CPU PERFORMANCE EVALUATION AND EXECUTION TIME PREDICTION USING NARROW SPECTRUM BENCHMARKING

by

Rafael H. Saavedra-Barrera

CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking

Rafael H. Saavedra-Barrera

Computer Science Division
University of California
Berkeley, CA 94720

ABSTRACT

Runs of a benchmark or a suite of benchmarks are inadequate either to characterize a given machine or to predict the running time of some benchmark not included in the suite. Further, the observed results are quite sensitive to the nature of the benchmarks, and the relative performance of two machines can vary greatly depending on the benchmarks used. In this dissertation we propose and investigate a new approach to CPU performance evaluation. The main idea is to represent machine performance and program execution in terms of a high level abstract machine model. The model is machine-independent and thus is valid on any uniprocessor. We have developed tools to measure the performance of a variety of machines, from workstations to supercomputers. We have also characterized the execution of many large applications, including the SPEC and Perfect benchmark suites. By merging these machine and program characterizations, we can estimate execution times quite accurately for arbitrary machine-program combinations. Another aspect of the research has consisted in characterizing the effectiveness of optimizing compilers.

Another contribution of this dissertation is to propose and investigate new metrics for machine and program similarity and the information that can be derived from them. We define the concept of pershapes, which represent the level of performance of a machine for different types of computation. We introduce a metric based on pershapes that provides a quantitative way of measuring how similar two machines are in terms of their performance distributions. This metric is related to the extent to which pairs of machines have varying relative performance levels depending on which benchmark is used. A similar metric for programs allows us to compare and cluster them according to their dynamic behavior. All this information helps to identify those parameters in machines and programs which are the most important in determining their execution times. Further, it provides a way for designers and users to identify potential bottlenecks in machines, compilers, and applications.

February 18, 1992

Table of Contents

List of Tables

1 Introduction

1.1. Summary

In this dissertation 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 mechanism 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. Finally, we show that this approach can be extended to include the effects of compiler optimization and program locality.

1.2. Background

The impressive increase in computer speed on large and medium computers in the last 30 years has been surpassed by an even larger rate of improvement in microprocessors during the last decade. This improvement, which has been mainly the result of advances in VLSI technology and architectural design, have extended over a range of almost three orders of magnitude. Furthermore, the time from system design to mass production and to the eventual replacement of these new systems has shrunk from years to months. Therefore, the problem of evaluating the performance of machines and comparing their characteristics has become even more important as machine designers need to assess how new designs will perform with respect to other systems; compiler writers need to quantify the quality of the code generated by their compilers; and users need to select the "best" machines in which to run their applications.

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 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. (d) The model is different for every machine architecture.

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 dissertation 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.

1.3. Dissertation Overview

Chapter 2, *Machine Characterization Based on an Abstract High Level Machine Model*, begins by introducing our methodology and discussing the benefits and limitations of using a linear model for CPU performance based on the concept of an abstract machine. We present the set of abstract operations defining the Fortran abstract machine and discuss how these are measured using "narrow spectrum" benchmarking. An important result here is that it is possible to characterize with good accuracy the execution times of the abstract operation across many machines ranging from workstation to supercomputer, even when using timers with poor resolution.

We present execution time measurements for a large number of machines and discuss some of the most relevant results. The basic operations can be combined into a small set of indices representing the performance of different aspects of the CPU. We introduce the concept of *performance shape* which represents a convenient way of graphically displaying the distribution of performance along the indices. We then present a measure to evaluate the amount of similarity between performance shapes. We cluster machines according to their similarity distance and show that this measure is closely related to the amount of dispersion found in the distribution of benchmark results between pairs of machines. We end the chapter by discussing the main limitations of the system characterizer.

Chapter 3, *Analysis of Benchmark Characteristics and Benchmark Performance Prediction*, discusses how we use our model to analyze the characteristics of a large number of benchmarks and to show that we can predict their execution times on many machines. We present these results, and use them to discuss variations in machine performance and weaknesses in individual benchmarks. Our main result here is that many large benchmarks with long execution times tend to spend most of their time in a small number of basic blocks which consist of a small set of basic operations. This puts into question the effectiveness of these programs as benchmarks.

We define a similarity metric for benchmarks based on their dynamic statistics. We show that this metric is related to the way benchmarks rate machines according to their relative performance.

The basic linear model of CPU performance presented in Chapters 2 and 3 has two important limitations. The first is that our characterization of programs ignores the effect of optimizing compilers in reducing the execution time. Up to this point we have assumed that programs are compiled without optimization and our predictions reflect this.

The second problem is that there are no terms in the basic model which deal with the memory delays due to cache and TLB misses. Every permutation on the sequence of memory references generated by the programs produces the same execution time prediction. In reality, permutations to the stream of references change the spatial and temporal locality of the program and consequently their execution time [Smit82]. A program with bad locality will incur a higher number of cache misses, TLB misses, and/or page faults requiring more accesses to lower levels of the memory hierarchy with an associated delay.

Chapter 4, *Performance Characterization of Optimizing Compilers*, deals explicitly with the problem of optimization. We begin by first observing that most of the performance benefits derived from using optimization are the results of transformations which improve sequences of machine instructions but do not change the sequence of abstract operations identified in our analysis of the source code. These type of transformations, which we called *invariant optimizations*, permit viewing the process of optimization as the execution of the same program in a new machine representing an 'optimized' implementation of the abstract machine, instead of executing an 'optimized' program on the same machine. The relevance of this approach is that by doing this we avoid dealing with the more difficult problem of attempting to predict how an optimizing compiler will transform an arbitrary program without having knowledge about the inner structure of the optimizer. Characterizing the new machine is relatively straightforward; we just run the

machine characterizer benchmark with optimization enabled. We show that by using this approach we can predict the execution time of most optimized programs.

Chapter 4 also presents a comparison of the type of optimizations that can be performed by current optimizing compilers. A special benchmark is used to identify if a compiler can detect a particular optimization and whether its application extends to all or only some data types or expressions. The chapter ends with an evaluation of the amount of optimization present on the SPEC and Perfect Club benchmark suites. We also discuss the shortcomings in the programs which can be exploited by clever optimizers to give the illusion of higher performance.

Chapter 5, *Locality Effects and Characterization of the Memory Hierarchy*, discusses the problem of locality in the execution time of programs and extends our model of program execution with an explicit term that models the delay incurred by cache and TLB misses. The new term combines actual cache and TLB measurements with the actual miss ratios of programs.

The parameters of the cache and TLB are obtained from profile curves which are generated using a special benchmark. These profiles allow us to measure: a) the size of the cache and TLB; b) the size of a cache line and the granularity of a TLB entry; c) the cache and TLB associativity; and d) the execution time needed to satisfy a cache or TLB miss. Using these results and miss ratios we quantify the amount of time delay that different machines experience during the execution of benchmarks. We also evaluate the impact of the memory systems on the overall CPU performance of the machines.

The dissertation concludes with Chapter 6, which summarizes the results and discusses some future directions for research.

1.4. References

[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.

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

[Smit82] Smith, A.J., "Cache Memories", *ACM Computing Surveys*, Vol.14, No.3, September 1982, pp. 473-530.

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

[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.

2 Machine Characterization Based on an Abstract High Level Machine Model

2.1. Summary

Runs of a benchmark or suite of benchmarks are inadequate to either characterize a given machine or to predict the running time of some benchmark not included in the suite. Furthermore, the observed results are quite sensitive to the nature of the benchmarks, and the relative performance of two machines can vary greatly depending on the benchmarks used.

In this chapter, we present a new approach to benchmarking and machine characterization. The idea is to create and use a machine characterizer, which measures the performance of a given system in terms of a Fortran abstract machine. Fortran is used because of its relative simplicity and its wide use for scientific computation. The analyzer yields a set of parameters which characterize the system and spotlight its strong and weak points; each parameter provides the execution time for some primitive operation in Fortran.

We present measurements for a large number of machines ranging from small workstations to supercomputers. We then combine these measurements into groups of parameters which relate to specific aspects of the machine implementation, and use these groups to provide overall machine characterizations. We also define the concept of pershapes, which represent the level of performance of a machine for different types of computation. We introduce a metric based on pershapes that provides a quantitative way of measuring how similar two machines are in terms of their performance distributions. This metric is related to the extent to which pairs of machines have varying relative performance levels depending on which benchmark is used.

2.2. Introduction

One approach to comparing the CPU performance of different machines is to run a set of benchmarks on each. Benchmarking has the advantage that, since real programs

are being run on real machines, the results are valid, at least for that set of benchmarks; such results are much more believable than estimates produced from models of the system, no matter how detailed. To the extent that the benchmark set is representative of some target workload, the observed performance differences will reflect differences in practice.

Considerable effort has been expended to develop benchmark suites that are considered to reflect real workloads. Among them are the Livermore Loops [McMa86], the NAS kernels [Bail85a, Bail85b]), and synthetic benchmarks (e.g. Dhrystone [Weic84, Weic88], Whetstone [Curr76]). Unfortunately, there are a number of shortcomings to benchmarking [Dong87, Worl84]: (1) It is very difficult to explain the benchmark results from the characteristics of the machines. (2) It is not clear how to combine individual measurements to obtain a meaningful evaluation of the various systems. (3) Given that there is almost never a good model of the machines being benchmarked, it is not possible to validate the results, nor to make predictions and/or extrapolations about expected performance for other programs. (4) Unless the benchmarks are tuned for each machine architecture, they may not take advantage of important architectural features. (5) The large variability in the performance of highly optimized computers is difficult to characterize with benchmarks. For example using benchmarks Harms et al. found that the relative performance between the Fujitsu VP-200 and the CRAY X-MP/22, varied from 0.41 to 5.39 on individual programs [Harm88]; the ratio for the whole workload was only 1.12.

In this dissertation we are interested primarily in CPU performance. Here and the following chapters we sometimes use the terms system performance and system characterization. These two terms should be interpreted only in the context of the CPU and the memory hierarchy. In addition, for the purposes of this dissertation, a workload consists only of CPU intensive applications. Normally, a system consists of hardware components such as the CPU, the memory hierarchy, the I/O, and the network. Moreover, there are software components such as user programs, the operating systems, and the libraries. For a more general discussion about computer system performance evaluation and workload modeling the reader is referred to [Ferr78, Ferr84].

Here we present a new approach to characterizing machine performance. We do this via "narrow spectrum" benchmarking, by which we measure the performance of a machine on a large number of very specific operations, in our case primitive operations in Fortran. This set of measurements characterizes each specific CPU. We separately analyze specific programs, ignoring at this stage of our research compiler optimizations and vector instructions. We can then combine the frequency of the primitive operations with their running times on various machines to predict the running time of any analyzed program on any analyzed machine. This approach also gives us considerable insight into both the machines and the programs, since the effects of individual parameters are immediately evident.

In this chapter, we provide an overall presentation of this work, but concentrate on the specific issue of machine characterization; prediction of the execution time of benchmarks is done in Chapter 3. We discuss how to extend the model to incorporate the effects of compiler optimization and the memory hierarchy in Chapters 4 and 5. Section 2.3 gives a somewhat more detailed overview of our research. In Section 2.4, we describe the machine characterizer, and also the program analyzer. The parameters used

to characterize a machine are explained in Section 2.5. The methodology used to make measurements is presented in Section 2.6, and the parameters derived from a number of machines are given in Section 2.7. A comparison of machines is also provided in that section. The concepts of performance distributions (pershapes) and pershape distances between machines are given in Section 2.8. Some unresolved issues are considered in Section 2.9.

2.3. System Characterization and Performance Evaluation

The idea behind our approach is to distinguish between two different activities often ignored in machine evaluation; these are system characterization and performance evaluation. We define *system characterization* as an n-value vector where each component represents the performance of a particular operation (P_i). This vector ($<P_1, P_2, \cdots, P_m>$) fully describes the system at some level of abstraction. The parameters we use are a set of primitive operations, as found in the Fortran programming language, and are defined in §2.5. We measure the values of the parameters using a *system characterizer*, that runs a set of 'software experiments', which detect, isolate and measure the performance of each basic operation. Using software experiments to measure each system allows us to validate our model and measurements by making predictions and checking the results with the execution of benchmarks and workloads. This approach is in contrast to studies which use a low-level machine architecture based model [Peut77] and is similar to the work presented in [Morg73] to model system performance.

The *performance evaluation* of a group of systems is the measurement of some number of properties during the execution of some workload. One property may be the total execution time to complete some job. It is important to note that the results depend on, and are only valid for, the set of programs used in the evaluation. The evaluation includes not only the machine, but also the compiler, the operating system and the libraries. In this research we focus on the execution time of computationally intensive programs as our metric for evaluating different architectures.

2.3.1. A Linear Model for Program Execution

Our research is based on the assumption that the execution time of a program can be partitioned into independent time intervals, each corresponding to the execution of some operation of an abstract Fortran machine. The abstract Fortran machine (AFM) is used as a general model for all the machines, each executing the object code produced by their Fortran compilers. Thus, each system represents a different implementation of the AFM. In this way, the AFM makes it possible to compare different architectures. As we show in Chapter 3, this assumption is reasonably accurate.

System designers use a similar approach, but at the hardware level, when they evaluate different implementation of the same architecture. In this case the model of the machine is defined by its instruction set, and they are interested in the mean instruction execution time [MacD84]. This quantity is equal to the sum of the mean nominal execution time, the mean pipeline delay caused by path conflicts and data dependencies, and the mean storage access delay caused by cache misses of instructions and operands. In our case, instead of having one single machine instruction to measure, we have a group of instructions corresponding to an abstract parameter. How each abstract parameter is

implemented in each machine depends on its instruction set, compiler, and libraries. In fact, normally there will be several sequences of instructions implementing each parameter. Which particular sequence is generated by the compiler depends on the context in which the operation appears in the source program.

Our model of the total execution time is the following: Let $\mathbf{P_M} = <P_1, P_2, \cdots, P_n>$ be the set of parameters that characterize the performance of machine M. Let $\mathbf{C_A} = <C_1, C_2, \cdots, C_n>$ be the dynamic distribution of operations in program A. We obtain the expected execution time of program A on machine M as

$$T_{A,M} = \sum_{i=1}^{n} C_i P_i = \mathbf{C_A} \cdot \mathbf{P_M} \tag{2.1}$$

In general, given machines $M_1, M_2, ..., M_m$, with characterizations $\mathbf{P_{M_1}}, \mathbf{P_{M_2}}, \cdots, \mathbf{P_{M_m}}$, and a workload W formed by programs $A_1, A_2, \cdots, A_l$ with dynamic distributions $\mathbf{C_{A_1}}, \mathbf{C_{A_2}}, \cdots, \mathbf{C_{A_l}}$, the expected execution time of machine M_k on workload **W** is

$$T_{\mathbf{W},M_k} = \sum_{j=1}^{l} \mathbf{C_{A_j}} \cdot \mathbf{P_{M_k}} \tag{2.2}$$

$T_{\mathbf{W},M_i}$ provides a way to make a direct comparison between several machines with respect to workload **W**. The expected execution time of workload W on machine M_i is less than that of machine M_j if $T_{\mathbf{W},M_i} \leq T_{\mathbf{W},M_j}$.

Using this model it is possible not only to compare two different machine architectures using any workload, but also to explain their results in terms of the abstract parameters. Let $\Phi_{\mathbf{M,A}} = <\phi_1, \phi_2, \cdots, \phi_n>$ be the normalized distribution of the execution time for program A executed on machine M. Define:

$$\phi_i = \frac{C_{A,i} P_{M_k,i}}{\mathbf{C_A} \cdot \mathbf{P_M}}$$

Vector $\Phi_{\mathbf{M,A}}$ decomposes the total execution time in terms of each parameter and makes it possible to identify which operations are the most time consuming. We would expect that different machines will have different distributions, even for different implementation of the same architecture or/and for different compilers. Once we have the machine characterizations, it is possible to study the effect of changes in the normalized dynamic distribution without writing real programs that correspond to these distributions, and in this way detect which parameters have a significant impact in the execution time for some machines.

An advantage of this scheme is that the $l \cdot m$ machine-program combinations only require that each machine be measured once to obtain its characterization, and also that each program be analyzed once. Making an evaluation using normal benchmarking techniques requires the execution of $l \cdot m$ programs. Moreover, once the machine has been measured, its characterization can be used at any time in the future for additional evaluations, in contrast to benchmarking in which access to the machine (same model, operating system, compiler, libraries) is needed for each new set of benchmarks.

2.3.1.1. Linear Models and Curve Fitting

Linear models are often used to fit a k-parametric "model" to a set of data represented by a matrix $A_{n \times m}$. In the context of this paper, the matrix could contain the execution times of n benchmarks on m machines [Pond90]. The fit is obtained by factoring matrix A into the product $X_{n \times k} \cdot Y_{k \times m}$ using statistical methods such as *principal component analysis* [Mard79]. These techniques attempt to reconstruct the original matrix from two matrices of rank k by minimizing the distance under some appropriate metric. The usefulness of curve fitting in benchmarking appears to be very limited, because normally there is no relation between the individual parameters of the model and the characteristics of the machines and/or the benchmarks. Therefore it is difficult to understand how changes in the programs or the machines would affect the model and the execution times. Furthermore, adding or removing a row or column to A tends to affect significantly all entries on matrices X and Y. This is counterintuitive as it implies that replacing the benchmark results of machine M_1 by those of machine M_2 in matrix A affects the way we explain the execution times of machine M_3.

Our approach is entirely different; the only similarity to curve fitting is the use of a linear model which happens to be a non-essential feature in our methodology. We do not attempt in any way to explain benchmark results by using curve fitting; on the contrary, we predict benchmark results by characterizing the set of operations that programs execute and by measuring the performance of these operations on different machines. These measurements are 'absolute' in the sense that including more programs and/or machines does not affect the measurements, and they represent actual program and machine characteristics, which allows us to relate changes in their values to real execution times.

2.3.1.2. Limits of the Linear Model

The only way in which the linear model can give acceptable results is if the following conditions hold: (1) The experimental measurements are representative of 'typical' occurrences of the parameters in real programs. (2) The errors caused by the low resolution and the intrusiveness of the measuring tools are small compared to the magnitude of the measurements. (3) Variability in the mean execution time caused by data dependencies, external concurrent activity, and nonreproducible conditions is small, and therefore does not significantly affect the results. In some cases, the above conditions cannot be satisfied, especially in highly pipelined machine where the execution time when there is a register dependency conflict is several times greater than the execution time without this delay. An example of this is the CYBER 205, where an add or multiply can take as little as 20 ns to execute, when the pipeline is full, or as much as 100 ns in the worst case [Ibbe82]. If we consider the following two statements

```
X9 = ((X1 + X2) * (X3 + X4)) + ((X5 + X6) * (X7 + X8))

X6 = ((X1 + X2) * X3 + X4) * X5
```

we find that the execution of the first statement takes approximately 360 ns, while the execution time of the second takes 400 ns. A simple linear model will estimate that the execution of the second statement will be less than that of the first statement, unless the

model contains information on how the execution time is affected by data dependencies. Branching and interrupts also prevent the pipeline from working at peak speed. Although it is difficult to detect and measure how each machine will execute different statements, it is always possible to create new parameters that take into account data dependencies and measure the extra penalty in the execution time. In practice, the number of parameters cannot be expanded without limit.

2.3.2. Fortran and Other Programming Languages

The model presented above can also be applied to other general purpose languages. We chose Fortran instead of other programming language for the following reasons: 1) most large scale scientific computation, accounting for most of the CPU time on super-computers, is done in Fortran; 2) the number of language constructs in Fortran is small; and 3) the execution time of most of the operations in Fortran does not depend on the value of the arguments. It is therefore natural to experiment first with a less complex programming language and test whether it is possible to make acceptable predictions. Most of the differences between Fortran and other general purpose languages do not prevent building an abstract machine model, although a model with a larger number of parameters and better experiments would be required.

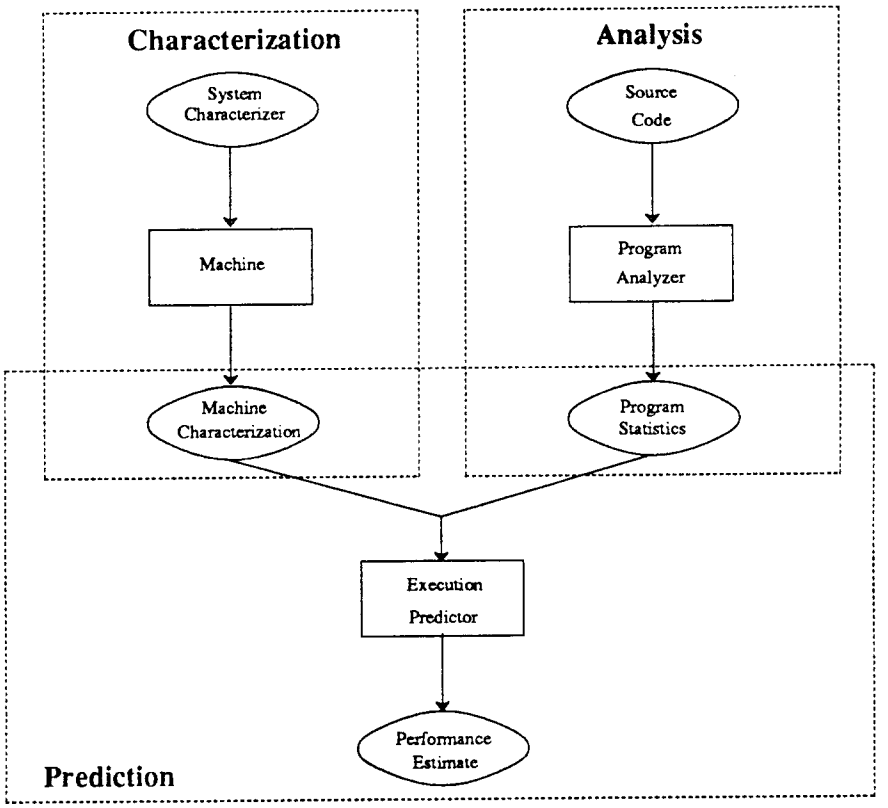


Figure 2.1: The process of characterization, analysis and prediction.

2.4. Description of the System

In the last section we showed what we need in order to characterize machines using the linear model and how to use this information to make predictions about the execution time of programs. We have implemented (a) a system characterizer and assembled a library of machine characterizations ($\mathbf{P_M}$); (b) a program analyzer that generates the dynamic distribution ($\mathbf{C_A}$) of Fortran programs; and (c) an execution predictor that takes $\mathbf{P_M}$ and $\mathbf{C_A}$, and estimates the expected execution time of the applications. The complete process, characterization, analysis, and prediction is shown in figure 2.1. In the next two subsections we give an overview of the program analyzer and the execution predictor. A more in depth presentation of the system characterizer follows.

2.4.1. Program Analyzer

The program analyzer (PA) decomposes Fortran programs statically and dynamically in terms of the abstract parameters. This provides a uniform model for the execution of different applications. In addition both models, the performance model associated with machines, and the execution model associated with the applications are identical. Thus, it is possible using the dynamic distribution to compare different programs, putting the emphasis not on their syntactic or semantic properties, but in how they affect the performance of different systems.

The PA is basically the front end of a Fortran compiler. It takes as its input a Fortran program and after making a lexical and syntactical analysis, it outputs an instrumented version of the original program, from which we obtain the dynamic statistics. In addition, the PA also gives the static statistics for each parameters for each basic block. The operation of the program analyzer is shown in figure 2.2.

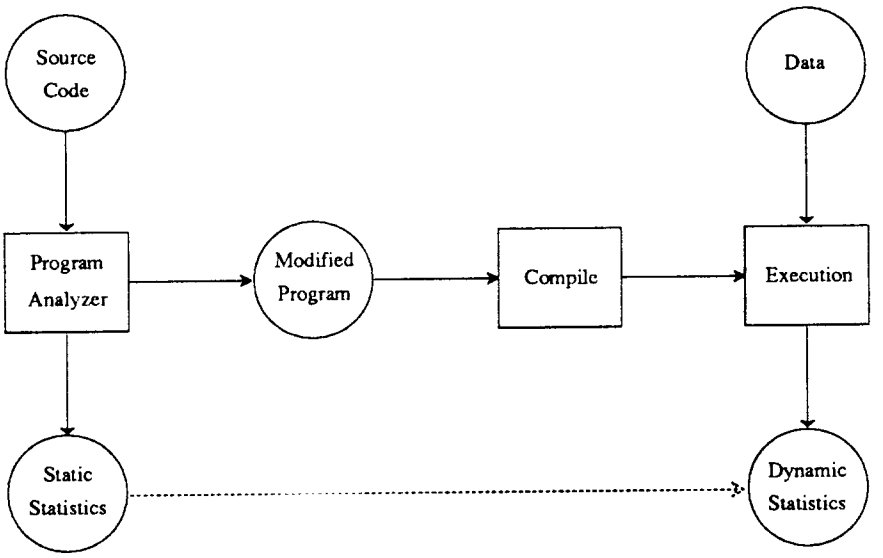


Figure 2.2: The static and dynamic statistics are obtained by parsing and instrumenting the source program.

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 parameter 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 $\mu_A=<\mu_1,\mu_2,\cdots,\mu_j>$ 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. Matrix $\mathbf{S_A}$ and vector μ_A are obtained by parsing and instrumenting the source code. Vector $\mathbf{C_A}$ (§2.2.1) and matrix $\mathbf{D_A}$ are related by the following equations

$$C_i = \sum_{j=1}^{m} d_{i\,j} \tag{2.3}$$

The dynamic statistics are independent of the code generated by each compiler, and they only depend on the source code and the data used in the execution.

2.4.2. Execution Predictor

The execution predictor (EP) combines the machine characterization $\mathbf{P_M}$ with the dynamic statistics $\mathbf{C_A}$ to obtain estimates of the expected execution time of programs, using equations (2.1) and (2.3). The execution time is computed for each statement of the program, and this makes it possible to compute estimates for different parts of the program. In our system, this is done by inserting two special comments at the beginning and end of the particular region in which we are interested. There is no limit to the number of these regions as long as they are either disjoint or one is contained in the other. In addition to the expected execution time, the EP also reports the variance of the estimate, and the expected execution time per parameter along with its variance. This is done by measuring vector $\sigma^2\mathbf{P_M}=<\sigma^2P_1,\sigma^2P_2,\cdots,\sigma^2P_n>$ with the system characterizer.

2.5. The Fortran Abstract Machine

By using a common parametric model, we are able to compare the performance of different architectures and make a fair comparison between them with respect to their execution of Fortran programs. How many parameters the system should have depends on how accurate we want our predictions to be, although this is limited by the resolution of our measuring tools; at some point increasing the number of parameters does not have any effect in improving our predictions. We are also limited in our accuracy by the fact that we do not analyze the code generated by the compiler, nor use information that is not contained in the source code of the program. In the next subsection we present the set of parameters in our model, and give a brief description of what they measure.

2.5.1. Parameters in the System Characterizer

Each parameter of the model can be classified in one of the following broad categories: arithmetic and logical, procedure calls, array references, branching and iteration, and intrinsic functions. 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. Although every basic operation in Fortran is characterized by some parameter, we have made some simplifications with some

operations which were rarely executed in the benchmarks we used. It is straightforward to include new parameters in the model, and to write new experiments for the system characterizer. The parameters are classified in eighteen different groups according to the semantics of the operation; tables 2.8 and 2.9 in Appendix 2.A present the 109 parameters.

2.5.2. Global vs. Local Variables

Most operators are characterized by several parameters, depending on the operand types and sizes, and storage class (common / local). Global variables in Fortran (COMMON) are sometimes treated differently from local variables. In some compilers, variables stored in COMMONs are treated as components of a structure using a base-descriptor for each COMMON block which points to the first element of the COMMON. An operand is loaded by first adding an offset to the base-descriptor and then loading the operand. This way of treating simple variables makes the execution slower when they are allocated as global variables as opposed to local.

2.5.3. Arithmetic and Logical Operations

Fortran is a language for scientific and numeric applications. For this reason the richness of the language lies in the arithmetic operators that it supports. In addition to the arithmetic operators, Fortran also provides six relational and six logical operators. Table 2.8 in Appendix 2.A (groups 1-8) gives the fifty-six arithmetic parameters grouped by data type (real, complex, or integer), size (single or double precision) and storage class (local or global). Not all combination are included; there is no double precision for integers or complex variables. In table 2.9 (Appendix 2.A) we find the logical and conditional parameters (groups 9-10). Each arithmetic mnemonic is formed by appending the first letter of the operation with the first letter of the data type (R, C, or I), plus a letter identifying the size of the operand (S or D), and at the end the first letter of the storage class (L or G).

The arithmetic operators defined in the system characterizer are: addition, multiplication, division (quotient), and exponentiation. The addition operator also includes subtraction. Addition (subtraction) between an array index and a constant is treated as a special parameter; most compilers add, at compile time, the constant to the base-descriptor of the array and eliminate the addition (see §2.5.6). In the case of exponentiation using a real base, we distinguish between two cases: one when the exponent is integer and the other when the exponent is real. When the exponent is integer, the result is computed by either executing the same number of multiplications as in the exponent (when this is small), or by binary decomposition. In the case of a real exponent, the result is computed using logarithms. If the base is integer, we have two cases, one with the exponent equal to two and another when the exponent is greater than two. Given that the number of exponentiations executed in most programs is small, these simplifications are sufficient.

Two groups of parameters require further explanation. One is the set of parameters that measure the overhead of the store operation (SRSL, SCRL, SISL, SRDL, SRSG, SCSG, SISG, and SRDG), and the other, what we called memory transfer parameters (TRSL, TCSL, TISL, TRDL, TRSG, TCSG, TISG, and TRDG). In Fortran loading an operand is not a separate operation, and so the time for a load is included in the execution time of the corresponding arithmetic or logical operation. A store, however, is a visible

operation whose time can be measured. In some cases, the time for a store is negligible, (part of the store operation overlaps with the execution of the previous or/and following operation), while in other cases the time can be significant.

Loads are visible and must be accounted for in one case. In a memory transfer statement where there are no operators on the right hand side of the equal sign, the execution time of the statement cannot be explained just by the store operation. For these kind of statements we use the memory transfer parameters to distinguish them from normal statements that include expressions.

2.5.4. Procedure Calls and Arguments

Function and subroutine call overhead also significantly affect the execution time of programs. This overhead can be identified with three actions: passing the arguments between procedures, transferring control from the caller to the callee, and returning the result and control from the callee to the caller. In Fortran all the arguments are passed by reference including values computed by expressions, so the argument passing overhead is limited to setting up the reference.

We use two parameters which characterize function and procedure calls: the first measures the joint execution of the prologue and epilogue of the call (PROC), and the second the time it takes to load the address of an argument, either into registers, the static environment of the callee subprogram or in the execution stack (some systems use the execution stack or registers to pass arguments between subprogram units).

2.5.5. Branch and Loop Control

Branches (table 2.9, group 13) affect the execution of a program in several ways. In pipelined machines, a penalty must be paid when a branch is taken and the target instruction has not been previously fetched[1], since all partially executed instructions in the pipeline must be discarded and the new stream of instructions must be fetched [LeeJ84]. For a machine with a cache memory, a branch to an instruction that is not in the cache may increase the miss ratio by changing the locality of execution [Smit82].

Although Fortran has three different types of GO TO statements: unconditional, assigned, and computed, we characterize all three with only two parameters. We make a distinction between a direct jump (GOTO) and a computed jump (GCOM). In the first case the target of the jump is known by the compiler and it is normally implemented as an unconditional jump. In the second case the target depends on the value of some expression and it should be computed before the branch can be executed. In our model, parameter GOTO is used for unconditional GO TOs, and GCOM for computed and assigned GOTOs.

DO loops (group 14) are flow of control constructs that are widely used in scientific programs. The overheads incurred by this instruction are the time to set the initialization, limit and step of the control index, and the time it takes to update the index and check if the number of iterations has been completed. In addition, some machines implement loops with unit step differently from non-unit step loops. In some cases the loop is

[1] Even in machines that have some kind of branch prediction circuitry, a penalty is incurred when the prediction is incorrect.

transformed to a loop with a unit step, which sometimes increases the execution time in non-unit loops. Four parameters (LOIN, LOOV, LOIX, LOOX) characterize loops with unit and non-unit increment.

Although Fortran has three different types of IF statements, neither of these has been found to need special parameters for their characterization. The block IF and the logical IF are decomposed into two parts: the evaluation of the predicate (arithmetic-logical expression) and a direct branch. The arithmetic IF is different only in the way it branches. We handle the branching part as a computed GOTO (GCOM).

2.5.6. Array References

Array variables in expressions are treated as ordinary variables plus an additional overhead to compute the address of the element. We have three parameters (ARR1, ARR2, ARR3, and ARR4) that characterize the dimension of an array reference (group 12). The overhead for variables in four and five dimensions is computed in the execution predictor using a linear combination of the three basic parameters. We found that most of the applications we examined had very few arrays with more than three dimensions and no examples of more than five.

As we mentioned in §2.5.2, integer addition between a variable and a constant in an array index is considered a special operation. Initially we treated them as normal integer adds, but the predictions thus obtained were off significantly from the measured times. We fixed this problem by creating a new parameter (IADD) that measures the execution time of an add using an index and a constant as operands.

2.5.7. Intrinsic Functions

Intrinsic functions form the last subset of parameters (groups 15-18). Although the number of times these instructions are executed in a program is small, and they only occur in some workloads, their execution times tend to be very large compared with that of a single arithmetic operation. We have twenty-four parameters in four groups which represent the intrinsic functions most used in scientific programs.

The execution time of an intrinsic function is not always constant and normally depends on the magnitude of the arguments. As an example, consider how the IBM 3090/200 computes the sine function [IBM87]. In the computation, the execution time of several steps depends not only in how large is the argument, but also in how small is its difference from the nearest multiple of π. Depending on the magnitude of this difference, a polynomial of degree one, three, five, or a table and additional arithmetic is needed to compute the result. The frequency of intrinsic functions is generally low, and the arguments unpredictable, and we have found that our assumption of constant execution time is a good enough approximation.

2.6. Machine Characterizer

The machine characterizer (MC) consists of 109 'software experiments' that measure the performance of each individual parameter needed to completely characterize a Fortran machine (see tables 2.8-2.9 in Appendix 2.A). The MC is written as a Fortran program and runs from 200 seconds, on machine with good clock resolution, to 2000 seconds on machines with 1/60'th second clock resolution. We have run the MC on

several 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.

2.6.1. Experiment Structure

Timing a benchmark is very different from making a detailed measurement of the parameters in the system characterizer. For some benchmarks the system clock is enough for timing purposes, and repetition of the measurements normally yields an insignificant variance in the averaged results. On the other hand, the measurement of the parameters in the system characterizer is more difficult due to a number of factors:

- The short execution time of most operations (20 ns - 10 μs)
- The resolution of the measuring tools (≥ 1 μs)
- The difficulty of isolating the parameters using a program written in Fortran
- The intrusiveness of the measuring tools
- Variations in the hit ratio of the memory cache
- External events like interrupts, multiprogramming, and I/O activity
- The need to obtain repeatable results and accuracy

Most of our primitive operations have execution times of from ten to thousands of nanoseconds, and are implemented with a single or a small number of machine instructions. For this reason direct measurement is not possible, especially since our tests should work for many different architectures. In addition, the need to isolate an operation for measurement normally requires robust tests to avoid optimizations[2] from the compiler that would eliminate the operation from the test and distort the results [Clap86]. Different techniques must be used, in particular avoiding the use of constants inside the test loops; using IF and GO TO instructions instead of the DO LOOP statements to control the execution of the test; and initializing variables in external procedures to avoid constant folding. Separate compilation of variable initialization procedures is used to make sure that the body of the test does not give enough information to the compiler to eliminate the operation being measured from inside the control test loop.

2.6.2. Test Structure and Measurement

The measurement tools we have are the system clock and the repeated execution of a sequence of statements. The resolution of the clock, the overhead of the timing routine and the overhead of the statements that control of measurements are the sources of error that we can control or work around. Variations in the hit ratio of the cache, interrupts, multiprogramming and I/O activity are more difficult to eliminate and measure (see §2.9).

---

[2] Even when we compile without optimization, compilers try to apply some standard optimizing techniques, like constant folding, short-circuiting of logical expressions, and computing the address of an element in an array.

We use three different methods to measure the execution time of the parameters. The first is by *direct* measurement, i.e. executing some operation for some number of times and in different contexts. The second is with a *composite* measurement. In this case we execute a number of different operations and subtract the execution time of the known parameters to obtain the value of the one that is unknown. The third possibility is with an *indirect* measurement. Some parameters of the model are 'coupled'; it is not possible to execute one without executing the other. The way to measure one of the parameters is to run two or more tests with a different number of operations; the solution of a set of linear equations gives the correct result. Figure 2.3 shows the basic structure of our tests. This same structure is used in all the tests.

```
      LIMIT = LIMIT0 * SPEEDUP * (TMAX - TMIN) / 2.
      DO 4 K = 1, REPEAT
1        COUNTER = 1
         TIME0 = SECOND ()
2        IF (COUNTER .GT. LIMIT)  GO TO 3
            ...
            body of the test
            ...
         COUNTER = COUNTER + 1
         GO TO 2
3        TIME1 = SECOND ()
         IF (TIME1 - TIME0 .GE. TMIN .AND. TIME1 - TIME0 .LE. TMAX)  GO TO 4
         LIMIT = .5 * LIMIT * (TMAX - TMIN) / (TIME1 - TIME0)
         GO TO 1
4     SAMPLE(K) = (TIME1 - TIME0) / LIMIT
      CALL STAT (REPEAT, SAMPLE, AVE, VAR)
```

Figure 2.3: The basic structure of an experiment. The statement `IF (TIME1 - TIME0` enforces the execution of each test for more than TMIN and less than TMAX seconds. If the execution time is outside this interval a new value of LIMIT is computed and the test is repeated.

The sequence of statements to measure correspond to the 'body of the test'. These statements are executed for some number of times (LIMIT) and the execution time is measured (function SECOND). This time is called an observation. TMIN, and TMAX control the minimum and maximum time that each observation should run ($t_{min} \leq time_1 - time_0 \leq t_{max}$). The two statements before the GO TO 1 enforce this condition. The DO loop is used to get several (REPEAT) observations to obtain a meaningful statistic. Because we don't know a priori how fast or slowly an operation executes in an arbitrary machine, we extrapolate by using the time it takes to run the test in the CRAY X-MP/48 and multiplying by their relative speeds. This is done using LIMIT0, which is the number of times the test runs in the CRAY X-MP/48, and SPEEDUP that gives the relative speed of the machine. The relative speed is computed by running a small test at the beginning of the characterization.

2.6.3. Experimental Error and Confidence Intervals

There are many known sources for the variability of the CPU time [Curr75, Merr83] and consequently in our measurements. Some of these factors are: timer resolution of the

clock, improper allocation of CPU for I/O interrupt handling, cycle stealing, and changes in cache hit ratios due to interference with concurrent tasks. Small errors in the measurements have considerable impact in the predictions we make, and we must measure and compensate for them. We will proceed to derive expressions for the variance and the confidence intervals of the measurements. The following definitions are used in the analysis:

T_{j_0}	::=	CPU time before the observation (TIME0)
T_{j_1}	::=	CPU time after the observation (TIME1)
$C_{overhead}$	::=	overhead involved in the timing function
$IF_{overhead}$	::=	overhead involved in the if-loop control
N_{limit}	::=	number of times the body is executed (LIMIT)
N_{repeat}	::=	number of observations in the experiment (REPEAT)
O_j	::=	observation j, equal to TIME1 – TIME0
$\hat{O}$	::=	sample mean of each observation (measurement)
$\hat{B}$	::=	sample mean time of one 'body of the test' execution
$\hat{P}_i$	::=	sample mean of parameter i
σ^2	::=	variance operator

We know that each observation O_j is equal to

$$O_j = T_{j_1} - T_{j_0}$$

then the mean value ($\hat{O}$) and variance of these observations are

$$\hat{O} = \frac{1}{N_{repeat}} \sum_{j=1}^{N_{repeat}} O_j \; ; \qquad \sigma^2 O = \frac{1}{N_{repeat} - 1} \sum_{j=1}^{N_{repeat}} (O_j - \hat{O})^2 \tag{2.4}$$

Now the mean value of each observation is equal to the time it takes to execute the body of the test N_{limit} times, plus the overhead of the timing function ($C_{overhead}$), and the extra instructions that control the test ($IF_{overhead}$).

$$\hat{O} = N_{limit} \; (\hat{B} + IF_{overhead}) + C_{overhead}$$

where $\hat{B}$ is the mean time it takes to execute once the body of the test. We can compute this value and the variance with the equations

$$\hat{B} = \frac{\hat{O} - C_{overhead}}{N_{limit}} - IF_{overhead} \; ; \qquad \sigma^2 B = \frac{\sigma^2 O + \sigma^2 C_{overhead}}{N_{limit}^{\;2}} + \sigma^2 IF_{overhead} \tag{2.5}$$

To obtain the mean value of parameter $\hat{P}_i$ we need to know if the test is direct, composite or indirect. Let N be the number of times parameter $\hat{P}_i$ is executed inside the body of the test, then the mean value and variance of parameter $\hat{P}_i$ in a direct test are

$$\hat{P}_i = \frac{\hat{B}}{N} \; ; \qquad \sigma^2 P_i = \frac{\sigma^2 B}{N^2} \tag{2.6}$$

In a composite test we have

$$\hat{P}_i = \frac{\hat{B} - \hat{W}_{extra}}{N} \; ; \qquad \sigma^2 P_i = \frac{\sigma^2 B + \sigma^2 W_{extra}}{N^2}$$

where W_{extra} is the additional work inside the body of the test or in the second test. In an indirect test $\hat{P}_i$ is a function of several measurements.

$$\hat{P_i} = f(\hat{B}_1, \hat{B}_2, \cdots, \hat{B}_n)$$

The normalized 90% confidence intervals are given by the expression

$$\left[\hat{P_i} - t_{.95} \left[\frac{\sigma^2 P_i}{N_{repeat}} \right]^{1/2}, \hat{P_i} + t_{.95} \left[\frac{\sigma^2 P_i}{N_{repeat}} \right]^{1/2} \right] \tag{2.7}$$

where $t_{.95}$ corresponds to the 95% percentile of the Student's t distribution. Looking at equations 2.4-2.7 we see that by increasing N, N_{limit}, and N_{repeat}, we can reduce the variance in our measurements.

2.6.4. Is the Minimum Better than the Average?

The measurements are obtained by computing the average of a number of observations. The sample represent the 'effective' execution time of the experiment plus an additional random variable that represents the 'noise' produced by concurrent activity and the resolution of our measuring tools. The error produced by the clock can be modeled as a random variable with mean zero and standard deviation of $1/6^{1/2}$ times the resolution of the clock. The concurrent activity is a positive random variable that depends on the load of the system. The 'effective' execution time must be less than or equal to (ignoring clock resolution) any observation in the sample. Therefore instead of taking the average as our measurement, it might be better to take the minimum of the sample. However, what we are trying to characterize is the execution time of programs under real conditions, and in this case the average would correspond better to these 'normal' conditions.

