–	0.0034	–	–	< 0.0001	0.0006
093 ABSD	–	–	0.0019	–	–	–	0.0003
094 MODD	–	–	–	–	–	–	0.0000
095 MAXD	–	–	0.0019	–	–	–	0.0003
096 LOGC	–	–	–	–	–	–	0.0000
097 EXPC	–	< 0.0001	–	–	–	–	0.0000
098 SINC	–	–	–	–	–	–	0.0000
099 SQRC	–	–	–	–	–	–	0.0000
100 ABSC	–	–	–	–	–	–	0.0000
101 MAXC	–	–	–	–	–	–	0.0000
102 SQRI	–	< 0.0001	< 0.0001	–	–	–	0.0000
103 ABSI	0.0004	< 0.0001	–	< 0.0001	–	–	0.0001
104 MODI	< 0.0001	0.0001	< 0.0001	< 0.0001	–	< 0.0001	0.0001
105 MAXI	–	< 0.0001	–	< 0.0001	–	–	0.0000
106 CMPX	–	–	–	–	–	0.0158	0.0026
107 REAL	–	–	–	–	–	0.0370	0.0062
108 IMAG	–	–	–	–	–	0.0367	0.0061
109 CONJ	–	–	–	–	–	–	0.0000

Table 23: Dynamic distributions for the Perfect Club benchmarks (part 4 of 4).

operation	ALAMOS	BASKETT	ERAS	LINPACK	LIVER	LOOPS	MAND	SHELL	SMITH	WHETS	Average
001 SRSL	< 0.0001	< 0.0001	< 0.0001	$_3$ 0.1326	$_3$ 0.1389	< 0.0001			$_2$ 0.2187	–	0.0010
002 ARSL	0.0300	< 0.0001	< 0.0001	$_4$ 0.1309	$_1$ 0.2460	0.0384	$_1$ 0.2206	–	0.0010	$_2$ 0.1357	0.0803
003 MRSL	< 0.0001	–	–	$_5$ 0.1251	$_4$ 0.0919	0.0026	$_3$ 0.2168	–	0.0005	0.0019	0.0439
004 DRSL	< 0.0001	–	–	0.0039	0.0016	0.0016	< 0.0001	–	0.0005	0.0047	0.0013
005 ERSL	–	–	–	–	0.0015	0.0012	–	–	–	–	0.0003
006 XRSL	–	–	–	–	–	–	–	–	< 0.0001	–	0.0000
007 TRSL	< 0.0001	< 0.0001	< 0.0001	0.0094	0.0026	< 0.0001	0.0075	< 0.0001	0.0143	< 0.0001	0.0034
008 SCDL	–	–	–	–	–	–	–	–	–	–	0.0000
009 ACDL	–	–	–	–	–	–	–	–	–	–	0.0000
010 MCDL	–	–	–	–	–	–	–	–	–	–	0.0000
011 DCDL	–	–	–	–	–	–	–	–	–	–	0.0000
012 ECDL	–	–	–	–	–	–	–	–	–	–	0.0000
013 XCDL	–	–	–	–	–	–	–	–	–	–	0.0000
014 TCDL	–	–	–	–	–	–	–	–	–	–	0.0000
015 SISL	< 0.0001	0.0053	0.0061	0.0001	0.0051	0.0123	0.0542	$_3$ 0.1332	$_4$ 0.1052	0.0025	0.0324
016 AISL	< 0.0001	$_5$ 0.1059	0.0061	0.0022	0.0104	0.0248	0.0542	$_4$ 0.1332	$_3$ 0.1109	0.0025	0.0450
017 MISL	–	0.0001	–	0.0038	< 0.0001	< 0.0001	–	< 0.0001	0.0119	0.0125	0.0029
018 DISL	< 0.0001	–	–	–	0.0001	0.0003	–	< 0.0001	0.0009	–	0.0002
019 EISL	< 0.0001	–	–	< 0.0001	–	–	–	–	–	–	0.0000
020 XISL	–	–	–	< 0.0001	–	–	–	–	< 0.0001	–	0.0000
021 TISL	< 0.0001	0.0078	$_5$ 0.1141	0.0040	0.0051	0.0087	0.0019	$_2$ 0.1365	$_2$ 0.1253	0.0004	0.0404
022 SRDL	–	–	–	–	–	–	–	–	0.0103	–	0.0010
023 ARDL	–	–	–	–	–	–	–	–	0.0058	–	0.0006
024 MRDL	–	–	–	–	–	–	–	–	0.0007	–	0.0001
025 DRDL	–	–	–	–	–	–	–	0.0005	–	–	0.0001
026 ERDL	–	–	–	–	–	–	–	–	–	–	0.0000
027 XRDL	–	–	–	–	–	–	–	–	–	–	0.0000
028 TRDL	–	–	–	–	–	–	–	–	0.0113	–	0.0011
029 SRSG	$_3$ 0.1302	–	–	–	0.0384	0.0929	–	–	–	0.0140	0.0276
030 ARSG	$_5$ 0.0701	–	–	–	0.0512	0.1241	–	–	–	0.0043	0.0250
031 MRSG	$_4$ 0.1202	–	–	–	0.0407	0.0987	–	–	–	$_5$ 0.0678	0.0327
032 DRSG	–	–	–	–	0.0011	0.0027	–	–	–	0.0293	0.0033
033 ERSG	–	–	–	–	0.0002	0.0006	–	–	–	–	0.0001
034 XRSG	< 0.0001	–	–	–	–	–	–	–	–	–	0.0000
035 TRSG	< 0.0001	–	–	–	0.0110	0.0262	–	–	–	0.0551	0.0092
036 SCDG	–	–	–	–	–	–	–	–	–	–	0.0000
037 ACDG	–	–	–	–	–	–	–	–	–	–	0.0000
038 MCDG	–	–	–	–	–	–	–	–	–	–	0.0000
039 DCDG	–	–	–	–	–	–	–	–	–	–	0.0000
040 ECDG	–	–	–	–	–	–	–	–	–	–	0.0000
041 XCDG	–	–	–	–	–	–	–	–	–	–	0.0000
042 TCDG	–	–	–	–	–	–	–	–	–	–	0.0000
043 SISG	< 0.0001	0.0011	–	–	0.0005	0.0012	–	–	–	0.0188	0.0022
044 AISG	0.0501	0.0011	–	–	0.0026	0.0064	–	–	–	0.0501	0.0110
045 MISG	–	–	–	–	< 0.0001	–	–	–	–	0.0313	0.0031
046 DISG	–	–	–	–	< 0.0001	< 0.0001	–	–	–	–	0.0000
047 EISG	–	–	–	–	–	–	–	–	–	–	0.0000
048 XISG	–	–	–	–	–	–	–	–	–	–	0.0000
049 TISG	–	0.0055	–	–	0.0014	0.0034	–	–	–	0.0309	0.0041
050 SRDG	–	–	–	–	–	–	–	–	–	–	0.0000
051 ARDG	–	–	–	–	–	–	–	–	–	–	0.0000
052 MRDG	–	–	–	–	–	–	–	–	–	–	0.0000
053 DRDG	–	–	–	–	–	–	–	–	–	–	0.0000
054 ERDG	–	–	–	–	–	–	–	–	–	–	0.0000
055 XRDG	–	–	–	–	–	–	–	–	–	–	0.0000
056 TRDG	–	–	–	–	–	–	–	–	–	–	0.0000

Table 24: Dynamic distributions for several small benchmarks (part 1 of 2).

