Performance Prediction by Benchmark and Machine Analysis†

Rafael H. Saavedra-Barrera‡
Alan Jay Smith‡

ABSTRACT

We present a new methodology for CPU performance evaluation based on the concept of an abstract machine model and contrast it with benchmarking. The model consists of a set of abstract parameters representing the basic operations and constructs supported by a particular programming language. The model is machine-independent, and is thus a convenient medium for comparing machines with different instruction sets. A special program, called the machine characterizer, is used to measure the execution times of all abstract parameters. Frequency counts of parameter executions are obtained by instrumenting and running programs of interest. By combining the machine and program characterizations we can and do obtain accurate execution time predictions. This abstract model also permits us to formalize concepts like machine and program similarity.

A wide variety of computers, from low-end workstations to high-end supercomputers, have been analyzed, as have a large number of standard benchmark programs, including the SPEC scientific benchmarks. We present many of these results, and use them to discuss variations in machine performance and weaknesses in individual benchmarks. We also present some of our results in evaluating optimizing compilers.

Introduction

Comparing the CPU performance of different machines is a problem that has confronted designers and users for many years. Nowadays, the most widely used method is benchmarking. It consists of running a set of programs and measuring their execution times [Pric89, Hinn88]. The advantage to benchmarking is that it yields measurements of real programs running on real computers. There are several shortcomings to benchmarking, however, one of which has been the problem that many standard benchmarks, e.g. Dhrystone [Weic84], are considered to be substantially unrepresentative of 'normal' workloads. Two efforts to create a set of realistic

† 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 grant MIP-8713274, by the State of California under the MICRO program, and by the International Business Machines Corporation, Philips Laboratories/Signetics, Apple Computer Corporation, and Digital Equipment Corporation.

‡ Computer Science Division, EECS Department, University of California, Berkeley, California 94720.

benchmarks are the SPEC [SPEC89] and Perfect Club [Cybe90] suites. However, even when a set of benchmarks is carefully assembled, there are some limitations to the technique [Worl84, Dong87]: 1) It is very difficult to explain benchmark results from the characteristics of the machines. 2) It is not clear how to combine individual measurements to obtain a meaningful evaluation of various systems. 3) Using benchmark results it is not possible to predict and/or extrapolate the expected performance for arbitrary programs. 4) The benchmarks may still not be representative, and/or may not do the kind of computation (e.g. integer vs. floating point) expected. 5) The large variability in the performance of highly optimized computers is difficult to characterize with benchmarks.

Another approach to machine performance evaluation is to model the machine at the instruction set level. This approach suffers from several problems: (a) It is very difficult to construct an accurate model of machine operation, including all pipeline delays. (b) A very large number of parameters are needed for such a model. (c) The frequency of machine operations may be hard to know or estimate.

We can consider benchmarking and machine models as being located at opposite extremes of a spectrum. Machine models are limited by their complexity and machine specificity. One the other hand, benchmarking lacks any kind of model, and without it, it is not possible to predict or explain benchmark results.

Our particular approach and the subject of this paper has been to develop a new methodology which attempts to overcome the limitations of both benchmarking and machine models, while retaining their particular advantages. Our solution, which we call *Abstract Machine Performance Characterization*, consists of building a machine-independent model based on the set of operations used in source programs. Because the model is machine-independent, it applies to all computer systems running that programming language, independent of their particular instruction sets.

Machine characterization is accomplished by measuring the execution time of individual parameters via 'narrow spectrum' benchmarking. This produces a vector of measurements for all the abstract operations that determine the execution time of programs. We characterize programs by the number and type of abstract operations they execute. Using the set of measurements from the machines and programs, both expressed in terms of the same abstract model, it is simple to combine these to produce execution time estimates for arbitrary machine-program combinations. Execution time can also be decomposed in terms of what the programs does and how the machine performs with respect to individual operations. This relation between machine characterization, program characterization, and execution time can be expressed with following equation.

$$T_{A,M} = C_{total} \sum_{i=1}^{n} C_i P_i = C_{total} \mathbf{C_A} \cdot \mathbf{P_M} \quad (1)$$

where $\mathbf{P_M} = \langle P_1, P_2, \cdots, P_n \rangle$ represents the machine characterization, $\mathbf{C_A} = \langle C_1, C_2, \cdots, C_n \rangle$ and C_{total} represent the program characterization, and $T_{A,M}$ the execution time of program A on machine M. A graphical representation of this process can be seen in figure 1.

It is important to stress that we don't consider our approach and benchmarking to be incompatible. On the contrary, we regard our approach as a more general methodology encompassing

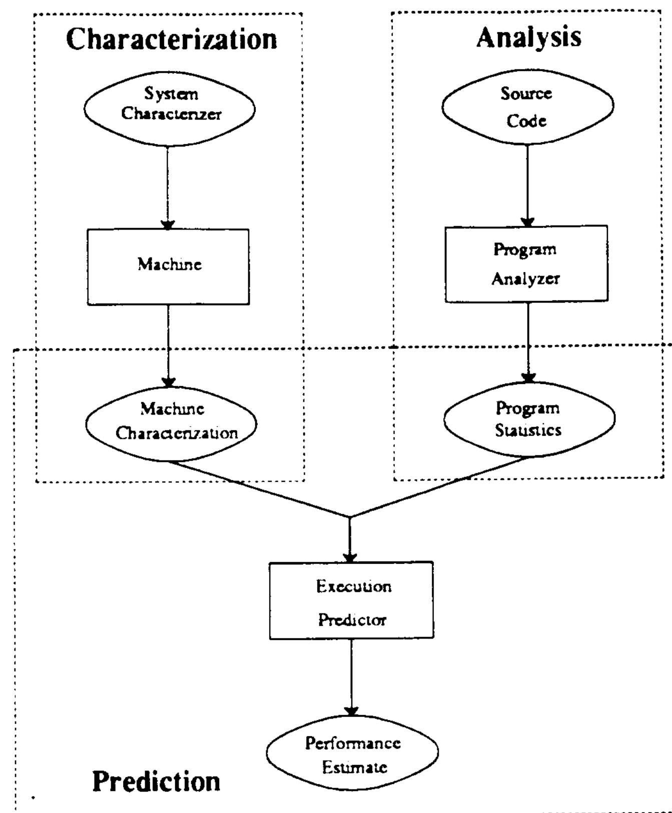

Figure 1: The process of characterization, analysis and prediction. There exists one performance vector ($\mathbf{P_M}$) for each machine, and a set of statistics ($\mathbf{C_A}$ and C_{total}) for each program. Execution time estimates are obtained by merging the machine characterizations with the program statistics.

benchmarking and building on it. The abstract machine model has several advantages not present in benchmarking or machine models. Some of these are: 1) A single benchmark (the machine characterizer) is used to completely characterize the performance of a machine, instead of using a large set of benchmarks that only provide unrelated observations of performance. 2) It is possible to compare different architectures or implementation of the same architecture in a machine independent way, and to study how different machines react to the most time-consuming sections of the program. 3) Machines can be compared at many different levels: individual abstract operations, functional units, programs, and workloads. 4) The machine abstract model can be used to easily select, from a large set of machines, those which satisfy some performance requirements. 5) Making a detailed analysis of benchmarks helps to understand those machine features that they test. 6) Benchmarking real machines is a time consuming activity; running N benchmarks in M machines require $N \cdot M$ steps. In contrast using the abstract machine model requires only $N + M$ machine and program characterizations[1].

[1] Although we still need to produce $N \cdot M$ predictions, the amount of work require to do this is negligible.

In what follows we discuss the abstract machine model in more detail, showing how it is used to characterize and compare machines, programs, and optimizing compilers. We give measurements for real machines and programs, and discuss the significance of some of the results. Then we present metrics for program and machine similarity. These allow us to quantify the extent to which performance on two machines will be proportional across a range of programs. We finish by presenting some of the work we are doing in characterizing optimizing compilers. We note that this paper is only a short summary overview of our work on this topic; further information can be found in [Saav89, Saav90, Saav91].