Table 2.1: **Estimates taking the average and the minimum**

Machine	Real Time	Average	Error	Minimum	Error
Convex C-1	543 sec	551 sec	1.47 %	499 sec	8.10 %

We decided which approach was better by using both schemes (minimum and average) in the system characterizer and using the measurements for the prediction of the execution time of a set of ten programs. The results (table 2.1) for one of the machines tested showed that using the average produces a better estimate. The programs and both system characterizers were run under similar conditions.

2.6.5. Reducing the Variance

Measuring small execution times from Fortran programs is difficult, especially for parameters that are measured using indirect tests. In a direct or composite test it is relatively easy to increase the number of operations executed inside the body of the test, and in this way reduce the variance (eq. 2.6). In the case of an indirect test, the parameters measured are coupled with other parameters and we cannot increase the execution of one without also increasing one or more of the others.

An example is the address computation for an array element. This parameter is measured by running some code using simple variables and then subtracting the running time from the execution time for the same code using array elements. Increasing the number of array elements in the test also increases the number of operations executed in

Table 2.2: **Mean and Standard Deviation**			
Parameter	Mean (μ)	Std. Dev. (σ)	σ/μ (%)
$\hat{B_1}$	69.9 μs	1.49 μs	2.13
$\hat{B_2}$	127.3 μs	5.74 μs	4.51
$\hat{B_3}$	107.4 μs	4.53 μs	4.22
LOIN	12.5 μs	9.11 μs	72.9
LOOV	1.99 μs	0.73 μs	36.8

Table 2.2: Relative magnitude of the standard deviation compared to the sample mean for parameters LOIN and LOOV. The $\hat{B_i}$s are experimental results used to computed the value of LOIN and LOOV. Each test consists of 5 observations executed for 1 second on a VAX-11/785.

the test. Moreover the variance of the difference of two random variables is the sum of the individual variances. Therefore for some parameters this produces an increase in variance, not a reduction. The way to reduce the variance in these cases is to increase the number of observations (N_{repeat}) and/or the number of times the body is executed (N_{limit}). How difficult it is to control the variance of some parameters can be seen in the following example. In this case parameters LOIN (loop initialization) and LOOV (loop overhead) are computed from the results of three tests using equations 2.8 and 2.9. Each $\hat{B_i}$ represents the result obtained in one of the three tests used in the measurement of LOIN and LOOV. Even when the sample standard deviation is small (< 5%) for the $\hat{B_i}$s, in the case of the DO loop parameters it is very large with respect to the mean. The three $\hat{B_i}$ represent the performance in each of the three experiments used to obtain parameters LOIN and LOOV.

$$\text{LOIN} = 2\cdot\hat{B_1} - \hat{B_2}\,; \qquad \sigma^2\text{LOIN} = 4\cdot\sigma^2 B_1 + \sigma^2 B_2 \tag{2.8}$$

$$\text{LOOV} = \frac{\hat{B_2} - \hat{B_3}}{N}\,; \qquad \sigma^2\text{LOOV} = \frac{\sigma^2 B_2 + \sigma^2 B_3}{N^2} \tag{2.9}$$

2.6.6. The Effect of N_{limit} and N_{repeat} on the Variance

One question we have not answered is what should be the magnitude of N_{limit} and N_{repeat} to obtain measurements which give a small σ/μ ratio. These parameters are system dependent and are mainly affected by the resolution of the clock, the concurrent activity on the system, and the particular parameter being measured. We ran several experiments using different values for N_{repeat} and N_{limit} in several machines. Figure 2.4 shows the normalized confidence interval of ten parameters for values of N_{limit} such that the each test is run for at least 0.1, 0.2, 0.5, 1.0, 2.0 and 4.0 seconds on a Vax-11/780. We also obtained measurements for N_{repeat} equal to 5, 10 and 20 observations.

We see that for a fixed value of N_{repeat} the width of the confidence interval of our measurements decreases as the time of the test increases, but for small values of N_{repeat}, there is a limit to how much we can decrease the confidence interval by only increasing the time of the test (N_{limit}). The reason for this is that by increasing the length of the test we reduce the variability due to short term variations in the concurrent activity of the system. However the probability of a change in the overall concurrent activity of the system

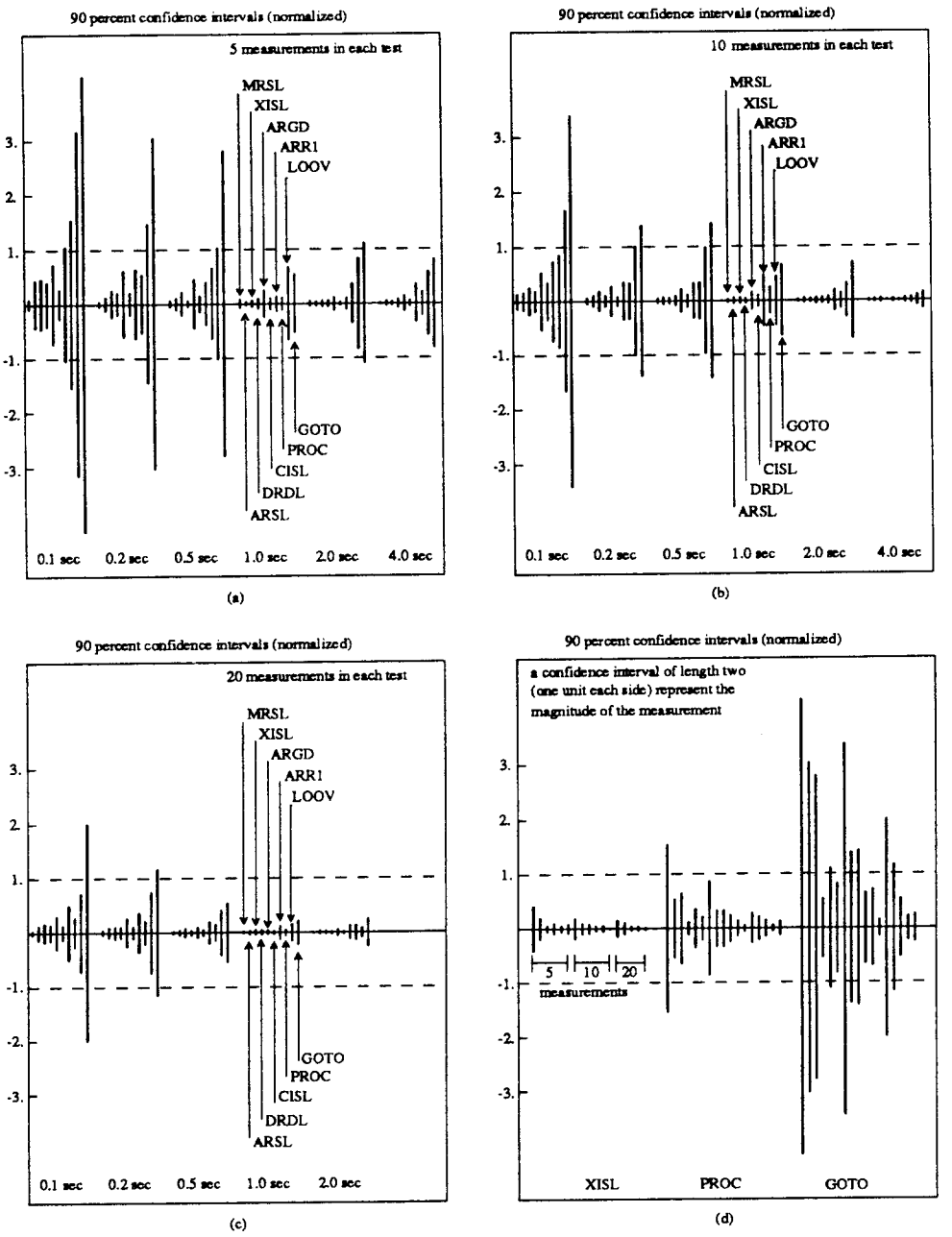


Figure 2.4: Normalized confidence intervals for ten different parameters. In (a), (b), and (c) we show how the length of the test (N_{limit}) and the number of observations (N_{repeat}) affect the confidence interval of the measurements, taken on a VAX 11/780. For a fixed number of observations, an increase in the execution time of the test tends to reduce the length of the confidence interval. Figure (d) shows for three parameters all their confidence intervals plotted together. All confidence intervals are normalized with respect to parameter P_i.

Table 2.3: Characteristics of the machines							
Machine	Name/Location	Operating System	Compiler version	Memory	Integer	Real	
					single	single	double
CRAY Y-MP/832	reynolds.arc.nasa.gov	UNICOS 4.0.8	CFT77 3.0	32 Mw	46	64	128
CRAY-2	navier.arc.nasa.gov	UNICOS 4.0.6	CFT77 3.0	128 Mw	46	64	128
CRAY X-MP/48	NASA Ames	COS 1.16	CFT 1.14	8 Mw	46	64	128
IBM 3090/200	cmsa.berkeley.edu	VM/CMS r.4	FORTRAN v2.3	32 MB	32	64	128
MIPS/1000	cassatt.berkeley.edu	UMIPS-BSD 2.1	F77 v1.21	16 MB	32	32	64
Sun 4/260	rosemary.berkeley.edu	SunOS r.4.0	F77	32 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 (fort)	pioneer.arc.nasa.gov	Ultrix 3.0	Fort v4.7	16 MB	32	32	64
VAX-11/785 (f77)	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
Sun 3/260 (f)	picasso.arc.nasa.gov	UNIX 4.2 r.3.2	F77 v1	16 MB	32	32	64
Sun 3/260	picasso.arc.nasa.gov	UNIX 4.2 r.3.2	F77 v1	16 MB	32	32	64
Sun 3/50	baal.berkeley.edu	UNIX 4.2 r.3.2	F77 v1	4 MB	32	32	64
IBM RT-PC/125	loki.berkeley.edu	ACIS 4.3	F77 v1	4 MB	32	32	64

Table 2.3: Characteristics of the machines. The size of the data type implementations are in number of bits. Sun 3/260 (f) uses the 68881 as a co-processor running at 20 MHz, while the CPU executes at 25 MHz. For the VAX-11/785 we used two FORTRAN compilers, the VAX FORT 4.7 and the Berkeley 1.1.

increases with a larger test. This change may produce a greater variance if the size of the sample statistic is small. We see that the best results are obtained for 20 observations and 1 to 2 seconds for the duration of the test. In machines with good clock resolution acceptable results are obtained with 10 observations and .2 seconds for each test.

2.7. Measurements and Some Results

We have run the system characterizer on the machines shown in table 2.3. Of the fifteen systems, four are supercomputers, each implementing single precision floating-point with 64 bits. On the other systems single precision variables are allocated using 32 bits. We gathered two sets of measurements for the Sun 3/260, one using the 68881 co-processor to execute floating-point arithmetic, and another emulating the same functions in software. We also measured the effect of using different Fortran compilers, the VMS FORT compiler and the UNIX BSD F77 both running on the VAX-11/785, in both cases with Ultrix as the operating system. By using the characterizer we can quantify how much each parameter is affected by the addition of a new hardware feature or by changing the compiler.

The measurements of all parameters are presented in Appendix 2.B in tables 2.10-2.14. The parameters are grouped according to tables 2.8 and 2.9, with all magnitudes in units of nanoseconds. Entries with magnitude '<1' represent parameters that were not detected by the characterizer. This happens when the execution time of the parameter is so small that most of its the execution overlapped with other operations; the total execution time of the program does not depend significantly on the occurrence of these parameters.

We can see some characteristics of the machines by looking at the results. For example, it is clear from the tables that the performance of the four supercomputers on the execution of double precision arithmetic is significantly lower than that for single

precision. Single precision arithmetic operations and intrinsic functions take one order of magnitude less time to execute on these machines than double precision. The greatest difference occurs on the IBM 3090/200 with double precision division. This operation takes almost 700ns using 64-bit operands, while the same operation with 128-bit operands takes around 75500ns. In contrast, the same operation takes less than 8000ns in any of the three CRAYs.

By looking at the results of the Sun 4/260 and Sun 3/260 (f), we can see the main differences between them. The greatest performance gap is found in floating-point (real and complex) arithmetic, intrinsic functions, procedure calls and parameter passing. For integer arithmetic this difference is smaller.

It is also possible to compare our results with the numbers reported by manufacturers. However, this is no easy task given that our parameters may not map directly to a particular sequence of instructions and that there are many factors affecting the execution times of instructions. For example, on the 68020 the effective address calculation can take from zero to twenty-four cycles depending on the addressing mode and whether a prefetch instruction or/and an operand read is needed [Moto85, Moto87]. Nevertheless, table 2.4 shows timing estimates for four intrinsic functions (single precision) and also for the sequence of instructions implementing a procedure call. Included in the table are the measurements obtained with the system characterizer.

Table 2.4: **Execution estimates vs. characterization results**

	units	LOGS	EXPS	SINS	TANS	PROC
Timing Est.	cycles	672	598	482	574	113
	nsec	33600	29900	24100	28700	4420
Measurement	nsec	43799	28548	25790	31478	5034
Error		30.5%	4.5%	7.0%	9.7%	13.9%

For the intrinsic functions we assumed that the cycle time was 50 nsec (20MHz), and for procedure call 40 nsec (25MHz). We made some simplifying assumptions that are not necessarily valid for the Sun 3/260. We see that except for the logarithm function, our measurements are sufficiently closed to the timing estimates. This large difference is easily explained by looking at the code generated by the compiler; several additional instructions are included to determine, at execution time, whether to compute $\log(x)$, or $\log(x+1)$.

The effect of different compilers can be seen in the results for the VAX-11/785. The FORT compiler produces code that is significantly faster for complex arithmetic and intrinsic functions, especially single precision intrinsics. There are some strange results in the case of the exponential operator.
While the F77 code is between 2 and 5 times faster using a real base and an integer exponent, the FORT compiler is more than 4 times faster in the case of a real base and a real exponent. A similar situation occurs when the base is integer.

The fact that procedure calls are expensive operations on the VAX architecture can be corroborated when we compare the time it takes to execute this instruction on the VAX 8600 against either the MIPS/1000 or the Sun 4/260. A procedure call is approximately six times slower on the VAX 8600. This large gap is also found in the other VAX

implementations, when we make the comparison against the Sun 3 or IBM RT-PC. This agree with previous studies done on the VAX-11/780 that found that procedure calls take on the average 45.25 cycles to execute, while the average VAX instruction takes only 10.6 cycles [Emer82, Clar82]. On their workload 14 percent of the time was spent executing this type of instructions[3].

2.7.1. A Reduced Representation of the Performance Measurements

The measurements obtained with the system characterizer makes it possible to compare different machine architectures either at the level of the parameters or by predicting the execution times of a set of programs using their parametric dynamic distributions. Predicting the execution time of a program is equivalent to reducing the set of basic measurements to a single number (the execution time) with the dynamic distribution acting as a weighting function. These two types of comparisons represent different extremes. On one side we have too much information with the raw measurements; it is difficult to identify those parameters that most affect performance without making reference to some particular workload. On the other extreme, a single number representing the execution time gives an illusion of precision by hiding the multidimensional aspects of program execution.

Therefore, it is convenient to represent the parameters in some 'reduced' form, in which overall performance is represented using a small number of dimensions, each associated with different aspects of the computation. In this way it is not only possible to compare the performance of a single operation or the overall performance with respect to a given workload, but also to focus on some particular mode of execution.

2.7.2. Combining Measurements and Selecting Weights

The two major issues when we reduce a large number of parameters into a smaller set are how to group the basic measurements, and how much weight to assign to each element.

For the first part we identified a small number of performance 'dimensions', each representing either a hardware or a software feature. These 'dimensions' should be as independent of each other as possible, and should reflect distinct components of the machine. A good selection of these new parameters will help us to better understand the behavior of the system. We use hardware, software and hybrid parameters.
Integer addition is representative of the first group; trigonometric functions of the second; and floating-point arithmetic, which in some machines is executed using special hardware and on others by software routines, belongs to the hybrid group.

The second issue, assigning weights to basic parameters, is a more difficult task, given that the impact of a parameter in the performance of a system is a function of the workload. However, this workload dependency is not as serious a problem as in the case of reducing all parameters to a single number. The relative proportion of integer and

---

[3] On the VAX 8800 series procedure calls and returns take only 27.8 cycles, while the average instruction requires 8.8 cycles. Even with this improvement, the VAX 8800 is 13.8 percent of the time executing procedure calls. For some procedure intensive programs this number can be as high as 62 percent [Clar87, Clar88].

Table 2.5: **Reduced Parameters**

1 memory bandwidth (single)			
TRSL	.250	TISL	.250
TRSG	.250	TISG	.250

2 memory bandwidth (double)			
TCSL	.250	TRDL	.250
TCSG	.250	TRDG	.250

3 integer addition			
AISL	.500	AISG	.500

6 floating-point addition			
ARSL	.500	ARSG	.500

4 integer multiplication			
MISL	.500	MISG	.500

7 floating-point multiplication			
MRSL	.500	MRSG	.500

5 integer arithmetic			
DISL	.400	DISG	.400
EISL	.090	EISG	.090
XISL	.010	XISG	.010

8 floating-point arithmetic			
DRSL	.400	DRSG	.400
ERSL	.090	ERSG	.090
XRSL	.010	XRSG	.010

9 complex precision arithmetic			
ACSL	.325	ACSG	.325
MCSL	.125	MCSG	.125
DCSL	.040	DCSG	.040
ECSL	.008	ECSG	.008
XCSL	.002	XCSG	.002

10 double arithmetic			
ARDL	.325	ARDG	.325
MRDL	.125	MRDG	.125
DRDL	.040	DRDG	.040
ERDL	.008	ERDG	.008
XRDL	.002	XRDG	.002

11 intrinsic functions (single)			
LOGS	.166	TANS	.166
EXPS	.166	SQRS	.166
SINS	.166	MODS	.166

12 intrinsic functions (double)			
LOGC	.100	LOGD	.100
EXPC	.100	EXPD	.100
SINC	.100	SIND	.100
SQRC	.100	SQRD	.100
TAND	.100	MODD	.100

13 logical operations			
ANDL	.250	CISL	.250
CRSL	.250	CDRL	.125
CCSL	.125		

14 pipelining			
GOTO	.900	GCOM	.100
15 procedure calls			
CALL	.750	ARGU	.250

16 address computation			
ARR1	.600	ARR3	.100
ARR2	.300		

17 iteration			
LOIN	.060	LOIX	.030
LOOV	.605	LOOX	.305

Table 2.5: The seventeen reduced parameters, including basic measurements and their respective weights. With the exception of memory bandwidth, the sum of weights for each reduced parameter equals one.

floating-point operations varies greatly from one program to another, but, if we focus only on floating-point, the relative distribution of these operations does not show the same degree of variability. We selected the weights based on extensive statistics of Fortran programs reported in the literature complemented with other statistics produced with

our program analyzer [Knut71, Weic84, Saav88].

In table 2.5 we present the set of raw measurements and weights that formed each of the seventeen reduced parameters. Parameters characterizing hardware functional units are: integer addition and multiplication, logical operations, procedure calls, looping, and memory bandwidth (single and double precision). Software characteristics are represented by trigonometric functions (single and double precision). Floating-point, double precision and complex arithmetic, pipelining, and address computation belong to the hybrid class.

2.7.3. Reduced Measurements and Kiviat Graphs

We present the same experimental measurements shown in tables 2.10-2.14 in Appendix 2.B in terms of the reduced parameters in figures 2.3 and 2.4. These results are also given in tables 2.15-2.17 (Appendix 2.C). In the tables, in addition to the magnitude we give the result normalized with respect to the shortest time, the CRAY Y-MP/832, and the VAX-11/780. For the VAX-11/780 we report the reciprocal. Both Kiviat graphs are logarithmic, with each circle representing a change of one order of magnitude with respect to its nearest neighbor. In figure 2.5 values are in units of nanoseconds with the circle closest to the center representing 50 nanoseconds. Quantities smaller than 50 nanoseconds are plotted in the direction of the center.

Sometimes it is convenient to express the performance distribution of the machine in terms of another machine which we defined as our standard unit of measure. The VAX-11/780 is usually arbitrarily rated as a 1 MIPS machine, and the performance of other machines is given in units of VAX-11/780 MIPS [MIPS88]. We applied a similar transformation to the graphs in figure 2.5 to produce the Kiviat graphs of figure 2.6. Each dimension is normalized with respect to the VAX-11/780. In this case the smallest circle corresponds to a performance equal to one tenth of a VAX-11/780. As in figure 2.5, two adjacent circles have a separation of one order of magnitude.

Using the results from figures 2.6-2.7, and tables 2.15-2.17, we can identify several differences and similarities between the machines. The memory bandwidth results indicate that the only machines that show the same performance in single and double precision memory bandwidth are the CRAY Y-MP, the CRAY-2, and the CRAY X-MP. Although the single precision memory bandwidth in the IBM 3090 is faster than the CRAY Y-MP (34ns vs. 45ns), for double precision this situation is reversed (63ns vs. 40ns)[4]. The memory bandwidth reported here does not necessarily match the numbers given by the manufacturer. Our measurements characterize the execution time of a memory transfer assignment in a Fortran program, and for an arbitrary system this transfer is affected by the availability of registers, data cache, write buffer, and other circuitry that improves the data transfer between the CPU and memory.

We can see the effect of different compilers on the VAX-11/785. The difference in performance between the code produced by the two compilers is less than 10% in the cases of memory bandwidth with single precision, integer arithmetic, and DO loops. The FORT compiler code is 30% faster for real multiplication, 90% percent for complex

[4] The difference between the memory bandwidth measured for the CRAY Y-MP/832 between single and double precision is a result of the measuring tools and the small execution times.

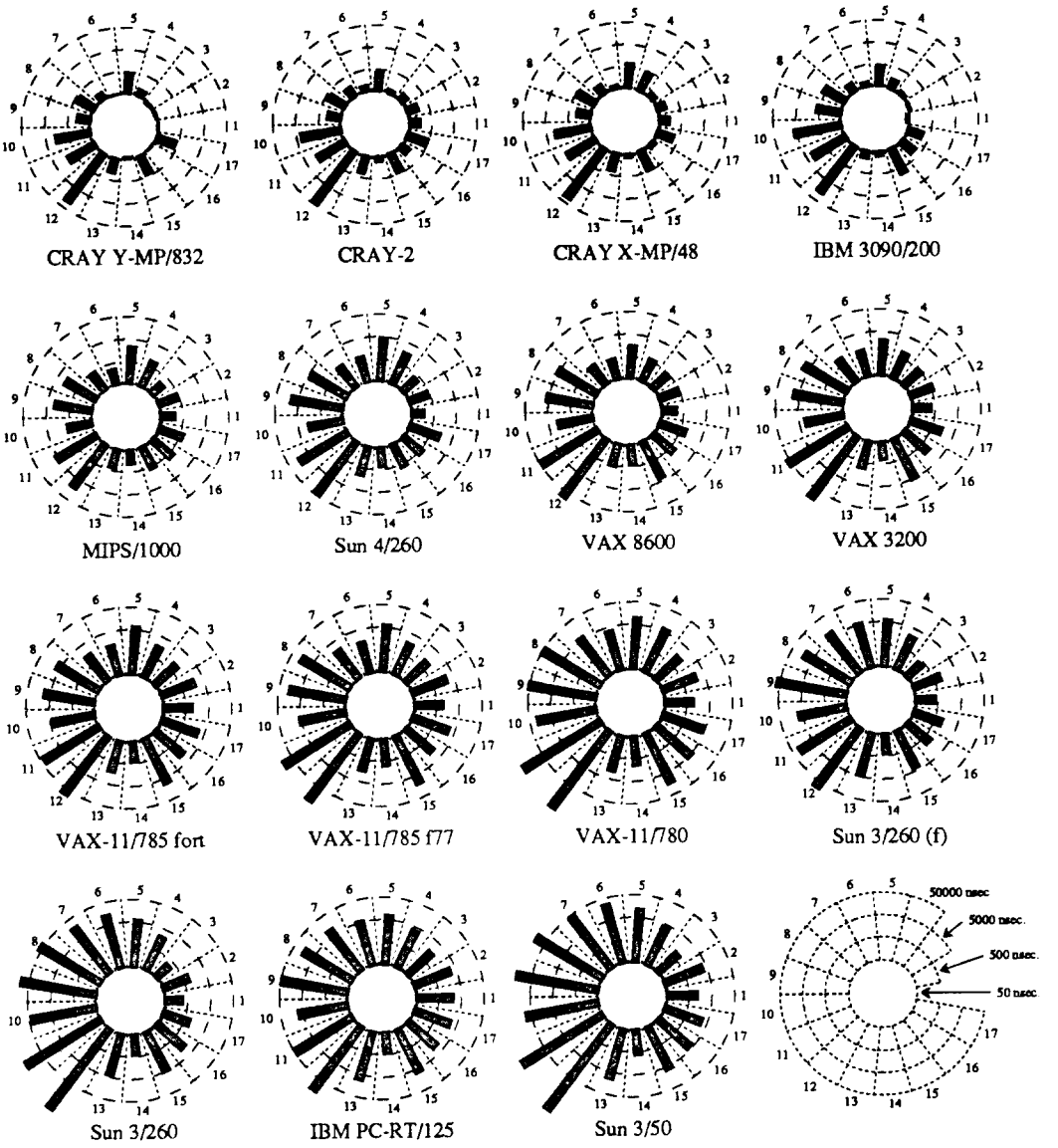


Figure 2.5: Performance of composite parameters. The four concentric circles represents 50, 500, 5000 and 50000 nanoseconds. Each Kiviat graph shows how different machines distribute their performance along different modes of execution.

arithmetic, 130% for real division and exponentials, and more than 3 times faster in intrinsic functions. On the other side, the F77 compiler code is less than 15% faster for integer division, and address computation. For intensive floating-point programs, the code of the FORT compiler clearly outperforms the F77 compiler.

The effect of the floating-point co-processor in the Sun 3/260 is also clear by looking at the results. Using the 68881 increases the performances of intrinsic functions by a factor of more than thirteen, and for floating-point arithmetic by a factor from two to

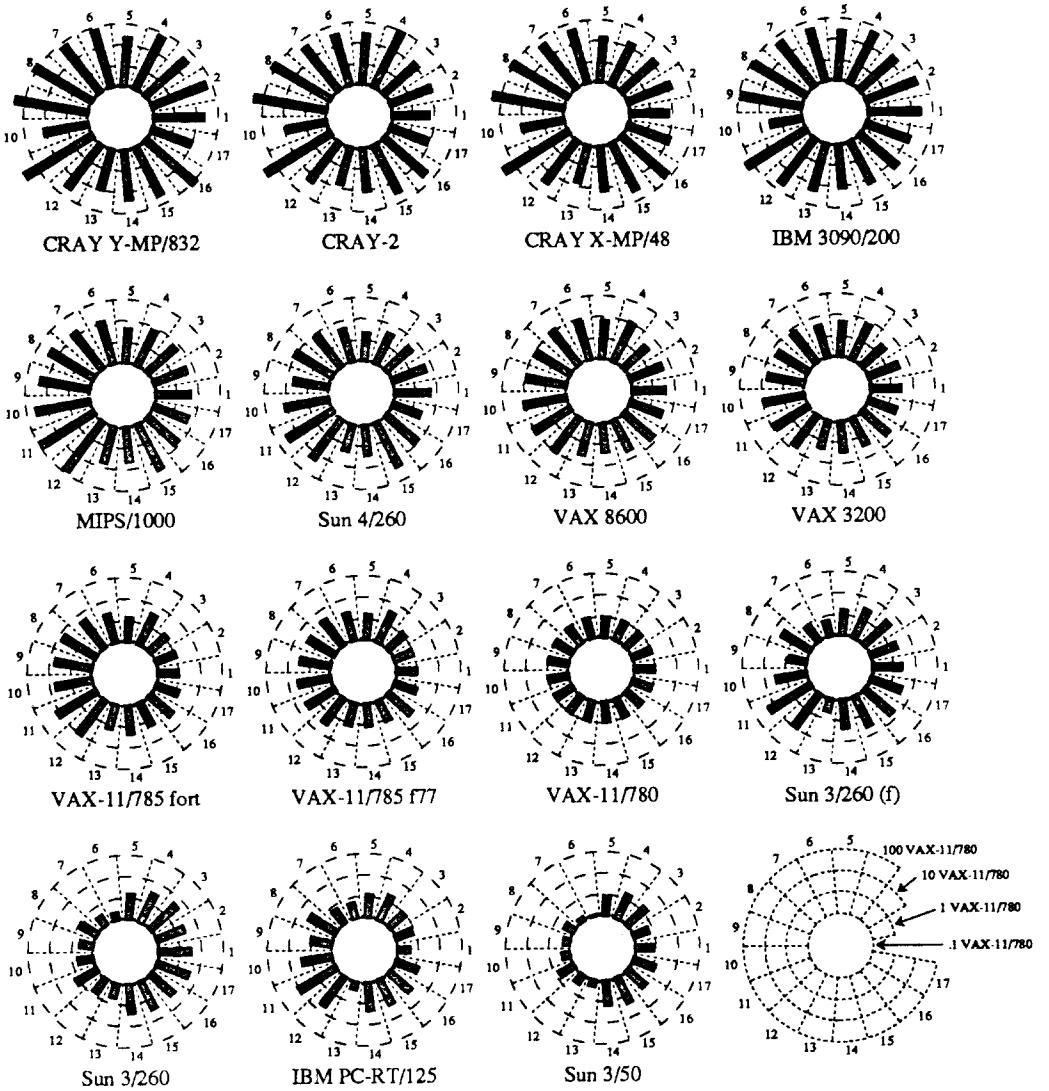


Figure 2.6: Performance of the reduced parameters with respect to the VAX-11/780. The concentric circles represents .1, 1, 10, and 100 times faster. The closest a performance shape (pershape) is to a circle, the closest the machine is to a VAX-11/780 in terms of how both machines distributed their performance along different computational modes.

five.

The CRAY Y-MP/832 has the fastest times for floating-point and complex arithmetic operations, function calls, array references, branching and single precision intrinsic functions (fig. 2.8 and tables 2.11-2.13). In particular, access to array elements is almost six times faster in the CRAY than in the IBM 3090/200 and 127 times faster than in the VAX-11/780. The CRAY machines are highly optimized for those parameters that are extensively used in scientific programs. The IBM 3090 is the fastest machine in double

precision trigonometric functions, single precision memory bandwidth, and logical operations. The CRAY-2 shows similar performance to the CRAY X-MP and Y-MP in integer arithmetic (except integer addition), complex arithmetic, and procedure calls, but memory bandwidth shows a larger difference in both single and double precision.

A comparison between the MIPS/1000 and the Sun-4 shows better performance for the Sun-4 only in memory bandwidth and address computation, and the difference in all cases is less than 15%. The MIPS/1000 has an advantage of more than 75% in integer multiplication and arithmetic, floating-point and complex arithmetic, intrinsic functions,

Floating-point performance on the IBM RT-PC and Sun-3 (50 and 260) is slow compared with other machines and although a co-processor provides a significant improvement, their performance doesn't match the performance of comparable minicomputers. For example, the IBM 3090/200 is less than 12 times faster than Sun 3/50 (15 MHz) in integer addition, but 60 times faster than the Sun 3/260 (25 MHz) with 68881 (20 MHz) in floating-point addition. The Sun 3/260 is between 4-6 times faster than the VAX-11/780 on procedure calls and array references, but the VAX-11/780 outperforms the Sun 3/260 on single precision floating-point addition and multiplication.

2.8. Similar Performance Distributions (Performance Shapes)

Consider two machines M_X and M_Y that are identical except for the clock rates. These machines have the property that, for any benchmark A, their performance ratio (the execution time on one machine divided by the execution time on the other machine) is always a constant; thus, only one benchmark is sufficient to evaluate one against the other. For two arbitrary machines, however, this performance ratio can vary significantly for different benchmarks; it is possible to obtain a wide variety of performance ratios by running a sufficient set of benchmarks. Therefore it is important to quantify how different is the performance distribution of an arbitrary pair of machines and in this way determine how large we can expect the variability in the performance ratio to be when running a large sample of programs. This metric should group machines according to their performance 'shapes' and not by the magnitude of their performance parameters. A *performance shape* (pershape) is the Kiviat graph representing how performance is distributed along the different computational modes (reduced parameters). A pershape tells us not how large a parameter or set of parameters is with respect to other machines but how different machines distribute their performance. In the next subsection we present a metric that measures how similar are the absolute and normalized pershapes of two arbitrary machines.

2.8.1. A Metric for Performance Shapes

We would like a metric that captures the notion of similarity explained in the previous paragraph. By looking at figure 2.5 or 2.6, we clearly see that the pershape of the CRAY Y-MP/832 is very different from the pershapes of the VAX-11/780 or the Sun 3/50. But if we compare the CRAY Y-MP with the CRAY X-MP, or the VAX 8600 with the VAX 3200, we find that, except for their relative sizes, the figures are very similar. It is this informal notion of similarity that we try to capture with the pershape distance.

First, there are several properties that we like our metric to satisfy in addition to the obvious properties required for distances. The pershape distance must be greater than or equal to zero, and the distance from any machine to itself must be zero. It should satisfy

the triangle inequality. It should be symmetric; the distance from A to B must be equal to the distance from B to A. One essential property for our metric is that, if the performance of one machine is increased or decreased by the same quantity in all the dimensions, the new distance does not change. By allowing two different pershapes vectors to have distance zero, we make the pershape distance a semi-metric[5] [Gile87]. Every parameter should have the same weight, and any arbitrary permutation of the dimensions in both machines should not affect the distance. This means that our metric should be a function only of the relative performances of the machines and not of how we plot them. Making each dimension equally important intends to make the distance workload independent. The last property that we require is that, if the performance in one dimension is changed in both machines by the same factor, their relative distance should not be affected.

The following discussion will give the rationale for allowing different pershapes vectors to have distance zero. It is important to understand that we are not trying to measure the difference in performance between two machines, but something completely different. We are interested in the variability of their expected performance. How fast one machine is compared to the other is always a function of the workload we use to evaluated them. What the pershape distance tries to measure is how large is the spectrum of possible comparative performance results when we use any possible workload composition. Therefore, given that two machines have a distance d, if in one machine we increase the performance of every dimension by the same factor (λ), the distance should not be affected. Obviously, the machine will be faster or slower depending on whether λ is greater or less than 1, but the distribution of its performance remains the same. Therefore, its distance to any possible machine should not change. A similar situation happens when we add a constant to a random variable; the mean is affected, but the variance does not change.

Formally, let $X = \langle x_1, x_2, \cdots, x_n \rangle$ and $Y = \langle y_1, y_2, \cdots, y_n \rangle$ be two performance vectors in $(0, \infty)^n$ representing the pershapes of machines M_X and M_Y. The metric

$$d(X,Y) = \left[\frac{1}{n-1} \sum_{i=1}^{n} \left[\log\left(\frac{x_i}{y_i}\right) - \frac{1}{n} \sum_{j=1}^{n} \log\left(\frac{x_j}{y_j}\right) \right]^2 \right]^{1/2} \tag{2.10}$$

satisfies the following set of axioms:

i) $d(X,Y) \geq 0$
ii) $d(X,Y) = 0$ *iff* $X = \lambda Y$ and $\lambda > 0$
iii) $d(X,Y) = d(Y,X)$
iv) $d(X,Y) \leq d(X,Z) + d(Z,Y)$
v) $d(X_\sigma, Y_\sigma) = d(X,Y)$ for any arbitrary permutation σ
vi) $d(\langle \lambda x_1, x_2, \cdots, x_n \rangle, \langle \lambda y_1, y_2, \cdots, y_n \rangle) = d(X,Y)$

Note that equation (10) is not the only possible distance satisfying the axioms; there are

[5] In some textbooks, this is called a pseudo-metric [Kell85, Bour89]. We will not use the prefix 'semi-' or 'pseudo-' and simply refer to it as a metric.

an infinity of different distance metrics with the same basic properties. The only metric property which provides any difficulty to verify is the triangle inequality. To verify axiom iv) we first rewrite equation (10) as follows:

$$d(X,Y)=\left[\sum_{i=1}^{n}\left[\frac{1}{(n-1)^{1/2}}\left[\log(x_i)-\frac{1}{n}\sum_{j=1}^{n}\log(x_j)\right]-\frac{1}{(n-1)^{1/2}}\left[\log(y_i)-\frac{1}{n}\sum_{j=1}^{n}\log(y_j)\right]\right]^2\right]^{1/2} \quad (2.11)$$

then consider the mapping $\phi : (0,\infty)^n \rightarrow R^n$ defined by

$$\phi(x_i)=\frac{1}{(n-1)^{1/2}}\left[\log(x_i)-\frac{1}{n}\sum_{j=1}^{n}\log(x_j)\right].$$

Now, if we substitute $\phi(X)$ and $\phi(Y)$ in equation (2.11)

$$d(X,Y)=\left[\sum_{i=1}^{n}(\phi(x_i)-\phi(y_i))^2\right]^{1/2} \quad (2.12)$$

we obtain the Euclidean metric for R^n, and the verification of the triangle inequality follows directly from the Cauchy-Schwarz inequality [Gile87].

In our presentation of the pershape distance, we did not specify whether vectors X and Y represent absolute or normalized pershapes. Computing a function on a normalized set of values does not always preserves some elementary properties. The output of the function may change when we normalize the inputs. It is important to see how our metric behaves when we normalize the set of reduced parameters.

Let X be an absolute pershape vector and X_Z a normalized vector obtained by dividing each component x_i of X by the same element in Z

$$X_Z=<\frac{x_1}{z_1},\frac{x_2}{z_2},\cdots,\frac{x_n}{z_n}>$$

In linear algebra terms, normalizing vector X with respect to vector Z means applying a linear transformation T to vector X, such that the transformation matrix associated with T is diagonal. The matrix is zero everywhere except on the diagonal, with $1/z_i$ as the diagonal element i. Now the normalized distance is given by

$$d(TX,TY)=d(X_Z,Y_Z)=d(<\frac{x_1}{z_1},\cdots,\frac{x_n}{z_n}>,<\frac{y_1}{z_1},\cdots,\frac{y_n}{z_n}>)=d(X,Y)$$

The rightmost equality follows from the fact that the logarithm of a quotient is equal to the difference of the logarithms. Hence, the sum of all the logarithms of the z_is for X_Z equals the corresponding sum for Y_Z. Therefore, both terms are eliminated from the equation. If we substitute the normalized parameters in equation (2.10), we see that the distance does not change. It is also easy to see that this property is enforced by axioms v) and vi). Thus, we say that distance $d(X,Y)$ is isometric with respect to diagonal linear transformations.

In addition to measuring the distance between two performance vectors, the metric also gives information on which parameters will most affect the benchmark results between two machines. By ordering the terms inside of the first summation in equation

(2.10), we find that the largest terms will be the ones that will contribute more to the summation, and therefore to the distance.

2.8.1.1. Similarity Results

Pershape distances were computed for all pairs of machines to detect which were the most and least similar machines. The most and least similar 25 are reported in table 2.6. The table shows that the most similar machines are the VAX 8600, VAX 3200, and the VAX-11/785, all using the F77 compiler. Other machines that are also close to each other are the Sun 3/50 and the Sun 3/260, both running without the 68881. The differences between these two machine are the clock, the cache and the memory. The Sun 3/50 runs at 15 MHz, does not have a cache and uses standard memory chips. The Sun 3/260 runs at 25 MHz, has 64 Kbyte of virtual address write-back cache, and uses ECC for memory.

Most Similar Machines

	machine	machine	distance
001	VAX 8600	VAX 3200	0.187
002	VAX 8600	VAX-11/785 (f77)	0.214
003	VAX 3200	VAX-11/785 (f77)	0.235
004	Sun 3/50	Sun 3/260	0.291
005	VAX 3200	VAX-11/780	0.425
006	VAX-11/785 (f77)	VAX-11/785 (fort)	0.432
007	CRAY Y-MP/832	CRAY X-MP/48	0.454
008	MIPS/1000	VAX-11/785 (fort)	0.478
009	MIPS/1000	Sun 4/260	0.493
010	VAX 8600	VAX-11/780	0.498
011	VAX 3200	VAX-11/785 (fort)	0.509
012	VAX 8600	VAX-11/785 (fort)	0.516
013	CRAY-2	CRAY X-MP/48	0.518
014	VAX-11/785 (f77)	VAX-11/780	0.519
015	IBM RT-PC/125	Sun 3/260 (f)	0.522
016	CRAY Y-MP/832	CRAY-2	0.532
017	CRAY Y-MP/832	IBM 3090/200	0.661
018	Sun 4/260	VAX-11/785 (fort)	0.663
019	VAX-11/785 (fort)	IBM RT-PC/125	0.672
020	Sun 4/260	IBM RT-PC/125	0.684
021	Sun 4/260	VAX 3200	0.712
022	Sun 4/260	VAX 8600	0.717
023	Sun 4/260	VAX-11/785 (f77)	0.736
024	MIPS/1000	VAX-11/785 (f77)	0.743
025	MIPS/1000	VAX 8600	0.752

Least Similar Machines

	machine	machine	distance
105	CRAY-2	Sun 3/260	1.753
104	CRAY X-MP/48	Sun 3/260	1.725
103	MIPS/1000	Sun 3/260	1.661
102	CRAY X-MP/48	Sun 3/50	1.648
101	CRAY-2	Sun 3/50	1.647
100	MIPS/1000	Sun 3/50	1.591
099	CRAY Y-MP/832	Sun 3/260	1.562
098	VAX-11/785 (fort)	Sun 3/260	1.523
097	CRAY Y-MP/832	Sun 3/50	1.503
096	VAX-11/785 (fort)	Sun 3/50	1.445
095	IBM 3090/200	Sun 3/260	1.434
094	IBM 3090/200	Sun 3/50	1.421
093	CRAY-2	Sun 3/260 (f)	1.420
092	Sun 4/260	Sun 3/260	1.345
091	CRAY X-MP/48	Sun 3/260 (f)	1.303
090	VAX-11/785 (f77)	Sun 3/260	1.300
089	VAX 8600	Sun 3/260	1.296
088	Sun 4/260	Sun 3/50	1.286
087	CRAY-2	VAX-11/780	1.264
086	VAX 8600	Sun 3/50	1.250
085	CRAY Y-MP/832	Sun 3/260	1.242
084	IBM RT-PC/125	Sun 3/260	1.233
083	VAX-11/785 (f77)	Sun 3/50	1.231
082	IBM 3090/200	Sun 3/260 (f)	1.228
081	VAX 3200	Sun 3/260	1.203

Table 2.6: Pairs of machines with the smallest and largest pershape distance.

It is possible to use the results in table 2.6 to identify not only pairs of machines with similar pershapes, but also clusters of machines. Figure 2.7 illustrates one possible diagram showing for all the machines a bidirectional arrow joining the machines that have a distance less than 0.7. Different arrows are used to show how close the machines are. In the diagram we see three connected components, one formed by the supercomputers, another by the small workstations without floating-point co-processors, and a large component mainly formed by two groups having a common neighbor. The closest of the two groups is formed by the machines implementing the VAX architecture and using the F77 compiler. The other group is formed by fast workstations. The VAX-

11/785 using the FORT compiler acts a bridge between the two groups.