operation	ALAMOS	BASKETT	ERAS	LINPACK	LIVER	LOOPS	MAND	SHELL	SMITH	WHETS	Average
057 ANDL	< 0.0001	0.0966	–	0.0019	< 0.0001	–	0.0561	–	0.0122	–	0.0167
058 CRSL	–	–	–	0.0037	0.0042	–	$_5$ 0.0561	–	0.0003	–	0.0064
059 CCSL	–	–	–	–	–	–	–	–	–	–	0.0000
060 CISL	< 0.0001	0.0057	$_2$ 0.2172	0.0077	0.0008	0.0018	0.0561	$_5$ 0.1331	0.0172	0.0025	0.0442
061 CRDL	–	–	–	–	–	–	–	–	0.0003	–	0.0000
062 ANDG	–	–	–	–	–	–	–	–	–	–	0.0000
063 CRSG	–	–	–	–	0.0024	0.0058	–	–	–	–	0.0008
064 CCSG	–	–	–	–	–	0.0008	–	–	–	–	0.0001
065 CISG	–	$_1$ 0.2439	–	–	< 0.0001	–	–	–	–	0.0309	0.0275
066 CRDG	–	–	–	–	–	–	–	–	–	–	0.0000
067 PROC	0.0003	0.0035	< 0.0001	0.0019	0.0030	0.0064	< 0.0001	< 0.0001	0.0019	0.0272	0.0044
068 ARGL	0.0013	0.0069	< 0.0001	0.0115	0.0050	0.0111	< 0.0001	< 0.0001	0.0090	$_4$ 0.0810	0.0126
069 GOTO	–	0.0103	0.0568	0.0037	0.0009	0.0022	$_4$ 0.0561	0.0486	0.0485	0.0330	0.0260
070 GCOM	–	–	–	–	< 0.0001	–	–	–	0.0411	–	0.0041
071 ARR1	$_1$ 0.4607	$_4$ 0.1224	$_1$ 0.3252	$_1$ 0.3831	$_2$ 0.1454	0.2074	–	$_1$ 0.3730	$_2$ 0.3535	$_1$ 0.1801	0.2551
072 ARR2	–	$_3$ 0.1370	–	0.0225	0.0627	0.1519	–	–	0.0030	–	0.0377
073 ARR3	< 0.0001	–	–	–	0.0076	0.0185	–	–	–	–	0.0026
074 ARR4	–	–	–	–	–	–	–	–	–	–	0.0000
075 ADDI	–	0.1019	–	0.0040	0.0346	0.0839	–	–	0.0143	0.0125	0.0251
076 LOIN	0.0033	0.0038	< 0.0001	0.0021	0.0008	0.0016	< 0.0001	< 0.0001	0.0071	< 0.0001	0.0019
077 LOOV	$_2$ 0.1338	$_2$ 0.1412	$_4$ 0.1201	$_2$ 0.1364	$_5$ 0.0761	0.0533	0.0019	0.0424	$_5$ 0.0918	0.0666	0.0864
078 LOIX	–	< 0.0001	0.0032	< 0.0001	0.0002	0.0005		–	–	–	–
079 LOOX	–	< 0.0001	$_3$ 0.1511	< 0.0001	0.0024	0.0058	–	–	–	–	0.0159
080 LOGS	–	< 0.0001	–	–	–	–	–	–	–	0.0028	0.0003
081 EXPS	–	–	–	–	0.0002	0.0005	–	–	–	0.0028	0.0003
082 SINS	–	–	–	–	–	–	–	–	–	0.0076	0.0008
083 TANS	–	–	–	–	–	–	–	–	–	0.0019	0.0002
084 SQRS	–	–	–	–	0.0002	0.0006	–	–	–	0.0028	0.0004
085 ABSS	–	–	–	0.0020	< 0.0001	–	–	–	–	–	0.0002
086 MODS	–	–	–	–	–	–	–	–	–	–	0.0000
087 MAXS	–	–	–	0.0038	0.0027	0.0017	–	–	–	–	0.0008
088 LOGD	–	–	–	–	–	–	–	–	–	–	0.0000
089 EXPD	–	–	–	–	–	–	–	–	–	–	0.0000
090 SIND	–	–	–	–	–	–	–	–	–	–	0.0000
091 TAND	–	–	–	–	–	–	–	–	–	–	0.0000
092 SQRD	–	–	–	–	–	–	–	–	–	–	0.0000
093 ABSD	–	–	–	–	–	–	–	–	–	–	0.0000
094 MODD	–	–	–	–	–	–	–	–	–	–	0.0000
095 MAXD	–	–	–	–	–	–	–	–	–	–	0.0000
096 LOGC	–	–	–	–	–	–	–	–	–	–	0.0000
097 EXPC	–	–	–	–	–	–	–	–	–	–	0.0000
098 SINC	–	–	–	–	–	–	–	–	–	–	0.0000
099 SQRC	–	–	–	–	–	–	–	–	–	–	0.0000
100 ABSC	–	–	–	–	–	–	–	–	–	–	0.0000
101 MAXC	–	–	–	–	–	–	–	–	–	–	0.0000
102 SQRI	–	–	–	–	–	–	–	–	–	–	0.0000
103 ABSI	–	–	–	–	–	–	–	–	–	–	0.0000
104 MODI	< 0.0001	–	–	0.0038	< 0.0001	–	–	–	–	–	0.0004
105 MAXI	–	–	–	–	< 0.0001	–	–	–	–	–	0.0000
106 CMPX	–	–	–	–	–	–	–	–	–	–	0.0000
107 REAL	–	–	–	–	–	–	–	–	–	–	0.0000
108 IMAG	–	–	–	–	–	–	–	–	–	–	0.0000
109 CONJ	–	–	–	–	–	–	–	–	–	–	0.0000

Table 25: Dynamic distributions for several small benchmarks (part 2 of 2).

Appendix C

Program	DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	GREYCODE	Average
program size	5329	2098	139	181	792	14660	3867
basic blocks	1709	337	43	67	258	6044	1410
lines per block	3.12	6.23	3.23	2.70	3.07	2.43	3.46
blocks executed	64.07%	69.73%	93.02%	83.58%	99.61%	33.32%	73.89%
arith. opers	44.68%	66.99%	41.52%	28.47%	31.47%	43.21%	42.72%
AbOps executed	$6.241x10^8$	$8.949x10^8$	$1.191x10^9$	$1.527x10^9$	$5.716x10^9$	$2.005x10^{10}$	$5.001x10^9$
assignments	63.88%	92.84%	76.32%	49.94%	60.16%	55.38%	66.42%
memory transfers	59.02%	8.71%	6.96%	0.50%	14.05%	80.66%	28.32%
expressions	40.98%	91.29%	93.04%	99.50%	85.95%	19.34%	71.68%
opers per expr	4.77	4.94	2.74	2.00	3.35	9.74	4.59
if statements	14.99%	2.13%	5.24%	< 0.01%	0.01%	8.99%	5.23%
procedure calls	2.68%	1.29%	< 0.01%	0.17%	1.07%	0.81%	1.01%
user routine	58.73%	51.40%	< 0.01%	99.01%	50.31%	31.26%	48.45%
args per call	4.27	1.26	1.00	6.01	1.00	2.64	2.70
intrinsic routines	41.27%	48.60%	100.00%	0.99%	49.69%	68.74%	51.55%
branches	4.50%	2.27%	5.24%	< 0.01%	< 0.01%	33.42%	7.58%
goto	91.96%	84.20%	100.00%	99.34%	100.00%	99.70%	95.87%
computed goto	8.04%	15.80%	0.00%	0.66%	0.00%	0.30%	4.13%
loop iterations	13.95%	1.46%	13.21%	49.90%	38.76%	1.40%	19.78%
iter per loop	7.64	3.13	255.00	300.00	163.75	8.27	122.97