The Abstract Machine Model

Every programming language can be viewed as defining an abstract machine model, as specified in its language constructs and basic operations. Therefore, a programming language allows us to consider different machines as emulators of a single abstract machine. Execution time depends only on the compiler and the underlying machine and thus programming languages provide an ideal vehicle for building machine-independent models.

In our research, we have focussed on performance in a scientific environment, and have therefore built our model around the Fortran language; despite its antiquity, Fortran is still the most widely used programming language for scientific applications. A similar abstract model could be created for most other algorithmic languages. In order to distinguish as much as possible between the compiler and the underlying machine, we have initially considered only unoptimized code; at the end of this paper, we discuss and evaluate optimizing compilers.

Our Fortran abstract machine model contains 109 parameters. Each can be classified in one of the following broad categories: arithmetic and logical, procedure calls, array references, branching and iteration, and intrinsic functions. For example, there are individual parameters for each of the arithmetic operations (add/subtract, multiply, division, and exponentiation), for each data type (real, integer, and complex), and in most cases for each precision (single, double). We decided which parameters to include in our model in an iterative manner. Initially we associated parameters with obvious basic operations, and after a first version of the system was running, new parameters were incorporated to distinguish between different uses and execution times of the 'same' abstract operation in the program. This was mainly the result of detecting a significant error between our predictions and real execution times. Thus, the number of parameters has increased from 74 in [Saav88], to 102 in [Saav89], and to 109 currently. Although every basic operation in Fortran is characterized by some parameter, we have made simplifications in operations which were rarely executed in the benchmarks we used.

We can illustrate how the abstract parameters of the model relate to characteristics of the machines and programs with an example. The following code fragment represents one of the most executed basic blocks found in the NAS Kernels [Bail85].

```
      DO 110 I = 1, L
         C(I,K) = C(I,K) + A(I,J) * B(J,K) + A(I,J+1) * B(J+1,K) +
     1            A(I,J+2) * B(J+2,K) + A(I,J+3) * B(J+3,K)
110   CONTINUE
```

A static analysis of the code decomposes the statements into abstract operations and associates these with a unique basic block. Hence the four lines of code can be 'compiled' into the set of abstract operations shown in table 1.

Opers.	Mnem	Description
1	LOIN	DO LOOP initialization and bounds check
1	SRDL	store of a double precision real
4	ARDL	double precision floating point add
4	MRDL	double precision floating point multiply
10	ARR2	references to 2-D array elements
6	ADDI	add between an index and a constant
1	LOOV	DO LOOP overhead: increment, check, branch

Table 1: Static statistics for one of the loops in the NAS Kernels.

By supplementing the static analysis given above with dynamic counters indicating the number of times each basic block is executed, we obtain a count of the number of times each operation occurs. Given the time for each operation, we can then compute the execution time of this source code fragment.

$$Time = N_1 \cdot T_{LOIN} + N_2 (T_{SRDL} + 4 \cdot T_{ARDL} + 4 \cdot T_{MRDL} + 10 \cdot T_{ARR2} + 6 \cdot T_{ADDI} + T_{LOOV}).$$

N_1 corresponds to the number of times that the basic block containing the loop executes, and N_2 is the number of times that the body of the loop executes.

Machine Characterizer

The machine characterizer (MC) consists of 109 'software experiments' that measure the performance of each individual abstract parameter. The MC is written as a Fortran program and runs from 200 seconds, on machines with good clock resolution, to 2000 seconds on machines with 1/60'th or /100'th second resolution. We have run the MC on many different machines ranging from low-end workstations to supercomputers. Each experiment tries to measure the execution time that each parameter takes to execute in 'typical' Fortran programs. This typical execution time was obtained by looking at real programs and also by modifying those experiments that were identified as generating the biggest error in our predictions.

The general approach to measuring the execution times of parameters has been to time two versions of a loop, one with the parameter of interest in it, and one without. The main difficulties in measuring the performance of abstract parameters lie in their very small execution times, ranging from nanoseconds to hundreds of microseconds and the crudeness of the timing tools available in most machines; clocks normally have a resolution of only 1/60th or 1/100th of a second. Accuracy is obtained by running through the loop many times, and then running the overall loop test itself many times. Some parameters can't be measured quite so easily, and must be derived as the difference of other operation times [Saav89]. Despite a careful statistical approach to these measurements, there are some residual sources of error: the resolution, overhead and intrusiveness of the measuring tools; external events like interrupts, multiprogramming and I/O activity; variations in the hit ratio of the memory cache, and paging [Clap86].

From Basic to Reduced Parameters

Vector P_M in eq. 1 corresponds to the characterization of machine M in terms of the Fortran basic operations. It represents our fundamental measurement of performance, and from it, all predictions and metrics are computed. Unfortunately, it is very difficult to understand the meaning of an 109 element vector. For this reason we have defined a set of seventeen 'reduced' parameters that consolidate the original 109. The reduced parameters are obtained from the basic ones by aggregating those that exercise a similar functional unit in the processor and assigning them weights according to how frequently they are executed by programs.

Reduced Parameters

1	memory bandwidth (single)	10	double precision arithmetic
2	memory bandwidth (double)	11	intrinsic functions (single)
3	integer addition	12	intrinsic functions (double)
4	integer multiplication	13	logical operations
5	integer arithmetic	14	pipelining
6	floating point addition	15	procedure calls
7	floating point multiplication	16	address computation
8	floating point arithmetic	17	iteration
9	complex arithmetic		

Table 2: The seventeen reduced parameters. Integer and floating point arithmetic refer to all arithmetic operations, except addition and multiplication.

Table 2 shows the list of the seventeen reduced parameters. Most of the parameters deal with arithmetic characteristics, as would be expected for a language like Fortran. There are hardware, software, and hybrid parameters. Hybrid parameters are those that are implemented in hardware on some machine and in software on others. Parameters characterizing hardware functional units are: integer addition and multiplication, logical operations, procedure calls, looping, pipelining, and memory bandwidth (single and double precision). Pipelining comprises the different types of Fortran 'goto' statements. Software characteristics are represented by (intrinsic) trigonometric functions (single and double precision). Floating point, double precision and complex arithmetic, and address computation belong to the hybrid class.

A very convenient graphical way of representing the reduced parameters is to use a modified version of Kiviat graphs. Here the absolute or normalized values of the parameters are plotted on a logarithmic scale around a circle. In this way we can convey very effectively how different machines distribute their performance. We call these figures *pershapes* (performance shapes), and each system has a unique pershape. Figure 2 shows the pershapes for 2 supercomputers, 2 mainframes, 11 workstations and several implementations of the VAX architecture. The performance differences between two compilers or the impact a floating point co-processor are clearly reflected in the different pershapes. For example, in figure 2 we see how the use of two different Fortran compilers determine the pershape, as in the case of the *fort* and *f77* compiler on the VAX-11/785. In a similar way, the impact of using a floating point co-processor is clearly seen on the pershapes for the Sun 3/260 (the one with the co-processor is indicated by an (f)).