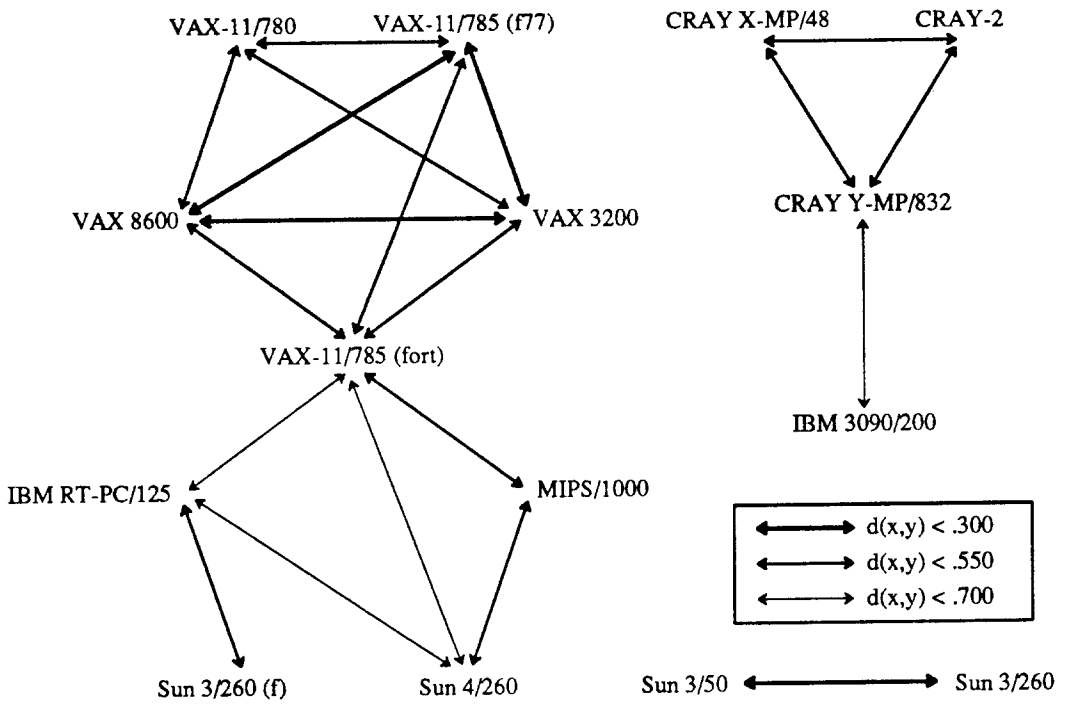


Figure 2.7: All machines with performance distance less than .700 are joined by a double arrow. The pershape distance identifies clusters of machines with similar performance distributions.

2.8.2. An Application of Pershape Distances

By using equation 10, it is possible not only to compute the distance between two machines but also to quantify which 'composite' parameters contribute most to unbalance the overall performance ratio between the two machines. In table 2.7 the execution times of nine programs are given for four of the machines. The table also includes the performance ratio between them, the maximum, minimum and geometric mean of their performance ratios, the maximum ratio of their relative performance and their pershape distance.

The programs used as benchmarks have different execution distributions and can be grouped in the following way: Shell, Erathostenes, and Baskett are integer programs; Alamos [Grif84, Simm87], Linpack [Dong85, Dong88], Livermore [McMa86], and Mandelbrot are floating-point intensive programs; Whetstone [Curr76] is a floating-point and intrinsic function program; and the Smith benchmark [Smit92] mixes floating-point, integer and logical operations. Baskett also executes a large proportion of function calls.

The results in the table show the relation between the pershape distance and the interval of possible benchmark results we can obtain when running a group of benchmarks. The pershape distance between the Sun 3/260 (without 68881) and the Sun 3/50 is only 0.29 and the interval of benchmark results is just 1.41. The difference between

the smallest ratio (1.59) and the largest (2.25) is 41%. The same small distance is found between the IBM RT-PC and the Sun 3/260 (which uses a co-processor). Machines with large distance pershapes also give a large interval in the benchmark results, but the relation is not as clear as in the other cases. A possible explanation is that our program sample is not large enough, and certain types of operations that contribute to a large distance are not present in a large enough proportion to skew the benchmark results. The results do show that the Sun 3/50 can be 1.6 times slower than the Sun 3/260 (with 68881) in a predominantly integer benchmark, but 7.2 times slower in a benchmark with a high number of intrinsic functions. This is consistent with the performance ratios of the parameters representing integer operations and intrinsic functions. By looking at the distances between a group of machines, it is possible to identify which characteristics of the benchmarks will give a more complete evaluation of the systems. In contrast, programs that only exploit one of two characteristics will give skewed results.

	Sun 3/260		IBM RT-PC	Sun 3/50	execution ratios					
program	(f) I	II	III	IV	II/I	III/I	IV/I	III/II	IV/II	IV/III
Alamos	1547.9 s	2838.9 s	3881.9 s	6273.2 s	1.83	2.51	4.05	1.37	2.21	1.62
Baskett	3.92 s	3.88 s	6.20 s	7.06 s	0.99	1.58	1.80	1.60	1.82	1.14
Erathostenes	0.64 s	0.64 s	1.10 s	0.90 s	1.00	1.72	1.59	1.72	1.59	0.93
Linpack	184.9 s	338.5 s	473.9 s	763.7 s	1.83	2.56	4.13	1.40	2.26	1.61
Livermore	507.1 s	1103.1 s	1610.1 s	2457.0 s	2.18	3.18	4.85	1.46	2.23	1.53
Mandelbrot	41.88 s	75.88 s	105.43 s	163.94 s	1.81	2.52	3.92	1.39	2.16	1.56
Shell	1.68 s	1.72 s	4.68 s	3.14 s	1.02	2.79	1.87	2.72	1.83	0.67
Smith	338.3 s	406.7 s	545.10 s	914.8 s	1.20	1.61	2.70	1.34	2.25	1.68
Whetstone	4.74 s	15.28 s	12.05 s	34.24 s	3.22	2.54	7.22	0.79	2.24	2.84
				minimum	0.99	1.58	1.59	0.79	1.59	0.67
				geom. mean	1.55	2.27	3.18	1.46	2.05	1.40
				maximum	3.22	3.18	7.22	2.72	2.26	2.84
				max/min	3.26	2.01	4.53	3.46	1.41	4.23
				d(x,y)	0.96	0.52	0.93	1.23	0.29	1.13

Table 2.7: Execution ratios between pair of machine and comparison against their performance distances. The columns on the left show the execution times (in seconds) for each program and machine. The upper right columns give the execution ratios. left Machines with a small performance distance have less The maximum and minimum entries correspond to the ratio of largest and smallest execution ratios. Machines with a small performance distance have less variability in their relative speed and this should correspond to a small max/min value. We give results for the Sun 3/260 with co-processor ((f) I), and without it (II).

2.9. Weak Points in the Characterizer

The current (new) version of the characterizer incorporates several additional parameters that were previously ignored. This has increased the number of parameters from 76 to 109. Even in this extended model there are several factors that have not been included in the model as it was presented in this chapter. In Chapter 4 and 5 we show how two of these factors: optimization and locality, can be incorporated in our methodology.

2.9.1. Optimization

In this chapter and in the following one we only consider unoptimized code by assuming that the characterizer is compiled and run with optimization disabled. Even when it is not difficult to detect which transformations an optimizer compiler can apply, it is not clear how we should modify the execution time model to include optimized programs.

We study the problem of optimization in detail in Chapter 4. There we show that we can predict the execution time of optimized programs when most of the optimizations exploited by the compiler in the program are invariant. An optimization is invariant with respect to our performance model when the abstract operations executed by the program do not change with optimization.

2.9.2. Locality and Cache Memory

Code that exhibits different locality than our experiments affects the cache hit ratio and in consequence the access time for data and/or instructions. In Chapter 5 we extend our model to include the amount of delay caused by cache and TLB misses. We measure using a special benchmark the main performance parameters that determine the average memory delay in many machines and use these results to evaluate how their memory systems affect the overall CPU performance.

2.9.3. Branching

The size of the branch affects the execution time by modifying the locus of execution. If the target of the branch is to a nonresident page this may involve a page fault and a context switch. A context switch normally involves flushing the cache and this forces a 'cold' start on cache references.

2.9.4. Hardware and/or Software Interlocks

In pipelined machines the time to produce a new result depends on the context in which the instruction is executed. This normally depends (in addition to the effective execution time) on the functional and data dependencies with respect to previously scheduled instructions. As in the previous two factors this is difficult to measure from a high-level program.

2.9.5. Machine Idioms

Special cases of some instructions are optimized to improve execution time. These idioms are used by the compiler whenever possible. Without knowledge of the architecture and the compiler, it is not possible to detect which are the idioms of a given machine. In machines with auto-increment and auto-decrement addressing modes, these modes may be used in statements like `i = i + 1`.

2.9.6. 'Random' Noise Produced by Concurrent Activity

Although we address this problem in §§2.6.5 and 2.6.6, there is still a problem left when we run in a loaded system. A small increase in the load of the system tends to affect the measurements of some parameters, in particular array address computation, branches and loop overhead.

2.10. Conclusions and Summary

In this chapter we have presented a model for machine characterization based on a large number of high-level parameters representing operations for an abstract Fortran machine. This provides a uniform model in which machines with different architectures can be compared on equal terms. It is possible to detect differences and similarities between machines with respect to individual parameters. In addition, we have presented a set of composite parameters that provide a more compact way of representing the effect of hardware or software features in the execution time of programs. Based on these composite parameters we presented the concept of performance shape to show how different machines distribute their possible performance in different ways. We defined a metric to measure the similarity between two pershapes and show how this distance can be used to classify machines and the metric's relation to the variation in benchmark results.

Using the characterization results or the reduced parameters, it is possible to make estimates for the execution time of programs and in this way study the sensitivity of the execution time with respect to variations in the workload. This last aspect will be presented in a next chapter. We think that our approach advances the state of the art of performance evaluation in several ways.

(1) A uniform 'high level' model of the performance of computer systems allows us to make better comparisons between different architectures and identify their differences and similarities when the systems execute a common workload.

(2) Using the characterization to predict performance provides us with a mechanism to validate our assumptions on how the execution time depends on individual components of the system.

(3) With a uniform model that can be used for all machines sharing a common mode of computation, it is possible to define metrics that permit more extensive comparisons and in this way obtain a better understanding of the behavior of each system.

(4) We can study the sensitivity of the system to changes in the workload, and in this way detect imbalances in the architectures.

(5) The results obtained with the system characterizer give insight into the implementation of the CPU architecture, and the machine designers can use the results to improve future implementations.

(6) Application programmers and users can identify the most time consuming parts of their programs and measure the impact of new 'improvements' on different systems.

(7) For procurement purposes this is a less expensive and more flexible way of evaluating computer systems and new architectural features. Although the best way to evaluate a system is to run a real workload, a more extensive and intensive evaluation can be made using system characterizers to select a small number of computers for subsequent on-site evaluation.

In the last thirty years we have seen an explosion of new ideas in many field of computer science, but one problem that has not received much attention is how to make a fair comparison between two different architectures. Given the impact that computers have in all aspects of society we cannot afford to continue characterizing the performance of such complex systems using MIPS, MFLOPS or DHRYSTONES as our units of measure.

2.11. References

[Bail85a] Bailey, D.H., and Barton, J.T., "The NAS Kernel Benchmark Program", NASA Technical Memorandum 86711, August 1985.

[Bail85b] Bailey, D.H., "NAS Kernel Benchmark Results", *Proc. First Int. Conf. on Supercomputing*, St. Petersburg, Florida, December 16-20, 1985, pp. 341-345.

[Bour89] Bourbaki, N., *Elements of Mathematics: General Topology*, Springer Verlag, 1989.

[Clar85] Clark, D.W., and Levy, H.M., "Measurement and Analysis of Instruction Set Usage in the VAX-11/780", *Proc. 9th Annual Symposium on Comp. Arch.*, April 1982.

[Clar87] Clark, D.W., "Pipelining and Performance in the VAX 8800 Processor" *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, October 1987, pp. 173-177.

[Clar88] Clark, D.W., Bannon, P.J., and Keller, J.B., "Measuring VAX 8800 Performance with a Histogram Hardware Monitor", *Proc. 15th Annual Int. Symposium on Comp. Arch.*, Honolulu, HI, June 1988.

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

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

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

[Dong85] Dongarra, J.J., "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment", *Computer Architecture News*, Vol.13, No.1, March 1985, pp. 3-11.

[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", *Computer Architecture News*, Vol.16, No.1, March 1988, pp. 47-69.

[Emer84] Emer, J.S. and Clark, D.W., "A Characterization of Processor Performance in the VAX-11/780", *Proceedings of the 11th Annual Symposium on Computer Architecture*, Ann Arbor, Michigan, June 1984.

[Ferr78] Ferrari, D., *Computer Systems Performance evaluation*, Englewood Cliffs, NJ, Prentice Hall, 1978.

[Ferr84] Ferrari, D., "On the Foundations of Artificial Workload Design", *Proc. 1984 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Cambridge, Massachusetts, August 21-24 1984, pp. 8-14.

[Gile87] Giles, J.R., *Introduction to the Analysis of Metric Spaces*, (Australian Mathematical Society, lecture series 3; Cambridge University Press, 1987).

[Grif84] Griffin, J.H., and Simmons, M.L., "Los Alamos National Laboratory Computer Benchmarking 1983", Los Alamos Technical Report No. LA-10151-MS, June 1984.

[Harm88] Harms, U., and Luttermann, H., "Experiences in Benchmarking the Three Supercomputers CRAY-1M, CRAY-X/MP, Fujitso VP-200 compared with the CYBER 76", *Parallel Computing*, 6 (1988) pp. 373-382.

[Ibbe82] Ibbett, R.N., *The Architecture of High Performance Computers* (Springer-Verlag, New York, 1982).

[IBM87] *IBM 3090 VS Fortran v.2 Language and Library Reference*, SC26-4221-02, 1987.

[Kell87] Kelley, J.L., *General Topology*, GTM; 27, Springer-Verlag, 1985.

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

[LeeJ84] Lee, J.K.F., and Smith, A.J., "Branch Prediction Strategies and Branch Target Buffer Design", *Computer*, Vol.17, No.1, January 1984, pp. 6-22.

[MacD84] MacDougall, M.H., "Instruction-Level Program and Processor Modeling", *Computer*, Vol.7 No.14, July 1982, pp. 14-24.

[McMa86] McMahon, F.H., "The Livermore Fortran Kernels: A Computer Test of the Floating-Point Performance Range", Lawrence Livermore National Laboratory, UCRL-53745, December 1986.

[Merr83] Merrill, H.W., "Repeatability and Variability of CPU timing in Large IBM Systems", *CMG Transactions*, Vol.39, March 1983.

[MIPS88] MIPS Computer Systems, Inc, "Performance Brief CPU Benchmarks", Issue 3.5, October 1988.

[Morg73] Morgan, D.E., and Campbell, J.A., "An Answer to a User's Plea?", *Proc. 1st ACM-SIGME Symp. on Measurement and Evaluation*, February 1973, pp. 112-120.

[Moto85] Motorola, Inc, *MC68020 32-Bit Microprocessor User's Manual*, Prentice-Hall, Inc, 1985.

[Moto87] Motorola, Inc, *MC68881/MC68882 Floating-Point Coprocessor User's Manual*, Prentice-Hall, Inc, 1987.

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

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

[Simm87] Simmons, M.L. and Wasserman H.J., "Los Alamos National Laboratory Computer Benchmarking 1986", Los Alamos National Laboratory, LA-10898-MS, January 1987.

[Smit82] Smith, A.J., "CPU Cache Memories", *ACM Computing Surveys*, Vol.14, No.3, September 1982, pp. 473-530.

[Smit92] Smith, A.J., paper in preparation, January 1992.

[Weic84] Weicker, R.,P., "Dhrystone: A Synthetic Systems Programming Benchmark", *Communications of the ACM*, Vol.27, No.10, October 1984.

[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.

Appendix 2.A

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 2.8: 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 2.9: Abstract operations in the System Characterizer (part 2 of 2)

Appendix 2.B

Group 1: Floating-Point Arithmetic Operations (single, local)

machine	SRSL	ARSL	MRSL	DRSL	ERSL	XRSL	TRSL
CRAY Y-MP/832	13	46	111	210	660	4150	96
CRAY-2	39	70	101	250	78	4180	112
CRAY X-MP/432	82	76	154	357	91	5035	281
IBM 3090/200	1<	82	140	684	129	4952	60
MIPS/1000	67	269	437	976	543	53018	499
Sun 4/260	104	755	788	2496	4724	60430	533
VAX 8600	72	425	575	1610	1097	217676	509
VAX 3200	262	805	999	2013	1847	361666	587
VAX-11/785 fort	263	1282	1524	3778	16305	82006	1799
VAX-11/785 f77	246	1371	1924	4034	3740	648082	2065
VAX-11/780	1086	3215	6739	9322	11041	2066420	1598
Sun 3/260 (f)	1978	5543	8709	11394	15998	58901	1293
Sun 3/260	1<	13580	19118	23003	31612	2205175	1286
Sun 3/50	1<	26420	40246	46476	60818	4743815	3076
IBM RT-PC/125	3639	5684	10715	12304	12437	231989	6235

Group 2: Floating-Point Arithmetic Operations (complex, local)

machine	SCSL	ACSL	MCSL	DCSL	ECSL	XCSL	TCSL
CRAY Y-MP/832	30	85	267	497	818	10466	147
CRAY-2	32	110	221	386	48	17167	199
CRAY X-MP/432	63	124	271	511	1<	13168	319
IBM 3090/200	26	215	679	3218	2940	13912	97
MIPS/1000	121	926	1727	12025	9004	72791	1097
Sun 4/260	1<	8034	11808	29356	7561	130805	663
VAX 8600	275	1438	3523	39419	17876	326399	974
VAX 3200	792	2287	6925	47240	30134	510817	1072
VAX-11/785 fort	531	2653	7542	53236	26842	314924	3514
VAX-11/785 f77	1074	4717	10206	88085	83278	966246	4703
VAX-11/780	1319	9679	38202	328270	170337	3584596	3796
Sun 3/260 (f)	436	27270	83719	353726	133222	446378	1382
Sun 3/260	1<	31812	109547	604151	183495	5417755	1095
Sun 3/50	265	63460	231185	1233098	453373	11405138	8310
IBM RT-PC/125	471	26969	47498	194262	183060	678778	5101

Group 3: Integer Arithmetic Operations (single, local)

machine	SISL	AISL	MISL	DISL	EISL	XISL	TISL
CRAY Y-MP/832	1<	39	106	271	1113	1131	82
CRAY-2	1<	61	62	324	126	131	114
CRAY X-MP/432	1<	91	414	714	396	755	320
IBM 3090/200	1<	76	143	439	163	358	73
MIPS/1000	1<	227	945	2577	1111	2146	475
Sun 4/260	1<	286	1634	3918	5882	7979	219
VAX 8600	1<	357	628	1591	896	1883	462
VAX 3200	1<	490	895	2206	1273	2592	750
VAX-11/785 fort	1<	1002	1615	7292	1760	28928	2259
VAX-11/785 f77	1<	1088	1789	7053	2309	5142	2182
VAX-11/780	1<	1327	6924	10502	7779	15803	2186
Sun 3/260 (f)	1<	298	2212	4011	13979	17174	393
Sun 3/260	1<	237	2280	4119	14708	17398	251
Sun 3/50	1<	813	3898	7039	29262	36348	856
IBM RT-PC/125	1<	1497	3438	8837	4063	7581	2478

Table 2.10: Characterization results for Group 1-3. A value 1< indicates that the parameter was not detected by the experiment.

Group 4: Floating-Point Arithmetic Operations (double, local)

machine	SRDL	ARDL	MRDL	DRDL	ERDL	XRDL	TRDL
CRAY Y-MP/832	2	917	1626	5473	4804	108121	13
CRAY-2	1<	1974	2752	7355	2072	194054	1<
CRAY X-MP/432	69	1122	1812	6392	1073	138645	206
IBM 3090/200	1<	424	964	75656	1493	48282	154
MIPS/1000	117	346	581	1556	838	49780	632
Sun 4/260	290	986	1228	4665	7046	133573	1058
VAX 8600	220	754	1725	5812	2841	208984	876
VAX 3200	276	1367	1896	4063	3256	353750	1178
VAX-11/785 fort	1047	2280	4243	7996	23386	177403	3920
VAX-11/785 f77	929	2893	5460	8921	9517	636736	5637
VAX-11/780	1142	10589	24687	48235	33181	2044644	5483
Sun 3/260 (f)	2159	5819	9272	11942	17793	112601	2610
Sun 3/260	1172	23804	49458	73051	46323	2482220	1364
Sun 3/50	2068	54245	100594	132284	110522	5504681	6682
IBM RT-PC/125	5765	6889	8125	12611	14110	200954	4623

Group 5: Floating-Point Arithmetic Operations (single, global)

machine	SRSG	ARSG	MRSG	DRSG	ERSG	XRSG	TRSG
CRAY Y-MP/832	13	46	111	210	660	4150	96
CRAY-2	95	124	216	392	72	3811	506
CRAY X-MP/432	90	73	152	354	83	5052	287
IBM 3090/200	12	80	129	685	152	4901	60
MIPS/1000	70	268	435	973	536	50784	364
Sun 4/260	145	778	855	2573	4739	60387	573
VAX 8600	254	483	598	1600	1040	215039	408
VAX 3200	400	878	1076	2159	1770	361567	554
VAX-11/785 fort	998	1244	1501	3619	16318	81494	1430
VAX-11/785 f77	219	1378	1928	4049	3727	651381	2070
VAX-11/780	1517	3304	6646	9539	10649	2056123	1164
Sun 3/260 (f)	1948	5616	8945	11662	16018	58321	1157
Sun 3/260	1<	13433	18920	22865	31453	2139632	716
Sun 3/50	1<	26943	40586	47035	58663	4671805	5092
IBM RT-PC/125	3156	5629	9684	12287	13054	230580	6636

Group 6: Floating-Point Arithmetic Operations (complex, global)

machine	SCSG	ACSG	MCSG	DCSG	ECSG	XCSG	TCSG
CRAY Y-MP/832	30	85	267	497	818	10466	147
CRAY-2	92	167	303	513	18	16738	512
CRAY X-MP/432	78	117	265	511	1<	13177	335
IBM 3090/200	27	230	682	3162	2983	13965	102
MIPS/1000	121	927	1730	12049	8992	73007	1101
Sun 4/260	63	8027	12078	29703	7573	130146	664
VAX 8600	551	1544	3802	38787	17812	326228	1159
VAX 3200	519	2859	7334	46918	31358	511323	1599
VAX-11/785 fort	1055	3210	8827	52768	24348	298978	4393
VAX-11/785 f77	1144	4695	10039	87745	83649	962812	4680
VAX-11/780	1684	9778	35853	322237	168501	3679780	3297
Sun 3/260 (f)	1<	27975	83103	352636	134477	453818	1519
Sun 3/260	3695	29210	107292	586288	191029	5307737	1<
Sun 3/50	2383	63526	231688	1233524	448297	11359785	8016
IBM RT-PC/125	555	26948	47435	197036	182374	693827	5216

Table 2.11: Characterization results for Group 4-6. A value 1< indicates that the parameter was not detected by the experiment.

Group 7: Integer Arithmetic Operations (single, global)

machine	SISG	AISG	MISG	DISG	EISG	XISG	TISG
CRAY Y-MP/832	1<	39	106	271	1113	1131	82
CRAY-2	1<	161	89	485	144	153	607
CRAY X-MP/432	1<	93	405	716	405	751	327
IBM 3090/200	1<	79	151	439	170	393	82
MIPS/1000	1<	227	942	2580	1110	2143	476
Sun 4/260	1<	421	1728	4022	6022	7972	252
VAX 8600	1<	522	606	1593	990	2010	622
VAX 3200	1<	594	1028	2202	1484	2852	826
VAX-11/785 fort	1<	1113	1888	7428	1857	29838	2269
VAX-11/785 f77	1<	1094	1788	7025	2279	5089	2167
VAX-11/780	1<	1616	7166	10731	8002	16036	2156
Sun 3/260 (f)	1<	438	2116	4015	14156	17771	427
Sun 3/260	1<	381	2121	4050	14212	16824	218
Sun 3/50	1<	937	3537	6887	29760	36609	738
IBM RT-PC/125	1<	1459	3422	8865	3956	7553	2438

Group 8: Floating-Point Arithmetic Operations (double, global)

machine	SRDG	ARDG	MRDG	DRDG	ERDG	XRDG	TRDG
CRAY Y-MP/832	2	917	1626	5473	4804	108121	13
CRAY-2	1<	2051	2858	7360	2283	202494	302
CRAY X-MP/432	69	1122	1821	6342	1108	138935	222
IBM 3090/200	1<	421	963	73307	912	41150	154
MIPS/1000	108	349	587	1561	854	58483	679
Sun 4/260	278	961	1167	4586	7078	133796	1060
VAX 8600	252	796	1611	5905	2828	206974	817
VAX 3200	268	1515	2175	4238	3307	353997	1080
VAX-11/785 fort	1207	2202	4106	8044	23236	171685	3882
VAX-11/785 f77	968	2268	4498	7916	9192	636084	4461
VAX-11/780	1274	10848	24648	47719	34214	2024890	4551
Sun 3/260 (f)	2354	5790	9275	11817	18065	112638	2477
Sun 3/260	549	23648	49458	72612	45612	2562978	2664
Sun 3/50	2436	54749	100942	133431	110630	5495105	4046
IBM RT-PC/125	3799	6017	10228	13526	14088	203231	7612

Group 9,10: Conditional and Logical Parameters

machine	ANDL	CRSL	CCSL	CISL	CRDL	ANDG	CRSG	CCSG	CISG	CRDG
CRAY Y-MP/832	14	287	315	282	335	14	287	315	282	335
CRAY-2	36	95	237	96	1873	85	317	430	337	1963
CRAY X-MP/432	45	226	335	229	1243	46	228	322	231	1256
IBM 3090/200	104	94	106	73	214	111	138	171	161	259
MIPS/1000	185	471	375	337	603	183	474	404	335	602
Sun 4/260	310	1217	3767	236	1566	455	1333	4168	655	1585
VAX 8600	304	653	680	464	867	321	868	991	743	1022
VAX 3200	389	1127	1295	767	1603	412	1207	1371	844	1602
VAX-11/785 fort	954	1378	1727	1033	2649	1013	1747	2348	1116	2264
VAX-11/785 f77	769	1937	1966	1467	2578	768	1909	1937	1477	2629
VAX-11/780	1091	2823	2768	1987	3914	1100	3057	3674	2481	4573
Sun 3/260 (f)	414	6814	16257	329	7171	588	7093	15922	741	7057
Sun 3/260	394	5542	10938	332	10730	604	5728	11985	847	9765
Sun 3/50	803	13243	29047	559	22442	1399	14382	27969	1625	25800
IBM RT-PC/125	1005	16166	16033	2012	15919	1029	15430	15700	2130	15993

Table 2.12: Characterization results for Group 7-10. A value 1< indicates that the parameter was not detected by the experiment.

Group 11,12: Function Call, Arguments and References to Array Elements

machine	PROC	ARGU	ARR1	ARR2	ARR3	IADD
CRAY Y-MP/832	512	61	42	49	60	12
CRAY-2	574	40	122	159	200	1<
CRAY X-MP/432	583	73	59	104	148	2
IBM 3090/200	1162	70	128	410	746	17
MIPS/1000	797	139	523	1044	1592	1<
Sun 4/260	918	67	384	1004	1490	13
VAX 8600	4670	610	478	1223	2137	1<
VAX 3200	6991	957	668	1934	3316	1<
VAX-11/785 fort	11678	1515	1320	2897	5578	844
VAX-11/785 f77	16421	1526	995	2701	5057	32
VAX-11/780	19931	1783	2126	9592	18518	1<
Sun 3/260 (f)	5034	397	448	1661	2600	2
Sun 3/260	6548	594	990	3834	3484	1<
Sun 3/50	8838	1535	2042	6396	8759	100
IBM RT-PC/125	9395	991	2212	2406	4536	1<

Group 13,14: Branching and DO loop Parameters

machine	GOTO	GCOM	LOIN	LOOV	LOIX	LOOX
CRAY Y-MP/832	1<	406	1015	315	627	368
CRAY-2	15	692	1263	353	264	513
CRAY X-MP/432	25	483	966	180	1307	293
IBM 3090/200	38	460	660	130	952	353
MIPS/1000	137	1010	1938	417	1643	945
Sun 4/260	302	984	3378	1007	2320	1638
VAX 8600	262	1705	2540	396	6223	1070
VAX 3200	128	2117	3916	975	5336	1634
VAX-11/785 fort	277	1691	13042	972	11747	3124
VAX-11/785 f77	332	4262	8323	1621	7644	2768
VAX-11/780	588	4783	2525	2552	17363	4558
Sun 3/260 (f)	258	1742	2863	567	3256	1509
Sun 3/260	268	1694	1657	524	1957	1411
Sun 3/50	394	3001	6558	1976	5765	3776
IBM RT-PC/125	119	3395	11368	1236	5425	3396

Group 15: Intrinsic Functions (single precision)

machine	EXPS	LOGS	SINS	TANS	SQRS	ABSS	MODS	MAXS
CRAY Y-MP/832	1453	1314	1423	1514	1038	1<	265	177
CRAY-2	1980	1855	2067	2136	266	25	383	328
CRAY X-MP/432	1826	1627	1846	1985	1356	1<	318	200
IBM 3090/200	2893	2887	2805	4119	2534	37	1094	435
MIPS/1000	6612	5680	5751	5156	6745	61	7215	1470
Sun 4/260	13560	14197	12081	20338	14520	450	23141	4758
VAX 8600	67798	52587	42683	70577	23883	1285	26471	3275
VAX 3200	109786	77167	63001	99637	32436	2108	38300	4563
VAX-11/785 fort	27212	28438	39474	70494	22634	215	42421	4101
VAX-11/785 f77	204824	240223	109462	138871	56848	2996	88497	8302
VAX-11/780	690106	765999	468763	857151	177536	4230	186125	12234
Sun 3/260 (f)	43799	28548	25790	31478	12627	464	15571	15528
Sun 3/260	367032	443458	574151	686006	61509	1<	49869	18798
Sun 3/50	770610	950878	1272922	1512997	92447	4700	129932	47730
IBM RT-PC/125	27466	22327	23168	26511	7014	47189	179593	41101

Table 2.13: Characterization results for Group 11-15. A value 1< indicates that the parameter was not detected by the experiment.

Group 16: Intrinsic Functions (double precision)

machine	EXPD	LOGD	SIND	TAND	SQRD	ABSD	MODD	MAXD
CRAY Y-MP/832	51052	58111	32289	71166	8689	28	9581	2200
CRAY-2	88428	94268	67440	146937	12100	431	19506	1021
CRAY X-MP/432	70511	64914	37931	83390	9751	21	9459	727
IBM 3090/200	20471	21893	19390	28520	10193	70	76571	858
MIPS/1000	8565	7508	7997	7747	9330	39	6985	2385
Sun 4/260	22261	22220	21184	36096	27382	504	18995	6106
VAX 8600	67151	52267	41751	69792	23310	1755	24244	5083
VAX 3200	108029	79491	63237	101020	31103	3001	35793	7175
VAX-11/785 fort	51621	51081	97932	158473	30389	693	71349	7050
VAX-11/785 f77	203491	238536	107896	137069	55608	5906	84380	17867
VAX-11/780	701933	776686	467842	856357	179556	7504	176955	22079
Sun 3/260 (f)	46526	32093	28500	32619	13966	312	17058	19584
Sun 3/260	965293	1096555	1009146	1132512	94349	1<	60492	22428
Sun 3/50	2080362	2332882	2210543	2418819	175124	1<	150656	67713
IBM RT-PC/125	38928	34208	34753	37343	13901	11669	120334	44149

Groups 17,18: Intrinsic Functions (integer and complex)

machine	ABSI	MODI	MAXI	EXPC	LOGC	SINC	SQRC	ABSC
CRAY Y-MP/832	76	563	127	6093	4478	5027	4282	1784
CRAY-2	51	545	202	9299	7827	9244	3761	1618
CRAY X-MP/432	58	1644	192	7913	5755	6553	5020	2309
IBM 3090/200	93	541	399	6948	5384	7081	6188	2302
MIPS/1000	169	2607	1415	21639	20454	25382	17361	6955
Sun 4/260	1027	3261	2957	87132	46483	123282	73181	38489
VAX 8600	1381	2546	2983	168775	145238	262011	87857	47671
VAX 3200	1498	3640	3816	266255	233369	438531	118507	62425
VAX-11/785 fort	563	11022	3367	156792	81107	88997	102146	54562
VAX-11/785 f77	2897	8970	8096	458645	491650	760555	199835	113818
VAX-11/780	4262	16165	10719	1749767	1637191	2510877	626909	299780
Sun 3/260 (f)	1665	3721	3825	178628	196966	231844	232360	26185
Sun 3/260	3186	6529	3414	2722348	2339489	3656209	649596	168115
Sun 3/50	9953	15760	13886	5734016	5100953	7775319	1338978	364009
IBM RT-PC/125	2549	9232	8196	410201	233930	511218	380805	258205

Table 2.14: Characterization results for Group 16-18. A value 1< indicates that the parameter was not detected by the experiment.

Appendix 2.C

Parameter 1: Memory bandwidth (single precision)

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	44.5	1.30	1.00	22.27
CRAY-2	167.4	4.87	3.76	5.92
CRAY X-MP/48	151.8	4.41	3.41	6.53
IBM 3090/200	34.4	1.00	0.77	28.80
MIPS/1000	226.6	6.59	5.09	4.37
Sun 4/260	197.0	5.73	4.43	5.03
VAX 8600	250.1	7.27	5.62	3.96
VAX 3200	339.5	9.87	7.63	2.92
VAX-11/785 (fort)	969.8	28.19	21.79	1.02
VAX-11/785 (f77)	1004.5	29.20	22.57	0.99
VAX-11/780	990.8	28.80	22.27	1.00
Sun 3/260 (f)	408.8	11.89	9.19	2.42
Sun 3/260	308.9	8.98	6.94	3.21
IBM RT-PC/125	2223.3	64.63	49.96	0.45
Sun 3/50	1220.1	35.47	27.42	0.81

Parameter 2: Memory bandwidth (double precision)

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	40.1	1.00	1.00	52.98
CRAY-2	168.9	4.21	4.21	12.58
CRAY X-MP/48	135.2	3.37	3.37	15.71
IBM 3090/200	63.4	1.58	1.58	33.51
MIPS/1000	438.6	10.94	10.94	4.84
Sun 4/260	430.7	10.74	10.74	4.93
VAX 8600	478.3	11.93	11.93	4.44
VAX 3200	616.0	15.36	15.36	3.45
VAX-11/785 (fort)	1963.7	48.97	48.97	1.08
VAX-11/785 (f77)	2282.2	56.91	56.91	0.93
VAX-11/780	2124.4	52.98	52.98	1.00
Sun 3/260 (f)	998.6	24.90	24.90	2.13
Sun 3/260	853.8	21.29	21.29	2.49
IBM RT-PC/125	2818.8	70.29	70.29	0.75
Sun 3/50	3381.7	84.33	84.33	0.63

Parameter 3: Integer addition

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	38.6	1.00	1.00	37.54
CRAY-2	110.8	2.87	2.87	13.08
CRAY X-MP/48	91.9	2.38	2.38	15.77
IBM 3090/200	77.3	2.00	2.00	18.74
MIPS/1000	227.4	5.89	5.89	6.37
Sun 4/260	353.4	9.16	9.16	4.10
VAX 8600	439.6	11.39	11.39	3.30
VAX 3200	541.9	14.04	14.04	2.67
VAX-11/785 (fort)	1057.3	27.39	27.39	1.37
VAX-11/785 (f77)	1108.2	28.71	28.71	1.31
VAX-11/780	1448.9	37.54	37.54	1.00
Sun 3/260 (f)	367.9	9.53	9.53	3.94
Sun 3/260	309.2	8.01	8.01	4.69
IBM RT-PC/125	1477.9	38.29	38.29	0.98
Sun 3/50	875.0	22.67	22.67	1.66

Parameter 4: Integer multiplication

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	105.6	1.40	1.00	65.76
CRAY-2	75.7	1.00	0.72	91.74
CRAY X-MP/48	409.5	5.41	3.88	16.96
IBM 3090/200	147.2	1.95	1.40	47.18
MIPS/1000	943.9	12.47	8.94	7.36
Sun 4/260	1681.0	22.21	15.92	4.13
VAX 8600	617.1	8.15	5.84	11.26
VAX 3200	961.6	12.70	9.11	7.22
VAX-11/785 (fort)	1751.4	23.14	16.59	3.97
VAX-11/785 (f77)	1810.2	23.91	17.14	3.84
VAX-11/780	6944.7	91.74	65.76	1.00
Sun 3/260 (f)	2164.3	28.59	20.50	3.21
Sun 3/260	2200.8	29.07	20.84	3.16
IBM RT-PC/125	3430.1	45.31	32.48	2.03
Sun 3/50	3717.4	49.11	35.20	1.87

Parameter 6: Floating-point addition (single)

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	46.3	1.00	1.00	71.57
CRAY-2	97.0	2.10	2.10	34.16
CRAY X-MP/48	74.4	1.61	1.61	44.54
IBM 3090/200	81.2	1.75	1.75	40.81
MIPS/1000	268.4	5.80	5.80	12.35
Sun 4/260	766.3	16.55	16.55	4.32
VAX 8600	454.0	9.81	9.81	7.30
VAX 3200	841.3	18.17	18.17	3.94
VAX-11/785 (fort)	1263.3	27.29	27.29	2.62
VAX-11/785 (f77)	1391.5	30.06	30.06	2.38
VAX-11/780	3313.8	71.57	71.57	1.00
Sun 3/260 (f)	5579.7	120.51	120.51	0.59
Sun 3/260	13506.7	291.72	291.72	0.25
IBM RT-PC/125	5656.3	122.17	122.17	0.59
Sun 3/50	26681.8	576.28	576.28	0.13

Parameter 7: Floating-point multiplication (single)

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	110.8	1.00	1.00	61.21
CRAY-2	158.2	1.43	1.43	42.87
CRAY X-MP/48	153.0	1.38	1.38	44.33
IBM 3090/200	134.5	1.22	1.22	50.43
MIPS/1000	436.0	3.94	3.94	15.56
Sun 4/260	821.4	7.41	7.41	8.26
VAX 8600	586.7	5.30	5.30	11.56
VAX 3200	1037.2	9.36	9.36	6.54
VAX-11/785 (fort)	1512.8	13.65	13.65	4.48
VAX-11/785 (f77)	1952.5	17.62	17.62	3.47
VAX-11/780	6782.1	61.21	61.21	1.00
Sun 3/260 (f)	8827.0	79.67	79.67	0.77
Sun 3/260	19018.9	171.65	171.65	0.36
IBM RT-PC/125	10199.2	92.05	92.05	0.67
Sun 3/50	40416.1	364.77	364.77	0.17

Table 2.15: Parameter values, and performance ratios with respect to fastest machine (m/f), CRAY Y-MP/832 (m/cray), and VAX-11/780 (v780/m) for reduced parameters 1-6. Numbers in column 'value' are given in nanoseconds.

Parameter 5: Integer arithmetic

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	439	1.25	1.00	23.01
CRAY-2	350	1.00	0.80	28.83
CRAY X-MP/48	659	1.88	1.50	15.34
IBM 3090/200	388	1.11	0.88	26.03
MIPS/1000	2305	6.58	5.25	4.39
Sun 4/260	4406	12.57	10.03	2.29
VAX 8600	1482	4.23	3.38	6.82
VAX 3200	2065	5.89	4.70	4.89
VAX-11/785 (fort)	6801	19.40	15.48	1.49
VAX-11/785 (f77)	6286	17.93	14.31	1.61
VAX-11/780	10108	28.83	23.01	1.00
Sun 3/260 (f)	6092	17.38	13.87	1.66
Sun 3/260	6212	17.72	14.14	1.63
IBM RT-PC/125	7953	22.69	18.11	1.27
Sun 3/50	11612	33.12	26.43	0.87

Parameter 8: Floating-point arithmetic (single)

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	369	1.06	1.00	133.02
CRAY-2	350	1.00	0.95	140.36
CRAY X-MP/48	400	1.15	1.09	122.63
IBM 3090/200	671	1.92	1.82	73.18
MIPS/1000	1914	5.47	5.18	25.66
Sun 4/260	4087	11.68	11.07	12.02
VAX 8600	5803	16.58	15.71	8.47
VAX 3200	9226	26.35	24.98	5.33
VAX-11/785 (fort)	7529	21.51	20.38	6.53
VAX-11/785 (f77)	17268	49.33	46.75	2.85
VAX-11/780	49138	140.36	133.02	1.00
Sun 3/260 (f)	13276	37.92	35.94	3.70
Sun 3/260	67471	192.72	182.65	0.73
IBM RT-PC/125	16756	47.86	45.36	2.93
Sun 3/50	142314	406.50	385.26	0.35

Parameter 9: Complex arithmetic

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	206	1.00	1.00	261.67
CRAY-2	242	1.18	1.18	222.98
CRAY X-MP/48	225	1.09	1.09	239.76
IBM 3090/200	661	3.20	3.20	81.79
MIPS/1000	2369	11.45	11.45	22.85
Sun 4/260	11087	53.59	53.59	4.88
VAX 8600	6295	30.43	30.43	8.60
VAX 3200	9041	43.70	43.70	5.99
VAX-11/785 (fort)	9547	46.15	46.15	5.67
VAX-11/785 (f77)	17514	84.65	84.65	3.09
VAX-11/780	54138	261.67	261.67	1.00
Sun 3/260 (f)	70687	341.65	341.65	0.77
Sun 3/260	113826	550.15	550.15	0.48
IBM RT-PC/125	50206	242.66	242.66	1.08
Sun 3/50	239606	1158.08	1158.08	0.23

Parameter 10: Double precision arithmetic

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	1846	1.00	1.00	12.63
CRAY-2	3229	1.75	1.75	7.22
CRAY X-MP/48	2127	1.15	1.15	10.96
IBM 3090/200	6626	3.59	3.59	3.52
MIPS/1000	673	0.37	0.37	34.65
Sun 4/260	1841	1.00	1.00	12.67
VAX 8600	2061	1.12	1.12	11.31
VAX 3200	2804	1.52	1.52	8.32
VAX-11/785 (fort)	4112	2.23	2.23	5.67
VAX-11/785 (f77)	5775	3.13	3.13	4.04
VAX-11/780	23323	12.63	12.63	1.00
Sun 3/260 (f)	7684	4.16	4.16	3.04
Sun 3/260	41962	22.73	22.73	0.56
IBM RT-PC/125	8380	4.54	4.54	2.78
Sun 3/50	89383	48.42	48.42	0.26

Parameter 11: Intrinsic functions (single)

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	1167	1.00	1.00	449.60
CRAY-2	1447	1.24	1.24	362.68
CRAY X-MP/48	1492	1.28	1.28	351.74
IBM 3090/200	2722	2.33	2.33	192.90
MIPS/1000	6193	5.30	5.30	84.79
Sun 4/260	16306	13.96	13.96	32.20
VAX 8600	47333	40.53	40.53	11.09
VAX 3200	70054	59.98	59.98	7.50
VAX-11/785 (fort)	38445	32.92	32.92	13.66
VAX-11/785 (f77)	145176	124.31	124.31	3.62
VAX-11/780	525084	449.60	449.60	1.00
Sun 3/260 (f)	26302	22.52	22.52	19.97
Sun 3/260	363671	311.39	311.39	1.45
IBM RT-PC/125	47679	40.83	40.83	11.01
Sun 3/50	788297	674.97	674.97	0.67

Parameter 12: Intrinsic functions (double)

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	25076	1.24	1.00	38.54
CRAY-2	45881	2.26	1.83	21.06
CRAY X-MP/48	30119	1.49	1.20	32.09
IBM 3090/200	20263	1.00	0.81	47.69
MIPS/1000	13296	0.66	0.53	72.68
Sun 4/260	47821	2.36	1.91	20.21
VAX 8600	94239	4.65	3.76	10.26
VAX 3200	147533	7.28	5.88	6.55
VAX-11/785 (fort)	88988	4.39	3.55	10.86
VAX-11/785 (f77)	285871	14.11	11.40	3.38
VAX-11/780	966457	47.69	38.54	1.00
Sun 3/260 (f)	101056	4.99	4.03	9.57
Sun 3/260	1372600	67.74	54.74	0.70
IBM RT-PC/125	181562	8.96	7.24	5.32
Sun 3/50	2931770	144.68	116.91	0.33

Table 2.16: Parameter values, and performance ratios with respect to fastest machine (m/f), the CRAY Y-MP/832 (m/cray), and the VAX-11/780 (v780/m) for reduced parameters 7-12. Numbers in column 'value' are given in nanoseconds.