Program	BENCHMARK	BIPOLE	DIGSR	MOSAMP2	PERFECT	TORONTO	Average
program size	14660	14660	14660	14660	14660	14660	14660
basic blocks	6044	6044	6044	6044	6044	6044	6044
lines per block	2.43	2.43	2.43	2.43	2.43	2.43	2.43
blocks executed	52.48%	34.89%	35.41%	36.42%	33.88%	34.96%	38.01%
arith. opers	41.14%	42.21%	45.37%	43.38%	39.58%	42.09%	42.30%
AbOps executed	$5.695x10^7$	$1.984x10^8$	$3.184x10^8$	$2.335x10^7$	$1.962x10^8$	$1.353x10^8$	$1548x10^7$
assignments	67.74%	59.95%	68.62%	70.56%	68.93%	66.01%	66.97%
memory transfers	47.64%	64.05%	49.72%	44.73%	47.10%	53.36%	51.10%
expressions	52.36%	35.95%	50.28%	55.27%	52.90%	46.64%	48.90%
opers per expr	3.08	4.90	3.70	3.15	3.11	3.60	3.59
if statements	10.62%	14.14%	11.04%	9.37%	9.08%	10.71%	10.83%
procedure calls	2.72%	1.29%	1.37%	1.99%	1.97%	1.61%	1.83%
user routine	31.92%	27.24%	18.42%	22.31%	21.97%	24.99%	24.48%
args per call	3.22	2.78	4.48	4.45	3.95	4.31	3.87
intrinsic routines	60.08%	72.76%	81.58%	77.69%	78.03%	75.01%	74.19%
branches	14.79%	21.73%	16.13%	15.00%	16.02%	18.58%	17.04%
goto	96.03%	98.96%	97.12%	95.21%	95.00%	96.86%	96.53%
computed goto	3.97%	1.04%	2.88%	4.79%	5.00%	3.14%	3.47%
loop iterations	4.13%	2.88%	2.84%	3.09%	4.00%	3.09%	3.34%
iter per loop	4.44	6.44	4.38	3.12	3.01	3.43	4.14

Table 26: Program and statements statistics for the SPEC benchmarks and several data sets for Spice2g6.

program	ADM	QCD	MDG	TRACK	BDNA	OCEAN	Average
program size	4164	1768	932	2110	3659	1908	2424
basic blocks	165	520	216	566	883	616	494
lines per block	25.24	3.40	4.31	3.73	4.14	3.10	7.32
blocks executed	45.25%	77.69%	78.20%	75.97%	59.34%	86.53%	70.50%
arith. opers	31.74%	32.44%	27.25%	38.14%	40.93%	40.37%	35.15%
AbOps executed	$1.388x10^9$	$9.799x10^8$	$7.804x10^9$	$2.286x10^8$	$2.014x10^9$	$5.670x10^9$	$3.014x10^9$
assignments	77.10%	55.61%	55.36%	35.65%	82.61%	67.50%	62.31%
memory transfers	20.35%	56.09%	9.30%	35.63%	6.77%	43.86%	28.67%
expressions	71.65%	43.91%	90.70%	64.37%	93.23%	56.14%	70.00%
opers per expr	2.35	6.10	1.95	3.13	1.79	3.75	3.18
if statements	1.99%	8.50%	6.00%	18.53%	1.84%	0.12%	6.16%
procedure calls	0.90%	4.62%	4.95%	0.76%	10.76%	0.01%	3.67%
user routine	52.79%	54.03%	20.01%	29.82%	75.97%	0.14%	38.79%
args per call	7.62	3.04	3.11	4.41	1.00	2.90	3.68
intrinsic routines	47.21%	45.97%	79.99%	70.18%	24.03%	99.86%	61.21%
branches	0.36%	1.29%	0.73%	18.46%	1.83%	0.02%	3.78%
goto	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
computed goto	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
loop iterations	19.65%	29.97%	32.96%	26.61%	2.95%	32.35%	24.08%
iter per loop	11.01	8.96	11.75	19.65	279.84	145.81	79.50

program	DYFESM	MG3D	ARC2D	FLO52	TRFD	SPEC77	Average
program size	3402	2459	2471	1855	412	3278	2312
basic blocks	773	598	571	602	202	1045	631
lines per block	4.40	4.11	4.33	3.08	2.04	3.14	3.51
blocks executed	67.14%	74.92%	73.73%	83.22%	42.08%	88.52%	71.60%
arith. opers.	29.27%	49.24%	35.83%	29.27%	29.43%	33.01%	34.34%
AbOps executed	$1.074x10^9$	$3.410x10^{10}$	$5.229x10^9$	$1.855x10^9$	$1.527x10^9$	$6.448x10^9$	$8.372x10^9$
assignments	52.02%	78.38%	78.17%	63.58%	59.76%	66.80%	66.45%
memory transfers	9.63%	31.22%	12.29%	5.37%	7.72%	10.81%	12.84%
expressions	90.37%	68.78%	87.71%	94.63%	92.28%	89.19%	87.16%
opers per expr	2.05	5.00	2.47	2.50	1.98	3.56	2.93
if statements	0.15%	0.21%	< 0.01%	< 0.01%	0.02%	1.14%	0.26%
procedure calls	0.11%	0.04%	< 0.01%	0.62%	< 0.01%	0.05%	0.14%
user routine	44.91%	37.77%	0.12%	25.71%	100.00%	0.07%	34.76%
args per call	3.82	10.22	1.56	2.02	1.02	3.98	3.77
intrinsic routines	55.09%	62.23%	99.88%	74.29%	0.00%	99.93%	65.24%
branches	0.02%	0.03%	< 0.01%	0.02%	0.00%	0.32%	0.07%
goto	100.00%	10.58%	66.99%	100.00%	0.00%	100.00%	62.93%
computed goto	0.00%	89.42%	33.01%	0.00%	0.00%	0.00%	20.41%
loop iterations	47.69%	21.35%	21.82%	35.77%	40.22%	31.70%	33.10%
iter per loop	20.20	45.09	153.84	55.60	21.74	14.57	51.84

Table 27: Program and statements statistics for the Perfect Club benchmarks.

Program	ALAMOS	BASKETT	ERAS	LINPACK	LIVER	Average
programs size	207	254	21	434	1365	456
basic blocks	58	106	13	158	374	141
lines per block	3.56	2.40	1.62	2.75	3.65	2.79
blocks executed	100.00%	97.17%	100.00%	58.86%	92.78%	89.76%
arith. opers	27.04%	45.33%	22.33%	27.90%	45.49%	33.61%
AbOps executed	6.989×10^8	5.598×10^6	9.989×10^5	7.140×10^7	1.711×10^8	1.578×10^8
assignments	49.28%	6.92%	21.49%	50.08%	70.02%	39.56%
memory transfers	0.00%	67.40%	94.95%	9.22%	9.92%	36.30%
expressions	100.00%	32.60%	5.05%	90.78%	90.08%	63.70%
opers per expr	2.08	5.44	1.00	2.00	2.45	2.59
if statements	0.00%	38.55%	19.82%	1.28%	1.58%	12.24%
procedure calls	0.10%	1.22%	< 0.01%	0.67%	1.03%	0.61%
user routine	99.44%	100.00%	100.00%	16.99%	48.79%	73.04%
args per call	4.99	2.00	1.00	5.90	1.66	3.11
intrinsic routines	0.56%	0.00%	0.00%	83.10%	51.21%	26.97%
branches	0.00%	3.63%	10.16%	1.28%	0.31%	3.07%
goto	0.00%	100.00%	100.00%	100.00%	99.92%	79.98%
computed goto	0.00%	0.00%	0.00%	0.00%	0.08%	0.02%
loop iterations	50.62%	49.68%	48.52%	46.70%	27.05%	44.51%
iter per loop	40.40	36.82	83.44	66.17	74.95	60.36