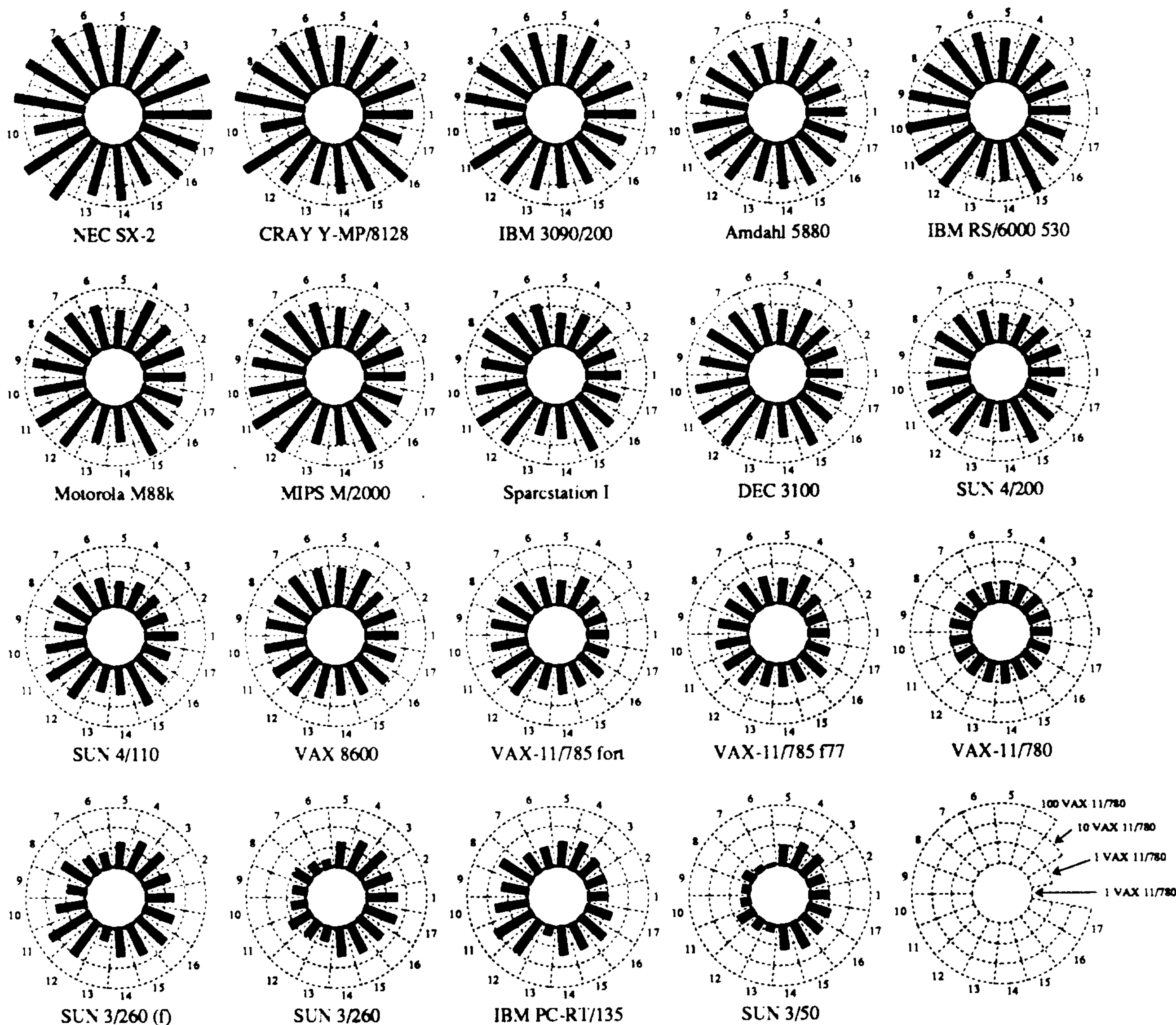

Figure 2: Performance of the reduced parameters with respect to the VAX-11/780. The concentric circles represent .1, 1, 10, and 100 times faster. The closest a performance shape (pershape) is to a circle, the closest the machine is to a VAX-11/780 in terms of how both machines distributed their performance along different computational modes.

We can trace the evolution of performance in workstations by looking at the pershapes. If we compare the pershapes for the VAX-11/780 and the Sun 3/50, we see that for floating point arithmetic (single and double precision), complex arithmetic, intrinsic functions, and logical operations the Sun 3/50 had worse performance than the VAX-11/780. Even in the Sun 3/260 with a co-processor (SUN 3/260 (f) in the figure), single precision and complex arithmetic lagged

behind that of the VAX-11/780. However, newer workstations, such as the IBM RS/6000 530, show almost two orders of magnitude improvement with respect to the Sun 3/260 in floating point and complex arithmetic. In contrast the performance improvement in integer and similar operations has been of only one order of magnitude.

Pershapes can be used to compare the relative performance of machines, and determine the extent to which they are similar. The idea is that two similar machines A and B will execute any arbitrary program P, K times faster on A than B. For dissimilar machines, for one program A may be faster, and for another, B may be faster. We define machine similarity (or pershape similarity) as the distance between two different performance shapes. We have created a similarity metric, shown below in equation (2), which is based on the 17 reduced parameters, and that has the right properties [Saav89].

$$d(X,Y) = \left[\frac{1}{n-1} \sum_{i=1}^{n} \left\{ \log(\frac{x_i}{y_i}) - \frac{1}{n} \sum_{j=1}^{n} \log(\frac{x_j}{y_j}) \right\}^2 \right]^{1/2} \qquad (2)$$

$X = <x_1, x_2, \cdots, x_n>$ and $Y = <y_1, y_2, \cdots, y_n>$ are two reduced performance vectors in $(0,\infty)^n$ representing the pershapes of machines M_X and M_Y. We can see that this metric has the desired similarity property. Consider two machines A and B, such that for every program P the ratio between their respective execution times is always a constant K. From eq. (2) we can see that, in this situation, their pershape distance is zero. Conversely, if the distance between two machines is very large, then their pershapes are very different and the spectrum of benchmark execution time ratios is large, independently of whether the machines have a similar average performance or not. A more formal discussion about the properties of this metric can be found in [Saav89].

Most Similar Machines

	machine	machine	distance
001	MIPS M/2000	DEC 3100	0.186
002	VAX 8600	VAX-11/785 f77	0.229
003	Sun 4/200	Sun 4/110	0.243
004	Sun 3/260	Sun 3/50	0.290
005	Sparcstation I	DEC 3100	0.326
006	MIPS M/2000	Sparcstation I	0.373
007	Sparcstation I	Sun 4/110	0.378
008	Sparcstation I	Sun 4/200	0.391
009	Amdahl 5880	VAX-11/785 f77	0.422
010	VAX-11/785 fort	VAX-11/785 f77	0.426
011	IBM RS/6000 530	DEC 3100	0.442
012	IBM RS/6000 530	MIPS M/2000	0.445
013	DEC 3100	Sun 4/110	0.453
014	Motorola M88k	Sun 4/200	0.456
015	DEC 3100	VAX-11/785 fort	0.462
016	MIPS M/2000	Sun 4/200	0.462
017	MIPS M/2000	Sun 4/110	0.468
018	IBM RS/6000 530	Motorola M88k	0.485
019	DEC 3100	Sun 4/200	0.495
020	VAX 8600	VAX-11/780	0.499

Least Similar Machines

	machine	machine	distance
170	IBM RS/6000 530	Sun 3/260	1.758
169	IBM RS/6000 530	Sun 3/50	1.669
168	MIPS M/2000	Sun 3/260	1.658
167	DEC 3100	Sun 3/260	1.645
166	Sparcstation I	Sun 3/260	1.605
165	MIPS M/2000	Sun 3/50	1.598
164	CRAY Y-MP/8128	Sun 3/260	1.583
163	DEC 3100	Sun 3/50	1.573
162	NEC SX-2	Sun 3/50	1.535
161	CRAY Y-MP/8128	Sun 3/50	1.527
160	Sparcstation I	Sun 3/50	1.521
159	VAx-11/785 fort	Sun 3/260	1.519
158	NEC SX-2	Sun 3/260	1.477
157	VAX-11/785 fort	Sun 3/50	1.443
156	IBM 3090/200	Sun 2/260	1.438
155	IBM 3090/200	Sun 3/50	1.425
154	Sun 4/110	Sun 3/260	1.399
153	Motorola M88k	Sun 3/260	1.370
152	Sun 4/200	Sun 3/260	1.344
151	Sun 4/110	Sun 3/50	1.326

Table 3: Pairs of machines with the smallest and largest pershape distance.