Parameter 13: Logical operations

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	227.1	2.10	1.00	9.82
CRAY-2	320.4	2.97	1.41	6.96
CRAY X-MP/48	322.2	2.98	1.42	6.92
IBM 3090/200	108.0	1.00	0.48	20.64
MIPS/1000	370.4	3.43	1.63	6.02
Sun 4/260	1107.2	10.25	4.88	2.01
VAX 8600	548.7	5.08	2.42	4.06
VAX 3200	933.1	8.64	4.11	2.39
VAX-11/785 (fort)	1388.3	12.86	6.11	1.61
VAX-11/785 (f77)	1650.1	15.28	7.27	1.35
Vax-11/780	2229.6	20.64	9.82	1.00
Sun 3/260 (f)	4817.7	44.61	21.22	0.46
Sun 3/260	4275.4	39.59	18.83	0.52
IBM RT-PC/125	8789.7	81.39	38.70	0.25
Sun 3/50	10087.6	93.40	44.42	0.22

Parameter 14: Pipelining

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	40.6	1.00	1.00	24.85
CRAY-2	83.1	2.05	2.05	12.14
CRAY X-MP/48	70.9	1.75	1.75	14.23
IBM 3090/200	79.8	1.97	1.97	12.64
MIPS/1000	224.2	5.52	5.52	4.50
Sun 4/260	370.6	9.13	9.13	2.72
VAX 8600	405.8	10.00	10.00	2.49
VAX 3200	327.0	8.06	8.06	3.09
VAX-11/785 (fort)	418.6	10.31	10.31	2.41
VAX-11/785 (f77)	800.7	19.72	19.72	1.26
VAX-11/780	1008.8	24.85	24.85	1.00
Sun 3/260 (f)	406.1	10.00	10.00	2.48
Sun 3/260	410.6	10.11	10.11	2.46
IBM RT-PC/125	446.7	11.00	11.00	2.26
Sun 3/50	654.7	16.13	16.13	1.54

Parameter 15: Procedure calls

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	399.3	1.00	1.00	38.47
CRAY-2	440.1	1.10	1.10	34.90
CRAY X-MP/48	455.7	1.14	1.14	33.71
IBM 3090/200	889.1	2.23	2.23	17.28
MIPS/1000	632.5	1.59	1.59	24.28
Sun 4/260	696.4	1.75	1.75	22.06
VAX 8600	3655.0	9.16	9.16	4.20
VAX 3200	5482.5	13.73	13.73	2.80
VAX-11/785 (fort)	8765.3	21.95	21.95	1.75
VAX-11/785 (f77)	12565.3	31.47	31.47	1.22
Vax.780.f77	15359.3	38.47	38.47	1.00
Sun 3/260 (f)	3875.0	9.71	9.71	3.96
Sun 3/260	5059.5	12.67	12.67	3.04
IBM RT-PC/125	7294.1	18.27	18.27	2.11
Sun 3/50	7012.4	17.56	17.56	2.19

Parameter 16: Address computation

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	45.6	1.00	1.00	127.28
CRAY-2	141.1	3.10	3.10	41.13
CRAY X-MP/48	81.6	1.79	1.79	71.13
IBM 3090/200	274.2	6.01	6.01	21.17
MIPS/1000	786.2	17.24	17.24	7.38
Sun 4/260	680.6	14.93	14.93	8.53
VAX 8600	867.6	19.03	19.03	6.69
VAX 3200	1312.4	28.78	28.78	4.42
VAX-11/785 (fort)	2219.0	48.66	48.66	2.62
VAX-11/785 (f77)	1941.8	42.58	42.58	2.99
VAX-11/780	5804.0	127.28	127.28	1.00
Sun 3/260 (f)	1027.1	22.52	22.52	5.65
Sun 3/260	2092.5	45.89	45.89	2.77
IBM RT-PC/125	2502.4	54.88	54.88	2.32
Sun 3/50	4020.0	88.16	88.16	1.45

Parameter 17: Iteration (DO loops)

Machine	value	m/f	m/cray	v780/m
CRAY Y-MP/832	382.3	1.50	1.00	9.78
CRAY-2	453.8	1.79	1.19	8.24
CRAY X-MP/48	295.3	1.16	0.77	12.66
IBM 3090/200	254.3	1.00	0.67	14.70
MIPS/1000	706.1	2.78	1.85	5.30
Sun 4/260	1380.9	5.43	3.61	2.71
VAX 8600	905.3	3.56	2.37	4.13
VAX 3200	1483.2	5.83	3.88	2.52
VAX-11/785 (fort)	2675.5	10.52	7.00	1.40
VAX-11/785 (f77)	2773.2	10.91	7.26	1.35
VAX-11/780	3739.0	14.70	9.78	1.00
Sun 3/260 (F)	1072.3	4.22	2.81	3.49
Sun 3/260	905.1	3.56	2.37	4.13
IBM RT-PC/125	2628.1	10.34	6.88	1.42
Sun 3/50	2913.6	11.46	7.62	1.28

Table 2.17: Parameter values, and performance ratios with respect to fastest machine (m/f), the CRAY Y-MP/832 (m/cray), and the VAX-11/780 (v780/m) for reduced parameters 13-17. Numbers in column 'value' are given in nanoseconds.

3 Analysis of Benchmark Characteristics and Benchmark Performance Prediction

3.1. Summary

In this chapter 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.

3.2. Introduction

It is generally accepted that the best way to make an evaluation of the CPU performance of different architectures is to use benchmarks and compare the results. 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]. It is also known that benchmarking is inadequate to explain benchmark execution times, or predict them for programs not included in the suite [Worl84, Dong87]. This is because benchmarking completely ignores what programs do, and without this information, it is not possible to determine how a machine will behave while executing a program. In this chapter 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.

The fundamental idea behind our approach is that CPU performance evaluation requires a machine-independent model to characterize both the basic performance of a machine and the behavior of the program. At the same time, the model should provide a way of quantifying the individual contributions of the machine and the program to the total execution time. This provides a way of explaining and predicting benchmark results by using the machine and program characterizations. These two properties are what

distinguishes our approach from benchmarking and machine-dependent performance models.

In Chapter 2 we discussed our methodology and gave an in-depth presentation on machine characterization. In this chapter we focus on program characterization and execution time prediction. We explain how programs are characterized and present extensive statistics for a large set of programs, which include the Perfect Club and SPEC benchmarks. We discuss what these benchmarks measure and evaluate their effectiveness. Our results clearly show that some of the benchmarks are either too simple or do not behave as expected.

We can also use the dynamic statistics, information obtained at run-time, to define a metric to measure similarity between the benchmarks' dynamic statistics. This similarity distance is related to the performance of the benchmarks when these are normalized with respect to a particular machine, i.e., programs having a small distance under the metric tend to exhibit similar relative performance across many machines.

The structure of the chapter is as follows. In Section 3.3 we review the previous approaches to execution time prediction, and in Section 3.4 we explain the main concepts and how the program analyzer works. In Section 3.5 we give the dynamic distributions and statistics for a large suite of benchmarks which includes the SPEC Fortran and Perfect Club benchmarks and present a metric for program similarity. Section 3.6 deals with execution predictions and discusses their accuracy when compared against real measurements. Lastly, Section 3.7 ends the chapter with a summary and some of our conclusions. The presentation is self-contained and does not assume familiarity with the previous chapter.

3.3. Previous Work

Several people 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 two proposals that are somewhat related to our concept of an abstract machine model and the use of static and dynamic program statistics.

The application of low level models to predict execution times has been used to compare different implementations of the same architecture. A comparison study of the IBM 370/168 and the Amdahl 470 V/6 was carried out by obtaining execution times for machine instructions under different conditions and predicting execution time of executable code [Peut77]. The main difficulty lies in obtaining accurate values from instruction timing formulas [MacD84]. A major limitation of this approach is that it can only be used on machines that share the same instruction set. In contrast, our abstract machine model applies to all uniprocessors independent of their instruction sets. Because of this machine-independent property, our model is easier to use and validate. Furthermore, the experimental nature of our approach simplifies enormously the performance characterization of machines.

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] a new technique based on the interval structure and control

dependence of the program is proposed to gather profiling information. The *interval structure* identifies all the loops of a possibly unstructured program and helps in selecting the minimum number of basic blocks that need to be instrumented in order to get dynamic statistics for the whole program. Profiling information is obtained only for each loop and not for each basic blocks, as in our approach; dynamic statistics for basic blocks, are derived from the loop statistics and the control dependence graph. Since the overhead added by the basic block profiling is quite small (around 15%) and programs only need to be profiled once, it is not clear that the benefits of 'loop' profiling justify its use.

3.4. The Program Analyzer

The analysis of programs consists of two phases: the *static analysis* and the *dynamic analysis*. These two phases are similar to what most execution profilers do [Powe83, Grah82]; the only difference being that in our case, instead of counting the execution of procedures or statements, our *program analyzer* identifies Fortran abstract operations. We make the static analysis by using the front end of a Fortran compiler, and during this phase the source code is instrumented, by including counters in each basic block, to produce the dynamic statistics at run time. Running the instrumented program corresponds to the dynamic phase.

3.4.1. Measuring Dynamic Statistics

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 $\mu_A=<\mu_1,\mu_2,\cdots,\mu_j>$ 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

$$C_i = \sum_{j=1}^{m} d_{i,j}. \tag{3.2}$$

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

As expected, computing the dynamic statistics increases the execution time of the program. In addition, there is extra code to initialize the counters and to write them to a file at the end of the run. This instrumented program only needs to be executed once to obtain all the statistics and then use these to get predictions for many different machines. This means that producing execution estimates for M machines and N programs requires only characterizing M machines and analyzing N programs. The same process in benchmarking takes $N \cdot M$ runs. The overhead involved is a function of the work inside a basic block compared to the code inserted for counting, as each block requires one counter. We have found that this overhead is less than 15 percent.

3.4.2. Precision and Time Dependent Execution

In the previous paragraph, we said that the instrumented version of the program has to be executed only once. This is not always true; in some cases the program may have different dynamic statistics depending on the machine used. The sequence of

instructions executed, or in our case abstract operations, is a function of the program and the input data. This *execution history*, however, sometimes changes from one machine to another. The source of this change depends on whether the execution history is precision dependent and/or time dependent. An execution history is *precision dependent*, if in two machines the emulation of an abstract operation gives different results for the same input, and this difference affects the flow of execution. An execution history is *time dependent*, if at some point in the execution the control flow depends on the time interval between two events.

Programs that solve a system of equations by iterating on an approximate solution may be precision dependent. Our *machine characterizer* is an example of a time dependent program. Here, the duration of an experiment depends on the quality of the measuring tools and the speed of the machine. For example, a Sun 3/50 is much slower than a Sparcstation I. The machine characterizer, however, runs each test for approximately 2 seconds (plus or minus 10 percent) in both machines to guarantee a good statistical sample. Thus, the same test, in both machines, executes for different numbers of iterations. Instrumented versions of precision and time dependent programs need to be executed in all machines that may produce a particular execution history[1]. However, the whole idea behind performance evaluation is to make a consistent and fair comparison between machines, and, to satisfy this, the set of benchmarks used should produce the same execution histories independent of where it runs. All programs we analyze in this chapter have execution histories that are precision and time independent[2].

3.5. Program Characterization

Benchmarks are tools that performance analysts use to measure and compare the performance of different computer systems. The strengths and shortcomings of a machine have a direct effect on the execution times of the benchmarks. The analysis of these results provides answers, in terms of the model (basic operations), about why the performance of one benchmark is better or worse than the majority of the benchmarks. Therefore, it is essential for us to know what are the operations that each benchmark tests and how similar or different these are. In our model this information is provided by the dynamic statistics of the program.

In the rest of this section we present the set of benchmarks used in this study and give a good characterization of their execution behavior by using their dynamic distributions.

---

[1] The situation is more complex for time-dependent programs; the execution histories for the original and instrumented programs differ, even when both are run in the same machine. Every attempt to characterize a time-dependent program by instrumenting it changes the execution history of the program. In this situation the static statistics correspond to the original program, while the dynamic counters belong to the instrumented version. We can solve this problem by making a static analysis of the instrumented program. Now we can predict the execution time for the instrumented program, but not for the original program.

[2] The original version of TRACK of the Perfect Club benchmarks had several execution histories due to an inconsistency in the passing of constant parameters. The version of the program used in this chapter does not have this problem.

3.5.1. Benchmark Suite

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 3.1 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. We use these in §3.5.8 to show how benchmarks can be improved, in terms of what they measure, by using their dynamic statistics.

3.5.1.1. Floating-Point Precision in Benchmarks Suites

In Fortran there are two ways of specifying the precision of a floating-point variable: one is with an absolute precision declaration and the other is with a relative precision declaration. An absolute precision declaration is one which includes in the declaration the number of bytes that should be allocated to the variable. In contrast, on a relative precision declaration the number of bytes is left unspecified and instead the words 'single' and 'double' are used. However, the interpretation of these declarative directives by the compiler is machine dependent. Normally, the meaning of single and double precision on workstation is 32 and 64 bits respectively, while on supercomputers it is 64 and 128 bits.

There are arguments for using either one of the two types of declarations. It is generally accepted, however, that benchmarks should use absolute declarations, because the goal of a benchmark should be to test different machines under controlled conditions. Changing the number the precision of the variables affects in many cases the execution time by altering the amount of 'work' needed to execute the program. Unfortunately, most of the benchmarks still use relative declarations, as this can be seen in table 3.1. The second column of the table shows that of the six Fortran SPEC benchmarks only three of them use absolute declarations, while at the same time none of the Perfect benchmark do.

3.5.1.2. The SPEC Benchmark Suite

The Systems Performance Evaluation Cooperative (SPEC) was formed in 1989 by several machine manufacturers with the specific purpose of creating a methodology for evaluating different computer system that could be accepted by users and computer manufacturers. 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 reports obtained using the SPEC suite. The rapid increase in the number of members of SPEC clearly shows how successful has been their efforts.

The SPEC suite consists of six Fortran and four C programs taken from the scientific and systems domains [SPEC89, SPEC90]. The main performance metrics proposed by SPEC for uniprocessors are the SPECratio and the SPECmark. The SPECratio is the normalization of benchmark results by taking the ratio between the execution time on a VAX-11/780 with that of the machine being measured. The SPECmark is the overall

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 3.1: 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.

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. It is a large

kernel extracted from the original program.

- *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. The input to the program is the number of atoms. The amount of computation to be done is proportional to the fourth power of the number of atoms.
- *TOMCATV* is a very small (less than 140 lines) highly vectorizable mesh generation program. It is a double precision 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.

3.5.1.3. The Perfect Club Suite

Supercomputer users in an attempt to identify those problems and their respective algorithms that can be efficiently executed in conventional and new architectures have assembled a set of thirteen scientific programs to measure machine performance [Cybe90]. In contrast with the SPEC approach, where performance is measured relative to an arbitrary machine, performance in the Perfect Club approach is defined as the number of MFLOPS (millions of floating-point operations per second) that a machine can execute. This metric is computed by taking the harmonic mean of the MFLOPS rates for individual programs. It is important to note that the number of FLOPS in a program is given in terms of the 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. This program can be used for steady and unsteady flows, but only for inviscid (zero viscosity) flows.
- *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 that consists of solving the dynamical equations of a Boussinesq fluid layer required to study the chaotic behavior of free-slip Rayleigh-Benard convection.
- *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 & 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 to solve for the displacements and stresses, along with the velocities and accelerations at each time step.

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.

3.5.1.4. 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].

3.5.2. Normalized Dynamic Distributions

The normalized dynamic statistics for all benchmarks, including the seven data sets for *SPICE2G6*, are presented in tables 3.16–3.25 in Appendix 3.A. For each program[3] we give the fraction, with respect to the total, that each abstract operation is executed. Those that are executed less 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. Although the normalized distribution is the basis for computing execution predictions, it contains too much detail and thus makes it difficult to abstract the important characteristics of the programs. For this reason, we focus on some metrics that are more intuitive. These are represented in figures 3.1–3.6 and 3.8–3.9. The precise numbers depicted in the figures are given in Appendix 3.B in tables 3.26–3.32. The most interesting results are discussed in the following subsections.

3.5.3. Basic Block and Statement Statistics

Figure 3.1 shows the distribution of statements classified into five different types. The values used in the figures are also reported in Appendix 3.B (tables 3.26–3.28). These five groups are: assignments, procedure calls, IF statements, branches, and DO loop iterations. On this and similar figures we cluster the benchmarks according to how similar are their respective 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 distribution differ significantly from those of other benchmarks in the suite. In particular, programs *QCD*, *MDG*, and *BDNA* execute a larger fraction of procedure calls; on other programs the fraction is much smaller. A similar observation can be made in the case of IF statements for programs *QCD*, *MDG*, and *TRACK*. And with respect to branches we see that benchmark *TRACK* executes a substantially large number of branches. As we mentioned, the Perfect Club methodology defines performance based only in the number of floating-point operations executed per unit of time. But the statistics in figure 3.1 show that some of the programs execute a significant fraction of operations that are not floating-point. Any performance measurement, even for scientific codes, should take into account that programs execute a variety of operations, not only floating-point. In §3.5.6 we discuss in more detail the proportion of the total execution time that programs spend executing floating-point operations.

The SPEC and Perfect suites have similar distributions. *SPICE2G6* using model *GREYCODE* and *DODUC* are two programs which execute a substantial fraction of IF statements and branches. In *GREYCODE*, 35% of all its statements are branches, and *DODUC* has a significant number of IF statements. The distribution of statements also provides additional data. For example, 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. This means that these benchmarks have loops containing a large number of assignments. This is corroborated by the code; the most important basic block in *FPPPP* contains more than 500 assignments.

---

[3] In the rest of the chapter 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.

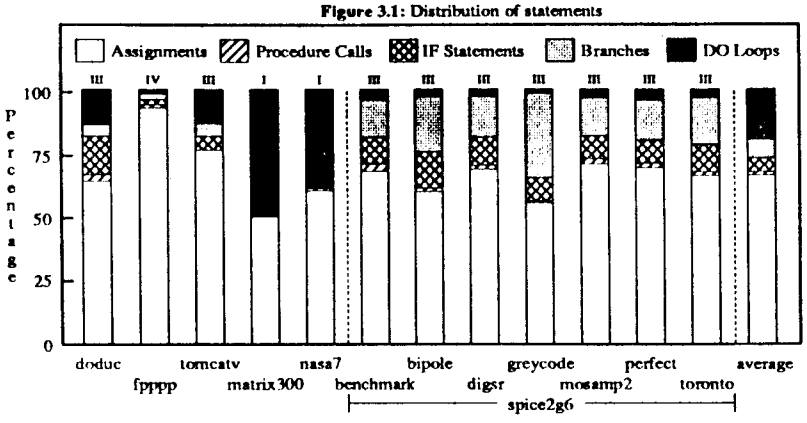


Figure 3.1: Distribution of statements

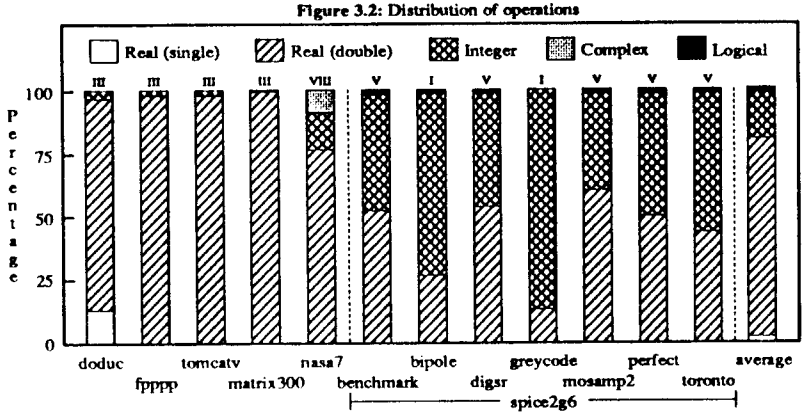
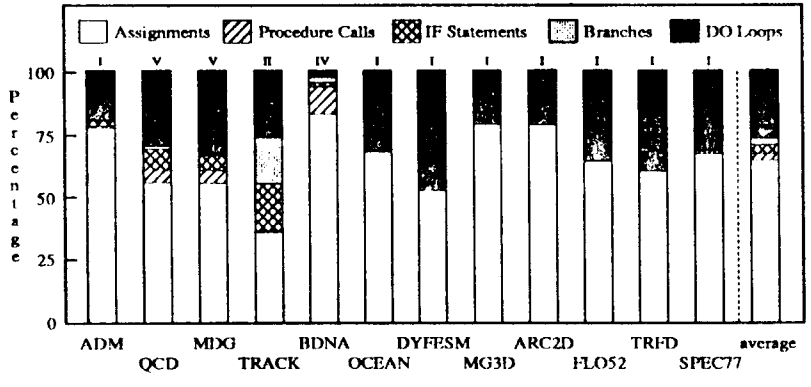
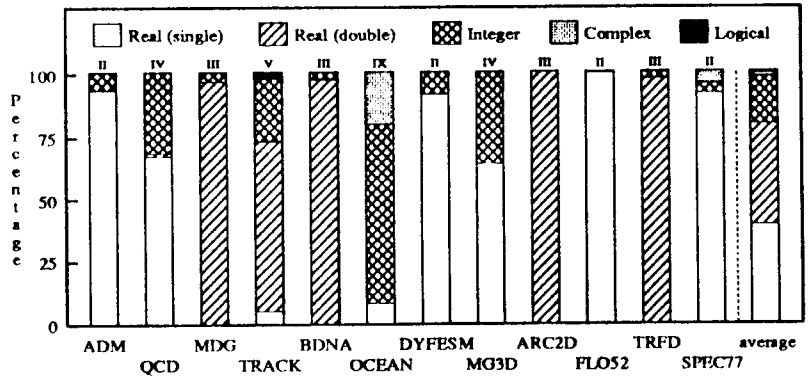
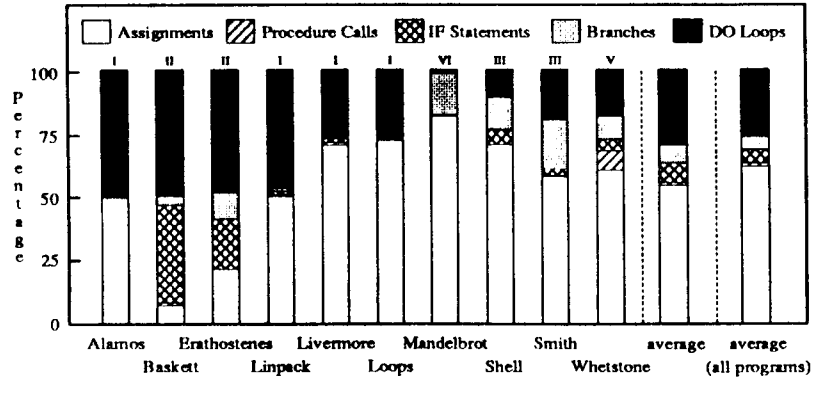
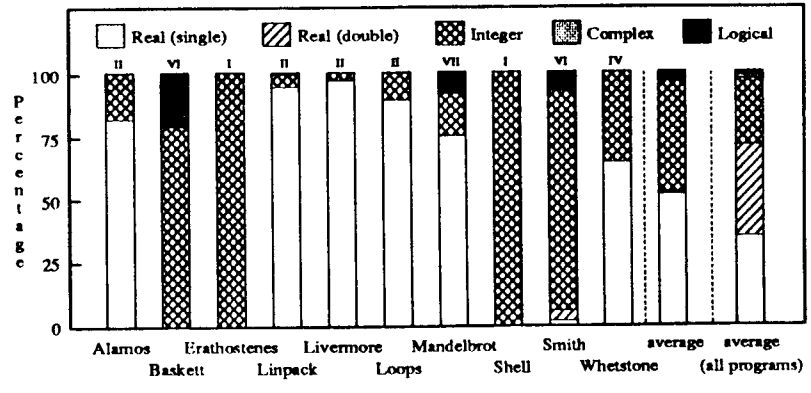


Figure 3.2: Distribution of operations









Figures 3.1 and 3.2: Distribution of statements, 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.

In table 3.2 we give the average distributions of statements for the SPEC, Perfect Club, and small benchmarks. We also indicate the average of all programs. These numbers correspond to the average dynamic distributions shown in figure 3.1.

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

3.5.4. Arithmetic and Logical Operations

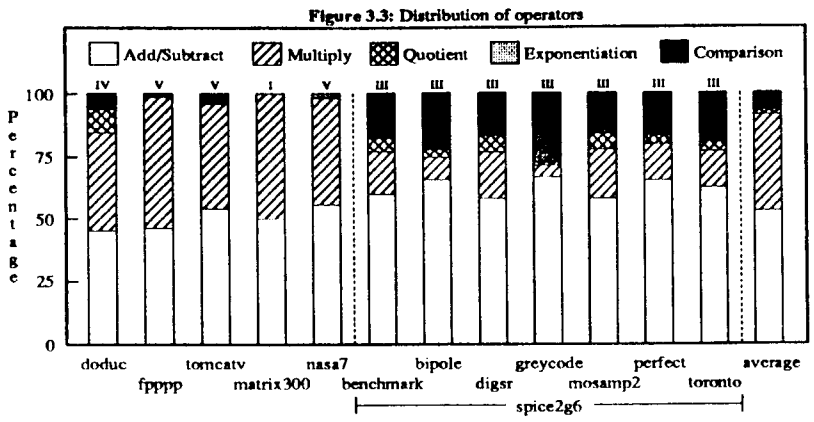
The distribution of arithmetic and logical operations represent another important statistic. Figures 3.2 and 3.3 depict the distribution of operations according to their type and what they compute. The values used in these figures are reported in tables 3.29–3.31 (Appendix 3.B). As it is clear from the graphs, most operators executed by these programs belong to one or two data types. 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 support even more our argument that scientific programs differ significantly from each other and that focusing in the number of MFLOPS is an oversimplification of their dynamic execution.

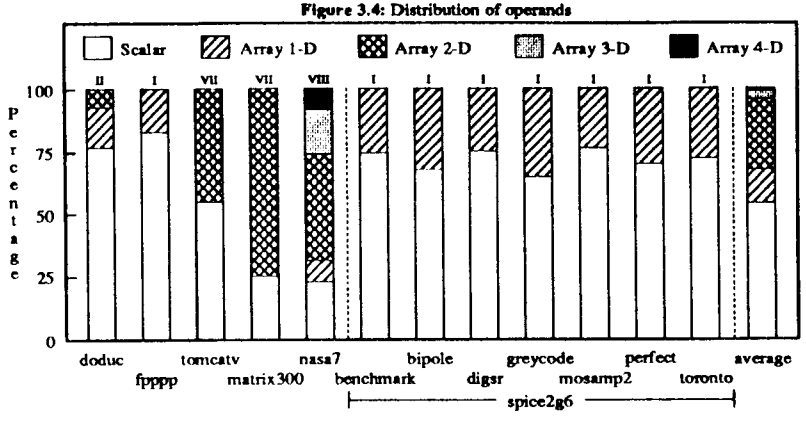
A similar classification can be obtained for the SPEC and the other benchmarks. Of all programs, *TRACK*, *OCEAN* and *NASA7* show more complex distributions for the type of operators executed.

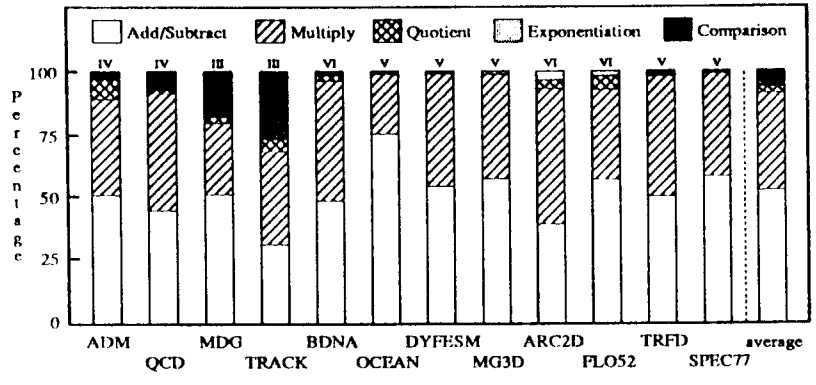
With respect to the distribution of arithmetic operators, figure 3.3 shows that the largest fraction correspond to addition and subtraction, followed by multiplication. Other operations like division, exponentiation and comparison amount for a small fraction. This is clearly seen in the distribution of the Perfect and SPEC benchmarks. The numbers for the average dynamic distributions are included in table 3.3.

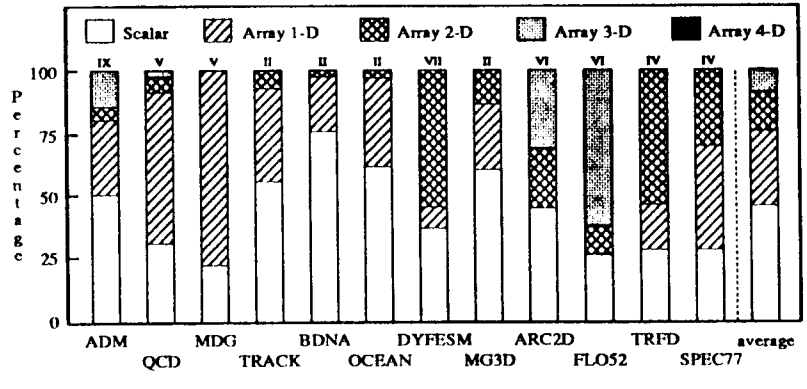
3.5.5. Distribution of Array and Simple Variables

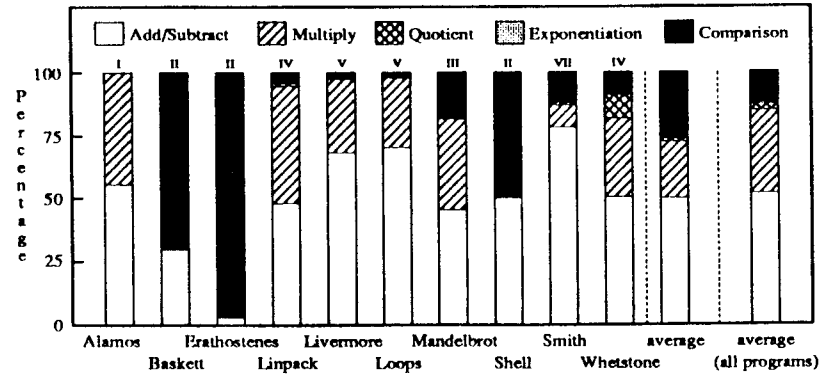
Another source of information about benchmarks is the ratio of simple to array variables and the distribution of the number of dimensions for array references. These statistics are shown in figure 3.4. We can see that for most of the Perfect benchmarks the proportion of array references is larger than 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 show that on most programs a large portion of the references are to 1-dimensional arrays with a smaller fraction in the case of two

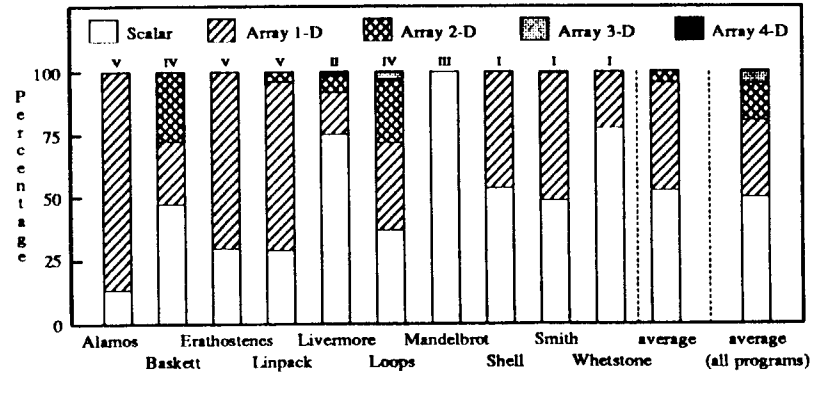












Figures 3.3 and 3.4: Distribution of operators, and distribution of operands. Bar *Loops* represents only the 24 computational kernels of benchmark *Livermore*, while ignoring the rest of the computation. Each bar is labeled with a roman numeral identifying those benchmarks with similar distributions. 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 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 3.3: Average dynamic distributions of arithmetic and logical operations for each of the suites and for all benchmarks.

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. If $X(i_1, i_2, \cdots, i_n)$ is an n-dimensional array reference, then its address ($ADDR$) is

$$ADDR[X(i_1, i_2, \cdots, i_n)] = ADDR[X(0, 0, \cdots, 0)] + Offset[X(i_1, i_2, \cdots, i_n)], \quad (3.3)$$

where

$$Offset[X(i_1, i_2, \cdots, i_n)] = B_{elem}((\cdots((i_n \cdot d_{n-1} + i_{n-1})\, d_{n-2} + i_{n-3})\cdots)\, d_1 + i_1). \quad (3.4)$$

In the equation, $\{d_1, d_2, \cdots, d_n\}$ represents the set of dimensions and B_{elem} the number of bytes per element. Note that element $X(0, 0, \cdots, 0)$ is not a valid Fortran array reference, and it is used only to reduce the amount of computation. 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.

The results in figure 3.4 show that the average number of dimensions in an array reference for the Perfect and SPEC benchmarks is 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).

With optimization enabled, if the array reference is inside a loop, then the amount of computation can be reduced to a single add. This requires detecting how the offset changes between iterations; instead of computing it from the indices, a register is used to store the offset for the duration of the loop, which is updated on every iteration by adding a constant increment. Computing the increment and the initial offset is done before

entering the loop. This represents an important optimization on scientific programs, especially when there are several array references in the loop that have the same offset or the same increment.

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

3.5.6. Execution Time Distribution

Scientific programs generally execute a large number of floating-point operations on large amounts of data stored in multidimensional arrays. How much of the total execution time corresponds to floating-point computation is clearly a function of the program and the machine. Figures 3.5 and 3.6 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 3.5 and 3.6. 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 on *GREY-CODE*. Here less than 10% of the total time on the RS/6000 530 is spent doing floating-point arithmetic. The numerical values for the benchmark suites and for all programs are given in table 3.5.

From the figures, it is evident that the time distributions of the RS/6000 530 and the CRAY Y-MP are very different even when all programs were 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 appear counterintuitive, as we would expect the fraction on the CRAY Y-MP to be lower given that this is a machine especially designed to execute floating-point intensive programs. The reason is that some of the programs execute in double precision, which on the CRAY corresponds to 128-bit quantities. Moreover, double precision arithmetic operations are not supported directly on hardware but are implemented by software using the single precision hardware. Hence, the amount of time spend on floating-point arithmetic increases. The effect is seen clearly in programs: *DODUC*, *SPICE2G6*, *MDG*, *TRACK*, *BDNA*, *ARC2D*, and *TRFD*. Using our program statistics, however, we can easily compute the average distribution when all programs execute using 64-bit quantities. All we need to do is to set a flag when we analyze the

Figure 3.5: Distribution of execution time (IBM RS/6000 530)

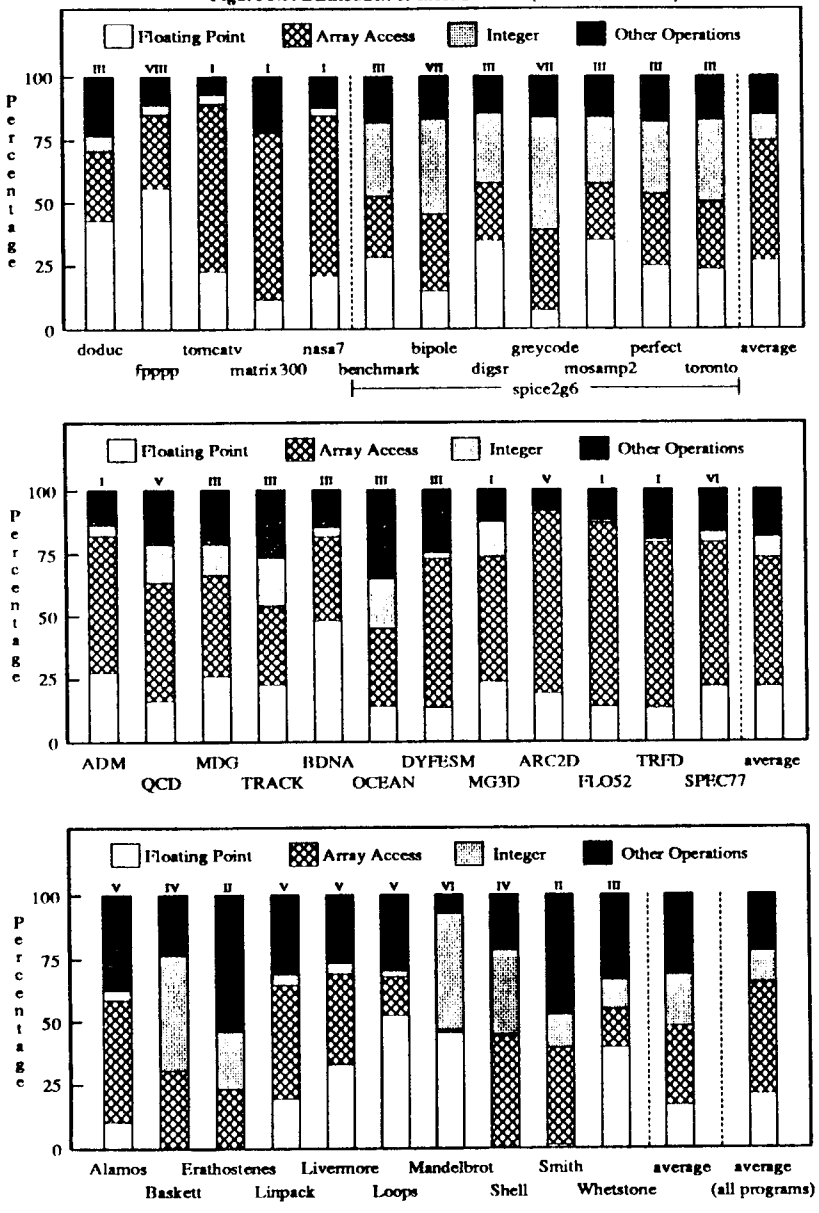
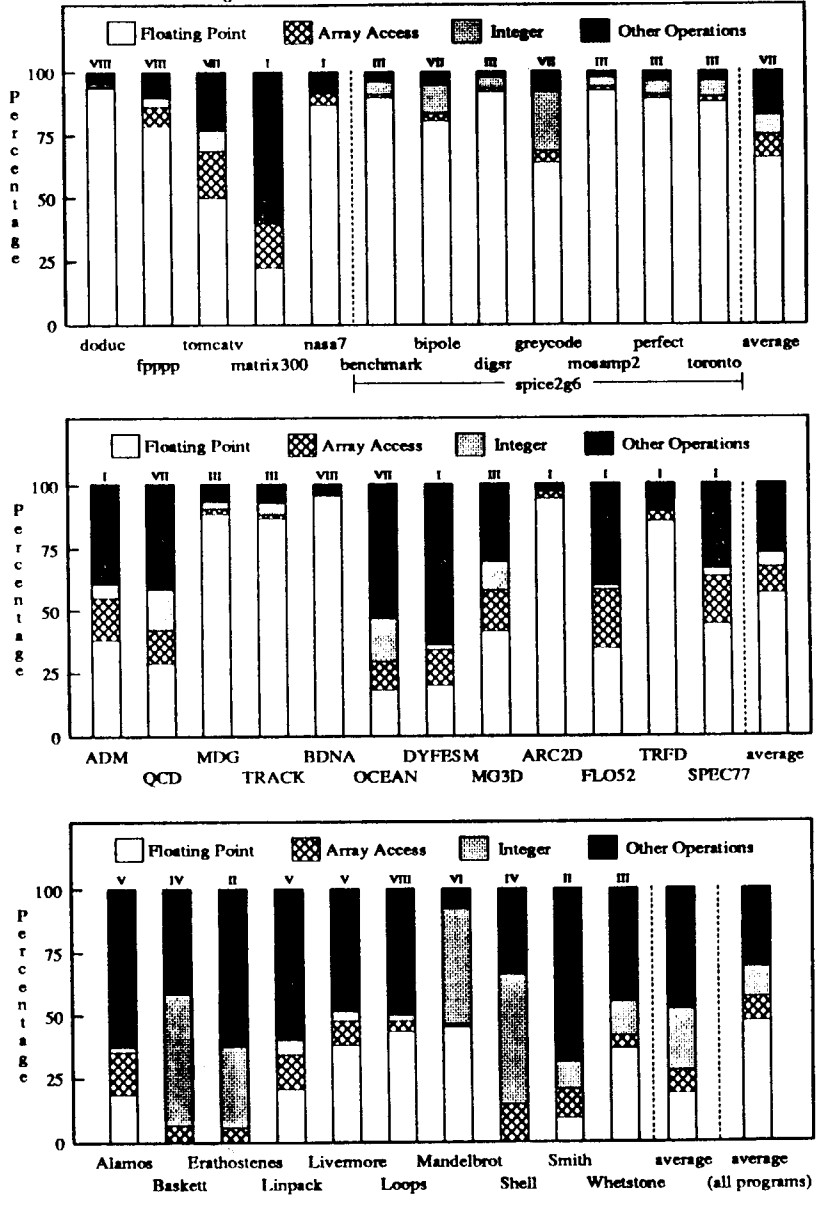


Figure 3.6: Distribution of execution time (CRAY Y-MP/832)



Figures 3.5 and 3.6: Distribution of execution time for the IBM RS/6000 530 and the CRAY Y-MP/832. The figures show the fraction of time corresponding to floating point computation. Bar *Loops* represents only the 24 computational kernels of benchmark *Livermore*, while ignoring the rest of the computation. 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 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 3.5: 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.

program to produce a new set of statistics. We then can use these to compute new execution time predictions. In this case the fraction of time represented by floating-point operations on the CRAY Y-MP decreases to 29%.