Program	LOOPS	MAND	SHELL	SMITH	WHETS	Average
program size	454	36	43	436	182	230
basic blocks	149	9	22	174	39	79
lines per block	3.05	4.00	1.96	2.51	4.67	3.24
blocks executed	90.73%	100.00%	100.00%	97.70%	97.44%	97.17%
arith. opers	55.01%	65.97%	26.63%	16.27%	36.36%	33.37%
AbOps executed	1.020×10^8	1.065×10^7	5.420×10^6	4.706×10^8	1.676×10^6	9.839×10^7
assignments	71.83%	82.51%	70.44%	57.76%	60.20%	68.55%
memory transfers	26.44%	3.33%	50.62%	56.46%	39.60%	35.29%
expressions	73.56%	96.67%	49.38%	43.54%	60.40%	64.71%
opers per expr	2.33	1.80	1.00	1.14	2.50	1.63
if statements	0.89%	0.55%	5.79%	2.64%	4.83%	2.94%
procedure calls	0.19%	< 0.01%	< 0.01%	0.41%	7.51%	1.63%
user routine	69.49%	100.00%	100.00%	100.00%	60.39%	85.98%
args per call	1.74	1.00	1.00	4.67	2.97	2.28
intrinsic routines	30.51%	0.00%	0.00%	0.00%	39.61%	14.02%
branches	< 0.01%	16.39%	12.69%	19.36%	9.09%	11.51%
goto	100.00%	100.00%	100.00%	54.15%	100.00%	90.83%
computed goto	0.00%	0.00%	0.00%	45.85%	0.00%	9.17%
loop iterations	27.09%	0.55%	11.08%	19.83%	18.37%	15.38%
iter per loop	27.44	100.50	14376.13	12.90	11165.00	5136.39

Table 28: Program and statements statistics for the small applications and synthetic benchmarks.

Program	DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	GREYCODE	Average
real (single)	12.60%	0.00%	< 0.01%	0.00%	0.34%	0.00%	2.15%
+ and –	87.75%	0.00%	0.00%	0.00%	83.05%	0.00%	56.92%
*	7.18%	0.00%	0.00%	0.00%	8.29%	0.00%	5.14%
/	5.07%	0.00%	100.00%	0.00%	8.55%	0.00%	37.86%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	11.05%	0.00%	0.00%	0.00%	0.12%	0.00%	3.72%
complex	0.00%	0.00%	0.00%	0.00%	9.98%	< 0.01%	1.66%
+ and –	0.00%	0.00%	0.00%	0.00%	60.90%	100.00%	80.43%
*	0.00%	0.00%	0.00%	0.00%	32.96%	0.00%	16.47%
/	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
**	0.00%	0.00%	0.00%	0.00%	6.13%	0.00%	3.06%
compare	0.00%	0.00%	0.00%	0.00%	0.01%	0.00%	0.00%
integer	2.98%	2.07%	2.66%	0.67%	13.89%	86.87%	18.19%
+ and –	26.33%	50.52%	100.00%	49.88%	99.92%	68.94%	65.93%
*	0.00%	6.30%	0.00%	24.85%	0.00%	0.27%	5.23%
/	2.64%	4.24%	0.00%	0.08%	0.00%	0.02%	1.16%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	71.04%	38.94%	0.00%	25.19%	0.08%	30.77%	27.67%
real (double)	84.04%	97.72%	97.34%	99.33%	75.79%	12.77%	77.83%
+ and –	39.50%	46.21%	52.69%	50.00%	46.66%	48.44%	47.25%
*	45.47%	53.16%	43.26%	50.00%	51.90%	33.80%	46.26%
/	10.14%	0.22%	1.35%	< 0.01%	0.88%	10.65%	3.87%
**	0.27%	0.09%	0.00%	0.00%	0.57%	0.00%	0.15%
compare	4.63%	0.32%	2.70%	< 0.01%	0.00%	7.11%	2.46%
logical	0.37%	0.21%	< 0.01%	0.00%	< 0.01%	0.36%	0.16%

Program	BENCHMARK	BIPOLE	DIGSR	MOSAMP2	PERFECT	TORONTO	Average
real (single)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
+ and –	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
*	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
/	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
complex	< 0.01%	< 0.01%	< 0.01%	< 0.01%	< 0.01%	< 0.01%	0.01%
+ and –	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
*	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
/	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
integer	45.95%	72.74%	45.32%	38.21%	48.65%	55.25%	51.02%
+ and –	71.59%	70.91%	73.78%	74.21%	74.03%	71.68%	72.70%
*	1.05%	0.35%	0.23%	0.51%	0.33%	0.29%	0.46%
/	0.79%	0.15%	0.11%	0.37%	0.26%	0.17%	0.30%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	26.58%	28.58%	25.88%	24.91%	25.37%	27.85%	26.52%
real (double)	52.20%	26.41%	53.48%	60.13%	49.66%	43.31%	47.53%
+ and –	48.93%	49.74%	44.17%	47.15%	56.18%	50.07%	49.37%
*	31.16%	33.38%	34.03%	31.91%	28.31%	32.65%	31.90%
/	9.75%	10.86%	12.48%	10.50%	6.68%	8.69%	9.82%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	10.16%	6.03%	9.32%	10.43%	8.83%	8.59%	8.89%
logical	1.85%	0.85%	1.20%	1.66%	1.69%	1.44%	1.44%

Table 29: Distribution of arithmetic and logical operations according to data type and precision for the SPEC Benchmarks.

Program	ADM	QCD	MDG	TRACK	BDNA	OCEAN	Average
real (single)	92.96%	66.74%	< 0.01%	5.18%	< 0.01%	8.10%	28.83%
+ and –	50.15%	44.44%	0.00%	0.01%	0.00%	32.63%	21.20%
*	40.27%	54.85%	0.00%	5.06%	0.00%	62.90%	27.18%
/	8.45%	0.46%	0.00%	0.03%	0.00%	4.46%	2.23%
**	0.66%	0.00%	0.00%	0.00%	0.00%	0.00%	0.11%
compare	0.47%	0.25%	100.00%	94.89%	100.00%	0.01%	49.27%
complex	< 0.01%	0.00%	0.00%	0.00%	0.00%	20.35%	3.39%
+ and –	32.50%	0.00%	0.00%	0.00%	0.00%	71.51%	51.99%
*	45.00%	0.00%	0.00%	0.00%	0.00%	26.29%	35.64%
/	22.50%	0.00%	0.00%	0.00%	0.00%	2.21%	12.33%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
integer	6.71%	32.08%	3.65%	24.47%	2.94%	71.53%	23.56%
+ and –	64.65%	46.24%	86.34%	15.30%	51.75%	81.15%	57.57%
*	5.36%	29.48%	0.00%	4.93%	25.03%	18.75%	13.92%
/	2.44%	2.67%	0.00%	0.00%	0.00%	0.06%	0.86%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	27.54%	21.61%	13.66%	79.76%	23.21%	0.05%	27.63%
real (double)	0.00%	1.19%	96.35%	67.35%	97.06%	0.00%	43.65%
+ and –	0.00%	33.33%	50.01%	39.61%	48.61%	0.00%	42.89%
*	0.00%	33.33%	29.36%	51.03%	48.25%	0.00%	40.49%
/	0.00%	33.33%	2.61%	7.20%	2.43%	0.00%	11.3%
**	0.00%	0.00%	0.00%	0.00%	0.01%	0.00%	0.00%
compare	0.00%	0.00%	18.01%	2.16%	0.70%	0.00%	5.21%
logical	0.34%	< 0.01%	< 0.01%	3.00%	< 0.01%	0.01%	0.56%