Pershapes and pershape distance can be used to compare and understand machine performance. For example, 1) normalized pershapes allow us to identify, relative to a particular

machine, the strengths and limitations of different machines. 2) We can observe the performance evolution of different implementations of the same architecture, and measure changes over time. 3) We can measure how advances in technology affect the way machine designers allocate resources to improve performance and identify those dimensions that are given more importance. 4) Pershapes metrics can be defined and used to cluster machines. In particular, machine similarity is a measurement of the potential variability of benchmark results between two machines.

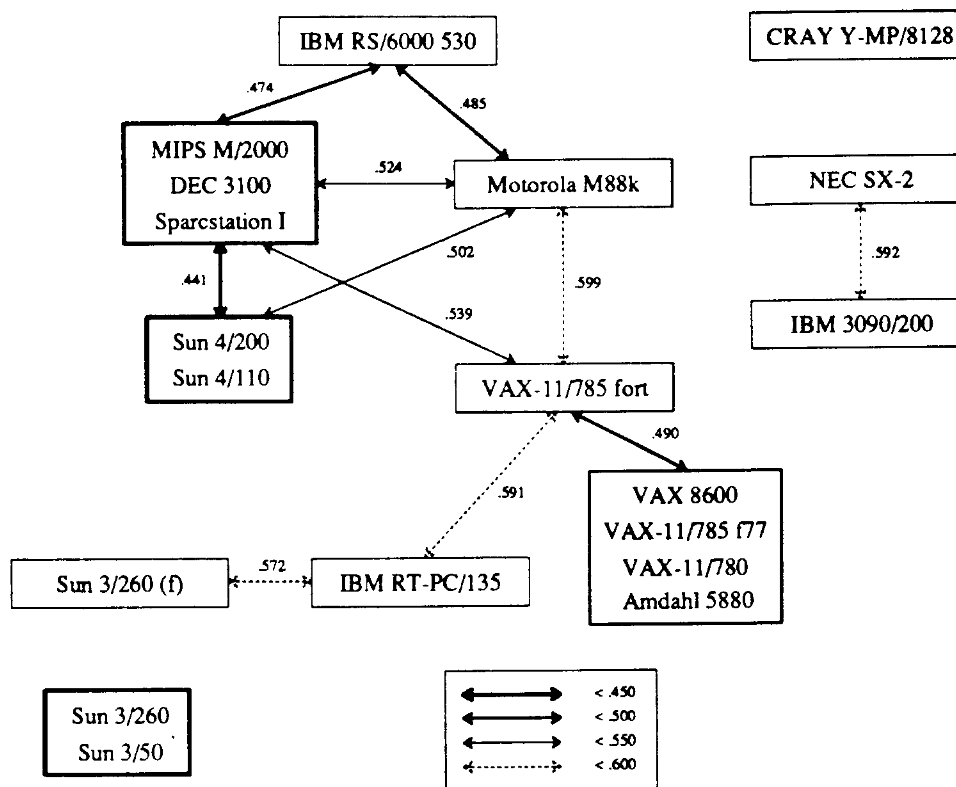

Figure 3: A clustering diagram based on machine distances. In order to make the figure more understandable, machines with very small pershape distances are depicted as a single unit.

In table 3 we give the list of the twenty smallest and twenty largest pershapes distances for the pairs of machines in figure 2. As expected, machines with similar characteristics, like the VAXes, show small distances, but there are also other interesting results. The Sparcstation I has a smaller distance to the DEC 3100 (a MIPS Co. RC2000 and RC2010 based machine) and MIPS M/2000 than to the any of the other two Sun 4 models.

Figure 3 shows the clustering of machines with respect to their distances. The distance between clusters represent the average of all distances between pairs of elements in the clusters. The figure illustrates the close similarity of modern RISC microprocessors. The supercomputers have performance distributions that are significantly different than those of the other machines;

only the NEC SX-2 and the IBM 3090/200 have a relatively small distance. A more complete exposition of pershapes and their metric can be found in [Saav89].

Program Characterization

One of our main arguments is that program characterization is important for CPU performance evaluation. Knowing the static and dynamic statistics of programs is what explains why a machine executes some programs faster, but others more slowly. There is an underlying assumption, by people who use benchmarks, that large real programs with a long execution times are always better for benchmarking purposes. In some cases this assumption is false and conclusions draw from these programs can be quite misleading. It is by making a detailed analysis of program execution that we discover what it is that a program measures and how to interpret its benchmark results.

Our program characterizer works similarly to most execution profilers [Powe83]. It does a static analysis of the source code at the basic block level and instruments the program with counters to measure, at run time, the number of times that each block is executed. Compiling the instrumented program and running it gives the dynamic counts. By combining the static statistics and the dynamic counts, we can compute the dynamic statistics and other interesting quantities. Merging the machine characterizations and the dynamic statistics allows us to make execution time predictions.

The programs we use in this paper to illustrate program characterization come from the SPEC suite [SPEC89]. The SPEC (System Performance Evaluation Cooperative) benchmarks are the results of an effort from computer manufacturers to assemble a suite of realistic and interesting benchmarks. These programs have been selected from a large sample of public domain applications. Currently the benchmark suite consists of ten programs, 6 in Fortran and 4 in C, covering scientific and system applications. Plans are underway to extend the suite to 30 programs, including some which measure I/O performance. Other efforts of SPEC have focused on the development of a methodology for measuring and reporting benchmark results. Their performance metric, the SPECmark, appears to correlate well with real performance. A major contribution of the SPEC group has been in raising the level of discussion between machine manufacturers with respect to machine performance. However, the SPEC people have focused mainly in devising better ways to measure machine performance, and have given little attention to the problem of explaining how and why machine performance is achieved, although this situation appears to be changing [SPEC90]. In this paper, and in [Saav90], we present some results of our analysis of the SPEC Fortran benchmarks; Pnevmatikatos and Hill [Pnev90] report on cache miss ratio for the four C SPEC benchmarks.

Figure 4 shows machine-independent dynamic statistics for the SPEC Fortran programs. The statistics we present are for the complete program, but in our system it is equally easy to limit them to individual subroutines or arbitrary groups of basic blocks. Figure 4 contains four graphs, each focusing on some particular aspect of the programs: the type of statements executed, the data type and precision of arithmetic and logical operations, the structure of the operands, and the distribution of blocks executed. From figure 4a, we see that the most executed statements are assignments and DO LOOPs. However, the number of branches in *spice2g6* is inordinate compared with the other benchmarks. The graph for arithmetic and logical operations (4b) is