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 this is a scalar code completely dominated by floating-point computation. The reader, however, should recall 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 of the computation 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 optimizations.

In figure 3.7 we show the overall average time distribution for a large number of 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. Furthermore, all programs are assumed to execute in scalar mode and without optimization. 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 integers operations exhibit less variation, ranging from 6 to 13%.

3.5.7. Dynamic Distribution of Basic Blocks

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 3.8 show, many of the programs in the Perfect and SPEC suites have very simple execution patterns, where only a small number of basic

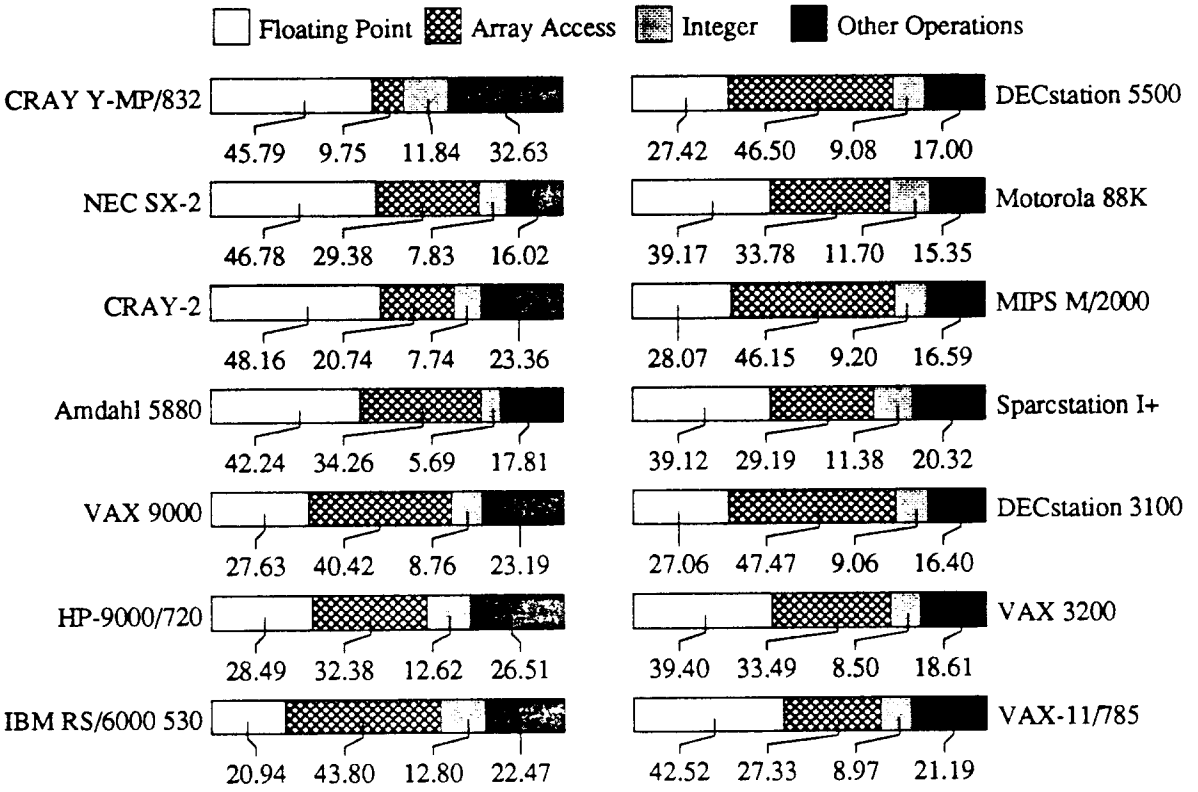
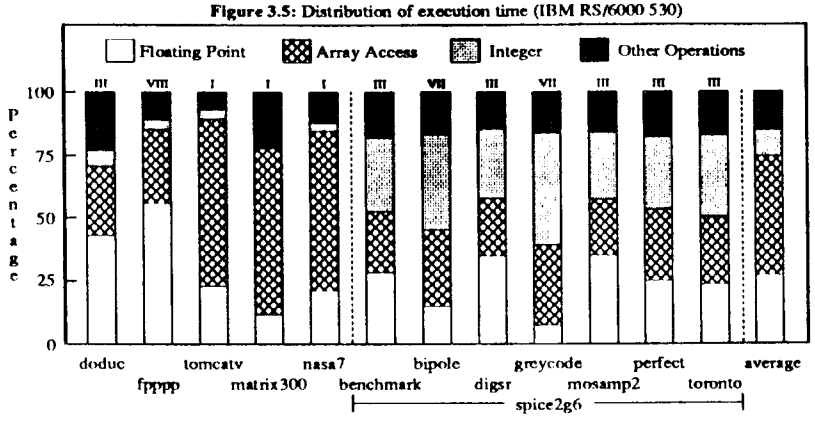


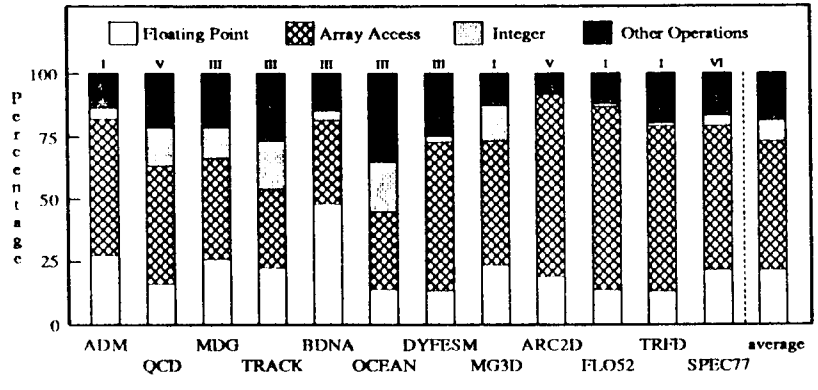
Figure 3.7: Average time distributions. The distributions are computed over all programs. Of the seven models for *spice2g6*, only *greycode* and *perfect* are considered in the computation of the averages.

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. How this characteristic of *MATRIX300* affects the performance of machines is discussed in the next paragraphs. On the average five blocks account for 55.45% and 71.85% of the total time on the Perfect and SPEC benchmarks.

The importance of measuring the complexity of benchmarks resides in identifying those programs that may give 'unstable' results across different machines or even compiler releases. Programs with unstable results are those where the execution time is determined by a small number of operations and/or basic blocks.

Figure 3.5: Distribution of execution time (IBM RS/6000 530)





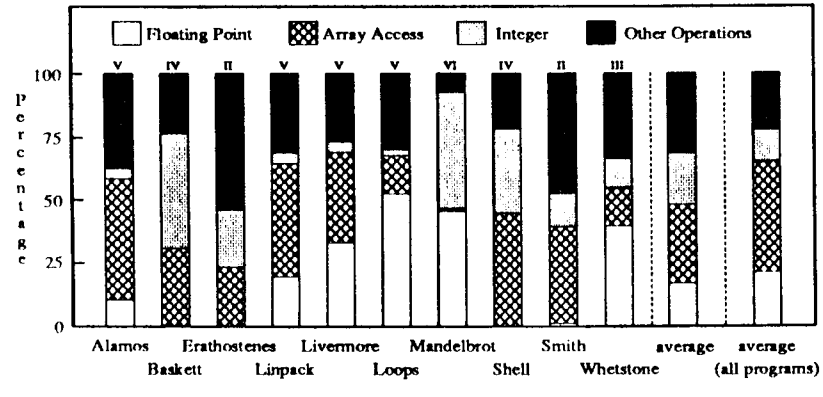
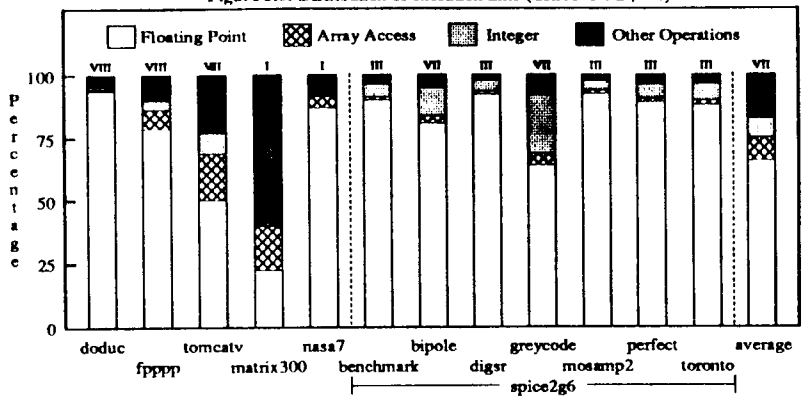
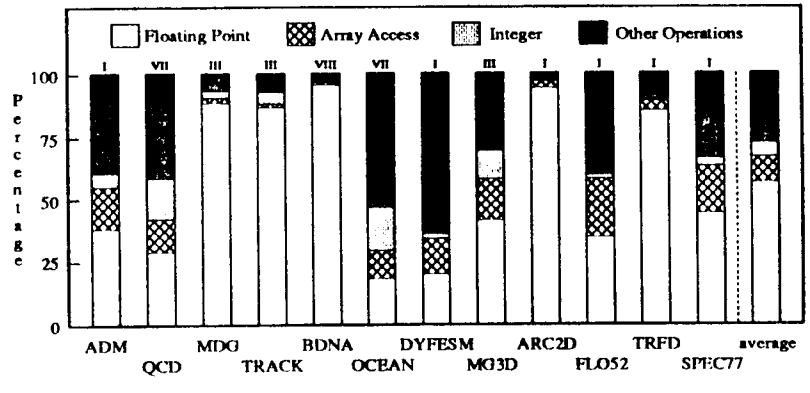
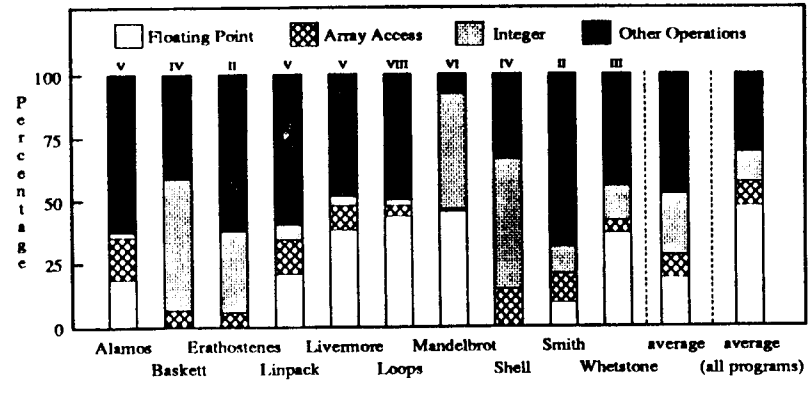


Figure 3.6: Distribution of execution time (CRAY Y-MP/832)







Figures 3.5 and 3.6: Distribution of execution time for the IBM RS/6000 530 and the CRAY Y-MP/832. The figures show the fraction of time corresponding to floating point computation. Bar *Loops* represents only the 24 computational kernels of benchmark *Livermore*, while ignoring the rest of the computation. 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 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 3.6: Average dynamic distributions of basic blocks for each of the suites and for all benchmarks.

3.5.7.1. Quantifying Benchmark Instability Using Skewness

We can quantify the notion of instability by characterizing the shape of the basic block distribution using the concept of skewness. *Skewness* is a measure of how asymmetric is a distribution; more specifically, it is a measure of how far, either to the left or the right, is the bulk of the distribution with respect to the mean. The coefficient of skewness is one of the statistics which measure the amount of departure from the normal distribution. However, this coefficient is not very useful in our situation, because we already know that neither the distribution nor the ordered distribution of basic blocks resemble a normal distribution. When we use the coefficient of skewness in the ordered distribution, it detects instability only on those programs which have a large number of basic blocks, but fails to detect it on small programs, even when it is clear that their distributions also are very skewed.

The ordered distributions of basic blocks have a lot of resemblance to the geometric distribution, which is a discrete, memoryless, monotonic decreasing function. In the previous section we said that an unstable benchmark is one where most of the execution time is localized in a small number of blocks. This means that a good measure of the skewness in an ordered distribution is the inverse of the expected value of the distribution. This is the case because, given two monotonically decreasing functions, the one which decays faster is the one whose inverse of its expectation is larger.

Table 3.7 gives the amount of skewness for all programs. The results show that *MATRIX300*, *MANDELBROT*, and *LINPACK* are the ones with the largest skewness.

3.5.7.2. Optimization and *MATRIX300*

The importance of detecting unstable programs is that it allows us to identify programs that have the potential of being artificially 'optimized' when they are used as standard benchmarks. Benchmark *MATRIX300* is a clear example of this situation; not only its amount of skewness is very high, but recent SPEC results on this program put in question its effectiveness as a benchmark. For example, on [SPEC91a] the SPECratio of the CDC 4330 on *MATRIX300* was reported as 15.7 with an overall SPECmark of 18.5, but on [SPEC91b] the SPECratio and SPECmark jumped to 63.9 and 22.4. A similar situation exists on 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

program	Skewness
01 Matrix300	0.983
02 Mandelbrot	0.790
03 Linpack	0.637
04 BDNA	0.567
05 Tomcatv	0.535
06 Baskett	0.466
07 Erathostenes	0.452
08 TRFD	0.405
09 Shell	0.385
10 DYFESM	0.250
11 Whetstone	0.229
12 Fpppp	0.201
13 OCEAN	0.171
14 Alamos	0.162

program	Skewness
15 Nasa7	0.155
16 MDG	0.145
17 Smith	0.136
18 QCD	0.133
19 Livermore	0.132
20 MG3D	0.108
21 Spice2g6	0.084
22 FLO52	0.078
23 ARC2D	0.073
24 TRACK	0.073
25 SPEC77	0.065
26 ADM	0.060
27 Doduc	0.049

Table 3.7: Skewness of ordered basic block distribution for the SPEC, Perfect and Small benchmarks. In an ordered distribution, the skewness is proportional to the inverse of the mean of the distribution.

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 algorithm. A *blocking algorithm* is one in which the matrices are divided into sub-blocks and the operations are is done on sub-blocks. Doing this is an effective way of increasing the locality of the algorithm, which results in a decrease in the number of cache and TLB misses. *MATRIX300* uses matrices of size 300x300 in order to test the memory system by forcing a substantial number of misses, because current data caches are not big enough to hold a single whole matrix. By changing the matrix multiply algorithm from non-blocking to blocking, however, the purpose of the benchmark is defeated[4].

3.5.7.3. 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 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 the kernels do not touch more than 2000 floating-point numbers, all the data sits comfortably in most caches. Thus, after the first iteration the memory system is not

[4] 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.

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. Benchmark *MATRIX300* is dominated by a single statement that the IBM Fortran compiler can optimize by decomposing it into a single multiply-add instruction. However, the SPECratio of the IBM RS/6000 530 on this program is lower than the overall geometric mean (SPECmark). In contrast, the SPECratio on program *TOMCATV* is 2.6 times larger than the SPECmark, even when the main 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; there is very little re-use of registers. Therefore, it is important that the behavior of benchmarks with respect to the memory system be studied in more detail. 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.

3.5.8. Distribution of Abstract Operations

In the last section we argued that some large and time consuming benchmarks spend most of their time in just a few basic blocks. Other information that interests us is the distribution of operations, or (in our model) abstract operations, executed by a benchmark, This information is useful because performance results using benchmarks which execute only a small number of different operations give a very limited view of the overall performance of a system.

Figure 3.9 shows the cumulative distribution for the different suites. In the figures we also show the relevant average distributions. Each bar indicates at the bottom the number of different abstract operations executed by the benchmark. The results show that most programs execute a small number of different operations and that there is small variation between them. The distribution for *MATRIX300* shows the expected profile of a benchmark that spends all of its execution in a small number of blocks. The averages for the three suites and for all programs are presented in table 3.8.

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 3.8: Average dynamic distributions of the number of abstract operations executed for each of the suites and for all benchmarks.

We can also compute the skewness of the ordered distribution of abstract operations in the same way as we did with basic blocks. In a similar way the amount of skewness is obtained from the inverse of the expected value of the distribution. The results are

shown in table 3.9. 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 abstract operations. People who use benchmarks normally are unaware that important differences exist in the complexity of different benchmarks, as it is illustrated by the distribution of *MATRIX300* and *DODUC*, and they assume that all programs in a suite are equally robust in terms of how they execute and what they measure. In §3.6.4 we use the skewness results of both the basic block and abstract operations to try to explain the amount of error in our predictions and the variability of normalized benchmark results.

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

program	Skewness
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 3.9: Skewness of ordered abstract operation distribution for the SPEC, Perfect and Small benchmarks. In an ordered distribution, the skewness is proportional to the inverse of the expected magnitude of the distribution.

3.5.8.1. Characterizing the Ordered Distribution of Abstract Operations

It has been argued that for an average program the distribution of the most executed operations (blocks) exhibits the memoryless property [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. This sequence corresponds to the geometric distribution. 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[5].

In figure 3.10 we show the fitted and actual average distributions for each suite and for all programs. The results show that the fitted distributions match well the actual distributions. The only significant discrepancy can be seen in the SPEC suite when n, the number of abstract operations, is small. This can be explained by the small number of programs in the suite. Furthermore, two of programs, *DODUC* and *MATRIX300*, have

[5] The n-th residual is given by $(1-\alpha)^n$. Thus, the cumulative distribution at point n is one minus the n-th residual.

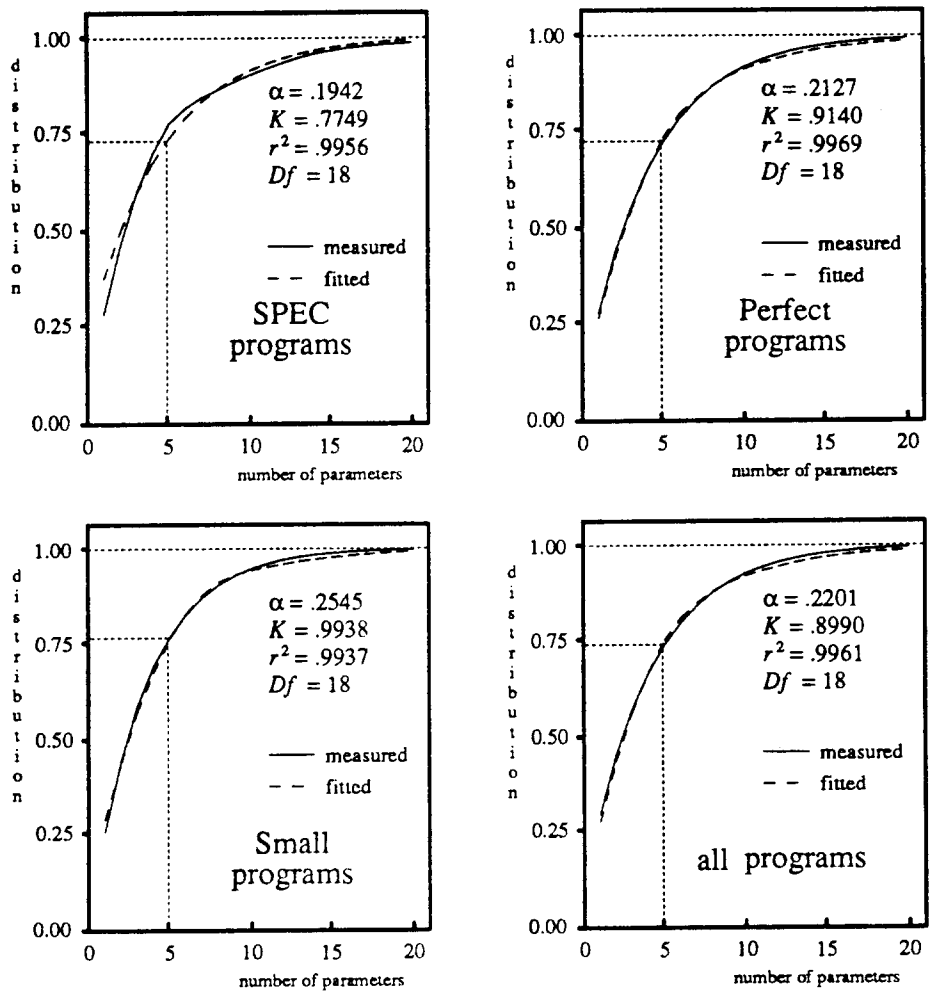


Figure 3.10: Fitted and actual cumulative distributions as a function of the n most important abstract operations executed by each benchmark. Equation $1 - K(1-\alpha)^n$ is used to fit the actual distributions. In addition to α and K, each graph indicates the values of the coefficient of correlation and the number of degrees of freedom. All coefficients of correlation are significant at the .9995 level.

distributions corresponding to the most and least uniform of all benchmarks. This is conspicuous in the results given in figure 3.9. Nevertheless, figure 3.10 clearly shows that, on the average, three operations account for 55-60% of all operations and five operations for almost 75%. This emphasizes that most programs consist of a small number of different operations executed many times. These operations, however, are not the same in all benchmarks.

3.5.9. The *SPICE2G6* Benchmark

Now we discuss in more detail the differences between the seven data sets used on the *SPICE2G6* benchmark. This will give us an opportunity to show how the characterization of a benchmark can be used to assess its effectiveness. *SPICE2G6* is normally considered, for performance purposes, a good example of a large CPU-bound scalar double precision floating-point benchmark. It is supposed to execute a small fraction of complex arithmetic. The amount of vectorization in the program is negligible and given its large size, it is a good test to stress the instruction and data caches. (The code and data sizes on a VAX-11/785 running ULTRIX are 325 Kbytes and 8 Mbytes respectively). 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 selected mainly because of its large execution time, but we shall see that its dynamic statistics indicate that its execution behavior is not typical, nor does it measures, what *SPICE2G6* is supposed to measure.

Table 3.26 (see Appendix 3.B) gives the general statistics for the seven data models of *SPICE2G6*. The results show that the number of abstract operations executed by *GREYCODE* (2.005×10^{10}) is almost two orders of magnitude larger than the maximum on any of the other models (3.184×10^{8}). However, all these operations touch only 33% of all basic blocks. In contrast, the number of basic blocks touch by *BENCHMARK* is 52%. Other 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 these. 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 3.2. 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 percent of floating-point operations is never less than 26% and it reaches 60% on 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. 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 show that *GREYCODE* is not an adequate benchmark for testing scalar double precision arithmetic, and that most of the execution is dominated by a small number of small blocks. Much better input models for *SPICE2G6* are *BENCHMARK*, *DIGSR*, or *PERFECT*.

3.5.10. Measuring Similarity Between Benchmarks

It is important that benchmark suites cover a wide spectrum of potential applications and that the users of these suites know which programs are similar. However, it is not obvious how to define benchmark similarity, nor which characteristics should determine its value. Many programs are similar with respect to some operations, but differ significantly in others. Moreover, programs sometimes are similar in terms of their static statistics, but at the same time have different dynamic behavior.

Our approach consists of defining a distance over a set of 13 reduced parameters. These parameters correspond to important (from the performance point of view) hardware and software aspects of the machines. The value of each reduced parameter is computed from the dynamic statistics of the program, and it characterizes one aspect of the machine that the program is exercising. Table 3.10 shows the list of the reduced parameters. These parameters are not identical to those used in Chapter 2. We decided to change them in order to achieve a better match, which was lacking in the previous set, between the reduced parameters and those aspects of machine architecture which are to some extent independent. Most of the parameters correspond to floating-point operations. This is normal given the characteristics of the programs and the programming language. Others are integer arithmetic, logical arithmetic, procedure calls, memory bandwidth, and intrinsic functions. Integer and floating-point division are assigned to a single parameter. Parameters 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 3.10: 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 distribution of all programs and the relative execution time of the individual parameters. 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

$$d(\mathbf{A}, \mathbf{B}) = \sum_{i=1}^{n} W_i (A_i - B_i)^2 \tag{3.5}$$

where W_i is the value of parameter i in the average distribution.

In table 3.11 we give the list of the 25 pairs of programs having the smallest and largest distances. These include all programs, except some input models of *SPICE2G6*.

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	Erathostenes	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	Erathostenes	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 3.11: Distance between programs. Distance is measured using the squared Euclidean distance.

Of these, only *GREYCODE* and *PERFECT* are considered, as they are the ones included in the Perfect and SPEC suites. The average distance between all programs is 1.1990 with a standard deviation of 0.8169. 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 block, 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) .
```

One difference between *TRFD* and *MATRIX300* and *DYFESM* and *LINPACK* is in their precision; double precision arithmetic is used in the first programs, while single precision is used on the other two. Newer machines, like the IBM RS/6000, have specialized hardware to speed up the execution of these type of statements. In these machines a multiply-add primitive 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 functions between the multiply and add, the execution time of this operation is significantly reduced compared to a multiply

followed by an add [Olss90]. Given the large number of scientific programs that contain loops based on the multiply-add operation it makes sense to create an optimized instruction implementing both operations.

Figure 3.11 shows the clustering of programs according to their distances. Pairs of programs having distance less than 0.7100 are joined by a bidirectional arrow. The thickness of the arrow is related to the magnitude of the distance. Three clusters are present in the diagram. One, with eight programs and containing *LINPACK* as a member, includes those programs that are dominated by single precision floating-point arithmetic. Another cluster, having also 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 of four 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.

The similarity distance given by equation (3.5) is based on on the dynamic statistics of the programs. We have assumed that two very similar programs, in terms of the metric, will tend to behave in in a similar way when they are used as benchmarks. In §3.6.6 we compare equation (3.5) against a similarity distance based on execution times. There we show that the two metrics have significant correlation.

We have investigated other methods of clustering programs. One of these consists in using Chernoff faces [Saav90], which maps program characteristics to facial features. The resemblance of the faces helps in identifying similarities between the programs. We have found that this technique is relatively useful. Chernoff faces, however, are very sensitive to the particular mapping chosen. It is necessary to try different variations to make sure that the correct clusters are found.

3.5.10.1. How Many Benchmarks are Needed in a Suite?

The main purpose in characterizing benchmarks is to understand which aspects of the machine the programs are testing and to identify similarities between them. With this information we can then construct a suite containing those benchmarks which as a whole represent a particular workload. We would like the number of programs to be as small as possible. The question is, what is the minimum number of programs that we need to make a good evaluation of the CPU? Is it possible to find a single application that will test the whole CPU? The results of the previous sections indicate, however, that most programs only test a few operations and concentrate their execution in an equally small number of basic blocks; so, in order to cover the performance space, we need a significant number of benchmarks. Consider for example the set of programs used in this research. The results in table 3.11 and figure 3.11 clearly show that programs *TRFD* and *MATRIX300*, and *DYFESM* and *LINPACK* are very similar and thus including one from each pair is sufficient. Nonetheless, the other programs, even when they have some similarities, have sufficient differences to warrant their inclusion in a suite. In particular, the six SPEC programs are quite different and represent different observations of the potential performance of a machine. Increasing the number of benchmarks in a suite, if they

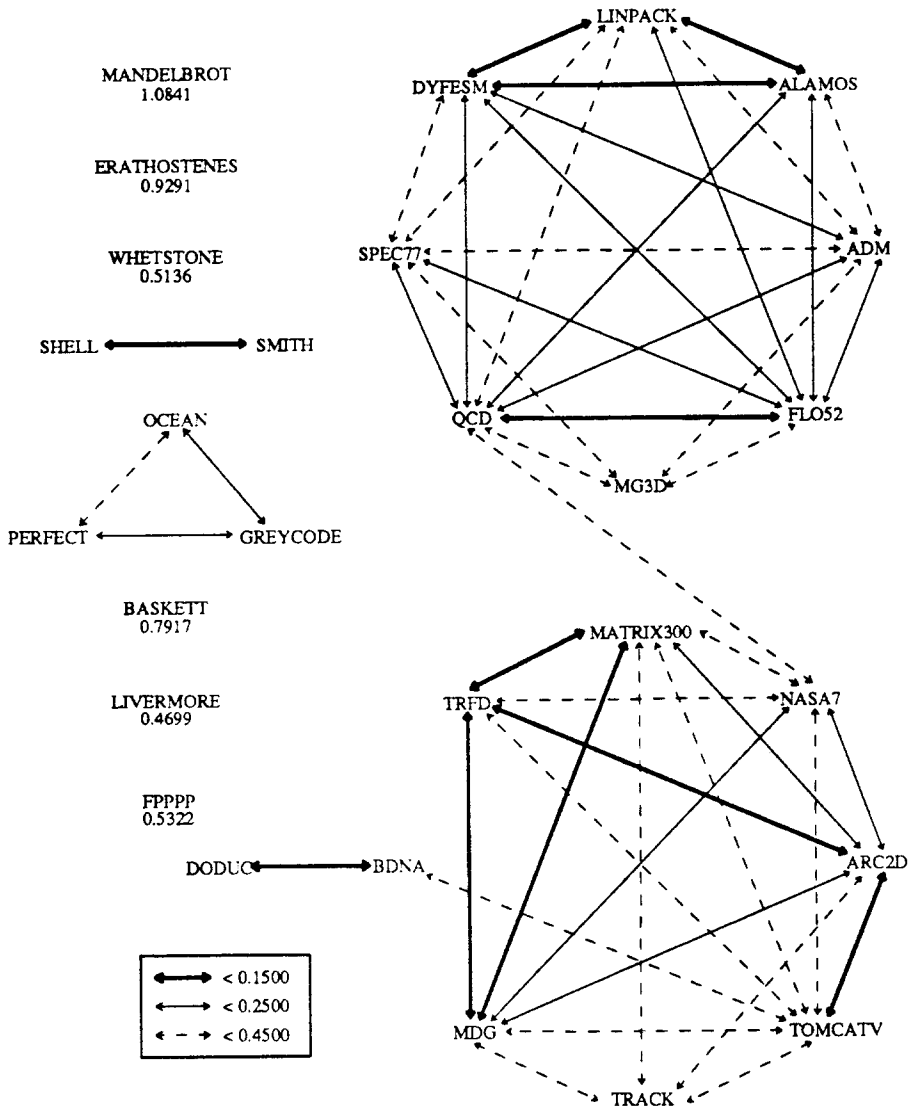


Figure 3.11: Principal clusters found in the Perfect, SPEC, and Small benchmarks. Distance is represented by the thickness of the arrow. The relative position between the programs is arbitrary. Programs whose smallest distance is greater than 0.4500 show under their name the magnitude of their smallest distance.

are carefully selected, is analogous to augmenting the resolution of a measuring tool. On the other hand, if the number of benchmarks is fixed, then a diagram like the one in figure 3.11 can be used to guide the selection process.

3.6. Execution Prediction

The most important limitation of benchmarking is its inability to predict the execution time of arbitrary programs. As we mentioned, this limitation is a direct consequence of not having an execution model in which the performance of a system can be represented. Without one, it is not possible to relate the performance parameters of the machine to program characteristics. In this section we show how to obtain execution estimates for the benchmarks presented in §3.5 on a variety of architectures. We then compare these estimates against real measurements. The predictions are quite accurate; half of these differ by less 10% from the actual execution times.

3.6.1. Extending the Model to Workloads

In our model extending the concepts of program distribution and execution prediction to workloads is straightforward. Consider a workload composed of m programs, each program j in the workload having $C_j = <C_{j,1}, \cdots, C_{j,n}>$ as its dynamic statistics, and a set of program weights $\{W_1, W_2, \cdots, W_m\}$ representing the importance of each program in the workload. The execution time for workload W on machine M ($T_{W,M}$) is given by

$$T_{W,M} = \sum_{j=1}^{n} \sum_{i=1}^{m} W_j C_{j,i} P_{M,i}, \tag{3.6}$$

where $P_{M,i}$, as in equation (3.1), is the value of operation i on machine M. Once we determine the dynamic statistics of the workload we can predict its execution time and study how it changes as a function of the programs and weights.

3.6.2. The Execution Predictor Module

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.

Figure 3.12 shows a sample output produced by the execution predictor. Each line gives the number of times that a particular operation is executed, and the fraction of the total that it represents. Next to it is the expected execution time contributed by the operation 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 3.12 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 distinguish if there is some kind of imbalance in any of the machines that makes its overall execution time larger than expected.

```
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 3.12: Execution time estimate for the TRFD benchmark program run on a IBM RS/6000 530.

3.6.3. Predicting Execution Times

We used the execution predictor to obtain estimates for the programs in table 3.1, and for the machines shown in table 3.12. These results are presented in figures 3.13 and 3.14. In addition, in tables 3.33 through 3.35 on Appendix 3.C 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 error and the root mean square across all machines and programs. From the results on Appendix 3.C we see that 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. Some of the reasons for this were internal compiler errors, run time errors, or invalid results. Livermore 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

Table 3.12: **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 X-MP/48	NASA Ames	COS 1.16	CFT 1.14	8 Mw	46	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	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	32 MB	32	32	64
Amdahl 5840	prandtl.nas.nasa.gov	UTS V	F77	32 MB	32	32	64

Table 3.12: Characteristics of the machines. The size of the data type implementations are in number of bits.

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.

From the predictions and real measurements we can see that we can estimate the relative performance of the machines on different benchmarks. To illustrate this, consider programs *QCD* and *MDG* running on the CRAY Y-MP and IBM RS/6000 530. The real times and predictions for *QCD* indicate that the CRAY Y-MP should be approximately 35% faster than the IBM RS/6000 530 (the real and predicted times on both machines are respectively: 90 and 121 seconds, and 93 and 134 seconds). But on program *MDG* the prediction correctly indicates that the IBM RS/6000 executes faster. Given that all programs in this study were analyzed and executed without making any changes to the source code, in interpreting these results the reader should keep in mind that the precision on CRAY machines is different than on other machines, and this affects the amount of memory allocated to program variables. This occurs when program declarations do not specify explicitly the number of bytes to allocate, but use the words 'single' or 'double'. On CRAYs, by default, single and double precision numbers are allocated on 64 and 128 bits, while the same declarations on workstations are allocated on 32 and 64 bits. On CRAYs the performance of double precision floating-point arithmetic is much slower than single precision, because the former are emulated in software[6]. This difference has a significant impact in the execution time of the benchmarks. On the other hand, in some workstations the difference in speed between 32-bit

[6] In addition to the extra arithmetic operations required in the double precision operations, the compiler generates extra code in order to pass the arguments to the function, store the result in the

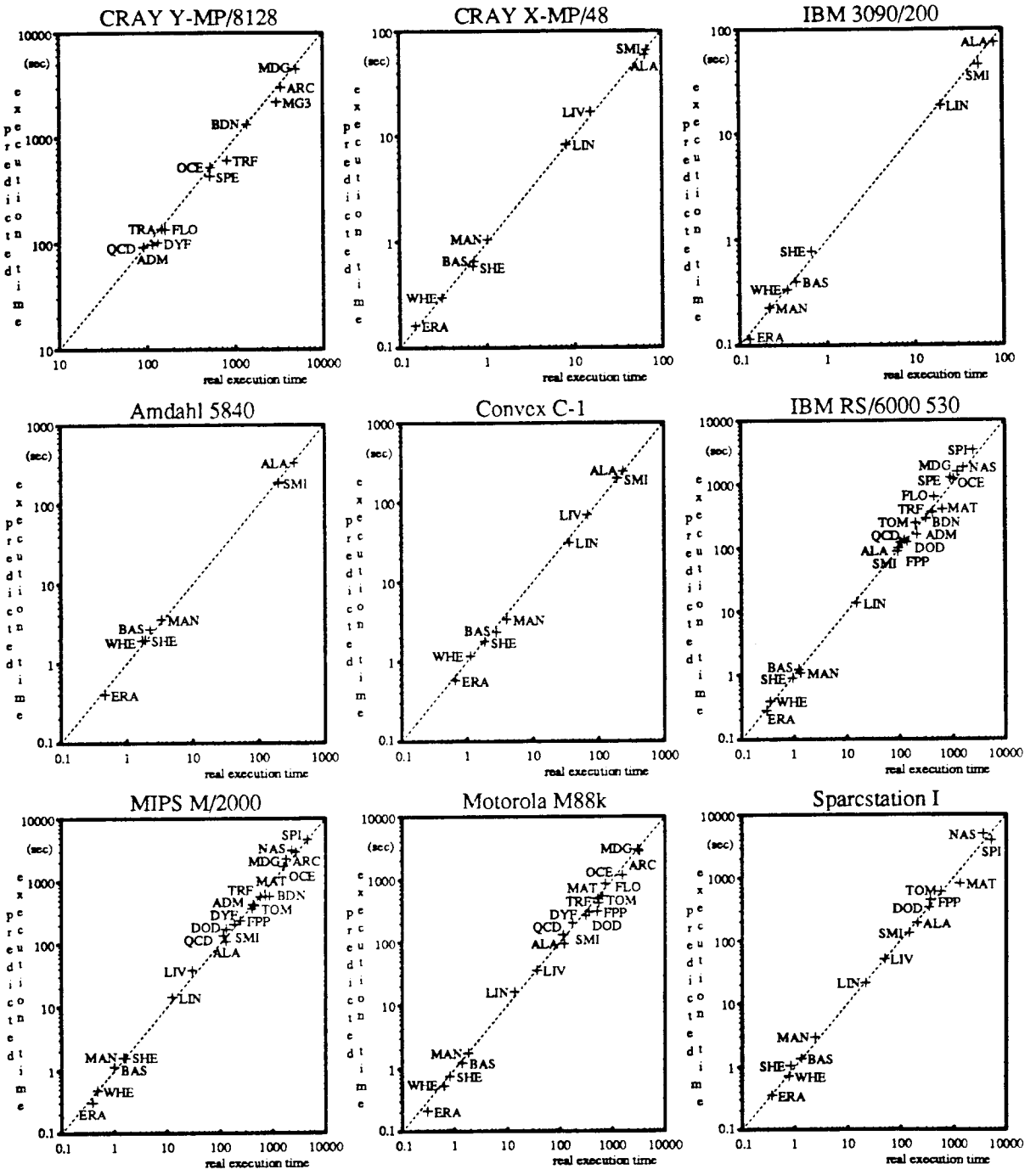


Figure 3.13: Comparison between real and predicted execution times (1 of 2). The predictions were computed using the program dynamic distributions and the machine characterizations. The vertical distance to the diagonal represents the predicted error.

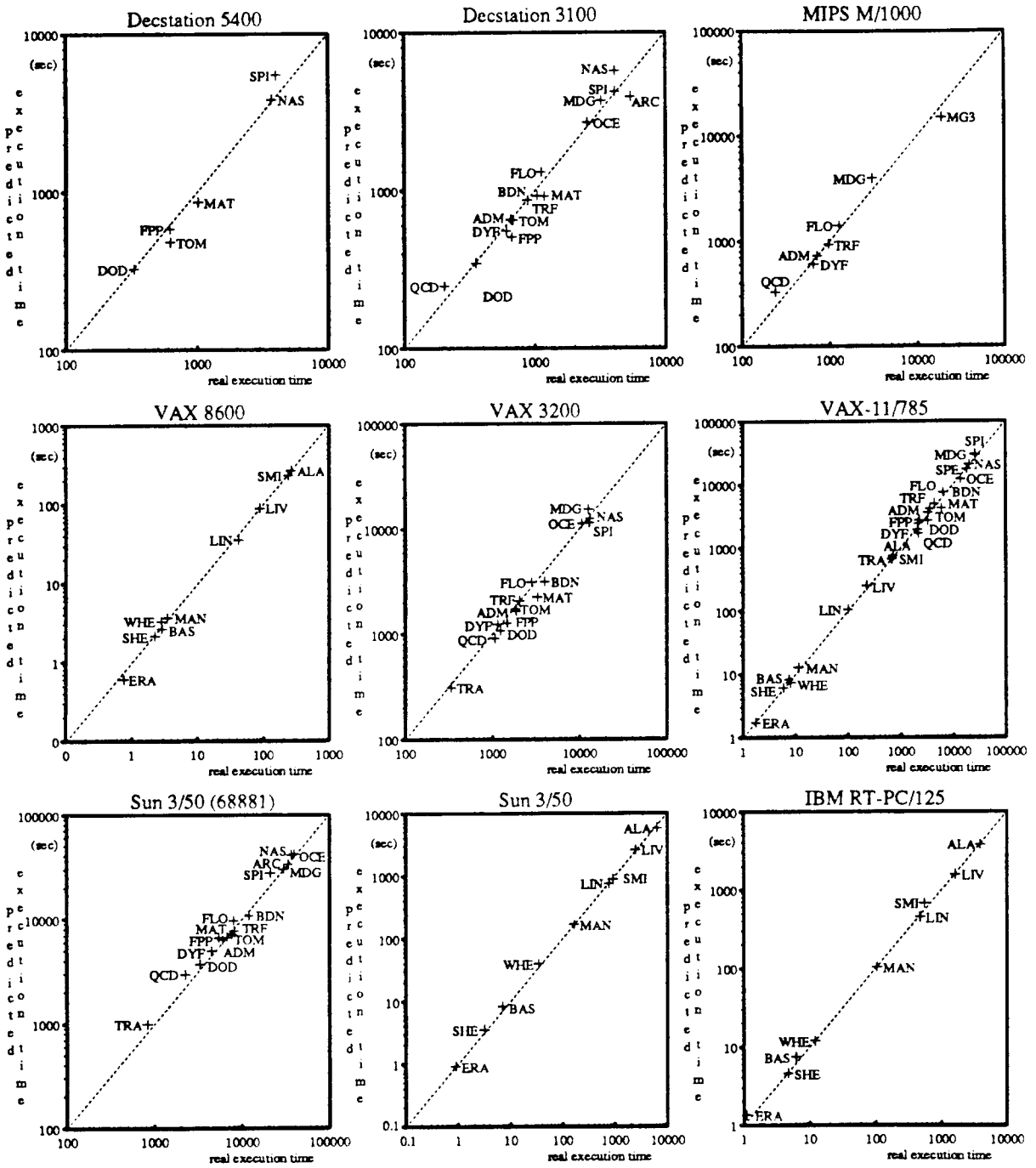


Figure 3.14: Comparison between real and predicted execution times (2 of 2). The predictions were computed using the program dynamic distributions and the machine characterizations. The vertical distance to the diagonal represents the predicted error.

and 64-bit arithmetic is negligible, as all operations are executed using at least 64 bits. Therefore, the observed difference of 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.

Table 3.12 also gives information on the number of bits used to represent different data types for all machines.

Table 3.13: **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 3.13: 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).