Program	DYFESM	MG3D	ARC2D	FLO52	TRFD	SPEC77	
real (single)	91.16%	63.69%	0.14%	99.61%	< 0.01%	90.75%	57.56%
+ and –	50.03%	33.63%	0.01%	56.77%	87.50%	54.83%	47.12%
*	49.20%	64.51%	0.00%	35.90%	0.00%	44.49%	32.35%
/	0.01%	1.81%	0.57%	5.33%	0.00%	0.49%	1.36%
**	0.00%	0.00%	0.00%	1.97%	0.00%	0.01%	0.33%
compare	0.76%	0.06%	99.42%	0.02%	12.50%	0.19%	18.82%
complex	0.00%	< 0.01%	0.00%	0.00%	0.00%	4.53%	0.75%
+ and –	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%	50.00%
*	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%	50.00%
/	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
integer	8.84%	36.31%	0.02%	0.39%	2.61%	3.89%	8.67%
+ and –	98.05%	99.15%	99.17%	94.48%	87.10%	96.25%	95.70%
*	0.02%	0.56%	0.03%	0.13%	0.00%	2.81%	0.59%
/	0.00%	0.10%	0.00%	0.10%	0.00%	0.00%	0.03%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	1.93%	0.19%	0.80%	5.28%	12.90%	0.94%	3.67%
real (double)	0.00%	0.00%	99.84%	< 0.01%	97.39%	0.81%	33.00%
+ and –	0.00%	0.00%	39.32%	0.00%	49.22%	25.28%	28.46%
*	0.00%	0.00%	53.87%	0.00%	49.07%	53.58%	39.12%
/	0.00%	0.00%	3.40%	0.00%	0.17%	21.12%	6.17%
**	0.00%	0.00%	3.41%	0.00%	0.00%	0.00%	0.84%
compare	0.00%	0.00%	0.00%	100.00%	1.54%	0.01%	25.38%
logical	< 0.01%	< 0.01%	< 0.01%	< 0.01%	< 0.01%	0.01%	0.01%

Table 30: Distribution of arithmetic and logical operations according to data type and precision for the Perfect Club Benchmarks.

Program	ALAMOS	BASKETT	ERAS	LINPACK	LIVER	
real (single)	81.48%	< 0.01%	< 0.01%	94.45%	96.92%	54.57%
+ and –	45.45%	100.00%	100.00%	49.66%	67.42%	72.50%
*	54.54%	0.00%	0.00%	47.46%	30.08%	26.41%
/	0.00%	0.00%	0.00%	1.46%	0.63%	0.41%
**	0.00%	0.00%	0.00%	0.00%	0.39%	0.07%
compare	0.00%	0.00%	0.00%	1.41%	1.48%	0.57%
complex	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
+ and –	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
*	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
/	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
integer	18.52%	78.68%	99.99%	4.88%	3.07%	41.02%
+ and –	99.99%	29.99%	2.72%	15.89%	93.19%	48.35%
*	0.00%	0.02%	0.00%	27.75%	0.25%	5.60%
/	0.00%	0.00%	0.00%	0.00%	0.83%	0.16%
**	0.01%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	0.00%	69.99%	97.28%	56.36%	5.73%	45.87%
real (double)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
+ and –	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
*	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
/	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
logical	0.00%	21.32%	0.00%	0.67%	0.00%	4.39%

Program	LOOPS	MAND	SHELL	SMITH	WHETS	Average
real (single)	89.01%	74.79%	0.00%	1.40%	64.28%	45.89%
+ and –	58.94%	44.70%	0.00%	45.15%	55.61%	51.10%
*	36.76%	43.94%	0.00%	20.46%	29.84%	32.75%
/	1.55%	0.00%	0.00%	20.46%	14.55%	9.14%
**	0.66%	0.00%	0.00%	< 0.01%	0.00%	0.16%
compare	2.09%	11.36%	0.00%	13.92%	0.00%	6.84%
complex	0.26%	0.00%	0.00%	0.00%	0.00%	0.05%
+ and –	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
*	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
/	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	100.00%	0.00%	0.00%	0.00%	0.00%	100.00%
integer	10.74%	16.71%	100.00%	86.58%	35.72%	49.95%
+ and –	93.79%	49.15%	50.00%	78.73%	40.52%	62.43%
*	0.08%	0.00%	0.00%	8.43%	33.77%	8.45%
/	0.85%	0.00%	0.00%	0.66%	0.00%	0.30%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	5.29%	50.85%	50.00%	12.19%	25.71%	28.80%
real (double)	0.00%	0.00%	0.00%	4.50%	0.00%	0.90%
+ and –	0.00%	0.00%	0.00%	78.90%	0.00%	78.90%
*	0.00%	0.00%	0.00%	9.40%	0.00%	9.40%
/	0.00%	0.00%	0.00%	7.37%	0.00%	7.35%
**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	0.00%	0.00%	0.00%	4.32%	0.00%	4.30%
logical	0.00%	8.50%	0.00%	7.52%	0.00%	3.20%

Table 31: Distribution of arithmetic and logical operations according to data type and precision for the several small programs.

program	DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	GREYCODE	Average
simple	76.34%	82.63%	54.51%	25.19%	22.71%	64.52%	54.31%
arrays	23.66%	17.37%	45.49%	74.81%	77.01%	35.48%	45.63%
1 dim	68.69%	99.98%	0.00%	0.00%	10.98%	100.00%	46.60%
2 dims	31.31%	0.02%	100.00%	100.00%	55.18%	0.00%	47.75%
3 dims	0.00%	0.00%	0.00%	0.00%	23.39%	0.00%	3.89%
4 dims	0.00%	0.00%	0.00%	0.00%	10.45%	0.00%	1.74%

program	BENCHMARK	BIPOLE	DIGSR	MOSAMP2	PERFECT	TORONTO	Average
simple	74.12%	67.27%	74.80%	76.15%	69.80%	71.90%	72.34%
arrays	25.88%	32.73%	25.20%	23.85%	30.20%	28.10%	27.66%
1 dim	99.99%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
2 dims	0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
3 dims	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
4 dims	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

program	ADM	QCD	MDG	TRACK	BDNA	OCEAN	Average
simple	50.72%	31.27%	22.67%	55.87%	75.79%	61.75%	49.67%
arrays	49.28%	68.73%	77.33%	44.13%	24.21%	38.25%	50.32%
1 dim	60.46%	87.50%	100.00%	83.83%	89.76%	92.12%	85.61%
2 dims	10.60%	9.11%	0.00%	16.12%	10.24%	7.88%	8.99%
3 dims	28.94%	3.39%	0.00%	0.05%	0.00%	0.00%	5.39%
4 dims	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

program	DYFESM	MG3D	ARC2D	FLO52	TRFD	SPEC77	Average
simple	37.02%	60.31%	44.94%	26.01%	28.54%	28.46%	37.54%
arrays	62.98%	39.69%	55.06%	73.99%	71.46%	71.54%	62.45%
1 dim	13.70%	65.97%	0.08%	1.00%	24.96%	57.72%	27.23%
2 dims	83.98%	29.37%	43.50%	15.13%	75.04%	42.18%	48.20%
3 dims	2.32%	0.90%	56.43%	83.87%	0.00%	0.09%	23.93%
4 dims	0.00%	3.77%	0.00%	0.00%	0.00%	0.00%	0.62%

program	ALAMOS	BASKETT	ERAS	LINPACK	LIVER	Average
simple	13.21%	47.35%	29.84%	29.02%	74.94%	38.87%
arrays	86.79%	52.65%	70.16%	70.98%	25.06%	61.12%
1 dim	100.00%	47.19%	100.00%	94.46%	67.40%	81.81%
2 dims	0.00%	52.81%	0.00%	5.54%	29.06%	17.48%
3 dims	0.00%	0.00%	0.00%	0.00%	3.54%	0.70%
4 dims	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

program	LOOPS	MAND	SHELL	SMITH	WHETS	Average
simple	36.95%	100.00%	53.71%	48.86%	77.50%	63.40%
arrays	63.05%	0.00%	46.29%	51.14%	22.50%	36.59%
1 dim	54.89%	0.00%	100.00%	99.15%	100.00%	88.50%
2 dims	40.20%	0.00%	0.00%	0.85%	0.00%	10.27%
3 dims	4.91%	0.00%	0.00%	0.00%	0.00%	1.23%
4 dims	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

Table 32: Distribution of simple and array variables for the SPEC, Perfect Club and several small benchmarks.