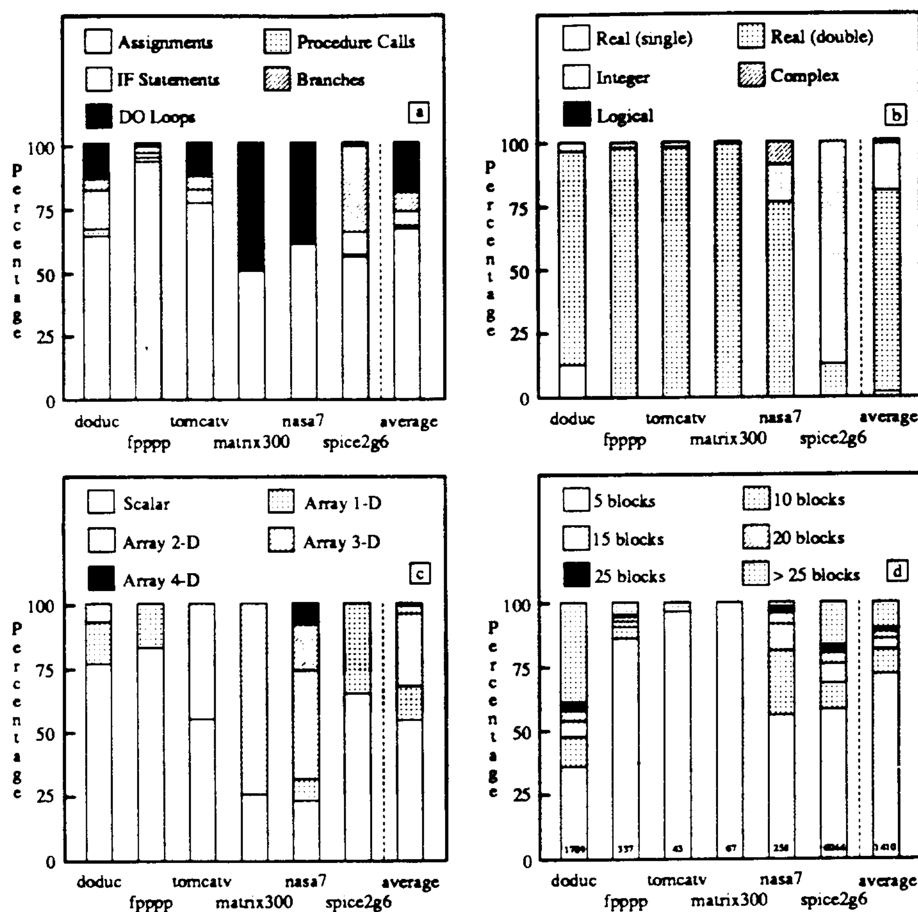

Figure 4: Dynamic statistics for the SPEC benchmarks. In 4d, the number at the bottom of each bar represents the basic blocks in the benchmark.

dominated by double precision floating point arithmetic, with the exception of *spice2g6* (using model *greycode*) in which integer arithmetic represents more than 80% of all operations. As we explain below, this is more a feature of the model used than the 'normal' behavior of *spice2g6*.

The most interesting graph is 4d, the one giving the distribution of operations with respect to basic blocks executed. At the bottom of each bar we indicate the number of basic blocks in each program. In programs *fpppp*, *tomcatv*, and *matrix300*, five blocks contain more than 85% of all operations. In fact for *matrix300*, a single basic block, containing a single statement, accounts for 99.8% of all operations. The situation is not much better for *nasa7* or *spice2g6*. *Spice2g6* represents a good example of a large program (18000 lines in 6044 blocks) that has a very large execution time (> 20000 seconds in a VAX-11/780), but where only five basic blocks account for more than 50% of all operations. The first four basic blocks are contained in less than 10 lines. From the number of blocks, we see that *tomcatv* and *matrix300* are very small programs (43 and 67 blocks). This is corroborated by the number of lines, 183 and 149 respectively (excluding

comments). These results underline the importance of taking into account, especially when we compute statistics from benchmark results, that some programs measure only very few things, while others have more complex executions. This is evident in the SPEC results for the Stardent 3010 [SPEC90]. For this machine the SPECratio results, excluding *matrix300*, range from 14.7 to 62.9[2]. However, the SPECratio for *matrix300* is 108.5. It is risky to draw conclusions from these numbers without knowing what each benchmark measures.

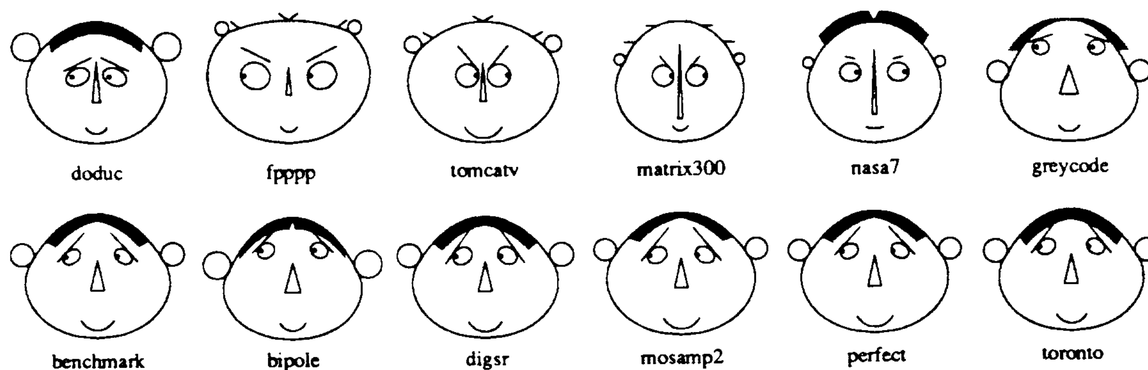

Figure 5: Chernoff faces for the SPEC Fortran benchmarks and six additional circuits for *spice2g6*. Each face is determined by 23 program characteristics.

When we analyze benchmarks in terms of a large number of characteristics, it is always useful to be able to classify them into different groups or clusters. A similar approach as the one we described for machine similarity can be used to classify benchmarks, although the properties we want to impose on the the metric are different. In [Saav90] we define a metric to benchmark similarity. In this paper we will present a more graphical approach using Chernoff faces to identify similar programs [Cher73]. A Chernoff face is a graphical method of mapping multidimensional data to facial features[3]. It has been observed that Chernoff faces are very useful for clustering, because humans have an extremely acute ability to recognize faces. In figure 5 we show examples of Chernoff faces. Each of the faces consists of 23 characteristics which include many of the measurements we showed in figure 4. The six faces in the upper row are for the SPEC benchmarks (*spice2g6* is labeled as *greycode*; the name of the circuit). On the lower portion of the figure we show Chernoff faces corresponding to six additional circuit models for *spice2g6*. If we focus only the first row we see that there are strong similarities between *tomcatv* and *matrix300*, and to a lesser degree between these two and *fpppp* and *nasa7*. On the other hand *doduc* and *greycode* show little resemblance to any of the other four faces.

[2] The SPECratio is defined as the ratio between the execution time of the measured machine and the execution time of the VAX-11/780.

[3] Standard Chernoff faces do not have hair and can be used to represent up to 20 parameters; we have made them hairy to accommodate 3 more parameters.

Interesting observations can be made on the Chernoff faces for the seven models of *spice2g6*. These are represented by the lower faces plus *greycode*. From these we immediately notice that five of the seven faces are extremely similar. Only *bipole* and *greycode* look different, but even here the difference from *greycode* to the other models is more conspicuous. We already mentioned that *spice2g6* using *greycode* is dominated by integer arithmetic and not by double precision floating point operations. Although it is not clear from the Chernoff faces, the other circuit models are dominated by double precision arithmetic, as users of this benchmark normally assume. Another difference between the seven circuits is that *greycode* requires two order of magnitude more time to execute. We have looked in some detail at the execution of *greycode* and found that what this model is measuring is more the performance limitations of the data structures in *spice2g6* when it analyzes large circuits, than the behavior of CAD algorithms.

Our study of the SPEC Fortran benchmarks indicates that there are some significant improvements that can be made to the suite. In particular, two of the benchmarks, *matrix300* and *spice2g6*, have problems. The current model for *spice2g6* should be replaced with a different model with a more interesting execution pattern. As we mentioned, the *greycode* model makes *spice2g6* behave as an integer benchmark and not as double precision floating point. Furthermore, a small number of simple basic blocks dominate most of the execution time. A better model should test a larger fraction of the program and exercise more of the main CAD algorithms. The *benchmark* model which contains several small circuits would be a good choice. Additional circuits could be added to increase its execution time.

With respect to *matrix300* we think that it should be replaced by a linear algebra application which spends a significant fraction of time in LINPACK routines other than SAXPY. As our statistics show, in *matrix300*, 99.8% of the operations are in a single basic block. This makes this benchmark a potential temptation for compiler writers to superoptimize. Once a suite of benchmarks establishes itself as a de facto 'standard', compiler writers and machine designers extensively analyze them and incorporate in their new machines and compilers changes which produce, for that particular suite, the largest improvements in the execution time. If the benchmarks are too simple or do not represent real workloads, the apparent improvements will not translate in actual gains for users' applications. The best way to prevent this is by knowing what each benchmark measures and replacing those that are inadequate.

Execution Time Prediction

Any execution time model needs to be able to make accurate execution time predictions for real programs in order to be considered credible and useful. It is only by comparing the predictions to actual measurements that we can quantify the accuracy of the model, region of validity, and robustness. Once we have validated the model we can use it to predict execution times, to study how sensitive the execution time is to changes in the workload, to assess the impact of new algorithms, etc.