In table 3.13 we show the overall accuracy of the predictions categorized by different error intervals. We report the number and the percentage of machine-program combinations having a prediction error of less than 5, 10, 15, 20, 30 percent. We also indicate how many of them have errors of more than 30%. The results show that 51% of all predictions fall within the 10% of the real execution times, and almost 79% are within 20%. These results are quite good if we consider that the characterization of machines and programs is done using a high level model. 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.

The maximum discrepancy in the predictions occurs on *MATRIX300* which has an average error of −24.51% and a root mean square error of 26.36%. Our predictions on this program consistently underestimate the execution time on all machines. The reason is that on this program the number of data cache misses and TLB misses is significant. Because our model, as presented in this chapter, ignores effects of locality, our predictions do not include the time spent satisfying misses. In Chapter 5 we extend our model to include the effects of locality and show that it is possible to improve our predictions to account for the effects of locality. Because most of the benchmarks in the SPEC and Perfect suite tend to have low cache and TLB miss ratios [GeeJ91, GeeJ92], our prediction errors do not present the same bias as in *MATRIX300*.

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

As we mentioned in §§3.5.7 and 3.5.8, an application which executes a small number of different operations or one where most of the execution time is spent in a few basic blocks is not a good program to use as a benchmark. In a similar way we can expect that programs with skewed distributions would tend produce larger prediction errors. The rationale behind this comes from applying a simple result from basic statistics to our particular situation. Assume that a program executes *NP* abstract operations,

---

appropriate register, and make the call. The execution time of the call is one order of magnitude larger than a single precision add or multiply.

and that the performance of all abstract operations have, in addition to their average execution time, an associated experimental error represented by the standard deviation of the measurement. For simplicity we assume that the standard deviation of all operations are equal (σ). Now, consider the following two scenarios: a) the program executes the same abstract operation NP times, and b) the program executes N different operations P times. In the former case, the standard deviation of the prediction is equal to $NP\,\sigma$, while in the later case the standard deviation is $N^{1/2}P\,\sigma$. This means that under the above conditions the experimental errors of a program which executes a large number of different abstract operations effectively cancel each other and thus produce a smaller standard deviation.

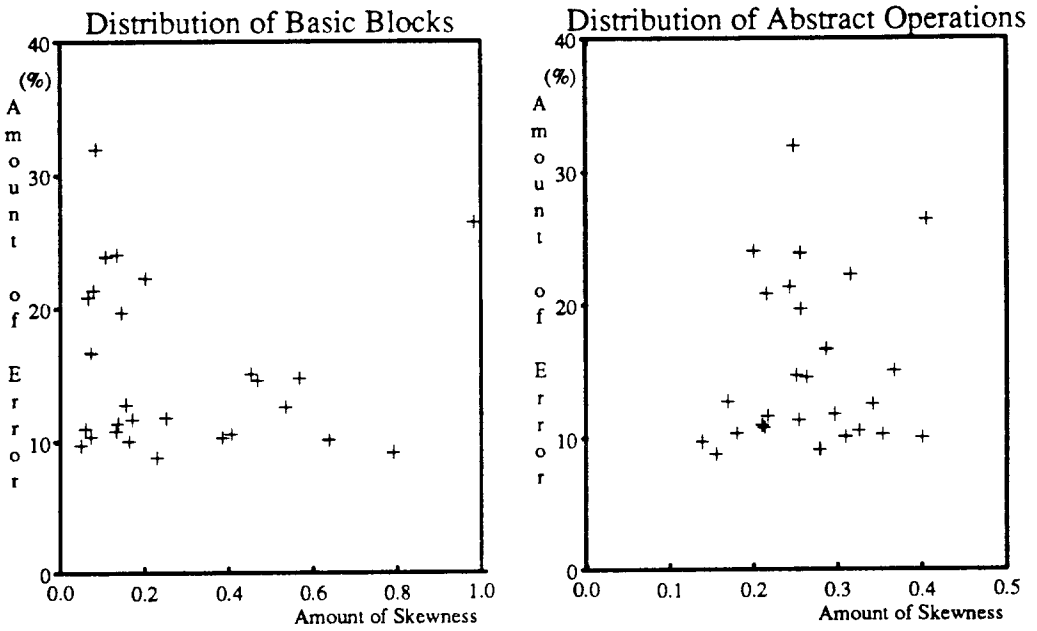


Figure 3.15: Scattergrams of the amount of skewness in the ordered distributions of basic blocks (a) and abstract operations (b) against the amount of error in the execution prediction. The results indicate that there is no sufficient evidence of correlation.

We tested the hypothesis that programs with a skewed distribution of either basic blocks or abstract operations will tend to produce larger prediction errors. The scattergrams for both distributions are shown in figure 3.15. As the figures clearly indicate there is no apparent correlation between the skewness and the magnitude of the errors. One reason is that the magnitudes of the experimental errors are not independent of the particular operations, and at the same time, programs with the very skewed distribution execute mostly floating-point operations, which are the operations that are easier to measure with good accuracy. Hence this tends to compensate somehow the negative effect of the skewness. Another reason is that other factors not included in the abstract model, like locality of reference, have a significant effect in the magnitude of the errors, and these are unrelated to the skewness of either basic blocks or abstract operations distributions.

3.6.5. Single Number Performance

In the end, benchmark results have to be 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.

	Cray X-MP/48	IBM 3090/200	Amdahl 5840	Convex C-1	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 3.14: 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.

In table 3.14 we present both the actual and predicted geometric means, including the percentage of error between them. We mention one more time that we do not attempt to make a comparison between the machines, and therefore no conclusions should be made about their actual relative performance. We can clearly see from the results that our estimates are very close to the actual geometric mean. In all cases the difference is less than 8%.

On some machines we also include their SPECmark numbers. Unfortunately, not all manufacturers have published their SPECmark results. The SPECmarks are higher than either the real or predicted geometric means. But the SPEC results are for optimized codes, and for some machines, like the IBM RS series, optimization is a very important factor in achieving high performance. This is the result not only of superior compiler technology, but also of major improvements in machine architecture.

3.6.6. Program Similarity and Benchmark Results

Our motivation in proposing a metric for program similarity in §3.5.10 was to identify groups of programs having similar characteristics. Intuitively we expect that similar benchmarks will produce similar performance results on many machines. Here we show how this metric of similarity, which was defined in terms of program characteristics, compares against a new metric based on actual benchmark results. First, we introduce the concept of benchmark equivalence in order to give a more precise definition of program similarity.

Definition: Two programs are said to be *benchmark equivalent* on a set of machines, if the relative performance of the machines under both programs are the same, that is, the two programs produce exactly the same rating of the machines.

By rating we refer not only to the ranking, i.e., one machine is faster than another, but also that there is agreement in the numbers representing their relative performance. 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

$$\frac{t_{A,M_i}}{t_{A,M_j}} = \frac{t_{B,M_i}}{t_{B,M_j}}. \tag{3.7}$$

This equation implies that the execution times obtained using program A differ from the execution times using program B, on all machines, by a multiplicative factor k

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

It is very difficult for two different programs to satisfy completely our definition of benchmark equivalence. Therefore, we will use a weaker concept, that of program similarity, to measure how far are two programs from true equivalence. Given two sets of benchmark results, we quantify how similar are the corresponding programs by computing the coefficient of variation of variable $z_{A,B,i} = t_{A,M_i}/t_{B,M_i}$[7]. The coefficient of variation measures how well the execution times of one program can be inferred from the execution times of the other program.

In figure 3.16 we show all the real execution times of the benchamrks (tables 3.33-3.35 in Appendix 3.C). The benchmark results for the IBM RS/6000 530, MIPS M/2000, Sparcstation I, and Motorola M88k are highlighted using special symbols for comparison. Recall that all the programs are executed in scalar mode and without using compiler optimization. The graph uses a logarithmic scale for the execution times, so multiplicative differences translate into additive offsets. This means that two programs that are benchmark equivalent will appear in the figure as having identical distributions but separated by an offset which is equal to k, the mean of variable $z_{A,B}$. For example, the individual distributions of figure 3.16 clearly show that programs *ALAMOS*, *LINPACK*, and *LIVERMORE* rate the machines in a similar way. For example, the relative speed of the machines according to the three programs is {2,3,7,8,9,5,13,15,16,19,18}[8]. Furthermore, the distance between consecutive machines, the relative performance difference, is also very similar on the three benchmarks. This can be contrasted with the permutation produced by the other benchmarks.

As we did in §3.5.10, we present in table 3.15 the 25 most and least similar programs, using in this case the coefficient of variation. In the results we only include pairs of programs having common benchmark results on at least 5 machines in order to have a significant statistic. As we expect, the results identify programs *ALAMOS*, *LINPACK*, and *LIVERMORE* as very similar. The same situation is found in *ADM*, *TOMCATV*, and *NASA7*. But in this case the number of common observations is less than that of the other programs, thus it is not clear that the same degree of similarity would remain if we increased the number of observations.

---

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

[8] We do not include machines 4 and 6 in the permutation, because on these machines at least one of the three benchmarks did not execute successfully.

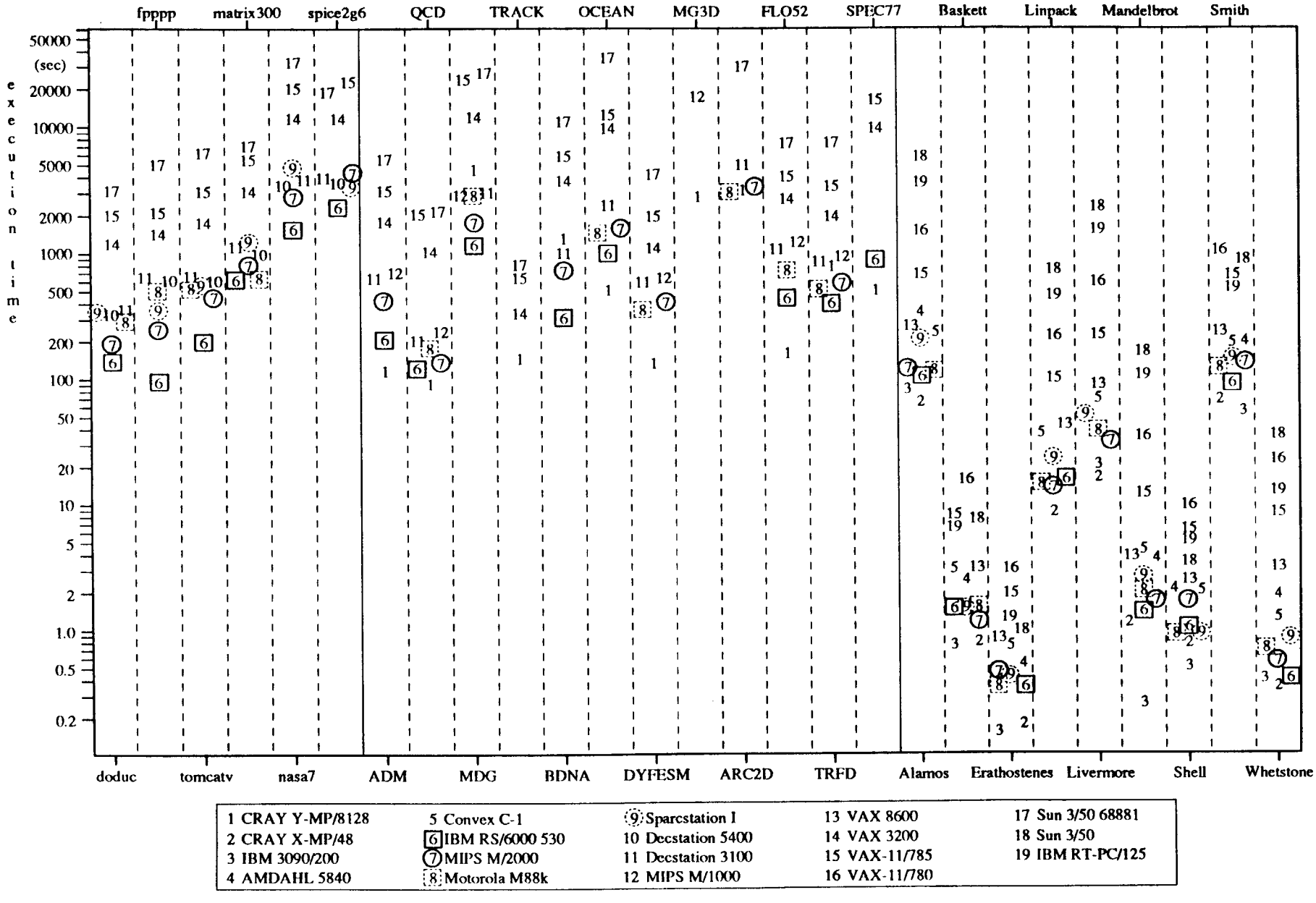


Figure 3.16: Distribution of execution times. Similar programs seem to produce similar distributions; the corresponding ratios of execution times on all machines are close to the same constant. *ALAMOS*, *LINPACK*, and *LIVERMORE* are clear examples of program similarity with respect to their execution time distributions.

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 3.15: 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.

Now that we have defined two different metrics for benchmark similarity, one based on program characteristics (see §3.5.10), and the other based on execution time results, we can compare the two metrics to see if there exists a high 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

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

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.[9]

[9] In computing the rank correlation coefficient we use the same set of program pairs for both metrics. The number of pairs for which there were enough benchmark results to compute the coefficient of variation is only half the total number of pairs. However, even if all pairs are included in the computation of the Spearman's coefficient the level of significance continues to be less than 0.00001.

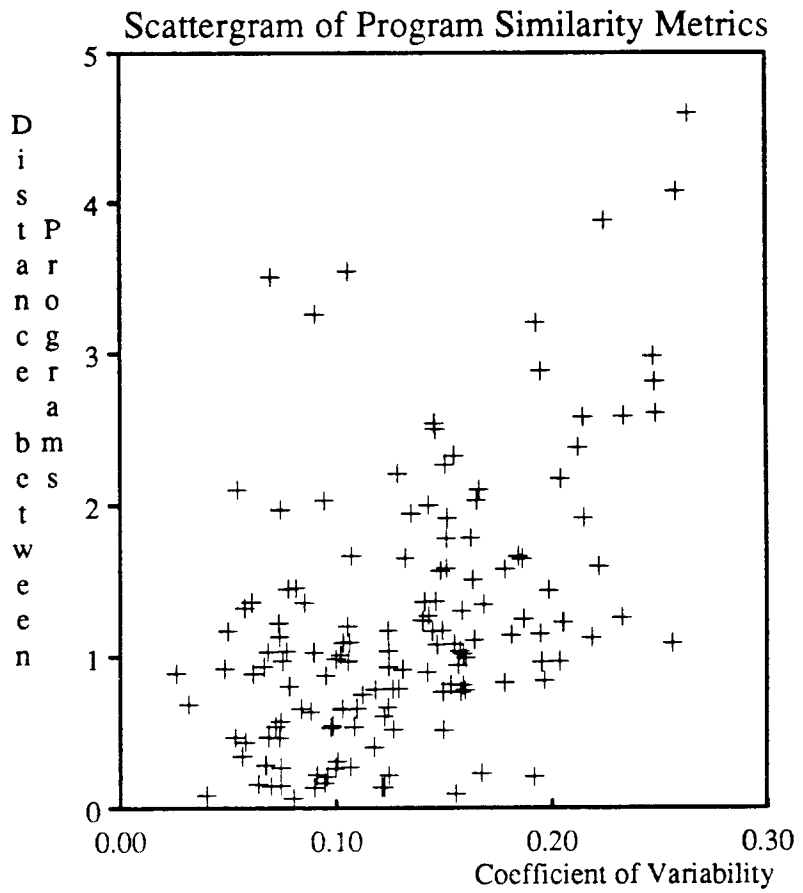


Figure 3.17: Scattergram of the two program similarity metrics. The horizontal axis corresponds to the metric computed from benchmark execution times, while the one on the vertical axis is computed from dynamic program statistics. The results exhibit a significant positive correlation.

A scattergram of two metrics is given in figure 3.17. 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. The results indicate that there is a significant positive correlation between the two metrics at the level of 0.0001. This means that our program similarity measure of §3.5.10 can be used to evaluate whether two programs would exercise the CPU of different machines in a similar way, and therefore we can expect their benchmark results to be similar. This kind of information can be used to aid in the selection of benchmarks without having to execute all of them in a large number of machines to identify the ones that produce similar results.

3.6.7. Execution Time Prediction and Optimization

In this and in the previous chapters, we have explicitly ignored the effect of optimization in our performance model, hence all our predictions were obtained under the assumption that the program is compiled without optimization. The reason behind this is

that small changes in the source code can substantially reduce the execution time of a program, especially when these changes affect the most executed basic blocks, and this information cannot be obtained without knowing in detail how a particular compiler works. In the next chapter, however, we extend our basic model to account for the effects of optimization and show that for a subset of optimizations, which we call invariant, we can still predict the execution time of programs when most of the optimization comes from these optimizations.

3.6.8. Other Factors Affecting the Abstract Model

There are some inherent limitations in using a linear high-level model and software experiments to characterize machine performance. Here we briefly mention the most important of them. For a more in-depth discussion see [Saav88, Saav89]. The main sources of error in our model can be grouped in two classes. The first corresponds to the non-reproducible nature of some architectural elements; some of these are: locality of reference and cache misses, branching, hardware or software interlocks, machine idioms, and non-linear interactions between consecutive machine instructions [Clap86]. In Chapter 5 we present an extension to our model which takes into account the locality of programs. This is done by adding a term which accounts for the time that machines spend satisfying cache and TLB misses. The last point deals with the non additive property of execution time measurements. The second group 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[10], multiprogramming) [Curr75].

3.6.8.1. Superscalar and Superpipelined Processors

Recently, manufacturers of high performance microprocessor have attempted to increase the performance of their processor architectures by making superscalar and/or superpipelined implementations. In a traditional pipelined processor several instructions can execute concurrently, with the only restriction that at most only one instruction can be issued on every cycle [Hene90]. In a superscalar processor, however, several instructions can be issued in one cycle as long as they belong to different functional units and their concurrent executions do not violate data and control dependencies. Under these conditions it is possible for a subunit to get ahead of the others for some number of cycles. This requires fully independent functional units, wider and multiple busses, a higher instruction bandwidth, and some form of scoreboarding. The IBM RS/6000 series is a example of a machine with a superscalar processor [Grov90]. In a superpipelined processor, on the other hand, the main idea is to re-use a functional unit more than once during a single pipeline stage. For example, a data cache in a superpipelined design will accept several requests from the processor in a single cycle. The new MIPS R4000 is an example of a processor which attempts to exploit superpipelining. The superscalar approach offers more potential to exploit concurrency, but its implementation is more difficult and complex.

---

[10] The number of page faults incurred by the program is in part dependent on its execution patterns and the environment in which the program executes.

In principle, our model should not be affected by the fact that the main processor is superscalar or superpipelined. This is because any improvement in the processor's design which affects its performance will have a corresponding effect in the measurement of abstract operations, and this will be reflected in the predictions. The potential problem lies not in the increased concurrency that these new techniques try to exploit, but in the fact that they are not orthogonal with respect to the instruction set. The amount of concurrency that a superscalar processor can exploit is a function of the execution stream and this dependency has a substantial effect in the observed performance. Because we tend to characterize each abstract operation with a single parameter, our measurements do not include the potential dynamic range of performance. Therefore we can expect, as a result of this, that the distribution of predicting errors for superscalar machines will exhibit a larger variance. 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 on all machines. Furthermore, the results on the RS/6000 also gives the maximum positive and negative errors (−35.9% and 44.0%). However, even when the errors tend to be larger than on other machines, the corresponding increase is not very large compared to the errors on the rest of the machines. Furthermore, one possible way of reducing these errors is to incorporate in our model information about the context in which operations are executed. These new operations can then be associated with the different contexts.

3.7. Summary and Conclusions

In this chapter 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 execution time estimates for arbitrary Fortran programs. The same approach can be used for other algebraic languages with different characteristics than Fortran. In most cases, however, a much 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.

The main results of this research are the following. We show that it is possible to build high-level models of execution time performance which provide good detail when they are used in the analysis of benchmark results and also in the measurement of individual abstract machine operations. Complementing this, several models can be defined from the set of basic operations, all having different amounts of detail, and use them to evaluate different aspects of the machine performance. In addition, our basic model can be used to predict the execution time of arbitrary Fortran programs and the results thus obtained are in most cases good; 96% of the predictions show less than 30% error.

We also indicated that benchmark characterization is essential for understanding what these programs measure and interpreting their results. Standard benchmarking ignores this aspect, thus it only allows us to observe performance but not explain it. We presented extensive statistics on the SPEC and Perfect Club benchmark suites, and illustrate how these can be used to identify deficiencies in the benchmarks. Finally, we introduced the concept of benchmark similarity which attempts to quantify how similar are the dynamic distributions of programs. We then compared similarity results against actual execution time distributions and showed that our metric can be used to identify those programs which produce similar results.

3.8. References

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

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

[Beel84] Beeler, M., "Beyond the Baskett Benchmark", *Computer Architecture News*, Vol.1, No.1, March 1986, pp. 20-31.

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

[Call88] Callahan, D., Dongarra, J., and Levine, D., "Vectorizing Compilers: A Test Suite and Results", *Proc. of the Supercomputing '88 Conf.*, Kissimmee, Florida, November 1988.

[Chow83] Chow, F., "A Portable Machine-Independent Global Optimizer - Design and Measurements", Ph.D. Thesis and Technical Report 83-254, Computer Systems Laboratory, Stanford University, December 1983.

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

[Cock80] Cocke, J. and Markstein, P., "Measurement of Program Improvement Algorithms", IBM Research Report RC 8111 (#35193), IBM Yorktown Heights, February 1980.

[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 Technical Report 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", *Computer Architecture News*, Vol.16, No.1, March 1988, pp. 47-69.

[Fato89] Fatoohi, R.A., "Vector Performance Analysis of Three Supercomputers: CRAY-2, CRAY Y-MP, and ETA10-Q", *Proc. Supercomputing '89*, Reno, Nevada, November 13-17, pp. 779-788.

[Fato90] Fatoohi, R.A., "Vector Performance Analysis of the NEC SX-2", *Proc. of the 1990 ACM International Conference on Supercomputing*, Brussels, Belgium, June 1990.

[GeeJ91] Gee, J., Hill, M.D., Pnevmatikatos, D.N., and Smith A.J., "Cache Performance of the SPEC Benchmark Suite", *submitted for publication*, also University of California, Berkeley, Technical Report No. UCB/CSD 91/648, October 1991.

[GeeJ92] Gee, J. and Smith, A.J., "TLB Performance of the SPEC Benchmark Suite", *paper in preparation*, January 1992.

[Grah82] Graham, S.L., Kessler, P.B., McKusick, M.K., "gprof: A Call Graph Execution Profiler", *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 17, No. 6, June 1982, pp. 120-126.

[Grov90] Groves, R.D. and Oehler, R., "RISC System/6000 Processor Architecture", *IBM RISC System/6000 Technology*, SA23-2619, IBM Corporation, 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 Transactions on Computers*, Vol. **C-32**, No. 9, September 1983, pp. 859-861.

[Koba84] Kobayashi, M., "Dynamic Characteristics of Loops", *IEEE Transactions 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).

[MacD84] MacDougall, M.H., "Instruction-Level Program and Processor Modeling", *Computer*, Vol.7 No.14, July 1984, pp. 14-24.

[Mard79] Mardia, K.V., Kent, J.T., and Bibby, J.M., *Multivariate Analysis*, New York, Academic Press, 1979.

[McMa86] McMahon, F.H., "The Livermore Fortran Kernels: A Computer Test of the Floating-Point Performance Range", Lawrence Livermore National Laboratory, 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 Corporation, 1990, pp. 34-43.

[Peut77] Peuto, B.L., and Shustek, L.J., "An Instruction Timing Model of CPU Performance", *The fourth Annual Symposium on Computer Architecture*, 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, *Computer Architecture News*, Vol. 18, No. 2, June 1990, pp. 53-68.

[Pond90] Ponder, C.G., "An Analytical Look at Linear Performance Models", Lawrence Livermore National Laboratory, Technical Report UCRL-JC-106105, September 1990.

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

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

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

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

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

[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.

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

[SPEC90a] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol.3, Issue 1, Winter 1990.

[SPEC90b] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol.3, Issue 2, Spring 1990.

[UCB87] U.C. Berkeley, CAD/IC group. "SPICE2G.6", EECS/ERL Industrial Liason Program, U.C. 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.

Appendix 3.A

operation	DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	GREYCODE
001 SRSL	0.0027	–	–	–	–	–
002 ARSL	0.0494	–	–	–	–	–
003 MRSL	0.0040	–	–	–	< 0.0001	–
004 DRSL	0.0029	–	< 0.0001	–	0.0001	–
005 ERSL	–	–	–	–	–	–
006 XRSL	–	–	–	–	–	–
007 TRSL	0.0161	–	–	–	–	–
009 ACDL	–	–	–	–	0.0190	< 0.0001
010 MCDL	–	–	–	–	0.0086	–
011 DCDL	–	–	–	–	–	–
012 ECDL	–	–	–	–	0.0019	–
013 XCDL	–	–	–	–	–	–
014 TCDL	–	–	–	–	0.0031	–
015 SISL	0.0047	0.0052	0.0110	0.0005	< 0.0001	0.0070
016 AISL	0.0035	0.0069	0.0110	0.0009	0.0437	0.0118
017 MISL	< 0.0001	0.0008	–	0.0005	< 0.0001	0.0010
018 DISL	< 0.0001	0.0006	< 0.0001	< 0.0001	< 0.0001	0.0001
019 EISL	–	–	–	–	< 0.0001	–
020 XISL	–	–	–	–	< 0.0001	< 0.0001
021 TISL	0.0429	0.0015	0.0001	< 0.0001	0.0002	3 0.1247
022 SRDL	4 0.0697	0.0489	4 0.1366	2 0.1414	0.0367	0.0116
023 ARDL	3 0.1285	2 0.2367	2 0.2130	3 0.1414	4 0.0792	0.0143
024 MRDL	2 0.1397	0.0271	3 0.1748	4 0.1414	5 0.0705	0.0076
025 DRDL	0.0335	0.0006	0.0055	< 0.0001	0.0020	0.0034
026 ERDL	0.0003	0.0005	–	–	0.0014	–
027 XRDL	0.0007	< 0.0001	–	–	–	–
028 TRDL	0.0593	0.0069	0.0110	0.0007	0.0114	0.0077
029 SRSG	–	–	–	–	–	–
030 ARSG	–	–	–	–	0.0009	–
031 MRSG	< 0.0001	–	–	–	0.0001	–
032 DRSG	–	–	–	–	< 0.0001	–
033 ERSG	–	–	–	–	–	–
034 XRSG	–	–	–	–	–	–
035 TRSG	–	–	–	–	–	–
036 SCDG	–	–	–	–	0.0006	–
037 ACDG	–	–	–	–	0.0001	–
038 MCDG	–	–	–	–	0.0018	–
039 DCDG	–	–	–	–	–	–
040 ECDG	–	–	–	–	–	–
041 XCDG	–	–	–	–	–	–
042 TCDG	–	–	–	–	< 0.0001	–
043 SISG	< 0.0001	< 0.0001	–	–	–	0.0009
044 AISG	< 0.0001	0.0001	–	–	< 0.0001	2 0.2471
045 MISG	< 0.0001	< 0.0001	–	–	–	< 0.0001
046 DISG	0.0003	–	–	–	–	< 0.0001
047 EISG	–	–	–	–	–	–
048 XISG	–	–	–	–	–	–
049 TISG	0.0004	< 0.0001	–	–	< 0.0001	< 0.0001
050 SRDG	0.0105	4 0.0796	–	–	0.0298	0.0125
051 ARDG	0.0199	5 0.0658	–	–	0.0321	0.0124
052 MRDG	0.0310	1 0.3208	–	–	0.0533	0.0111
053 DRDG	0.0046	0.0008	–	–	0.0001	0.0024
054 ERDG	–	0.0001	–	–	–	–
055 XRDG	–	–	–	–	–	–
056 TRDG	0.0076	0.0044	< 0.0001	–	0.0006	0.0010

Table 3.16: Dynamic distributions for the SPEC benchmarks (part 1 of 2).

operation	DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	GREYCODE
057 ANDL	0.0017	0.0014	–	–	–	0.0015
058 CRSL	–	–	–	–	< 0.0001	–
059 CCSL	–	–	–	–	< 0.0001	–
060 CISL	0.0028	0.0054	< 0.0001	0.0005	< 0.0001	4 0.1040
061 CRDL	0.0156	0.0018	0.0109	< 0.0001	–	0.0034
062 ANDG	–	–	–	–	–	–
063 CRSG	–	–	–	–	–	–
064 CCSG	–	–	–	–	–	–
065 CISG	0.0067	< 0.0001	–	–	–	0.0115
066 CRDG	0.0018	0.0003	–	–	–	0.0005
067 PROC	0.0090	0.0020	< 0.0001	0.0005	0.0019	0.0024
068 ARGL	0.0383	0.0025	0.0001	0.0028	0.0021	0.0063
069 GOTO	0.0139	0.0030	0.0109	< 0.0001	< 0.0001	0.0994
070 GCOM	0.0012	0.0006	–	< 0.0001	–	5 0.0003
071 ARR1	1 0.1421	3 0.1672	–	–	0.0453	1 0.2706
072 ARR2	5 0.0648	< 0.0001	1 0.3332	1 0.4264	1 0.2274	< 0.0001
073 ARR3	–	–	–	–	2 0.0964	–
074 ARR4	–	–	–	–	0.0431	–
075 ADDI	0.0108	0.0033	5 0.0326	–	3 0.0871	0.0135
076 LOIN	0.0061	0.0007	0.0001	0.0005	0.0004	5 0.0005
077 LOOV	0.0467	0.0023	0.0275	2 0.1425	0.0704	0.0042
078 LOIX	–	–	–	–	< 0.0001	–
079 LOOX	–	–	–	–	< 0.0001	–
080 LOGS	–	–	–	–	–	–
081 EXPS	–	0.0002	–	–	–	–
082 SINS	–	–	–	–	–	–
083 TANS	–	< 0.0001	–	–	–	–
084 SQRS	–	0.0004	–	–	–	–
085 ABSS	–	0.0013	–	< 0.0001	–	–
086 MODS	–	–	–	–	–	–
087 MAXS	–	–	–	–	–	–
088 LOGD	< 0.0001	–	–	–	0.0001	0.0003
089 EXPD	0.0013	–	–	–	–	0.0004
090 SIND	–	–	–	–	< 0.0001	< 0.0001
091 TAND	–	–	–	–	–	< 0.0001
092 SQRD	0.0008	–	–	–	0.0004	0.0001
093 ABSD	0.0030	–	0.0218	–	0.0004	0.0036
094 MODD	–	–	–	–	0.0001	–
095 MAXD	0.0011	–	< 0.0001	–	< 0.0001	0.0010
096 LOGC	–	–	–	–	0.0009	–
097 EXPC	–	–	–	–	0.0001	–
098 SINC	–	–	–	–	–	–
099 SQRC	–	–	–	–	–	–
100 ABSC	–	–	–	–	< 0.0001	–
101 MAXC	–	–	–	–	< 0.0001	–
102 SQRI	–	–	–	–	–	–
103 ABSI	–	–	–	< 0.0001	–	< 0.0001
104 MODI	–	< 0.0001	–	< 0.0001	< 0.0001	< 0.0001
105 MAXI	< 0.0001	–	–	–	< 0.0001	< 0.0001
106 CMPX	–	–	–	–	–	–
107 REAL	–	–	–	–	–	–
108 IMAG	–	–	–	–	–	–
109 CONJ	–	–	–	–	< 0.0001	–

Table 3.17: Dynamic distributions for the SPEC benchmarks (part 2 of 2).

operation	BENCHMARK	BIPOLE	DIGSR	MOSAMP2	PERFECT	TORONTO
001 SRSL	–	–	–	–	–	–
002 ARSL	–	–	–	–	–	–
003 MRSL	–	–	–	–	–	–
004 DRSL	–	–	–	–	–	–
005 ERSL	–	–	–	–	–	–
006 XRSL	–	–	–	–	–	–
007 TRSL	–	–	–	–	–	–
008 SCDL	< 0.0001	–	–	–	–	–
009 ACDL	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001
010 MCDL	–	–	–	–	–	–
011 DCDL	–	–	–	–	–	–
012 ECDL	–	–	–	–	–	–
013 XCDL	–	–	–	–	–	–
014 TCDL	0.0001	–	–	–	–	–
015 SISL	0.0192	0.0130	0.0119	0.0145	0.0177	0.0122
016 AISL	0.0121	0.0129	0.0111	0.0083	0.0087	0.0097
017 MISL	0.0019	0.0011	0.0005	0.0008	0.0006	0.0007
018 DISL	0.0015	0.0005	0.0002	0.0006	0.0005	0.0004
019 EISL	–	–	–	–	–	–
020 XISL	< 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
022 SRDL	$_4$ 0.0577	0.0254	$_4$ 0.0660	$_4$ 0.0738	$_5$ 0.0593	$_5$ 0.0564
023 ARDL	$_3$ 0.0766	0.0304	$_3$ 0.0830	$_3$ 0.0954	$_3$ 0.0821	$_3$ 0.0681
024 MRDL	0.0384	0.0159	0.0512	0.0460	0.0287	0.0333
025 DRDL	0.0137	0.0075	0.0223	0.0186	0.0080	0.0095
026 ERDL	–	–	–	–	–	–
027 XRDL	–	–	–	–	–	–
028 TRDL	0.0393	0.0166	0.0318	0.0444	0.0402	0.0325
029 SRSG	–	–	–	–	–	–
030 ARSG	–	–	–	–	–	–
031 MRSG	–	–	–	–	–	–
032 DRSG	–	–	–	–	–	–
033 ERSG	–	–	–	–	–	–
034 XRSG	–	–	–	–	–	–
035 TRSG	–	–	–	–	–	–
036 SCDG	–	–	–	–	–	–
037 ACDG	–	–	–	–	–	–
038 MCDG	–	–	–	–	–	–
039 DCDG	–	–	–	–	–	–
040 ECDG	–	–	–	–	–	–
041 XCDG	–	–	–	–	–	–
042 TCDG	–	–	–	–	–	–
043 SISG	0.0056	0.0034	0.0012	0.0026	0.0032	0.0023
044 AISG	$_2$ 0.1232	$_2$ 0.2048	$_2$ 0.1406	$_2$ 0.1147	$_2$ 0.1338	$_2$ 0.1570
045 MISG	< 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
047 EISG	–	–	–	–	–	–
048 XISG	–	–	–	–	–	–
049 TISG	0.0011	0.0001	0.0004	0.0007	0.0006	0.0005
050 SRDG	0.0251	0.0244	0.0216	0.0228	0.0237	0.0220
051 ARDG	0.0284	0.0250	0.0242	0.0276	0.0283	0.0232
052 MRDG	0.0285	0.0213	0.0314	0.0372	0.0268	0.0263
053 DRDG	0.0072	0.0046	0.0080	0.0088	0.0051	0.0063
054 ERDG	–	–	–	–	–	–
055 XRDG	–	–	–	–	–	–
056 TRDG	0.0102	0.0018	0.0083	0.0113	0.0128	0.0090

Table 3.18: Dynamic distributions for the Spice2g6 benchmarks (part 1 of 2).

operation	BENCHMARK	BIPOLE	DIGSR	MOSAMP2	PERFECT	TORONTO
057 ANDL	0.0076	0.0036	0.0054	0.0072	0.0067	0.0060
058 CRSL	–	–	–	–	–	–
059 CCSL	–	–	–	–	–	–
060 CISL	0.0320	5 0.0657	0.0346	0.0222	0.0279	0.0438
061 CRDL	0.0127	0.0058	0.0135	0.0154	0.0130	0.0099
062 ANDG	–	–	–	–	–	–
063 CRSG	–	–	–	–	–	–
064 CCSG	–	–	–	–	–	–
065 CISG	0.0182	0.0221	0.0186	0.0191	0.0209	0.0210
066 CRDG	0.0092	0.0010	0.0091	0.0118	0.0044	0.0058
067 PROC	0.0082	0.0040	0.0040	0.0058	0.0056	0.0048
068 ARGL	0.0267	0.0112	0.0181	0.0259	0.0222	0.0210
069 GOTO	0.0431	4 0.0660	0.0457	0.0416	0.0433	0.0543
070 GCOM	0.0018	0.0007	0.0014	0.0021	0.0023	0.0018
071 ARR1	1 0.2128	1 0.2586	1 0.2153	1 0.2016	1 0.2380	1 0.2302
072 ARR2	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001
073 ARR3	–	–	–	–	–	–
074 ARR4	–	–	–	–	–	–
075 ADDI	5 0.0577	0.0329	0.0336	5 0.0515	4 0.0617	0.0414
076 LOIN	0.0028	0.0014	0.0019	0.0029	0.0038	0.0027
077 LOOV	0.0125	0.0088	0.0083	0.0090	0.0114	0.0093
078 LOIX	–	–	–	–	–	–
079 LOOX	–	–	–	–	–	–
080 LOGS	–	–	–	–	–	–
081 EXPS	–	–	–	–	–	–
082 SINS	–	–	–	–	–	–
083 TANS	–	–	–	–	–	–
084 SQRS	–	–	–	–	–	–
085 ABSS	–	–	–	–	–	–
086 MODS	–	–	–	–	–	–
087 MAXS	–	–	–	–	–	–
088 LOGD	0.0013	0.0006	0.0015	0.0019	0.0014	0.0013
089 EXPD	0.0012	0.0009	0.0014	0.0016	0.0012	0.0011
090 SIND	< 0.0001	< 0.0001	0.0002	< 0.0001	< 0.0001	< 0.0001
091 TAND	< 0.0001	< 0.0001	0.0002	< 0.0001	< 0.0001	< 0.0001
092 SQRD	0.0023	0.0002	0.0045	0.0034	0.0009	0.0010
093 ABSD	0.0084	0.0065	0.0064	0.0084	0.0104	0.0068
094 MODD	–	–	–	–	–	–
095 MAXD	0.0042	0.0024	0.0034	0.0048	0.0060	0.0044
096 LOGC	–	–	–	–	–	–
097 EXPC	–	–	–	–	–	–
098 SINC	–	–	–	–	–	–
099 SQRC	–	–	–	–	–	–
100 ABSC	–	–	–	–	–	–
101 MAXC	–	–	–	–	–	–
102 SQRI	–	–	–	–	–	–
103 ABSI	< 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
105 MAXI	0.0002	0.0001	< 0.0001	0.0001	< 0.0001	< 0.0001
106 CMPX	0.0001	–	–	–	–	–
107 REAL	–	–	–	–	–	–
108 IMAG	< 0.0001	–	–	–	–	–
1090.0001	0.0115	0.0050	0.0111	< 0.0001	< 0.0001 CONJ	–

Table 3.19: Dynamic distributions for the Spice2g6 benchmarks (part 2 of 2).

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

Table 3.20: Dynamic distributions for the Perfect Club benchmarks (part 1 of 4).