Appendix D

	DODUC			FPPPP			TOMCATV		
System	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)
IBM RS/6000 530	135	125	–7.56	93	101	+9.11	196	244	+24.62
MIPS M/2000	187	208	+11.69	247	239	–3.21	452	415	–8.14
Motorola M88k	309	271	–12.42	511	313	–38.78	556	422	–24.04
Decstation 5400	330	325	–1.38	625	480	–23.18	619	583	–5.92
Decstation 3100	352	346	–1.52	664	510	–23.10	674	648	–3.76
Sparcstation I	344	341	+0.06	361	446	+23.43	571	603	+5.69
VAX 3200	1232	1078	–12.46	1476	1272	–13.82	1829	1735	–5.13
VAX-11/785	2114	2397	+13.41	2217	2708	+22.20	3272	3535	+8.04
Sun 3/50 (68881)	3313	3736	+12.76	5396	6669	+23.56	6707	6734	+0.40
average			+0.29			–2.64			–0.92
r.m.s.			9.72			22.25			12.57

	MATRIX300			NASA7			SPICE2G6			average	r.m.s.
System	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	error (%)	error (%)
IBM RS/6000 530	630	404	–35.85	1601	1815	+13.36	2438	3385	+38.85	+7.09	24.90
MIPS M/2000	816	614	–24.77	2906	2634	–9.36	4576	4539	–0.81	–5.77	12.35
Motorola M88k	651	538	–17.28	–	2964	–	–	4237	–	–23.13	25.17
Decstation 5400	1017	863	–15.17	3695	3824	+3.49	3994	5462	+36.76	–0.90	19.01
Decstation 3100	1176	922	–21.64	4103	4207	+2.53	4102	5702	+38.99	–1.42	20.60
Sparcstation I	1300	803	–38.21	5118	3906	–23.68	3594	4911	+36.64	+0.66	25.64
VAX 3200	3270	2251	–31.17	12891	11406	–11.52	12723	15289	+20.16	–8.99	17.72
VAX-11/785	5931	4171	–29.68	22457	20794	–7.41	25456	30533	+19.94	+3.49	20.11
Sun 3/50 (68881)	7674	7149	–6.83	36620	41310	+12.81	20973	27671	+31.94	+12.44	18.02
average			–24.51			–1.77			+28.93	–1.20	
r.m.s.			26.36			12.77			31.96		20.63

Table 33: Execution estimates and actual running times for the SPEC benchmarks. All real times and predictions are in seconds; errors in percentage.

	ADM			QCD			MDG			TRACK			BDNA			OCEAN		
System	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)
CRAY Y-MP/8128	114	98	–14.03	90	93	+3.33	4928	4511	–8.46	144	139	–3.47	1357	1338	–1.42	521	524	+0.57
IBM RS/6000 530	208	165	–20.67	121	134	+9.70	1209	1558	+28.86	–	49	–	307	288	–6.18	1025	1206	+17.65
MIPS M/2000	424	426	+0.47	131	176	+34.48	1796	2254	+25.50	–	71	–	733	582	–20.60	1618	1722	+6.47
Motorola 88000	–	407	–	175	205	+17.41	3005	2989	–0.54	–	82	–	–	823	–	1510	1157	–23.42
Decstation 3100	649	657	+1.29	202	248	+23.02	3212	3705	+15.34	–	111	–	1034	929	–10.12	2524	2682	+0.62
MIPS M/1000	715	723	+1.11	238	328	+37.82	3026	3979	+39.49	–	116	–	–	978	–	–	2968	–
VAX 3200	1865	1659	–11.05	1060	909	–14.24	13166	12502	–5.04	337	312	+7.41	3988	3162	–20.71	10628	11250	+5.85
VAX-11/785	3324	2883	–13.27	2141	1701	–20.55	26401	29037	+9.98	654	667	+1.98	6333	7446	+17.57	13651	12230	–10.41
Sun 3/50 (68881)	5964	6353	+6.52	2252	2966	+31.71	29717	30273	+1.87	836	994	+18.90	11986	10786	–10.01	39505	42015	+6.35
average			–6.20			+13.63			+11.81			–6.20			–6.89			+0.46
r.m.s.			10.99			24.01			19.66			10.35			14.73			11.65

	DYFESM			MG3D			ARC2D			FLO52			TRFD			SPEC77			average	r.m.s.
System	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	error (%)	error (%)
CRAY Y-MP/8128	131	103	–21.37	2966	2174	–26.70	3337	3025	–9.34	158	136	–13.92	803	611	–23.91	516	431	–16.47	–11.27	14.68
IBM RS/6000 530	–	266	–	–	6098	–	–	1516	–	441	635	+43.99	403	360	–10.66	901	1241	+37.74	+12.55	25.44
MIPS M/2000	407	370	–9.10	–	9041	–	3484	2470	–29.10	–	853	–	577	566	–1.87	–	2169	–	+0.78	20.12
Motorola 88000	358	304	–15.02	–	7606	–	3216	2788	–13.32	742	847	+14.17	522	496	–5.07	–	1628	–	–3.68	14.55
Decstation 3100	604	555	–8.16	–	13752	–	5372	3923	–26.98	1112	1310	+17.84	876	871	–0.56	–	2825	–	+5.26	11.79
MIPS M/1000	651	610	–6.29	19019	15089	–20.66	–	4126	–	1271	1406	+10.62	965	935	–3.10	–	3717	–	+8.43	22.61
VAX 3200	1136	1243	+9.41	–	28850	–	–	10017	–	2822	3126	+10.77	2047	2069	+1.07	10628	11250	+5.71	–1.08	10.52
VAX-11/785	2059	1936	–5.97	–	50743	–	–	20082	–	4335	4928	+13.67	3581	4153	+15.97	17846	17523	–1.81	–0.72	12.65
Sun 3/50 (68881)	4496	4986	+10.89	–	146824	–	33768	33556	–0.63	8024	9710	+21.01	8118	7715	–4.96	–	28616	–	+8.17	14.61
average			–5.70			–23.68			–13.10			+14.33			–3.68			+6.29	+1.46	
r.m.s.			11.80			23.87			16.67			21.34			10.57			20.81		16.69

Table 34: Execution estimates and actual running times for the Perfect benchmarks. All real times and predictions are in seconds; errors in percentage. The measurement missing couldn't be obtained due to compiler errors or invalid benchmark results. Benchmark *MG3D* was not executed on some system due to insufficient disk space; the program requires a 94 MB file. In some machines, *ARC2D*, using 64-bit double precision numbers, gave a run time error. Results for *TRACK* were invalid in several machines.