We have predicted and compared execution time predictions for a large set of programs and machines, and we have found that our predictions are quite accurate. For a sample of more than 244 machine-program combinations, using 20 machines and 28 programs, we have found that 55% of the predictions lie within 10% of the real execution time, 80% within 20%, and 94% within 30%. In figure 6 we show two graphs, that for two machines (IBM RS/6000 530 and

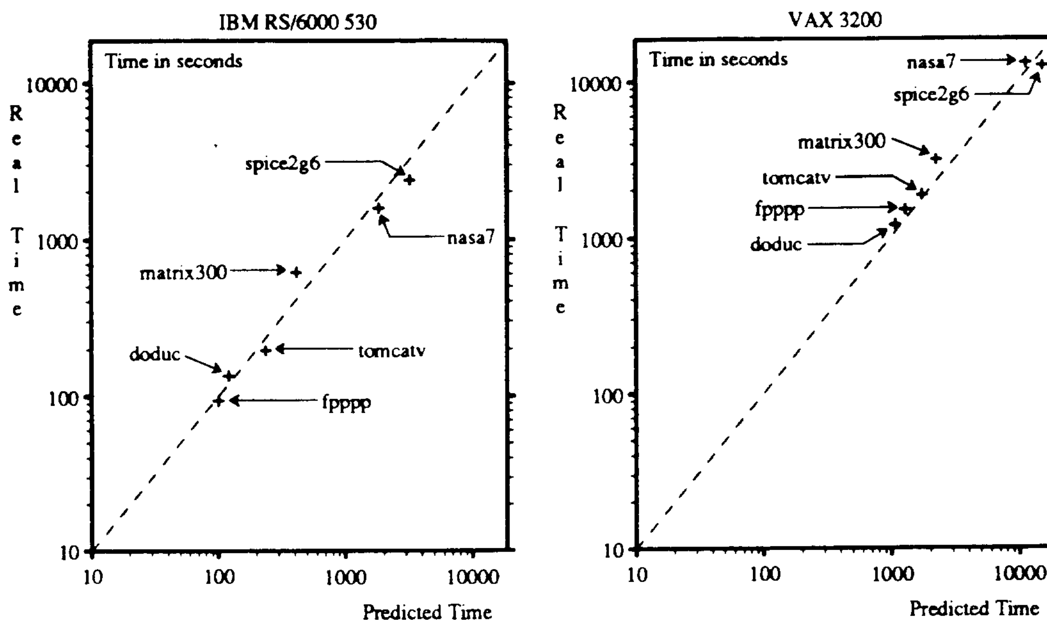

Figure 6: Comparison between real execution times and predictions for the IBM RS/6000 530 and VAX 3200.

VAX 3200) and the six SPEC Fortran programs, present a comparison between actual execution times and predictions. The execution times range from less than 100 seconds to more than 15000.

Benchmark results are often reduced to a single number representing the absolute performance of the machine or its relative performance with respect to a baseline machine. A common approach in benchmarking is to compute the geometric mean of the execution times normalized by the results of the VAX-11/780. This is the approach taken in the SPEC benchmarks. Here we will show, using the same performance metric, how close the single number performance estimates computed using our predictions and those obtained from real execution times are.

In table 4 we present both the actual and predicted geometric means for twelve machines ranging from the CRAY X-MP to the Sun 3/50. All programs were executed in scalar mode and without optimization, which accounts for the fact that many of our figures are smaller than reported SPECmarks. In all cases the difference between the predicted and real value is less than 6%; the average is +0.22%. Clearly, our methodology yields good predictions of relative performance.

Considerable improvement over the figures we report can be obtained by enabling compiler optimization. For example, the execution time for benchmark *tomcatv* decreases from 196.1 to 36.1 seconds on the IBM RS/6000 530 when optimization is enabled. In the next section we will see that much of the high performance on the IBM RS/6000 series depends on its highly optimizing compiler.

	Cray X-MP/48	IBM 3090/200	Amdahl 5840	Convex C-1	IBM RS/6000 530	Sparcstation I
actual mean	26.25	33.79	6.47	7.36	16.29	11.13
prediction	26.07	32.27	6.71	6.99	15.69	10.58
difference	+0.69%	−4.50%	+3.71%	−5.03%	−3.68%	−4.94

	Motorola 88k	MIPS M/2000	VAX 8600	VAX−11/785	IBM RT−PC/125	Sun 3/50
actual mean	14.24	13.88	5.87	2.01	0.95	0.69
prediction	15.34	13.70	5.63	2.12	0.99	0.72
difference	+7.72%	−1.30%	−4.09%	+5.47%	+4.21%	+4.35%

Table 4: Real and predicted geometric means of normalized benchmark results. Execution times are normalized with respect to the VAX-11/780.

Compiler Optimization

Thus far, we have presented our linear model as if a given source program were simply and directly translated into the corresponding machine code. In reality, compilers optimize the code, even when the optimizer is nominally inactive (optimization level 0). Including the effect of optimizations in our analysis is a difficult problem and is currently under study; we summarize some of the issues and results in this section.

The problem of evaluating the effects of compiler optimizers can be broken down into three subproblems. 1) The experimental detection of which optimizations can be applied by the optimizer –i.e. what can the compiler optimizer do. 2) The measurement of the performance improvement that a particular optimization will produce in a program. 3) The measurements of the possible optimizations present in the source code. Each is discussed below.

The first point is similar to machine characterization, but instead of measuring the performance of some operation, we are interested in detecting which optimizations can be applied by the compiler and in which cases. This may appear as an easy task, but unfortunately, in many compilers, optimizations are implemented ad-hoc; e.g. a given transformation may be used for only one data type and not another, although the transformation is type independent [Lind86].

We have written a benchmark program to detect scalar optimizations in Fortran compilers; see [Aho86] for a discussion of compiler optimization. It contains experiments that check whether individual optimizations can be detected inside a basic block (local) and across basic blocks (global). We also test that the optimizations work on integers, reals, and mixed (integer and real) expressions. All optimizations we detect are machine independent.

In table 5 and 6 we present a summary of the results found for eight Fortran compilers for local optimizations; see [Saav91] for further discussion. In addition to 'no' or 'yes' that represent the obvious situations when the optimizer cannot detect any opportunity or detects all instances, we also use 'marginal' and 'partial'. Marginal is used when the optimization is only detected in some special cases. Partial means that the optimization is detected in most, but not all situations. For example, if an optimization is applied on expressions containing integers or reals, but not when both are present, we use 'partial'. If it is only applied on one data type we use 'marginal'.

compiler	constant folding	common subexpr elim	code motion	copy propagation	dead code elimination
Ultrix F77 1.1	no	partial	marginal	partial	no
Mips F77 1.21 -O2 Mips F77 1.21 -O1	partial marginal	yes yes	yes no	partial marginal	yes no
Sun F77 -O3 Sun F77 -O2 Sun F77 -O1	marginal marginal no	yes yes no	yes yes no	no no no	yes partial no
Ultrix Fort 4.5	yes	yes	yes	yes	yes
Amdahl F77 2.0	no	no	no	no	no
CRAY CFT77 4.0.1	yes	yes	yes	yes	yes
IBM XL Fortran 1.1	yes	partial	yes	partial	yes
Motorola F77 2.0b3	marginal	yes	yes	no	no

Table 5: Summary of the effectiveness of compilers in applying local optimizations (1 of 2).

compiler	strength reduction	address calculation	inline substitution	loop unrolling
Ultrix F77 v.1	partial	marginal	no	no
Mips F77 1.21 -O2 Mips F77 1.21 -O1	yes no	yes yes	marginal no	yes no
Sun F77 -O3 Sun F77 -O2 Sun F77 -O1	partial partial no	marginal no no	no no no	yes yes yes
Ultrix Fort 4.5	yes	yes	no	no
Amdahl F77 2.0	no	no	no	no
CRAY CFT77 4.0.1	yes	yes	yes	yes
IBM XL Fortran 1.1	yes	yes	partial	yes
Motorola F77 2.0b3	partial	no	no	no

Table 6: Summary of the effectiveness of compilers in applying local optimizations (2 of 2).