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

Table 3.21: Dynamic distributions for the Perfect Club benchmarks (part 2 of 4).

operation	DYFESM	MG3D	ARC2D	FLO52	TRFD	SPEC77
001 SRSL	3 0.1335	0.0739	–	0.0374	–	0.0612
002 ARSL	4 0.1327	4 0.1049	< 0.0001	0.0720	< 0.0001	2 0.1492
003 MRSL	5 0.1300	2 0.1920	< 0.0001	4 0.0822	–	3 0.1171
004 DRSL	< 0.0001	0.0057	< 0.0001	0.0153	–	0.0013
005 ERSL	–	–	–	0.0057	–	< 0.0001
006 XRSL	–	–	–	< 0.0001	–	< 0.0001
007 TRSL	0.0123	0.0440	–	0.0039	–	0.0055
008 SCDL	–	< 0.0001	–	–	–	0.0135
009 ACDL	–	–	–	–	–	0.0135
010 MCDL	–	< 0.0001	–	–	–	–
011 DCDL	–	–	–	–	–	–
012 ECDL	–	–	–	–	–	–
013 XCDL	–	–	–	–	–	–
014 TCDL	–	–	–	–	–	0.0028
015 SISL	0.0073	0.0243	0.0001	0.0011	0.0043	0.0089
016 AISL	0.0221	3 0.1769	0.0001	0.0011	0.0062	0.0113
017 MISL	< 0.0001	0.0010	< 0.0001	< 0.0001	< 0.0001	0.0003
018 DISL	–	0.0002	–	< 0.0001	< 0.0001	< 0.0001
019 EISL	–	–	–	< 0.0001	–	–
020 XISL	–	–	–	–	–	–
021 TISL	0.0028	0.0006	0.0003	< 0.0001	0.0001	0.0012
022 SRDL	–	–	4 0.1441	–	2 0.1416	0.0003
023 ARDL	–	–	5 0.1402	–	3 0.1411	0.0006
024 MRDL	–	–	2 0.1912	–	4 0.1406	0.0013
025 DRDL	–	–	0.0122	–	0.0005	0.0005
026 ERDL	–	–	0.0122	–	< 0.0001	< 0.0001
027 XRDL	–	–	< 0.0001	–	–	< 0.0001
028 TRDL	–	–	0.0200	–	0.0121	0.0005
029 SRSG	0.0009	< 0.0001	–	5 0.0783	–	0.0002
030 ARSG	0.0008	0.0003	< 0.0001	3 0.0935	–	0.0003
031 MRSG	0.0013	0.0099	–	0.0225	–	0.0042
032 DRSG	< 0.0001	–	–	0.0002	–	< 0.0001
033 ERSG	–	–	–	< 0.0001	–	–
034 XRSG	–	–	–	< 0.0001	–	< 0.0001
035 TRSG	< 0.0001	< 0.0001	–	0.0027	–	< 0.0001
036 SCDG	–	–	–	–	–	0.0001
037 ACDG	–	–	–	–	–	0.0001
038 MCDG	–	–	–	–	–	–
039 DCDG	–	–	–	–	–	–
040 ECDG	–	–	–	–	–	–
041 XCDG	–	–	–	–	–	–
042 TCDG	–	–	–	–	–	0.0002
043 SISG	< 0.0001	–	< 0.0001	0.0002	< 0.0001	–
044 AISG	0.0033	< 0.0001	< 0.0001	< 0.0001	0.0005	–
045 MISG	< 0.0001	< 0.0001	< 0.0001	–	< 0.0001	–
046 DISG	–	< 0.0001	–	–	–	–
047 EISG	–	–	–	–	–	–
048 XISG	–	–	–	–	–	–
049 TISG	< 0.0001	–	< 0.0001	< 0.0001	< 0.0001	< 0.0001
050 SRDG	–	–	0.0005	–	–	–
051 ARDG	–	–	0.0005	–	–	–
052 MRDG	–	–	0.0015	–	–	–
053 DRDG	–	–	< 0.0001	–	–	–
054 ERDG	–	–	< 0.0001	–	–	–
055 XRDG	–	–	–	–	–	–
056 TRDG	–	–	< 0.0001	–	–	–

Table 3.22: Dynamic distributions for the Perfect Club benchmarks (part 3 of 4).

operation	DYFESM	MG3D	ARC2D	FLO52	TRFD	SPEC77
057 ANDL	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001
058 CRSL	0.0020	0.0002	0.0005	< 0.0001	–	0.0005
059 CCSL	–	–	–	–	–	–
060 CISL	< 0.0001	0.0003	< 0.0001	< 0.0001	0.0010	0.0001
061 CRDL	–	–	< 0.0001	< 0.0001	0.0044	< 0.0001
062 ANDG	–	–	< 0.0001	–	–	–
063 CRSG	< 0.0001	–	< 0.0001	< 0.0001	< 0.0001	–
064 CCSG	–	–	–	–	–	–
065 CISG	0.0005	–	< 0.0001	0.0001	–	< 0.0001
066 CRDG	–	–	–	–	< 0.0001	–
067 PROC	0.0003	0.0001	< 0.0001	0.0012	< 0.0001	0.0001
068 ARGL	0.0013	0.0007	0.0002	0.0025	0.0002	0.0004
069 GOTO	0.0001	< 0.0001	< 0.0001	< 0.0001	–	0.0004
070 GCOM	–	< 0.0001	< 0.0001	–	–	–
071 ARR1	0.0523	1 0.2035	0.0003	0.0040	5 0.1089	1 0.2021
072 ARR2	1 0.3207	5 0.0906	3 0.1648	0.0604	1 0.3274	3 0.1477
073 ARR3	0.0089	0.0028	1 0.2138	1 0.3349	–	0.0003
074 ARR4	< 0.0001	0.0116	–	–	–	–
075 ADDI	0.0156	0.0166	0.0443	2 0.1065	–	5 0.1160
076 LOIN	0.0071	0.0003	0.0003	0.0012	0.0049	0.0013
077 LOOV	2 0.1430	0.0321	0.0460	0.0687	0.1064	0.0177
078 LOIX	< 0.0001	0.0006	–	< 0.0001	< 0.0001	0.0018
079 LOOX	0.0008	0.0068	–	0.0008	< 0.0001	0.0272
080 LOGS	–	–	–	< 0.0001	–	–
081 EXPS	–	–	–	< 0.0001	–	0.0002
082 SINS	< 0.0001	< 0.0001	–	< 0.0001	–	< 0.0001
083 TANS	–	–	< 0.0001	< 0.0001	–	–
084 SQRS	< 0.0001	< 0.0001	< 0.0001	0.0006	–	< 0.0001
085 ABSS	< 0.0001	< 0.0001	–	0.0016	–	0.0002
086 MODS	–	–	–	–	–	–
087 MAXS	–	< 0.0001	–	0.0013	–	< 0.0001
088 LOGD	–	–	–	–	–	–
089 EXPD	–	–	–	–	–	< 0.0001
090 SIND	–	–	< 0.0001	–	–	< 0.0001
091 TAND	–	–	< 0.0001	–	–	< 0.0001
092 SQRD	–	–	0.0034	–	–	< 0.0001
093 ABSD	–	–	0.0019	–	–	–
094 MODD	–	–	–	–	–	–
095 MAXD	–	–	0.0019	–	–	–
096 LOGC	–	–	–	–	–	–
097 EXPC	–	< 0.0001	–	–	–	–
098 SINC	–	–	–	–	–	–
099 SQRC	–	–	–	–	–	–
100 ABSC	–	–	–	–	–	–
101 MAXC	–	–	–	–	–	–
102 SQRI	–	< 0.0001	< 0.0001	–	–	–
103 ABSI	0.0004	< 0.0001	–	< 0.0001	–	–
104 MODI	< 0.0001	0.0001	< 0.0001	< 0.0001	–	< 0.0001
105 MAXI	–	< 0.0001	–	< 0.0001	–	–
106 CMPX	–	–	–	–	–	0.0158
107 REAL	–	–	–	–	–	0.0370
108 IMAG	–	–	–	–	–	0.0367
109 CONJ	–	–	–	–	–	–

Table 3.23: Dynamic distributions for the Perfect Club benchmarks (part 4 of 4).

operation	ALAMOS	BASKETT	ERAS	LINPACK	LIVER	LOOPS	MAND	SHELL	SMITH	WHETS
001 SRSL	< 0.0001	< 0.0001	< 0.0001	3 0.1326	3 0.1389	< 0.0001			2 0.2187	–
002 ARSL	0.0300	< 0.0001	< 0.0001	4 0.1309	1 0.2460	0.0384	1 0.2206	–	0.0010	2 0.1357
003 MRSL	< 0.0001	–	–	5 0.1251	4 0.0919	0.0026	3 0.2168	–	0.0005	0.0019
004 DRSL	< 0.0001	–	–	0.0039	0.0016	0.0016	< 0.0001	–	0.0005	0.0047
005 ERSL	–	–	–	–	0.0015	0.0012	–	–	–	–
006 XRSL	–	–	–	–	–	–	–	–	< 0.0001	–
007 TRSL	< 0.0001	< 0.0001	< 0.0001	0.0094	0.0026	< 0.0001	0.0075	< 0.0001	0.0143	< 0.0001
008 SCDL	–	–	–	–	–	–	–	–	–	–
009 ACDL	–	–	–	–	–	–	–	–	–	–
010 MCDL	–	–	–	–	–	–	–	–	–	–
011 DCDL	–	–	–	–	–	–	–	–	–	–
012 ECDL	–	–	–	–	–	–	–	–	–	–
013 XCDL	–	–	–	–	–	–	–	–	–	–
014 TCDL	–	–	–	–	–	–	–	–	–	–
015 SISL	< 0.0001	0.0053	0.0061	0.0001	0.0051	0.0123	0.0542	3 0.1332	4 0.1052	0.0025
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
017 MISL	–	0.0001	–	0.0038	< 0.0001	< 0.0001	–	< 0.0001	0.0119	0.0125
018 DISL	< 0.0001	–	–	–	0.0001	0.0003	–	< 0.0001	0.0009	–
019 EISL	< 0.0001	–	–	< 0.0001	–	–	–	–	–	–
020 XISL	–	–	–	< 0.0001	–	–	–	–	< 0.0001	–
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
022 SRDL	–	–	–	–	–	–	–	–	0.0103	–
023 ARDL	–	–	–	–	–	–	–	–	0.0058	–
024 MRDL	–	–	–	–	–	–	–	–	0.0007	–
025 DRDL	–	–	–	–	–	–	–	0.0005	–	–
026 ERDL	–	–	–	–	–	–	–	–	–	–
027 XRDL	–	–	–	–	–	–	–	–	–	–
028 TRDL	–	–	–	–	–	–	–	–	0.0113	–
029 SRSG	3 0.1302	–	–	–	0.0384	0.0929	–	–	–	0.0140
030 ARSG	5 0.0701	–	–	–	0.0512	0.1241	–	–	–	0.0043
031 MRSG	4 0.1202	–	–	–	0.0407	0.0987	–	–	–	5 0.0678
032 DRSG	–	–	–	–	0.0011	0.0027	–	–	–	0.0293
033 ERSG	–	–	–	–	0.0002	0.0006	–	–	–	–
034 XRSG	< 0.0001	–	–	–	–	–	–	–	–	–
035 TRSG	< 0.0001	–	–	–	0.0110	0.0262	–	–	–	0.0551
036 SCDG	–	–	–	–	–	–	–	–	–	–
037 ACDG	–	–	–	–	–	–	–	–	–	–
038 MCDG	–	–	–	–	–	–	–	–	–	–
039 DCDG	–	–	–	–	–	–	–	–	–	–
040 ECDG	–	–	–	–	–	–	–	–	–	–
041 XCDG	–	–	–	–	–	–	–	–	–	–
042 TCDG	–	–	–	–	–	–	–	–	–	–
043 SISG	< 0.0001	0.0011	–	–	0.0005	0.0012	–	–	–	0.0188
044 AISG	0.0501	0.0011	–	–	0.0026	0.0064	–	–	–	0.0501
045 MISG	–	–	–	–	< 0.0001	–	–	–	–	0.0313
046 DISG	–	–	–	–	< 0.0001	< 0.0001	–	–	–	–
047 EISG	–	–	–	–	–	–	–	–	–	–
048 XISG	–	–	–	–	–	–	–	–	–	–
049 TISG	–	0.0055	–	–	0.0014	0.0034	–	–	–	0.0309
050 SRDG	–	–	–	–	–	–	–	–	–	–
051 ARDG	–	–	–	–	–	–	–	–	–	–
052 MRDG	–	–	–	–	–	–	–	–	–	–
053 DRDG	–	–	–	–	–	–	–	–	–	–
054 ERDG	–	–	–	–	–	–	–	–	–	–
055 XRDG	–	–	–	–	–	–	–	–	–	–
056 TRDG	–	–	–	–	–	–	–	–	–	–

Table 3.24: Dynamic distributions for several small benchmarks (part 1 of 2).

operation	ALAMOS	BASKETT	ERAS	LINPACK	LIVER	LOOPS	MAND	SHELL	SMITH	WHETS
057 ANDL	< 0.0001	0.0966	–	0.0019	< 0.0001	–	0.0561	–	0.0122	–
058 CRSL	–	–	–	0.0037	0.0042	–	5 0.0561	–	0.0003	–
059 CCSL	–	–	–	–	–	–	–	–	–	–
060 CISL	< 0.0001	0.0057	2 0.2172	0.0077	0.0008	0.0018	0.0561	5 0.1331	0.0172	0.0025
061 CRDL	–	–	–	–	–	–	–	–	0.0003	–
062 ANDG	–	–	–	–	–	–	–	–	–	–
063 CRSG	–	–	–	–	0.0024	0.0058	–	–	–	–
064 CCSG	–	–	–	–	–	0.0008	–	–	–	–
065 CISG	–	1 0.2439	–	–	< 0.0001	–	–	–	–	0.0309
066 CRDG	–	–	–	–	–	–	–	–	–	–
067 PROC	0.0003	0.0035	< 0.0001	0.0019	0.0030	0.0064	< 0.0001	< 0.0001	0.0019	0.0272
068 ARGL	0.0013	0.0069	< 0.0001	0.0115	0.0050	0.0111	< 0.0001	< 0.0001	0.0090	4 0.0810
069 GOTO	–	0.0103	0.0568	0.0037	0.0009	0.0022	4 0.0561	0.0486	0.0485	0.0330
070 GCOM	–	–	–	–	< 0.0001	–	–	–	0.0411	–
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
072 ARR2	–	3 0.1370	–	0.0225	0.0627	0.1519	–	–	0.0030	–
073 ARR3	< 0.0001	–	–	–	0.0076	0.0185	–	–	–	–
074 ARR4	–	–	–	–	–	–	–	–	–	–
075 ADDI	–	0.1019	–	0.0040	0.0346	0.0839	–	–	0.0143	0.0125
076 LOIN	0.0033	0.0038	< 0.0001	0.0021	0.0008	0.0016	< 0.0001	< 0.0001	0.0071	< 0.0001
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
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	–	–	–	–
080 LOGS	–	< 0.0001	–	–	–	–	–	–	–	0.0028
081 EXPS	–	–	–	–	0.0002	0.0005	–	–	–	0.0028
082 SINS	–	–	–	–	–	–	–	–	–	0.0076
083 TANS	–	–	–	–	–	–	–	–	–	0.0019
084 SQRS	–	–	–	–	0.0002	0.0006	–	–	–	0.0028
085 ABSS	–	–	–	0.0020	< 0.0001	–	–	–	–	–
086 MODS	–	–	–	–	–	–	–	–	–	–
087 MAXS	–	–	–	0.0038	0.0027	0.0017	–	–	–	–
088 LOGD	–	–	–	–	–	–	–	–	–	–
089 EXPD	–	–	–	–	–	–	–	–	–	–
090 SIND	–	–	–	–	–	–	–	–	–	–
091 TAND	–	–	–	–	–	–	–	–	–	–
092 SQRD	–	–	–	–	–	–	–	–	–	–
093 ABSD	–	–	–	–	–	–	–	–	–	–
094 MODD	–	–	–	–	–	–	–	–	–	–
095 MAXD	–	–	–	–	–	–	–	–	–	–
096 LOGC	–	–	–	–	–	–	–	–	–	–
097 EXPC	–	–	–	–	–	–	–	–	–	–
098 SINC	–	–	–	–	–	–	–	–	–	–
099 SQRC	–	–	–	–	–	–	–	–	–	–
100 ABSC	–	–	–	–	–	–	–	–	–	–
101 MAXC	–	–	–	–	–	–	–	–	–	–
102 SQRI	–	–	–	–	–	–	–	–	–	–
103 ABSI	–	–	–	–	–	–	–	–	–	–
104 MODI	< 0.0001	–	–	0.0038	< 0.0001	–	–	–	–	–
105 MAXI	–	–	–	–	< 0.0001	–	–	–	–	–
106 CMPX	–	–	–	–	–	–	–	–	–	–
107 REAL	–	–	–	–	–	–	–	–	–	–
108 IMAG	–	–	–	–	–	–	–	–	–	–
109 CONJ	–	–	–	–	–	–	–	–	–	–

Table 3.25: Dynamic distributions for several small benchmarks (part 2 of 2).

Appendix 3.B

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

Program	BENCHMARK	BIPOLE	DIGSR	MOSAMP2	PERFECT	TORONTO
program size	14660	14660	14660	14660	14660	14660
basic blocks	6044	6044	6044	6044	6044	6044
lines per block	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%
arith. opers	41.14%	42.21%	45.37%	43.38%	39.58%	42.09%
params executed	5.695×10^7	1.984×10^8	3.184×10^8	2.335×10^7	1.962×10^8	1.353×10^8
assignments	67.74%	59.95%	68.62%	70.56%	68.93%	66.01%
memory transfers	47.64%	64.05%	49.72%	44.73%	47.10%	53.36%
expressions	52.36%	35.95%	50.28%	55.27%	52.90%	46.64%
opers per expr	3.08	4.90	3.70	3.15	3.11	3.60
if statements	10.62%	14.14%	11.04%	9.37%	9.08%	10.71%
procedure calls	2.72%	1.29%	1.37%	1.99%	1.97%	1.61%
user routine	31.92%	27.24%	18.42%	22.31%	21.97%	24.99%
args per call	3.22	2.78	4.48	4.45	3.95	4.31
intrinsic routines	60.08%	72.76%	81.58%	77.69%	78.03%	75.01%
branches	14.79%	21.73%	16.13%	15.00%	16.02%	18.58%
goto	96.03%	98.96%	97.12%	95.21%	95.00%	96.86%
computed goto	3.97%	1.04%	2.88%	4.79%	5.00%	3.14%
loop iterations	4.13%	2.88%	2.84%	3.09%	4.00%	3.09%
iter per loop	4.44	6.44	4.38	3.12	3.01	3.43

Table 3.26: Program and statements statistics for the SPEC benchmarks and several data set for Spice2g6.

program	ADM	QCD	MDG	TRACK	BDNA	OCEAN
program size	4164	1768	932	2110	3659	1908
basic blocks	165	520	216	566	883	616
lines per block	25.24	3.40	4.31	3.73	4.14	3.10
blocks executed	45.25%	77.69%	78.20%	75.97%	59.34%	86.53%
arith. opers	31.74%	32.44%	27.25%	38.14%	40.93%	40.37%
params executed	1.388×10^9	9.799×10^8	7.804×10^9	2.286×10^8	2.014×10^9	5.670×10^9
assignments	77.10%	55.61%	55.36%	35.65%	82.61%	67.50%
memory transfers	20.35%	56.09%	9.30%	35.63%	6.77%	43.86%
expressions	71.65%	43.91%	90.70%	64.37%	93.23%	56.14%
opers per expr	2.35	6.10	1.95	3.13	1.79	3.75
if statements	1.99%	8.50%	6.00%	18.53%	1.84%	0.12%
procedure calls	0.90%	4.62%	4.95%	0.76%	10.76%	0.01%
user routine	52.79%	54.03%	20.01%	29.82%	75.97%	0.14%
args per call	7.62	3.04	3.11	4.41	1.00	2.90
intrinsic routines	47.21%	45.97%	79.99%	70.18%	24.03%	99.86%
branches	0.36%	1.29%	0.73%	18.46%	1.83%	0.02%
goto	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%
loop iterations	19.65%	29.97%	32.96%	26.61%	2.95%	32.35%
iter per loop	11.01	8.96	11.75	19.65	279.84	145.81

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

Table 3.27: Program and statements statistics for the Perfect Club benchmarks.

Program	ALAMOS	BASKETT	ERAS	LINPACK	LIVER
programs size	207	254	21	434	1365
basic blocks	58	106	13	158	374
lines per block	3.569	2.396	1.615	2.747	3.650
blocks executed	100.00%	97.17%	100.00%	58.86%	92.78%
arith. opers	27.04%	45.33%	22.33%	27.90%	45.49%
params executed	6.989×10^8	5.598×10^6	9.989×10^5	7.140×10^7	1.711×10^8
assignments	49.28%	6.92%	21.49%	50.08%	70.02%
memory transfers	0.00%	67.40%	94.95%	9.22%	9.92%
expressions	100.00%	32.60%	5.05%	90.78%	90.08%
opers per expr	2.08	5.44	1.00	2.00	2.45
if statements	0.00%	38.55%	19.82%	1.28%	1.58%
procedure calls	0.10%	1.22%	< 0.01%	0.67%	1.03%
user routine	99.44%	100.00%	100.00%	16.99%	48.79%
args per call	4.99	2.00	1.00	5.90	1.66
intrinsic routines	0.56%	0.00%	0.00%	83.10%	51.21%
branches	0.00%	3.63%	10.16%	1.28%	0.31%
goto	0.00%	100.00%	100.00%	100.00%	99.92%
computed goto	0.00%	0.00%	0.00%	0.00%	0.08%
loop iterations	50.62%	49.68%	48.52%	46.70%	27.05%
iter per loop	40.40	36.82	83.44	66.17	74.95

Program	LOOPS	MAND	SHELL	SMITH	WHETS
program size	454	36	43	436	182
basic blocks	149	9	22	174	39
lines per block	3.047	4.000	1.955	2.506	4.667
blocks executed	90.73%	100.00%	100.00%	97.70%	97.44%
arith. opers	55.01%	65.97%	26.63%	16.27%	36.36%
params executed	1.020×10^8	1.065×10^7	5.420×10^6	4.706×10^8	1.676×10^6
assignments	71.83%	82.51%	70.44%	57.76%	60.20%
memory transfers	26.44%	3.33%	50.62%	56.46%	39.60%
expressions	73.56%	96.67%	49.38%	43.54%	60.40%
opers per expr	2.33	1.80	1.00	1.14	2.50
if statements	0.89%	0.55%	5.79%	2.64%	4.83%
procedure calls	0.19%	< 0.01%	< 0.01%	0.41%	7.51%
user routine	69.49%	100.00%	100.00%	100.00%	60.39%
args per call	1.74	1.00	1.00	4.67	2.97
intrinsic routines	30.51%	0.00%	0.00%	0.00%	39.61%
branches	< 0.01%	16.39%	12.69%	19.36%	9.09%
goto	100.00%	100.00%	100.00%	54.15%	100.00%
computed goto	0.00%	0.00%	0.00%	45.85%	0.00%
loop iterations	27.09%	0.55%	11.08%	19.83%	18.37%
iter per loop	27.44	100.50	14376.13	12.90	11165.00

Table 3.28: Program and statements statistics for the small applications and synthetic benchmarks.

Program	DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	GREYCODE
real (single)	12.60%	0.00%	< 0.01%	0.00%	0.34%	0.00%
+ and –	87.75%	0.00%	0.00%	0.00%	83.05%	0.00%
*	7.18%	0.00%	0.00%	0.00%	8.29%	0.00%
/	5.07%	0.00%	100.00%	0.00%	8.55%	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%
complex	0.00%	0.00%	0.00%	0.00%	9.98%	< 0.01%
+ and –	0.00%	0.00%	0.00%	0.00%	60.90%	100.00%
*	0.00%	0.00%	0.00%	0.00%	32.96%	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%
compare	0.00%	0.00%	0.00%	0.00%	0.01%	0.00%
integer	2.98%	2.07%	2.66%	0.67%	13.89%	86.87%
+ and –	26.33%	50.52%	100.00%	49.88%	99.92%	68.94%
*	0.00%	6.30%	0.00%	24.85%	0.00%	0.27%
/	2.64%	4.24%	0.00%	0.08%	0.00%	0.02%
**	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%
real (double)	84.04%	97.72%	97.34%	99.33%	75.79%	12.77%
+ and –	39.50%	46.21%	52.69%	50.00%	46.66%	48.44%
*	45.47%	53.16%	43.26%	50.00%	51.90%	33.80%
/	10.14%	0.22%	1.35%	< 0.01%	0.88%	10.65%
**	0.27%	0.09%	0.00%	0.00%	0.57%	0.00%
compare	4.63%	0.32%	2.70%	< 0.01%	0.00%	7.11%
logical	0.37%	0.21%	< 0.01%	0.00%	< 0.01%	0.36%

Program	BENCHMARK	BIPOLE	DIGSR	MOSAMP2	PERFECT	TORONTO
real (single)	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%
complex	< 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%
*	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	45.95%	72.74%	45.32%	38.21%	48.65%	55.25%
+ and –	71.59%	70.91%	73.78%	74.21%	74.03%	71.68%
*	1.05%	0.35%	0.23%	0.51%	0.33%	0.29%
/	0.79%	0.15%	0.11%	0.37%	0.26%	0.17%
**	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%
real (double)	52.20%	26.41%	53.48%	60.13%	49.66%	43.31%
+ and –	48.93%	49.74%	44.17%	47.15%	56.18%	50.07%
*	31.16%	33.38%	34.03%	31.91%	28.31%	32.65%
/	9.75%	10.86%	12.48%	10.50%	6.68%	8.69%
**	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%
logical	1.85%	0.85%	1.20%	1.66%	1.69%	1.44%

Table 3.29: Distribution of arithmetic and logical operations according to data type and precision for the SPEC Benchmarks.

Program	ADM	QCD	MDG	TRACK	BDNA	OCEAN
real (single)	92.96%	66.74%	< 0.01%	5.18%	< 0.01%	8.10%
+ and –	50.15%	44.44%	0.00%	0.01%	0.00%	32.63%
*	40.27%	54.85%	0.00%	5.06%	0.00%	62.90%
/	8.45%	0.46%	0.00%	0.03%	0.00%	4.46%
**	0.66%	0.00%	0.00%	0.00%	0.00%	0.00%
compare	0.47%	0.25%	100.00%	94.89%	100.00%	0.01%
complex	< 0.01%	0.00%	0.00%	0.00%	0.00%	20.35%
+ and –	32.50%	0.00%	0.00%	0.00%	0.00%	71.51%
*	45.00%	0.00%	0.00%	0.00%	0.00%	26.29%
/	22.50%	0.00%	0.00%	0.00%	0.00%	2.21%
**	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	6.71%	32.08%	3.65%	24.47%	2.94%	71.53%
+ and –	64.65%	46.24%	86.34%	15.30%	51.75%	81.15%
*	5.36%	29.48%	0.00%	4.93%	25.03%	18.75%
/	2.44%	2.67%	0.00%	0.00%	0.00%	0.06%
**	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%
real (double)	0.00%	1.19%	96.35%	67.35%	97.06%	0.00%
+ and –	0.00%	33.33%	50.01%	39.61%	48.61%	0.00%
*	0.00%	33.33%	29.36%	51.03%	48.25%	0.00%
/	0.00%	33.33%	2.61%	7.20%	2.43%	0.00%
**	0.00%	0.00%	0.00%	0.00%	0.01%	0.00%
compare	0.00%	0.00%	18.01%	2.16%	0.70%	0.00%
logical	0.34%	< 0.01%	< 0.01%	3.00%	< 0.01%	0.01%

Program	DYFESM	MG3D	ARC2D	FLO52	TRFD	SPEC77
real (single)	91.16%	63.69%	0.14%	99.61%	< 0.01%	90.75%
+ and –	50.03%	33.63%	0.01%	56.77%	87.50%	54.83%
*	49.20%	64.51%	0.00%	35.90%	0.00%	44.49%
/	0.01%	1.81%	0.57%	5.33%	0.00%	0.49%
**	0.00%	0.00%	0.00%	1.97%	0.00%	0.01%
compare	0.76%	0.06%	99.42%	0.02%	12.50%	0.19%
complex	0.00%	< 0.01%	0.00%	0.00%	0.00%	4.53%
+ and –	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%
*	0.00%	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%
compare	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%
+ and –	98.05%	99.15%	99.17%	94.48%	87.10%	96.25%
*	0.02%	0.56%	0.03%	0.13%	0.00%	2.81%
/	0.00%	0.10%	0.00%	0.10%	0.00%	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%
real (double)	0.00%	0.00%	99.84%	< 0.01%	97.39%	0.81%
+ and –	0.00%	0.00%	39.32%	0.00%	49.22%	25.28%
*	0.00%	0.00%	53.87%	0.00%	49.07%	53.58%
/	0.00%	0.00%	3.40%	0.00%	0.17%	21.12%
**	0.00%	0.00%	3.41%	0.00%	0.00%	0.00%
compare	0.00%	0.00%	0.00%	100.00%	1.54%	0.01%
logical	< 0.01%	< 0.01%	< 0.01%	< 0.01%	< 0.01%	0.01%

Table 3.30: 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%
+ and –	45.45%	100.00%	100.00%	49.66%	67.42%
*	54.54%	0.00%	0.00%	47.46%	30.08%
/	0.00%	0.00%	0.00%	1.46%	0.63%
**	0.00%	0.00%	0.00%	0.00%	0.39%
compare	0.00%	0.00%	0.00%	1.41%	1.48%
complex	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%
compare	0.00%	0.00%	0.00%	0.00%	0.00%
integer	18.52%	78.68%	99.99%	4.88%	3.07%
+ and –	99.99%	29.99%	2.72%	15.89%	93.19%
*	0.00%	0.02%	0.00%	27.75%	0.25%
/	0.00%	0.00%	0.00%	0.00%	0.83%
**	0.01%	0.00%	0.00%	0.00%	0.00%
compare	0.00%	69.99%	97.28%	56.36%	5.73%
real (double)	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%
compare	0.00%	0.00%	0.00%	0.00%	0.00%
logical	0.00%	21.32%	0.00%	0.67%	0.00%

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

Table 3.31: Distribution of arithmetic and logical operations according to data type and precision for the several programs.

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

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

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

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

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

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

Table 3.32: Distribution of simple and array variables for the SPEC, Perfect Club and small benchmarks.

Appendix 3.C

System	DODUC			FPPPP			TOMCATV		
	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

System	MATRIX300			NASA7			SPICE2G6			average	r.m.s.
	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 3.33: Execution estimates and actual running times for the SPEC benchmarks. All real times and predictions are in seconds; errors in percentage.

System	ADM			QCD			MDG			TRACK			BDNA			OCEAN		
	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

System	DYFESM			MG3D			ARC2D			FLO52			TRFD			SPEC77			average	r.m.s.
	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 3.34: 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.

System	ALAMOS			BASKETT			ERATHOSTENES			LINPACK			LIVERMORE			MANDELBROT		
	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

System	SHELL			SMITH			WHETSTONE			average	r.m.s
	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 3.35: 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.

4 Performance Characterization of Optimizing Compilers

4.1. Summary

In this chapter we focus on the problem of characterizing the performance improvement due to compiler optimization and also in extending our performance methodology to include the effects of optimization. We do this by addressing three different subproblems: 1) extending our model to include optimization and using this new model to quantify and predict the execution time of optimized programs; 2) evaluating the effectiveness of different optimizing compilers in their ability to apply standard optimizations; and 3) evaluating the amount of optimization found in the SPEC suite and identifying distinctive features in the benchmarks which can be exploited by good optimizing compilers.

4.2. Introduction

In Section 4.3 we begin by discussing the relevant work done with respect to evaluating the effectiveness of optimizing compilers. We continue in Section 4.4 by discussing the inherent limitations of our model with respect to compiler optimization. We then proceed by extending our methodology to account for the performance improvements due to optimization by using the concept of invariant optimizations. An optimization is invariant with respect to our abstract machine model if it is still possible to identify in the optimized sequence of machine instructions the original abstract operations embodied in the source code. The advantage of the invariant optimization concept is that it makes it possible to view optimization as defining an 'optimized' implementation of the abstract machine and we can assume that the decomposition of an optimized program with respect to the abstract model remains unchanged. This approach eliminates the need for having to predict how the internal representation of the program changes as a result of the optimization process. Our main result is that in the case when most of the improvement comes from applying invariant optimizations, we can explain and predict the amount of execution time improvement experienced by a program at different levels of optimization.

In Section 4.5 we address the problem of characterizing and comparing different optimizing compilers in their ability to apply standard optimizations. We use a special benchmark consisting of a set of small kernels, each containing a single optimization, which detect the set of optimizations that optimizers can apply and the context in which they are detected. We show that even when most optimizers attempt to apply the same set of optimizations, there are some differences in their relative effectiveness that can significantly affect the performance of some programs.

Finally, in Section 4.6, we analyze the potential optimizations present in the SPEC Fortran benchmarks and how well current optimizers can detect them. We also discuss some problems in the benchmarks which can exploited by smart compilers to artificially improve the performance of the machine.

4.3. Previous Performance Studies in Compiler Optimization

In this section we review some of the work done in evaluating the effectiveness of optimizing compilers. We argue that most performance studies about optimizing compilers have focused on showing that they actually improve the execution time of programs, but have ignored other important aspects like evaluating their effectiveness in detecting optimizations and how often these optimizations occur in real applications.

Knuth, in 1971, was the first who attempted to quantify the potential improvement due to optimization [Knut71]. He statically and dynamically analyzed a large number of Fortran programs and measured the speedup that could be obtained by hand-optimizing them. He found that on the average a program could be improved by a factor of 4[1].

Papers reporting on the effectiveness of real optimizers were not published until the beginning of the eighties [Cock80, Chow83, BalH86, John86, Wolf85, Much86, Jaza86]. Most of these studies, however, have been done by compiler writers, whose main goal has not been to characterize the compilers, but to show that the compilers actually optimize. These studies describe the set of optimizations that can be detected by the optimizers, but without specifying if they are detected on all data types or only on a small subset. As we will see in §4.5.2, very few optimizers are able to detect optimizations on all data types. This can result in a significant loss of execution time improvement as a result of a small change in the precision or type of the data declarations. Another problem with these studies is that the programs used are very small programs which are not representative of real applications and that in some cases are too easy to optimize.

The performance of IBM's PL/1L experimental compiler is evaluated in [Cock80]. The compiler has 3 levels of optimizations. Although the paper describes which optimizations are carried out at each level, only the aggregate speedup is reported. On four programs the amount of speedup obtained at the maximum level of optimization was 1.312. Chow wrote the Uopt, a portable global optimizer, which is the basis of the MIPS Co.

[1] In the rest of this chapter we quantify the improvement produced by an optimizer in terms of the speedup experienced by the program, i.e., the ratio between the unoptimized execution time to the optimized time. Hence, a speedup of 2 means that the optimized execution time is 50% less than the unoptimized time. The overall speedup on all benchmarks is computed by taking the geometric mean of the individual speedups. In order to be consistent, we also follow these two rules when we describe work done by others. Therefore some of our numbers are not the same as those found in the original papers.

compilers. In [Chow83] he gives statistics about the number of times that each optimization was detected and for some of the optimizations he reports the amount of improvement produced. On 13 very small Pascal programs, with no program larger than 160 lines, the average speedup was 1.705. Applying only register allocation or backward code motion resulted in speedups of approximately 1.423 and 1.431 respectively[2]. Bal and Tanenbaum [BalH86] have found using the Amsterdam Compiler Kit optimizer that the speedup produced on toy programs was 1.851, while on larger programs it was only 1.220. Because the larger programs used in the experiments were modules taken from the optimizer itself and all these were written by the a small group of people, it is not clear whether the difference in speedups can be attributed to the complexity of the programs or the ability of the programmers. A performance study based on the HP Precision Architecture global optimizer reported in [John86] found that on the same toy programs used by Chow the average speedup was 1.381. With respect to individual optimizations, the speedup contributed by code motion was only 1.087. However the distribution was very skewed: on three of the eight programs the speedup was close to 1.250, while on the other five it was only 1.004.

The programs used in these studies have been written in C, Pascal, or a dialect of PL/1 and the speedups found were in all cases smaller than two. Later in the chapter we see that Fortran compilers produce speedups significantly larger. Although some amount of this speedup can be attributed to improvements in compiler technology over the last 10 years, nonetheless we believe that most of it is the result of Fortran itself. First, Fortran programs appear to offer more opportunities for optimization as most of the work is inside highly nested loops. Second, although Fortran programs use large amounts of data stored in arrays, the access to this data follows simple regular patterns which can be easily optimized. Third, subroutines in Fortran programs tend to be larger than on other languages, so there is more opportunity to find optimizations. Fourth, typical Fortran code inside basic blocks contain longer sequences of arithmetic statements, rather than complicated sequences containing lots of procedure calls, case statements, and other program structuring techniques, which impede optimization. Lastly, there are no pointers in Fortran, so detecting aliases is no problem for the optimizer. Normally, optimizers are forced to make worst case assumptions in the presence of aliases.

One of the problems in comparing the performance improvement of different optimizers is how to factor out from the speedups the quality of the unoptimized code. It is always possible to increase the speedup produced by an optimizer by generating worse unoptimized code. Another factor is the effect of the architecture. Some machines are easier to generate code for them than others. One of the arguments of the RISC movement [Patt85] was that a simple architecture makes it easier to write better optimizers, as the number of combinations to consider is significantly smaller. An attempt to evaluate the effect of optimization on different architectures using again the Uopt optimizer is reported in [Cude89]. This study found that register architectures tend to benefit the most from optimization, as optimization tends to introduce temporaries which get assigned to registers and eliminate instructions associated with loads and stores.

---

[2] The product of the individual speedups can be larger than the overall speedup because in some cases one optimization prevents the application of the other.

Lindsay has proposed measuring many different parameters of the system in order to characterize its performance [Lind86a, Lind86b]. However, he does not propose any model which these parameter can be plugged into to obtain other useful performance metrics or to predict the execution times of programs. The problem with his approach, which focuses only in collecting measurements, is that it is very difficult to interpret the results and conclude something that is meaningful about the machines. With respect to optimization, he has written some benchmarks to detect which transformations can be applied by compilers [Lind86c]. In [Lind86d] he reports some of his findings on the IBM VS-FORTRAN and the DEC VAX/VMS FORTRAN compilers.

There are other studies dealing with other aspects of optimization. Arnold [Arno83] reports on the effectiveness of the CYBER 205 vectorizing compiler to produce either vector or scalar versions of a loop as a function of the number of iterations. Richarson and Ganapathi [Rich89] have shown that certain types of interprocedural data flow analysis provides marginal improvement on most of the programs in their suite. Callahan, Dongarra, and Levine have collected a large suite of tests for vectorizing compilers and have evaluated a large number of them [Call88]. Most existing vectorizing compilers are based either on the VAST or KAP pre-compilers developed at Pacific Sierra Research and Kuck and Associates, which are compared in [Bras88]. Singh and Hennessy [Sing91] are studying the potential and limitations of automatic parallelization.

Most of the effectiveness of vectorizing compilers comes from making a dependence analysis of the program. People have realized that dependency analysis techniques [Bane88] can also be used to improve the performance of scalar machines. Recent research in optimization have proposed using this information to improve register allocation of subscript variables [Call90], increase locality inside nested loops [Port89, Ferr91, Wolf91], and reduce memory latency by using software prefetching in conjunction with lockup-free caches [Call91]. Although it will take some time before these techniques are incorporated in commercial compilers, they appear to be the best way of improving the performance of scientific programs. As optimizers become more and more powerful and complex, the need to evaluate their effectiveness and characterize how they affect real programs becomes increasingly important.

4.4. Program Optimization and the Abstract Machine Model

In this section we discuss the limitation of the model with respect to compiler optimization and how they can be overcome using the concept of invariant optimizations. We then show that it is possible to predict the execution time of optimized programs when most of the execution time improvement is the result of applying invariant optimizations.

4.4.1. Limitation of Our Model in the Presence of Optimization

Characterizing the performance of machines and predicting the execution times of arbitrary programs is substantially more difficult to achieve in the presence of compiler optimization. The main reason is that optimizers can apply arbitrary transformations to programs, and without knowing how the optimizer works, it is not possible to predict how the source code will change. In principle, an optimizer is allowed to transform a program, or any of its intermediate representations, in any way that reduces its execution time or its use of resources, as long as the relationship between the input and the output is

preserved. Although optimizers still have a long way to go to achieve dramatic improvements, it is typical for current state-of-the-art optimizers to reduce the execution time of Fortran programs on the average by a factor in the 2–4 range.

It is clear that optimizers present a problem to our abstract machine model. In this research we have deliberately avoided using any kind of machine- and compiler-dependent information, either in characterizing the performance of machines or in analyzing benchmarks. The advantage is that it permits constructing a single uniform model of performance which does not have to be extended when we need to evaluate a new architecture or compiler. The shortcoming of this approach, when we consider optimizing compilers, is that it makes it very difficult to know how an arbitrary optimizer would transform the program without knowing in detail how the optimizer works.

Let us start by considering our equation for the execution time of a program

$$T_{A,M} = \sum_{i=1}^{n} C_{i,A} P_{i,M} = \mathbf{C_A} \cdot \mathbf{P_M} \tag{4.1}$$

Here $C_{i,A}$ is the number of abstract operations of type i that program A executes, and $P_{i,M}$ is the execution time of operation i on machine M. In general, when we include optimization, both the decomposition of the program in terms of the abstract model ($\mathbf{C_A}$) and the performance of the abstract operations ($\mathbf{P_M}$) may change. $\mathbf{C_A}$ changes when the optimizer eliminates some part of the computation, or replaces it by a more efficient one. In both cases the distribution of abstract operations is altered. The raw performance measurements represented by $\mathbf{P_A}$ change, because the compiler generates different sequences of machine instructions at different levels of optimization. Therefore, in general, the execution time equation when using an optimizing compiler should be

$$T_{A,M,O} = \sum_{i=1}^{n} C_{i,A,O} P_{i,M,O} = \mathbf{C_{A,O}} \cdot \mathbf{P_{M,O}} \tag{4.2}$$

Our problem here is to obtain $\mathbf{C_{A,O}}$ and $\mathbf{P_{M,O}}$ from $\mathbf{C_A}$ and $\mathbf{P_M}$, by only making an analysis of the program and running experiments with optimization enabled.

4.4.2. Extending the Abstract Model to Include Optimization

Depending on whether we can visualize the effects of a particular optimization as either eliminating some abstract operations from the program, or just improving the sequence of machine instructions which implement specific abstract operations, we can classify optimizations into three classes.

In the first class (type I) we have optimizations which change the program's distribution of abstract operations, either by removing or replacing some amount of code. Common subexpression elimination is one example of this type of optimizations. Here all the abstract operations forming the subexpression are eliminated and replaced by a reference to the previously computed result. Applying a type I optimization has the effect, on equation (4.1), of changing $\mathbf{K_A}$, but without affecting $\mathbf{P_M}$. The problem in characterizing the performance improvement due to these optimizations is that we need to know how $\mathbf{K_A}$ changes, but without having any information about how an arbitrary optimizer works.

In the second class (type II) we have optimizations which only improve the sequence of machine instructions generated by the compiler to implement an abstract operation, but do not remove any abstract operations. Strength reduction is an example of these optimizations. Here one or several expensive operations are replaced by a faster but equivalent sequence of operations. Type II optimizations change $\mathbf{P_M}$, while leaving $\mathbf{K_A}$ unchanged. We call these optimizations *invariant* with respect to the abstract decomposition of the program. The advantage to us of invariant optimizations over type I optimizations is that we can characterize the performance improvement of the former by just running our machine characterizer with optimization enabled. If the optimizer changes the code it generates when it encounters an abstract operation in a program, it does the same action when it encounters it in the machine characterizer. Because there is a particular test for each abstract operation, we can measure the execution time of the optimized version.

The third group (type III) of optimizations are those which change both $\mathbf{K_A}$ and $\mathbf{P_M}$. Characterizing these optimizations is as difficult as type I optimizations. Most optimizations, however, belong to either classes one or two.

Whether an optimization is of type I, II, or III depends mainly on the level at which we define the abstract machine. If the abstract machine is defined at the level of the machine's instruction set, then all optimizations are of type I. The reason is that every machine instruction eliminated affects the decomposition of the program. If, on the other hand, the abstract operations consists of different algorithms, then almost all optimizations are of type II. As long as the algorithm is not eliminated, changes to it are considered only as different implementations of the same abstract operation. Given the level of abstraction of our model, it happens that most source to source transformation are optimizations of type I, and low level transformations are optimizations of type II.

To illustrate the difference between invariant and non invariant optimizations, consider the following code excerpt

```
        DO 2 I = 1, N
            DO 1 J = 1, N
                X(I) = X(I) + Y(J,K) * Z(J,L)
1           CONTINUE
2       CONTINUE
```

During program analysis we identify the different abstract operations, e.g., a floating-point add (ARSL), floating-point multiply (MRSL), computing the addresses of a 1 and 2-dimensional array elements (ARR1 and ARR2), etc. Combining this static decomposition with information of how many times each basic block is executed we can then obtain the contribution of this code to the total execution time

$$Time = (P_{SRSL} + 2 \cdot P_{ARR2} + 2 \cdot P_{ARR1} + P_{MRSL} + P_{ARSL} + P_{LOOV}) N^2 + (P_{LOIN} + P_{LOOV}) N + P_{LOIN}.$$

In table 4.1 we show the sequence of assembler instructions generated by the MIPS Co. *f77* compiler version 1.21 for the innermost loop without and with maximum optimization. We have made inconsequential changes to the syntax of the machine instructions to make the code more readable. On the left side of the table we indicate the abstract operations representing the innermost loop along with their corresponding subsequences

of machine instructions.

4.4.3. Optimization Viewed as an Optimized Implementation of the Abstract Machine

What the above example shows is that even when the two sequences of machine instructions, one unoptimized and the other optimized, are very different, we can still identify in both the original abstract operations. Hence, in this case, the optimizer has reduced the execution time, but the characterization of the program excerpt, in terms of our abstract machine, has not changed. We refer to these type of optimizations, which improve the execution time of a program but do not change the distribution of abstract operations, as being invariant with respect to the abstract machine model. An example of an optimization which is not invariant is common subexpression elimination. Here the compiler actually eliminates some of the computations, therefore we cannot map the abstract operations present in the source code with sequences of machine instructions.

Now in the case when all optimizations are invariant, predicting the execution of the optimized version requires only taking the dot product between the unchanged abstract characterization of the program excerpt and the 'optimized' set of machine parameters. This 'optimized' machine characterization is obtained by running the optimized version of the machine characterizer.

The relevance of viewing optimization, not as an attempt to improve the object code which executes on the same machine, but as running the same abstract set of instructions on an 'optimized' machine, is that we effectively avoid having to predict how an arbitrary optimizer would transform the program.