	ALAMOS			BASKETT			ERATHOSTENES			LINPACK			LIVERMORE			MANDELBROT		
System	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)
CRAY X-MP/48	63.8	58.7	−7.99	0.70	0.66	−5.71	0.149	0.161	+8.05	8.05	8.29	+2.98	15.3	16.9	+10.46	1.002	1.057	+5.49
IBM 3090/200	80.5	73.4	−8.82	0.66	0.78	+18.18	0.130	0.114	−12.31	–	9.77	–	19.5	18.5	−5.13	0.220	0.226	+2.73
Amdahl 5840	345.9	327.2	−5.41	2.23	2.67	+19.73	0.463	0.408	−11.88	–	44.43	–	–	92.6	–	3.344	3.546	+6.04
Convex C-1	236.1	243.6	+3.18	2.75	2.32	−15.64	0.650	0.580	−10.77	35.4	31.48	−11.07	67.9	69.9	+2.96	3.948	3.380	−14.39
IBM RS/6000 530	102.2	122.9	+20.25	1.30	1.08	−16.92	0.300	0.280	−6.67	14.8	13.74	−7.41	–	28.5	–	1.210	1.230	+1.65
MIPS M/2000	118.3	138.6	+16.95	1.00	1.13	+13.00	0.390	0.307	−21.28	12.7	14.50	+13.94	30.0	38.6	+28.80	1.500	1.592	+6.00
Motorola M88k	115.1	131.6	+14.34	1.40	1.22	−12.86	0.300	0.210	−30.00	13.6	16.40	+20.59	36.9	36.0	−2.44	1.800	1.770	−1.67
Sparcstation I	205.9	192.8	−6.39	1.32	1.36	+3.03	0.370	0.350	−5.41	21.9	21.17	−3.33	50.2	51.3	+2.17	2.400	2.970	+23.75
VAX 8600	265.3	266.7	+0.53	2.82	3.24	+14.89	0.750	0.603	−19.64	41.6	35.43	−14.83	88.2	88.7	+0.57	3.490	3.614	+3.55
VAX-11/785	701.7	758.3	+8.07	7.38	8.27	+12.06	1.733	1.726	−0.40	99.7	106.15	+6.47	223.3	255.9	+14.60	11.36	12.82	+12.85
VAX-11/780	1581.7	1702.7	+7.65	14.85	16.17	+8.89	2.766	2.462	−10.99	220.1	227.53	+3.38	611.0	653.5	+6.96	33.42	32.13	−3.86
Sun 3/50	6273.2	5795.8	−7.61	7.06	8.315	+17.78	0.900	0.916	+1.78	763.7	752.96	−1.41	2457.0	2583.7	+5.16	163.94	165.81	+1.14
IBM RT-PC/125	3881.9	3810.0	−1.85	6.20	7.40	+19.35	1.100	1.354	+23.09	473.9	448.47	−5.37	1610.1	1573.8	−2.25	105.43	104.09	−1.27
average			+2.53			+5.83			−7.42			+0.36			+5.62			+3.23
r.m.s.			10.04			14.58			15.05			10.09			10.78			9.12

	SHELL			SMITH			WHETSTONE			average	r.m.s
System	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	error (%)	error (%)
CRAY X-MP/48	0.683	0.593	−13.18	66.7	65.77	−1.39	0.302	0.296	−1.99	−0.36	7.37
IBM 3090/200	0.440	0.395	−10.23	53.2	45.3	−14.85	0.350	0.335	−4.29	−4.34	10.82
Amdahl 5840	1.893	1.965	+3.80	198.0	185.4	−6.36	1.697	1.942	+14.44	+2.91	11.08
Convex C-1	1.828	1.770	−3.17	193.1	197.2	+2.12	1.111	1.170	+5.31	−4.61	9.14
IBM RS/6000 530	0.920	0.900	−2.17	90.0	88.1	−2.11	0.350	0.390	+11.43	−0.24	10.83
MIPS M/2000	1.640	1.590	−2.44	132.4	112.5	+15.05	0.480	0.480	−0.06	+8.72	15.74
Motorola M88k	0.800	0.760	−5.00	120.6	94.4	−21.72	0.620	0.530	−14.52	−5.92	16.37
Sparcstation I	0.820	1.050	−28.05	145.7	134.1	−7.98	0.760	0.710	−6.58	−3.20	13.14
VAX 8600	2.233	2.140	−4.16	238.7	230.0	−3.64	2.870	2.631	−8.33	−3.45	10.22
VAX-11/785	5.800	6.110	+5.34	683.9	691.6	+1.13	7.950	7.385	−7.11	+5.89	8.89
VAX-11/780	9.183	8.803	−4.14	1087.5	1018.8	−6.32	21.57	21.74	+0.79	+0.26	6.59
Sun 3/50	3.140	3.522	+12.17	914.8	877.4	−4.09	34.24	39.50	+15.36	+4.48	9.47
IBM RT-PC/125	4.680	4.610	−1.50	545.1	675.3	+23.89	12.05	11.95	−0.82	+5.92	13.00
average			−3.41			−2.02			+0.28	+0.47	
r.m.s.			10.26			11.35			8.78		11.34

Table 35: Execution estimates and actual running times for the small programs. All real times and predictions in seconds; errors in percentage. In the last row r.m.s. is the root mean square error. The LINPACK benchmark was not available when the experiments were run on the IBM 3090 and Amdahl 5840, and Livermore did not run on the Amdahl 5840 or IBM RS/6000 530.

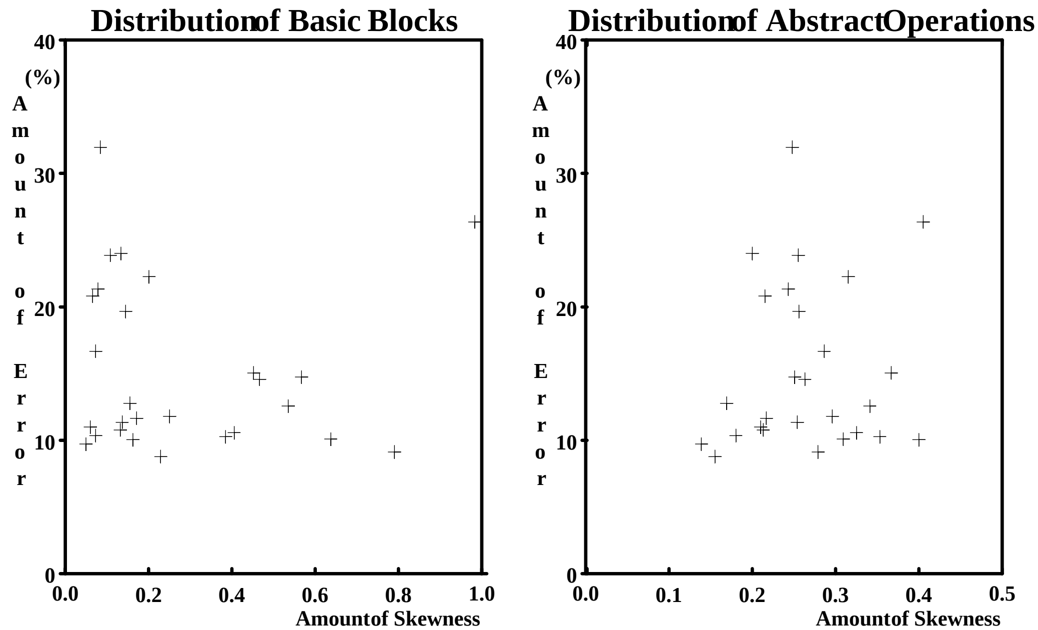


Figure 16: Scattergrams of the amount of skewness in the ordered distributions of basic blocks (a) and abstract operations (b) against the amount of error in the execution prediction.

num.	Most Similar Programs		
001	TRFD	Matrix300	0.0172
002	DYFESM	Linpack	0.0302
003	ARC2D	Tomcatv	0.0775
004	Alamos	Linpack	0.0855
005	QCD	FLO52	0.0913
006	DYFESM	Alamos	0.1224
007	MDG	TRFD	0.1332
008	ARC2D	TRFD	0.1346
009	MDG	Matrix300	0.1363
010	Shell	Smith	0.1423
011	BDNA	Doduc	0.1480
012	ADM	FLO52	0.1573
013	FLO52	SPEC77	0.1650
014	DYFESM	FLO52	0.1652
015	QCD	SPEC77	0.1781
016	FLO52	Alamos	0.1808
017	Greycode	Perfect	0.1823
018	ARC2D	Matrix300	0.1833
019	FLO52	Linpack	0.1970
020	MDG	Nasa7	0.2044
021	MDG	ARC2	0.2050
022	ADM	QCD	0.2129
023	OCEAN	Greycode	0.2154
024	ADM	DYFESM	0.2184
025	ARC2D	Nasa7	0.2295

num.	Least Similar Programs		
378	Fpppp	Whetstone	4.7040
377	Baskett	Whetstone	4.2341
376	Fpppp	Mandelbrot	4.2181
375	OCEAN	Fpppp	3.9908
374	Fpppp	Baskett	3.9733
373	SPEC77	Fpppp	3.9624
372	Erasthostenes	Whetstone	3.7693
371	Doduc	Baskett	3.6196
370	MG3D	Fpppp	3.5637
369	Baskett	Mandelbrot	3.5558
368	MDG	Mandelbrot	3.5173
367	Fpppp	Livermore	3.5135
366	Fpppp	Erasthostenes	3.4947
365	TRFD	Whetstone	3.4689
364	Matrix300	Whetstone	3.4518
363	SPEC77	Whetstone	3.4291
362	Doduc	Mandelbrot	3.4289
361	Matrix300	Mandelbrot	3.4168
360	Tomcatv	Whetstone	3.4084
359	SPEC77	Doduc	3.3582
358	Mandelbrot	Whetstone	3.3332
357	Tomcatv	Mandelbrot	3.3313
356	TRFD	Mandelbrot	3.3310
355	OCEAN	Doduc	3.3224
354	ARC2D	Mandelbrot	3.3201