Inline substitution can be used to illustrate how tricky is to evaluate an optimizing compiler. Calls to leaf procedures, those that do not call other procedures, can be eliminated, and potential optimization exposed, by inserting the callee's code at the point of call, after making a proper substitution for the formal parameters. In table 6 we see that our test detected that three compilers have some ability to inline procedures, but only the CFT77 compiler takes full advantage of it. In the case of MIPS f77 1.21, the compiler does not perform an actual inline substitution. The only transformation done is that the compiler does not use a new stack frame for the leaf procedure, but instead execution is carried out on the caller's frame. In contrast, a real inline substitution is done by IBM XLF 1.1, but here the insertion of unnecessary extra code obscures optimizations that inlining should have exposed. Only the CFT77 compiler was abled to detect all optimizations present after proper inlining.

The second subproblem mentioned above deals with quantification of the performance benefits of individual transformations. Several studies have tried to measure the performance effect of some algorithms for code improvement [Cock80, Chow83]. Most have been carried out in the process of engineering an optimizing compiler and performance evaluation has not been its main driving force; [Rich89] is an exception. In addition, some of the studies have used a set of small programs with a very regular structure like matrix multiplication, FFT, Baskett puzzle, etc, in which the execution time is significantly reduced by applying one or two transformations.

compiler	average	std. dev.
Ultrix F77 1.1	.7781	.1295
Mips F77 1.21 -O2	.5117	.1898
Mips F77 1.21 -O1	.8420	.0564
Sun F77 -O3	.5230	.2421
Sun F77 -O2	.5537	.2038
CRAY CFT77 4.0.1	.5640	.2933
IBM XL Fortran 1.1	.2849	.1381
Motorola F77 2.0b3	.6869	.1404

Table 7: Average decrease (the ratio between optimized and non-optimized execution time) produced by several optimizers and levels of optimization observed on the Perfect Club benchmarks.

The third subproblem refers to the amount of optimizable code present in real applications. Detecting which optimizations are done by optimizers and measuring their performance effects is only part of the problem; we also need to know the extent to which programs contain optimizable code. The measurement of optimization opportunities in programs could potentially be done by modifying an existing highly optimizing compiler.

Although the last two problems are important, and each will require a significant effort to solve them, in our research we have adopted a different strategy. We have used a suite of large scientific programs to measure the average improvement produced by different optimizing compilers, and have also used these results to investigate the amount of correlation found between optimizers. Table 7 gives results with respect to the average change (ratio of final to initial) in execution time obtained when optimization is enabled. We also give the standard deviation. It is important to realize that a larger reduction in execution time does not necessarily mean that the optimization is better, but only that the improvement to the original object code was more significant. A code generator that produces very bad code before optimization will yield a larger reduction in execution time after optimization.

An interesting question to consider is whether different compilers correlate in their ability to improve the execution time of individual programs. First, recall that we can predict execution times for nonoptimized programs. Now, if we find that there exist a significant correlation between two optimizers, then knowing how much one optimizer reduces execution the time will allow us estimate the reduction of the other optimizer. Thus, this give us a way of predicting execution times before and after optimization. In Figure 7 we show, for those compilers in table 7, graphs of execution time reductions for all pairwise combinations of compilers. We include the

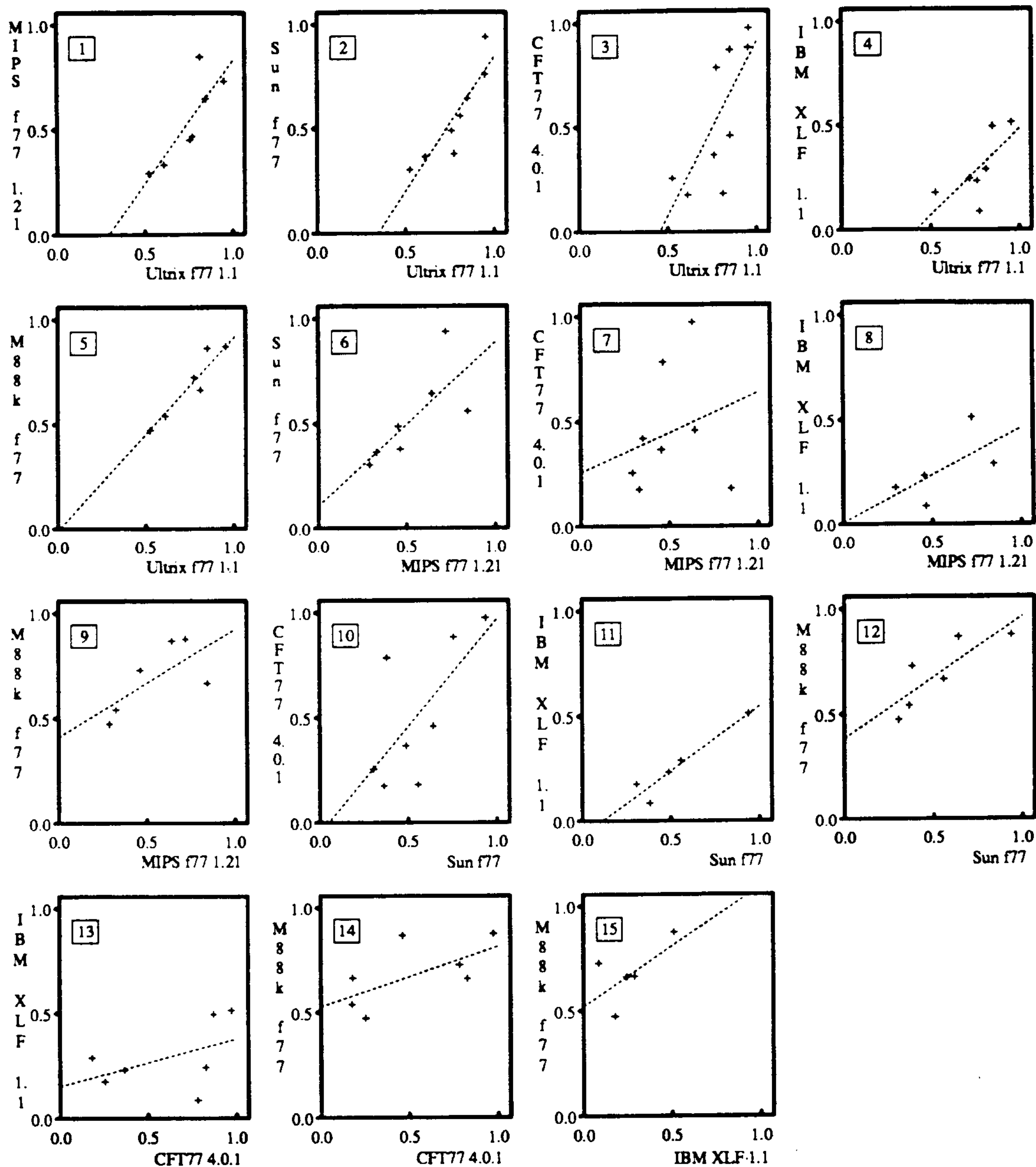

Figure 7: Correlation between execution time improvements of various optimizing compilers. Each graph includes the best linear fit.

best linear fit to the set of points.

As expected there is a positive correlation between all optimizers; on the average more improvement by one compiler means more improvement in the other. All the coefficients of correlation, except three, are greater than .6300. However only those for graphs 1, 2, 3, 5, 6, 11,

and 12, are statistically significant at a level of 0.025. An analysis of the code produced by the optimizers on our suite shows that the optimizations having the most effect are: 1) Reducing the number of loads and unnecessary stores by keeping temporaries in registers. 2) The elimination of expensive address computations by computing a base address before the first iteration and updating it with an add in all subsequent iterations. 3) Moving invariant code outside the innermost loops.