Table 36: Distance between programs. Distance is measured using the squared Euclidean distance.

num.	Most Similar Programs		
001	ADM	Tomcatv	0.0261
002	ADM	Nasa7	0.0318
003	Alamos	Linpack	0.0400
004	MDG	Doduc	0.0486
005	Doduc	Nasa7	0.0498
006	Doduc	Tomcatv	0.0530
007	Baskett	Smith	0.0544
008	Tomcatv	Nasa7	0.0564
009	Erathostenes	Shell	0.0577
010	BDNA	Tomcatv	0.0579
011	Alamos	Livermore	0.0608
012	Linpack	Livermore	0.0615
013	ADM	FLO52	0.0639
014	ADM	Doduc	0.0666
015	Matrix300	Nasa7	0.0673
016	Erathostenes	Smith	0.0684
017	FLO52	Nasa7	0.0685
018	Matrix300	Mandelbrot	0.0691
019	Shell	Smith	0.0699
020	Matrix300	Alamos	0.0719
021	BDNA	Nasa7	0.0735
022	ADM	OCEAN	0.0737
023	Baskett	Erathostenes	0.0739
024	BDNA	Doduc	0.0742
025	Matrix300	Linpack	0.0742

num.	Least Similar Programs		
164	Alamos	Mandelbrot	0.2634
163	Baskett	Mandelbrot	0.2578
162	Livermore	Mandelbrot	0.2559
161	Fpppp	Linpack	0.2484
160	Fpppp	Erathostenes	0.2480
159	Fpppp	Alamos	0.2471
158	Fpppp	Erathostenes	0.2480
157	Fpppp	Linpack	0.2484
156	Livermore	Mandelbrot	0.2559
155	Baskett	Mandelbrot	0.2578
154	Alamos	Mandelbrot	0.2634
153	Fpppp	Shell	0.2813
152	Alamos	Smith	0.2870
151	OCEAN	Spice2g6	0.3000
150	OCEAN	ARC2D	0.3064
149	Linpack	Smith	0.3104
148	Alamos	Baskett	0.3353
147	MDG	DYFESM	0.3415
146	BDNA	DYFESM	0.3457
145	Livermore	Smith	0.3832
144	Mandelbrot	Smith	0.4239
143	MDG	OCEAN	0.4325
142	Alamos	Shell	0.4371
141	Alamos	Erathostenes	0.4480
140	BDNA	OCEAN	0.4537

Table 37: Distance between programs. Distance is computed using the real execution times and the coefficient of variation of variable $z_{A,B,i} = t_{A,i}/t_{B,i}$. Only pairs of programs with five or more benchmark results on the same machines are reported.

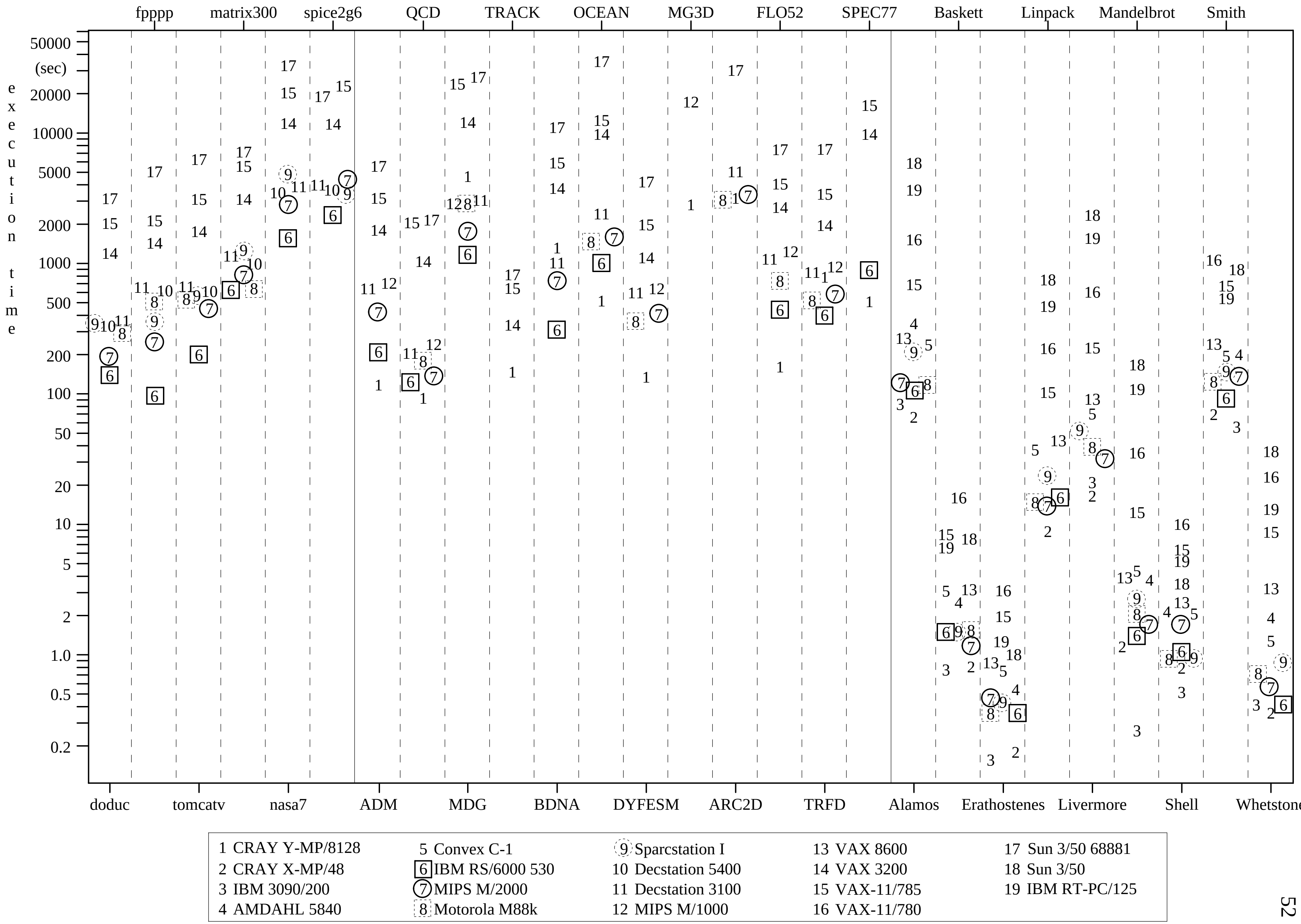


Figure 17: Distribution of execution times. Similar programs seem to produce similar distributions; the corresponding ratios of execution times on all machines are close to the same constant. *ALAMOS*, *LINPACK*, and *LIVERMORE* are clear examples of program similarity with respect to their execution time distributions.