Several problems arise when we deal with compiler optimizers and attempt to measure their effects. The first is that most optimizations cannot be measured in isolation. It is common that the opportunity to apply some optimization is the result of a previous transformation, and in some cases the first transformation may not necessarily improve the execution time of the program. A second problem is machine-dependent optimizations. Proposing a general framework for the evaluation of optimizers hampers our ability to evaluate machine-dependent optimizations. Most of these optimizations do not have an equivalent in other architectures, and it may be argued that these transformations are not really optimizations of the source code, but deal with the efficient use of the machine resources. However, some attempt should be made to measure the effect of machine-dependent optimizations and compare their effectiveness against those that are machine-independent.

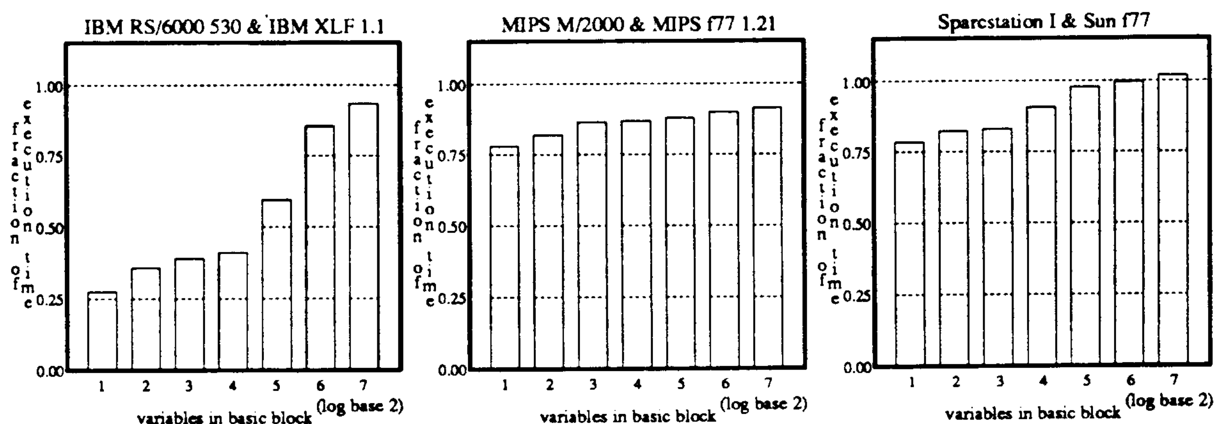

Figure 8: The effect of register allocation for three different machine-compiler combinations. The numbers on the x-axis are the logarithm base 2 of the number of variables that the optimizer must consider as candidates for registers.

A machine-dependent optimization that we have attempted to measure is register allocation. The speedup obtained by this optimization depends not only on the quality of the algorithm, but also on the number of registers and the time differential between using a register as opposed to loading the value from the cache or memory. Even when the program does not show opportunities for optimization, the optimizer improves the code by doing peephole optimization, so it is not possible to measure the register allocation speedup in isolation. Nevertheless, it is possible to detect the range of improvement derived from this optimization. We have written tests where the

only opportunities for improvement are register allocation and peephole optimization. By increasing the variables referenced inside a loop, while keeping everything else constant, we make it more difficult for the optimizer to keep all variables in registers. As a smaller fraction of variables are kept in registers, the optimized execution time tends to get closer to the non-optimized version.

We present results for three machine-compiler pairs in figure 8. Each graph shows how much optimization changes the execution time as a function of the logarithm of the number of distinct variable referenced inside the main loop. The results indicate that the gap between optimized and non-optimized execution times decreases as the number of variables increases; this is clearest for the IBM XLF compiler. The IBM RS/6000 contains in addition to 32 integer and 32 floating point registers, register renaming to prevent stalling, and a very effective register allocation algorithm. The MIPS f77 and Sun f77 compilers show similar improvement from register allocation, but the effect of increasing the 'size' of the basic blocks is more steep for the Sun f77 optimizer.

Conclusions

We have presented a new model for CPU performance characterization and discussed some of its more prominent features. The main advantage of this approach is that it permits one to model performance using a machine-independent model that is applicable to arbitrary uniprocessors. Machines are characterized with respect to set of abstract parameters that are measured experimentally. Likewise, using the same model, dynamic statistics of programs are obtained. These statistics represent the behavior of the program at execution.

By combining the machine and program characterizations, it is possible to predict with good accuracy execution times for arbitrary large programs over a wide spectrum of machines. In addition, it is possible to combine individual measurements into a set of reduced parameters that better represent hardware or software components. This permits us to identify the performance strengths and weaknesses of machines. We have defined abstract concepts like machine and program similarity that further increase our insight about how machines and programs behave. Currently we are extending our study to include the improvement produced by the use of optimizing compilers. We believe that the results we have obtained with the abstract machine model gives ground to our optimism that a much complete and insightful methodology than benchmarking exists in comparing the performance of different machines.

References

[Aho86] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass., 1986.

[Bail85] D.H. Bailey and J.T. Barton, "The NAS Kernel Benchmark Program", NASA Technical Memorandum 86711, August 1985.

[Cher73] H. Chernoff, "The Use of Faces to Represent Points in k-Dimensional Space Graphically", *J. of the American Statistical Association*, Vol. 68, No. 342, June 1973, pp. 361-368.

[Chow83] F. Chow, "A Portable Machine-Independent Global Optimizer - Design and Measurements", Ph.D. dissertation, Technical Report 83-254, Computer Systems

Laboratory, Stanford University, (December 1983).

[Clap86] R.M. Clapp, L. Duchesneau, R.A. Volz, and T. Schultze, " Toward Real-Time Performance Benchmarks for ADA", *Communications of the ACM*, Vol. 29, No. 8, August 1986, pp. 760-778.

[Cock80] J. Cocke and P. Markstein, "Measurement of Program Improvement Algorithms", *Research Report*, IBM, Yorktown Heights, RC 8110, July 1980.

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

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

[Hinn88] David Hinnant, "Accurate Unix Benchmarking: Art, Science or Black Magic?", IEEE MICRO, October, 1988, pp. 64-75.

[Lind86] D.S. Lindsay, "Methodology for Determining the Effect of Optimizing Compilers", *CMG 1986 Conference Proceedings*", Las Vegas, Nevada, pp. 379-385, December 9-12, 1986.

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

[Powe83] L.R. Powers, "Design and Use of a Program Execution Analyzer", *IBM Systems Journal*, Vol. 22, No.3, 1983, pp. 271-292.

[Pric89] Walter Price, "A Benchmark Tutorial", IEEE MICRO, October, 1989, pp. 28-43.

[Rich89] S. Richardson and M. Ganapathi, "Interprocedural Optimization: Experimental Results", *Software–Practice and Experience*, Vol. 19, No. 2, pp. 149-170, February 1989.

[Saav88] R.H. Saavedra-Barrera, "Machine Characterization and Benchmark Performance Prediction", University of California, Berkeley, Technical Report No. UCB/CSD 88/437, June 1988.

[Saav89] R.H. Saavedra-Barrera, A.J. Smith, and E. Miya, "Machine Characterization Based on an Abstract High Level Language Machine", Technical Report UCB/CSD 89/494, March 13, 1989, UC Berkeley, CS Division.

[Saav90] R.H. Saavedra-Barrera and A.J. Smith, "Analysis of Benchmark Characteristics and Performance Prediction", paper in preparation.

[Saav91] R.H. Saavedra-Barrera and A.J. Smith, "Benchmarking Optimizer Compilers", paper in preparation.

[SPEC89] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol. 1, Issue 1, Fall 1989.

[SPEC90] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol. 2, Issue 2, Spring 1990.

[Weic84] Reinhold Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark", CACM, 27, 10, October, 1984, pp. 1013-1030.

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