Although it is not always possible to know how optimization in general will affect a program, it is possible, for many programs, to obtain reasonable predictions by assuming that most of the optimization improvement comes from applying invariant optimizations. Under this assumption the execution time of an optimized program is

$$T_{A,M,O} = \sum_{i=1}^{n} C_{i,A} P_{i,M,O} = \mathbf{C_A} \cdot \mathbf{P_{M,O}} \tag{4.3}$$

There are three main reasons why this approach works. First, optimizations are applied at a low level when most of the program structure is not present any more, so most of the improvement derived is from optimizing sequences of machines instructions and not from eliminating abstract operations. Second, optimizers are consistent in detecting optimizations. If an optimizer is capable of improving the code emitted by the compiler in the expansion of a particular abstract operation, then it can also do it in most of the other instances where the same sequence appears. Lastly, the execution time of programs is normally determined by a small number of basic blocks, and programmers try to eliminate obvious machine-independent optimizations on these blocks to guarantee that the programs will execute efficiently on many machines. This appears to be the case in many benchmarks, although it may not be true for arbitrary programs.

The second argument in the previous paragraph is worth discussing in more detail. Even when a type I optimization changes the distribution of abstract operations of programs by eliminating some operations, it can be considered an invariant optimization as long as the same operations are eliminated from all occurrences in all programs,

abstract operation	assembler code without optimization	assembler code with optimization
arr2	l_int r14, 80444(sp) l_int r15, 40036(sp) mul_i r24, r15, 100 add_i r25, r14, r24 sub_i r8, r25, 101 mul_i r9, r8, 4 add_i r10, r9, -40424 add_i r11, sp, 80464 add_i r12, r10, r11 l_flt f6, 0(r12)	l_flt f4, 24708(r3) add_i r3, r3, 4
arr2	l_int r13, 32(sp) mul_i r15, r13, 100 add_i r24, r14, r15 sub_i r25, r24, 101 mul_i r8, r25, 4 add_i r9, r8, -80428 add_i r10, r9, r11 l_flt f8, 0(r10)	l_flt f6, 17472(r4) add_i r4, r4, 4
mrsl	mul_f f10, f6, f8	mul_f f8, f4, f6
arr1	sub_i r12, r14, 1 mul_f r13, r12, 4 add_i r15, r13, -424 add_i r24, sp, 80464 add_i r25, r15, r24 l_flt f16, 0(r25)	l_flt f16, -428(r2)
arsl	add_f f18, f16, f10	add_f f10, f16, f8
srsl	s_flt f18, 0(r25)	s_flt f10, -428(r2)
loov	l_int r8, 80444(sp) add_i r9, r8, 1 s_int r9, 80444(sp) l_int r11, 80440(sp) br_ne r9, r11, r34	add_i r2, r2, 4 br_ne r2, r6, r35

Table 4.1: Nonoptimized and optimized assembler code for the innermost loop. On the left side we show the abstract machine operations represented by the assembler code.

including our machine characterizer. For example, suppose that a very good compiler is capable of eliminating at compile time all multiply operations in programs. As long as the optimizer is always successful, we can take this optimization in our predictions, because our measurements with the machine characterizer will indicate that the execution time of the multiply operation is zero or close to zero. The corresponding execution time computed using this value will correspond to the actual execution time. Our focus in this subsection is in quantifying the performance effect of optimization and not in finding out which optimizations are applied. In §4.5 we characterize the particular optimizations that compilers can apply.

4.4.4. Limitations of Invariant Optimizations

The above approach to optimization works as long as the optimizer attempts to reduce the execution time of the programs without changing the original computations embodied in the source code. This is not always the case, however. For a long time vectorizing compilers have used data dependency analysis on programs to determine which loops can be executed in parallel or in vector mode. These techniques have begun to be used in high performance scalar computers to make loop transformations to reduce the

total execution time. Consider again the previous code excerpt. If we apply loop interchange, code motion, and loop unrolling [Paud86] we can dramatically reduce the number of operations and consequently the execution time[3]. These two source to source transformations produce the following equivalent piece of code

```
      TMP = 0.0
      DO 1 J = 1, N
         TMP = TMP + Y(J,K) * Z(J,L)
1     CONTINUE
      DO 2 I = 1, N
         X(I) = TMP
2     CONTINUE
```

The contribution of this code to the total execution time is

$$Time = N(P_{MRSL} + P_{ARSL} + 2 \cdot P_{ARR2} + P_{ARR1} + P_{SRSL} + P_{TRSL} + 2 \cdot P_{LOOV}) + 2 \cdot P_{LOIN} + P_{TRSL}.$$

This equation is now linear with respect to the number of iterations instead of quadratic. The example shows that, in general, without detailed knowledge of which transformations were applied by the optimizer it is not possible to predict what would be the execution time.

4.4.5. Machine Characterizations Results with Optimization

In the previous section we argued that we can easily extend our model to include invariant optimizations, if we consider them as defining a faster machine rather than optimizing the object code. This 'optimized machine' has its own machine performance vector which is obtained by executing the system characterizer with optimization enabled. Thus the performance vector shows how the implementation of each abstract operation changes with different levels of optimization. Furthermore we can apply to the performance vector the same metrics as in the unoptimized case. Thus, the concept of performance shape and machine similarity, presented in Chapter 2, are well defined and provide useful information with respect to the effectiveness of optimization. In this section we compare different machine characterizations under various levels of compiler optimization.

We ran the system characterizer using different optimization levels on three high performance workstations. The complete results, including those without optimization, are shown in Appendix 4.A (tables 4.16-4.20). Table 4.2 shows the set of reduced parameters which are obtained from the basic measurements. These reduced values correspond to the hardware and software characteristics of the system which most affect the execution time of Fortran programs. The value of each parameter, which is given in nanoseconds, corresponds to the weighted execution time of the set of abstract operations associated with the reduced parameter. In figure 4.1 we show the inverse normalized performance shapes (pershapes). Here the reduced parameters of table 4.2 are

---

[3] Loop interchange transposes the order of the loops. This allows the compiler to the detect that the expression `Y(J,K) * Z(J,L)` is invariant with respect to the induction variable `I` and hence can be moved out from the loop. The compiler can then identify that all elements of array X get the same value, which can be computed only once and the result added to all elements.

reduced parameters	HP 720			MIPS M/2000			Sparcstation 1+		
	-O0	-O1	-O2	-O0	-O2	-O3	-O0	-O2	-O3
memory bandwidth	108	104	61	173	135	52	545	511	269
integer add	95	60	29	165	89	69	247	188	41
integer multiply	442	419	170	574	463	489	1132	1265	950
logical operations	193	139	95	229	245	144	586	598	383
single prec. add	99	53	45	175	139	95	319	299	233
single prec. multiply	128	96	35	260	201	200	406	343	133
double prec. add	100	73	45	223	167	117	488	427	233
double prec. multiply	129	117	35	348	279	276	799	598	254
division	300	234	188	780	669	575	2648	2592	1933
procedure calls	99	96	72	328	238	161	215	76	77
address	136	76	42	462	262	147	426	267	166
branches & iteration	151	97	41	286	164	95	318	135	105
intrinsic functions	2561	2490	2477	3306	3246	3405	7442	7747	8568

Table 4.2: Optimization performance results in terms of the reduced parameters. Each parameter represents a particular characteristic of the machine and is computed from a subset of basic abstract machine parameters. All units are in nanoseconds. On the Sparcstation 1+ the results for optimization levels 0 and 1 were almost identical, so we only report results for level 0.

normalized with respect to the corresponding value on the MIPS M/2000 under no optimization. The inverse of the normalization is used on the pershapes.

The results clearly show that some abstract and reduced parameters benefit more from optimization than others. The parameters that benefit most are: memory bandwidth, integer addition, floating-point arithmetic operations, address computation, and branching and iteration. On the other extreme, intrinsic functions show marginal improvement. This is because normally the same libraries are used at all levels optimization. In fact, the average execution time for intrinsic functions on the Sparcstation 1+ increases with the level of optimization, and on the MIPS M/2000 the average time at the maximum level of optimization is larger than on other two cases.

The way we measure a particular intrinsic function is by running two experiments in which the only difference between them is that one contains a call to the function we are interesting in characterizing. At the lowest level of optimization the machine instructions executed by both experiments differ only by a call to the library function. With optimization, however, the experiment that contains the call has a few extra instructions more than the other experiment in addition to the call, because the compiler is prevented from applying some optimizations by the presence of the call. The net result is that even when the experiments run faster as we increase the level of optimization, the differential time contributed by the function call increases with the optimization level. Hence, the execution time we measure includes the *nominal time* of the intrinsic function plus the 'loss' of optimization. This *effective time*, however, is a better characterization of what occurs in real programs where the presence of a subroutine call also affects the effectiveness of the optimizer.

We can illustrate how different optimizations affect the performance of the abstract parameters by considering how the execution time needed to compute the address of an

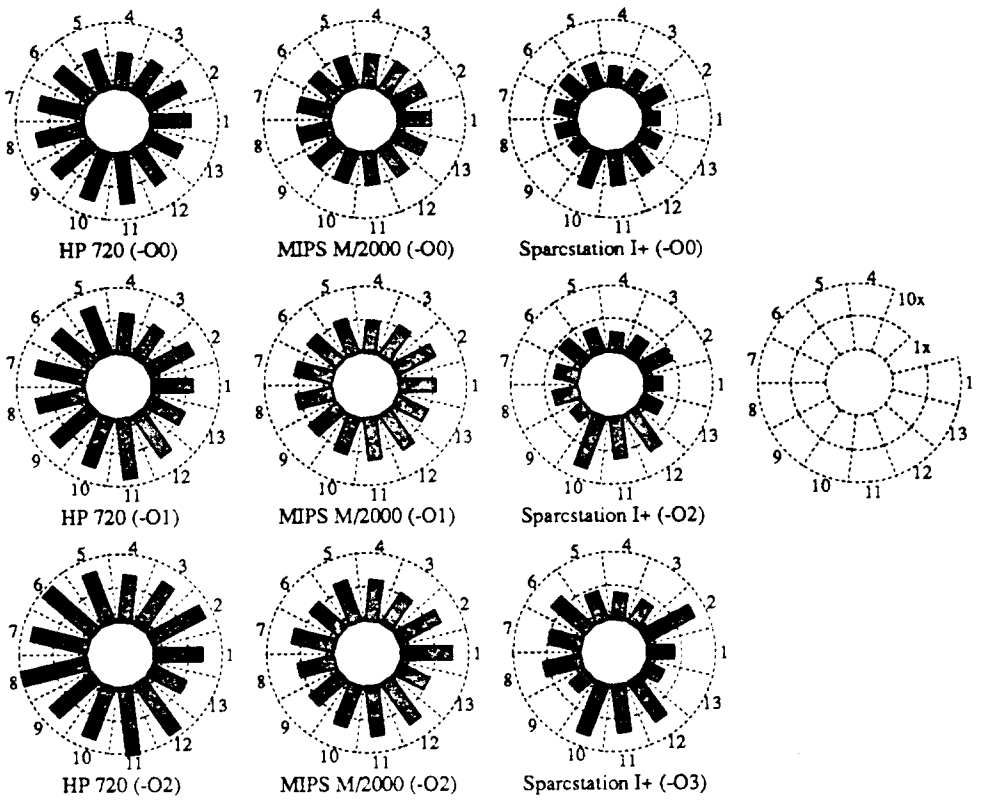


Figure 4.1: Performance shapes (pershapes) of different optimization levels. The thirteen dimensions correspond to the same thirteen parameters used in table 4.1. All dimensions are normalized with respect to the MIPS M/2000 with optimization level 0.

array element improves with optimization. Table 4.19 in Appendix 4.A shows that without optimization the execution time to compute the address of a 3-dimensional array element is between 3 to 5 times larger than the corresponding time for a 1-dimensional array. In general, the code generated by the compiler when it encounters a n-dimensional array reference contains n adds and n multiplies. By using several optimizations like strength reduction, backward code motion, and register allocation, it is possible to eliminate most of the computation inside loops by computing the address of the elements of the first iteration outside the loop, storing the values on registers and then updating these by adding the offset of consecutive reference to the same reference. The effect of these optimizations is evident in the results of table 4.19, where the effective time to compute the address of a 3-dimensional array element is only a factor of 2 larger than the 1-dimensional case and a factor of 5 smaller than the unoptimized measurements.

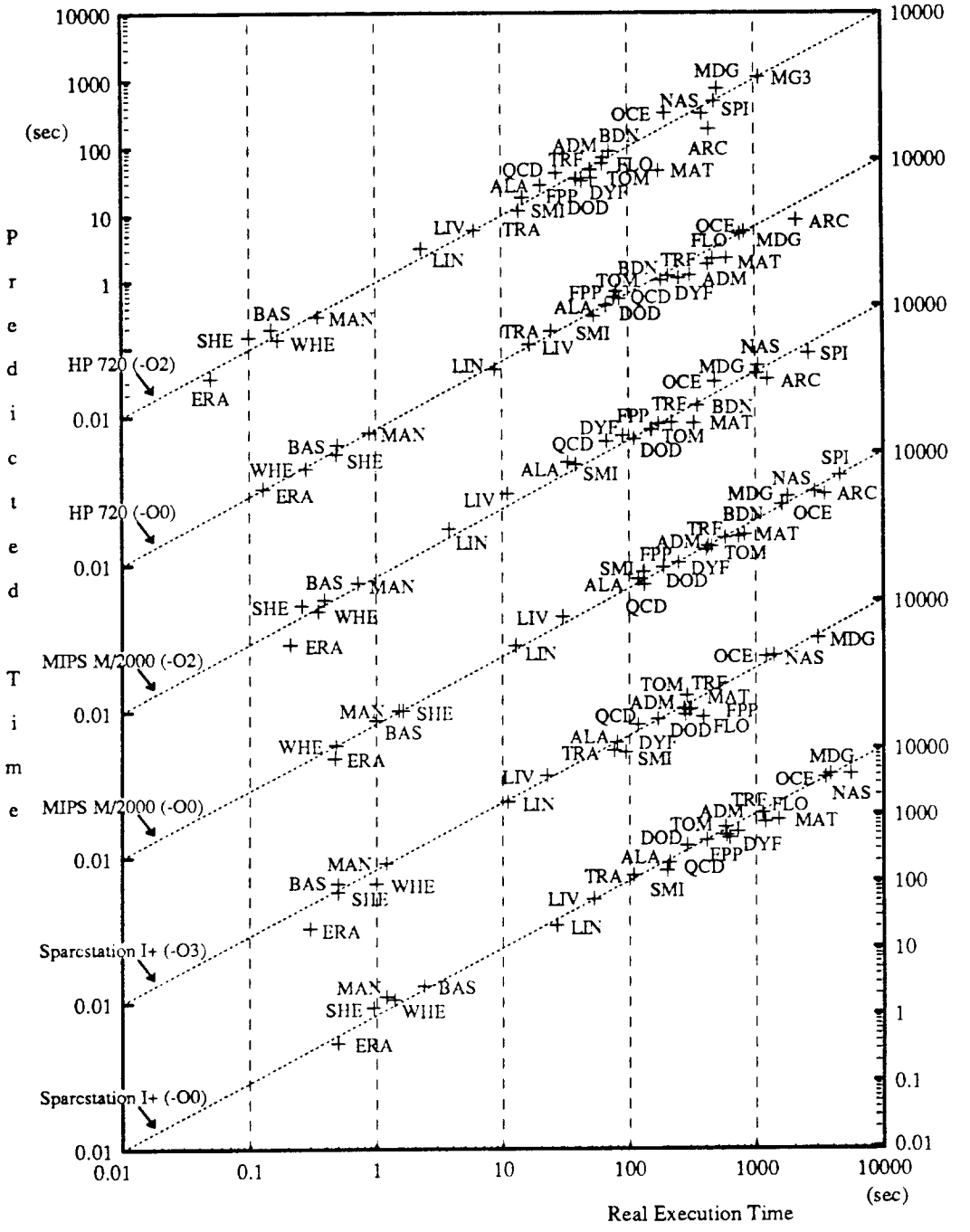


Figure 4.2: Each broken diagonal line, which corresponds to a particular machine and optimization level combination, shows the accuracy of the predictions compared to the real execution times. The left end point of each diagonal line maps to (0.01, 0.01) and the right end point to (10000, 10000). Points along the diagonal are of the form (T, T). All scales are in seconds.

4.4.6. Execution Time Prediction of Optimized Code

In this section we show that we can predict, reasonably well, the execution time of optimized programs when most of the optimization improvement comes from the application of invariant transformations. The experiments were done using a large set of Fortran programs taken from the SPEC and Perfect Club suites, and also some popular benchmarks. A description of the programs and their dynamic statistics can be found in Chapter 3. First, we compiled the programs using different levels of optimization and measured their respective execution times. At the same time we collected machine characterizations for the different levels of optimization. Using machine characterizations and the dynamic statistics of the programs we predicted the expected execution times. Our assumption that most of the improvement produced are invariant is reflected in the fact that the dynamic statistics of a program are the same with or without optimization, as reflected in equation 4.3.

In figure 4.2 we show the comparison between the real and predicted execution times for both optimized and unoptimized programs. For each graphs the vertical distance to the diagonal represents the error of the prediction. Because the scale is logarithmic, the errors appear smaller than they are. Nevertheless, it is clear from the figure that the predictions even at the maximum optimization level match relatively well the actual execution times. Tables 4.21-4.23 on Appendix 4.B gives the exact execution times and relative errors. Summaries of the predictive errors, by machine and program, are presented in tables 4.3-4.4. In both tables we indicate the average and root mean square (RMS) errors of the predictions at the minimum and maximum optimization levels. The RMS error is the square root of the average of the square of the individual errors. As expected the magnitude of the error increases with the optimization level. However, this increase is relative small with an average error of less than 11% in all cases. The root mean square error of all programs increases from 21% to 32%. A similar increase can be observed across machines.

	Minimum opt level		Maximum opt level	
Machine	Average	RMS	Average	RMS
HP-9000/720	−8.51 %	21.84 %	+3.42 %	35.60 %
MIPS M/2000	+1.95 %	16.81 %	+10.64 %	33.67 %
Sparcstation 1+	−7.52 %	22.87 %	−6.03 %	25.34 %

Table 4.3: Summary of execution time errors by machine at the minimum and maximum levels of optimization. RMS represents the root mean square error. The plus (negative) sign for average errors indicate that the predictions were above (below) the real execution times.

Figure 4.2 (see also tables 4.21-4.23) clearly shows that some programs benefit more than others from optimization. For example, the execution time improvement of *WHETSTONE* on the four machines is only 20 percent; the smallest of all benchmarks. The reason for this minor change is the relatively large number of intrinsic functions executed by the program. As we mentioned above, intrinsic functions are not affected by the optimization level. Therefore, even when only 2 percent of the abstract operations are intrinsic functions, their contribution to the total execution time increases with optimization. On the average, intrinsic function account for 25 and 40 percent of the total time

for the unoptimized and optimized versions respectively. That the value is significantly larger when optimization is enabled is a direct consequence of Amdahl's law, since the operations least affected by optimization tend to determine more the total execution time at higher optimization levels.

Program	Minimum Opt. Level		Maximum Opt. Level	
	Average	RMS	Average	RMS
Doduc	+4.10 %	6.76 %	−14.06 %	16.93 %
Fpppp	+5.93 %	11.74 %	−19.39 %	29.97 %
Tomcatv	−11.92 %	13.54 %	−18.44 %	21.60 %
Matrix300	−37.87 %	39.00 %	−46.00 %	51.33 %
Nasa7	−18.01 %	19.98 %	−4.14 %	12.30 %
Spice2g6	+11.87 %	17.36 %	+17.66 %	35.87 %
ADM	−18.17 %	22.73 %	−7.40 %	7.43 %
QCD	+30.72 %	31.04 %	+43.98 %	54.11 %
MDG	+12.15 %	15.43 %	+3.23 %	13.43 %
TRACK	+16.71 %	17.16 %	−14.78 %	15.54 %
BDNA	−13.18 %	14.27 %	+0.32 %	15.19 %
OCEAN	+3.45 %	4.26 %	+45.84 %	48.85 %
DYFESM	−25.76 %	28.62 %	+10.40 %	27.87 %
ARC2D	−35.28 %	35.63 %	−26.75 %	35.38 %
TRFD	−22.04 %	26.37 %	−8.44 %	15.31 %
FLO52	−19.50 %	22.86 %	+34.12 %	47.42 %
Alamos	+2.71 %	11.15 %	+27.69 %	34.48 %
Baskett	+16.33 %	16.58 %	+24.42 %	25.28 %
Erathostenes	−14.58 %	18.54 %	−45.02 %	47.01 %
Linpack	−6.25 %	15.74 %	+23.01 %	31.54 %
Livermore	+12.41 %	17.16 %	+21.98 %	31.45 %
Mandelbrot	+6.17 %	8.01 %	+1.65 %	10.48 %
Shell	+6.21 %	21.60 %	+33.60 %	39.38 %
Smith	−8.11 %	19.86 %	+1.27 %	28.78 %
Whetstone	−11.82 %	16.87 %	−22.69 %	25.58 %
Totals	−4.55 %	20.59 %	+2.48%	31.89 %

Table 4.4: Summary of execution time errors by program at the minimum and maximum levels of optimization for the programs on table 4.3. The real and predicted execution times are given in tables 4.19-4.21 in Appendix 4.B. RMS represents the root mean square error.

By modeling the execution time of a program using the abstract machine model in combination with the tools we have developed, we can get an understanding of how much optimization really affects the execution time of a program across many machines. We talk in more detail about this in §4.4.9.

4.4.7. Accuracy in Predicting the Execution Time of Optimized Programs

Our assumption that most of the performance improvement obtained from optimization is due to invariant optimizations is a simplification which is not necessarily valid on all programs. Nevertheless, the results of the previous section show that for most programs the assumption holds. In table 4.5 we compare the distribution of errors for both non-optimized and optimized programs. As expected, at the maximum level of optimization, the average errors are larger than in the non-optimized case, as it is evident in the

number of programs that fall within a certain error distance. For the results shown on figure 4.2, table 4.5 shows that while 85% of the non-optimized predictions are within 30% of the real execution time, this value decreases to 68% for optimized programs. Moreover, almost 13% of the predictions have errors of more than 50%, while none of the non-optimized prediction have errors of that magnitude.

Table 4.5: **Error distribution for execution time predictions**					
level	< 5 %	< 10 %	< 15 %	< 20 %	< 30 %
no optimization	15 (22.06)	26 (36.76)	39 (54.41)	46 (64.71)	61 (85.29)
max optimization	11 (12.86)	19 (24.29)	28 (37.14)	36 (48.57)	47 (68.29)

level	> 30 %	> 40 %	> 50 %
no optimization	11 (14.71)	3 (4.41)	0 (0.00)
max optimization	27 (35.71)	17 (22.86)	10 (12.86)

Table 4.5: Error distribution for the predicted execution times with and without optimization. For each error interval, we indicate the number of programs having errors that fall inside the interval (percentages inside parenthesis). The error is computed as the relative distance to the real execution time.

If a program exhibits a significantly larger positive prediction error at the maximum optimization level than it does with no optimization, then it is probably the case that the error is the result of ignoring non-invariant optimizations. In table 4.4 we see that four programs are in this situation: *QCD*, *OCEAN*, *FLO52*, and *SHELL*. An analysis of the source code shows that optimizers, on these programs, are applying optimizations that are not invariant with respect to our program decomposition. For example, the following code excerpt from *QCD* contributes significantly to total execution time:

```
DO 2 I = 0, 2
  DO 2 P = 0, 2
    DO 2 J = 0, 2
      DO 2 Q = 0, 2
        DO 2 K = 0, 2
          IF (EPSILO(I+1,J+1,K+1) .NE. 0) THEN
            DO 3 R = 0, 2
              IF (EPSILO(P+1,Q+1,R+1) .NE. 0) THEN
                FAC = EPSILO(I+1,J+1,K+1) * EPSILO(P+1,Q+1,R+1)
                TOT(1) = TOT(1) + FAC * U1(1,3*I+P+1) * U2(1,3*J+Q+1) *
                                        U3(1,3*K+R+1)
                TOT(1) = TOT(1) - FAC * U1(2,3*I+P+1) * U2(2,3*J+Q+1) *
                                        U3(1,3*K+R+1)
                TOT(1) = TOT(1) - FAC * U1(1,3*I+P+1) * U2(2,3*J+Q+1) *
                                        U3(2,3*K+R+1)
                TOT(1) = TOT(1) - FAC * U1(2,3*I+P+1) * U2(1,3*J+Q+1) *
                                        U3(2,3*K+R+1)
                TOT(2) = TOT(2) + FAC * U1(1,3*I+P+1) * U2(1,3*J+Q+1) *
                                        U3(2,3*K+R+1)
                TOT(2) = TOT(2) + FAC * U1(1,3*I+P+1) * U2(2,3*J+Q+1) *
                                        U3(1,3*K+R+1)
```

```
                    TOT(2) = TOT(2) + FAC * U1(2,3*I+P+1) * U2(1,3*J+Q+1) *
                                            U3(1,3*K+R+1)
.                   TOT(2) = TOT(2) - FAC * U1(2,3*I+P+1) * U2(2,3*J+Q+1) *
.                                            U3(2,3*K+R+1)
                  ENDIF
                CONTINUE
              ENDIF
2     CONTINUE
```

This code contains many opportunities for the compiler to apply common subexpression elimination (`3*I+P+1`, `3*J+Q+1`, and `3*K+R+1`) and thus significantly reduce the execution time.

Common subexpression elimination in this context is not an invariant optimization as defined in §4.4.2. Replacing an arithmetic expression by a reference to a previously computed equivalent value eliminates the abstract operations involved and thus has the potential of distorting our predictions. This is what happens on *QCD* where all our predictions are greater than the real time; on two of the machines machines the errors are as high as 47% and 81% (tables 4.21 and 4.22). This indicates that the optimizers are effectively eliminating the extraneous operations.

4.4.8. Improving Predictions in the Presence of Non-Invariant Optimizations

As we showed in the previous section, it is easy to identify which parts of the program have the potential of violating our assumption of optimization invariance and thus use this information to compute better execution estimates. We illustrate this by using the previous example. By applying common subexpression elimination optimizations to the code we obtain the following equivalent version:

```
DO 2 I = 0, 2
  DO 2 P = 0, 2
    DO 2 J = 0, 2
      DO 2 Q = 0, 2
        DO 2 K = 0, 2
          IF (EPSILO(I+1,J+1,K+1) .NE. 0) THEN
            DO 3 R = 0, 2
              IF (EPSILO(P+1,Q+1,R+1) .NE. 0) THEN
                FAC = EPSILO(I+1,J+1,K+1) * EPSILO(P+1,Q+1,R+1)
                I3 = 3 * I + P + 1
                J3 = 3 * J + Q + 1
                K3 = 3 * K + R + 1
                T11 = U1(1,I3) * U2(1,J3)
                T12 = U1(1,I3) * U2(2,J3)
                T21 = U1(2,I3) * U2(1,J3)
                T22 = U1(2,I3) * U2(2,J3)
                U31 = U3(1,K3)
                U32 = U3(2,K3)
                T111 = T11 * U31
                T112 = T11 * U32
                T121 = T12 * U31
                T122 = T12 * U32
```

```
                    T211 = T21 * U31
                    T212 = T21 * U32
                    T221 = T22 * U31
                    T222 = T22 * U32
                    TOT(1) = TOT(1) + FAC * (T111 - T221 - T122 - T212)
                    TOT(2) = TOT(2) + FAC * (T112 + T121 + T211 - T222)
                  ENDIF
                CONTINUE
              ENDIF
2       CONTINUE
```

Here the value of common subexpressions is computed once and stored in variables *I3*, *J3*, and *K3*[4]. In a similar way, we can eliminate other common subexpressions and in this way reduce the number of integer operations from 60 to 15 and the floating-point operations from 37 to 23.

Because common subexpression elimination is a machine-independent transformation, after applying these optimizations by hand we are left with an equivalent program whose execution time can be predicted in the same way as in the original program. After doing this we found that on all machines the prediction errors were less than 30%. By distinguishing the invariant and non-invariant optimizations, we can assess the performance impact of each, because the performance improvement due to non-invariant optimizations is equal to the difference between our prediction, considering only invariant optimizations, and the real execution time.

4.4.9. Amount of Optimization on Benchmarks

By comparing the execution times before and after optimization on different compilers we can measure how much potential optimization exists in programs. This information is quite useful in the development of a new optimizing compiler, as it gives us a way of measuring its effectiveness. It also helps users in understanding which are the characteristics of the programs that allow optimizers to produce the highest speedups. In table 4.6 we show the program speedup achieved by each optimization level with respect to the non-optimized execution time. We also give the ranking of the programs for the maximum level of optimization observed on each machine. In the same way, the geometric mean of each program's speedup is computed using the results from each machine obtained at the maximum level of optimization.

In §4.3 we mentioned that previous studies on the effectiveness of optimizing compilers for languages like C, Pascal, and PL/1 report speedups of less than a factor of 2. The results in table 4.6, however, show that at the maximum level of optimization the speedups observed on Fortran programs are larger than 2 with some programs experiencing speedups of more than a factor of 5.

The results of table 4.6 show that speedups on *FLO52*, *DYFESM*, *TRFD*, *ARC2D*, and *SHELL* are the highest of all programs, while those of *DODUC*, *FPPPP*, *TRACK*, *MDG*, and *WHETSTONE* are the lowest. The analysis of the source code shows that the

---

[4] Although *I3* and *J3* are invariant with respect to the induction variables of the two innermost loops, it is not profitable to move the code outside the loops because the two IFs eliminate a large fraction of the innermost iterations.

program	HP 720		MIPS M/2000		Sparcstation 1+		Geom. Mean
	-O1	-O2	-O1	-O2	-O2	-O3	Max. Opt.
Doduc	1.307	2.123 21	1.255	1.701 21	1.439	1.468 20	1.744 21
Fpppp	1.344	2.000 22	1.222	1.437 23	1.479	1.541 19	1.642 22
Tomcatv	1.504	3.497 10	1.445	2.994 10	1.866	1.927 16	2.722 14
Matrix300	1.377	3.413 11	1.263	2.475 14	3.788	4.854 2	3.448 7
Nasa7	1.477	3.318 14	1.300	2.817 11	3.759	3.953 3	3.331 9
Spice2g6	1.345	2.560 19	1.250	1.739 19	1.231	1.462 21	1.867 20
ADM	1.305	4.000 6	1.372	–	2.506	2.646 10	3.253 10
QCD	1.374	2.793 16	1.351	1.957 18	1.443	1.621 18	2.069 18
MDG	1.215	1.698 24	1.250	1.701 20	1.208	1.238 25	1.529 24
TRACK	1.316	1.786 23	1.377	1.700 22	1.318	1.403 23	1.621 23
BDNA	1.414	2.890 15	1.381	2.088 16	1.237	1.440 22	2.056 19
OCEAN	1.370	3.891 7	1.408	3.344 8	2.066	2.924 8	3.363 8
DYFESM	1.468	6.993 3	1.335	4.525 3	4.367	5.263 1	5.502 2
ARC2D	1.340	4.878 5	1.368	3.417 7	2.118	3.606 6	3.917 5
TRFD	1.664	7.143 2	1.361	4.338 4	3.690	3.891 5	4.940 3
FLO52	1.460	8.333 1	1.360	6.008 2	3.610	3.937 4	5.820 1
Alamos	1.397	3.344 12	1.311	3.571 5	1.362	2.558 11	3.126 11
Baskett	1.316	3.333 13	1.370	2.564 13	2.331	2.801 9	2.882 13
Erathostenes	1.300	2.597 18	1.305	2.237 15	1.667	1.667 17	2.132 17
Linpack	1.600	3.831 8	1.410	3.344 9	2.584	3.268 7	3.472 6
Livermore	1.473	2.703 17	1.570	2.725 12	2.045	2.326 12	2.578 15
Mandelbrot	1.348	2.545 20	1.429	2.083 17	2.000	2.000 15	2.197 16
Shell	1.634	4.902 4	1.592	6.289 1	1.357	2.093 14	4.011 4
Smith	1.350	3.597 9	1.282	3.472 6	2.000	2.105 13	2.973 12
Whetstone	1.218	1.647 25	1.200	1.372 24	1.300	1.300 24	1.432 25
Geom. Mean	1.392	3.271	1.348	2.665	1.973	2.296	2.722

Table 4.6: Optimization speedups under different optimization levels. Each speedup is computed by taking the ratio between the nonoptimized and optimized execution times. The last column gives the geometric mean of the machine speedups obtained at the maximum level of optimization. The small number on the right of each speedup indicates its relative magnitude, with the numeral 1 representing the largest speedup. Program *ADM* did not execute correctly on the MIPS M/2000 at the maximum optimization.

programs on each group share similar characteristics. For example, the size of most time-consuming basic blocks of the programs with the highest speedups is quite small. These consist of a few arithmetic statements where most of the operands are elements of multi-dimensional arrays. Most of the optimization improvement comes from collapsing the computation of the array addresses, good register allocation, and eliminating loads and stores of temporary values.

The programs with the smallest speedups are different. They tend to have substantially larger basic blocks. For example, the largest basic block on *FPPPP* has 590 lines of mostly scalar code. Here register files having as many as 32 or 64 registers cannot keep most of the variables in registers between their definition and use. Furthermore, on these programs, most of the operands are either scalars or one-dimensional arrays, so address collapsing, the elimination of time consuming address calculations in multi-

dimensional arrays, does not produce very much improvement. They also tend to execute a larger number of intrinsic functions whose execution is mostly unaffected by optimization. This is the case for *MDG* and *WHETSTONE*.

machines	Coefficient of Correlation	level of significance	Spearman's Rank Correlation	level of significance
HP 720 and MIPS M/2000	0.8677	.0002	0.9417	.0003
HP 720 and Sparc 1+	0.7390	.0009	0.7954	.0012
MIPS M/2000 and Sparc 1+	0.5656	.0070	0.7652	.0020

Table 4.7: Coefficient of correlation and Spearman's rank correlation of pairwise optimization speedup results. The statistical significance level gives the probability that there is not a positive correlation involved.

It is dangerous to draw conclusions on the effectiveness of the different optimizers from the speedup results of table 4.6. The overall speedup is as much a function of the quality of the non-optimized object code as it is of the optimizer, since it is always possible to improve the overall speedup by generating worse non-optimized code. This is true on the HP 720, where the overall speedup is significantly higher because the compiler generates native code for the 700 series only at the maximum level of optimization. For compatibility reason, the object code on low levels of optimization is for the 800 series which is emulated on the 720 in software.

Program *SHELL* is a good example of how the quality of nonoptimized code affects the amount of speedup observed on different programs. This benchmark is one of the few integer programs in our suite and consists of sorting an array of integers using the Shell sort algorithm. As Table 4.6 indicates, *SHELL* is the program with the largest speedup on the MIPS M/2000 (6.289), and is one of the top four on the HP 720 (4.902). On the other hand, the speedup on the Sparcstation 1+ is significantly lower (2.093). The speedup here is even lower than the overall improvement on all program (2.296). The reason for this is not because the Sun's optimizer fails to improve the code, but is due to the fact that under no optimization the MIPS M/2000 and HP 720 generate code of less quality. This is evident on the number of machine instructions generated by each compiler at different optimization levels for the most time consuming part of the program. On the Sparcstation 1+ the number of instructions changes from 41 without optimization to 23 with optimization. The corresponding numbers on the MIPS M/2000 are 74 and 16. The respective static speedups are 1.873 and 4.625. This discrepancy is clearly present in the actual execution times. Benchmark results normally rate the MIPS M/2000 as being at least 50% faster than the Sparcstation 1+. The results for *SHELL* on tables 4.22 and 4.23 (Appendix 4.B), however, indicate that at low levels of optimization the Sparcstation 1+ is faster than the MIPS M/2000 (0.95 sec vs. 1.64 sec and 0.70 sec vs. 1.03 sec). It is only at the maximum optimization level where the MIPS M/2000 exhibits a smaller execution time (0.43 sec vs. 0.26 sec). Therefore, concluding which optimizer is better by only comparing the speedup produced on programs can be misleading.

We can test if there is positive correlation between the amount of speedup produced by pairs of optimizers on these benchmarks, by computing either the coefficient of correlation or the Spearman's rank correlation coefficient. Table 4.7 gives the value of the

coefficients and the level of significant for the three combinations. Both statistics show positive correlation below the level of .007, with the largest correlation between the HP and the MIPS results under the .0003 level.

4.5. The Characterization of Compiler Optimizations

In the last section we discussed how to measure and predict the performance improvement produced by optimizers. In this section we characterize the set of optimizations that compilers can apply. We want to assess not only how fast programs execute under some level of optimization, but we also want to identify which standard transformations can be performed by the compiler and in which contexts. The context indicates whether a particular optimization can be performed on all data types or only on a subset of them. We are also interested in knowing if the optimization is detected when it is present inside a basic block and/or across basic blocks. In what follows, we refer to a local optimization as one that is detected inside a basic block and a global optimization when it spans more than one basic block.

As we mentioned in §4.3, most performance studies on optimizers have been done by compiler writers, and the emphasis there has been in showing that optimizers indeed improve performance, but little attention has been paid to detecting how well optimizations can be performed. We think performance analysts and compiler writer should attempt to measure the effectiveness of optimizers and relate their results to improvements in program execution. In this section we attempt to do this.

Our approach to detecting optimizations is similar in some respects to the way we characterize basic machine performance (see §2.5). We have developed a Fortran program consisting of some number of tests which detect individual optimizations under different conditions. We test for each optimization, whether it can be applied on integers, floating-point, or mixed (integer and floating-point) expressions. In those cases where it makes sense to make the distinction we test local and global optimizations

We detect whether a particular optimization is applied or not by running experiments which show a difference in their execution time only when the optimization is performed. In this way we avoid having to analyze the assembler code. Each optimization test consists of two almost identical experiments where the only difference between them is that the second experiment contains a potential optimization. If the optimization of the second experiment is not applied, then the execution time of both experiments should be the same. If it is applied, however, the execution time of the second experiment should be significantly smaller. Each experiment is repeated 20 times to collect a large statistic and a post-processor computes the average execution times of each experiment ($\hat{\mu}_1$ and $\hat{\mu}_2$) and the significance level of the following statistical test: $\hat{\mu}_1 \leq \hat{\mu}_2$. If there is sufficient evidence to reject the null hypothesis, then we can assume that the optimization was detected by the optimizer. The level of significance represents the probability that random variations in our measurements would appear as supporting the conclusion that the optimization was detected when in fact it was not. Nevertheless, in all the results we have double checked that the optimizations were applied by analyzing the assembler code.

Figure 4.3 illustrates the basic structure of our experiments. This example is one of the tests for detecting local dead code elimination. The two corresponding innermost

```
      DO 2 J = 1, 20
          T0 = SECOND (P)
          DO 1 I = 1, ITER
              W1 = X * W1 + (A * (B * C))
              W2 = X * W2 + ((A * B) * C)
              W3 = Y * W3 + ((C * A) * B)
              A = XA - A
              B = XB - B
              C = XC - C
              W1 = Y * W1 + (A * (B * C))
              W2 = Y * W2 + ((A * B) * C)
              W3 = X * W3 + ((C * A) * B)
              A = XA - A
              B = XB - B
              C = XC - C
1         CONTINUE
          T(J) = SECOND (P) - T0
2     CONTINUE
```

```
      DO 4 J = 1, 20
          T0 = SECOND (P)
          DO 3 I = 1, ITER
              W1 = X * W1 + (A * (B * C))
              W2 = X * W2 + ((A * B) * C)
              W3 = Y * W3 + ((C * A) * B)
              A = XA - A
              B = XB - B
              C = XC - C
              W1 = X * A  + (A * (B * C))
              W2 = Y * W1 + ((A * B) * C)
              W3 = Y * W2 + ((C * A) * B)
              A = XA - A
              B = XB - B
              C = XC - C
3         CONTINUE
          T(J) = SECOND (P) - T0
4     CONTINUE
```

Figure 4.3: A particular experiment to detect dead code elimination. On the left hand experiment all definitions inside the innermost loop are used at least once, while on the other experiment the topmost definitions of $W1$, $W2$, and $W3$ are not used. The three definitions can be eliminated by the optimizer.

loops are almost identical with only one difference: in the right hand side experiment, the first set of definitions of variables $W1$, $W2$, and $W3$ are not used subsequently by any other statement. Furthermore, these definitions are killed by the second set of definitions to the same variables. Formally, we say that there are no forward dependencies having as source the first definitions. Hence, if the compiler can detect this, it can eliminate their computation. In contrast, this does not occur on the left side where every definition is the source of a forward dependency. Eliminating the first three statements on the second experiment reduces the execution time between 25 and 50% on most machines; this is detected by the statistical test.

4.5.1. Standard Optimizations Detected

The types of optimizations we are interesting in detecting are machine-independent. This is consistent with our methodology that permits comparing different machines, and in this case their compilers, by providing a unified representation of the execution while ignoring machine-level details. Machine-dependent optimizations, like those performed by peephole optimizers, are invariant with respect to our model. They consist on finding small sequences of machine instructions[5] and replacing them with shorter and faster sequences [Davi80, Tane82, Kess86], thus the performance improvement resulting from these optimizations is detected by the machine characterizer.

[5] Some portable peephole optimizers work on low level intermediate representations of the program.

Most machine-independent optimizations detected by current optimizers have been known for many years. A good reference describing these optimizations and the general problem of compiler optimization is [AhoA86]. [Chow83 and BalH86] describe how optimizations are implemented in a real compiler. The following sections discuss the optimizations we currently detect.

4.5.1.1. Constant Folding

The idea behind this optimization is to replace symbolic constants by their actual values and evaluate the resulting expressions at compile time. If during this process other variables get a recently computed constant value, then their values are again propagated until no more constant expressions remain. The current emphasis on program modularity and portability has increased the use of symbolic constants and correspondingly the importance of applying this optimization.

4.5.1.2. Common Subexpression Elimination

It consists in identifying two or more identical subexpressions in a region without an intervening definition of any of the relevant variables. By computing the subexpression at the beginning and replacing subsequent computations by a reference to a temporary variable holding the result of the computation, the execution time is reduced.

4.5.1.3. Code Motion

This optimization is especially important in highly nested loops. The idea is to identify expressions or statements which are invariant with respect to the induction variables of the loop and are computed unnecessarily on every iteration, and to move them out of the loop. The performance improvement obtained is proportional to the number of times the loop is executed. In scientific programs this is one of the most important optimizations along with address collapsing. Both of them are used in conjunction in the optimization of array references.

4.5.1.4. Dead Code Elimination

In some programs there are pieces of code which can be statically proved never to be executed or whose execution does not have any semantic effect on the final computation. This code can be safely eliminated by the compiler to reduce the execution time and/or the object code size. Although this optimization does not appear very promising, as most programmers do not deliberately write needless code, occasionally some statements become dead as the result of applying other optimizations.

4.5.1.5. Copy Propagation

Some optimizations like common subexpression elimination, code motion, and address collapsing create large number of copy instructions, e.g., `x = y`. By replacing uses of the copy with the original variable it is possible to simplify the code and expose new optimizations. Optimizations that benefit from copy propagations are common subexpression elimination and register allocation.

4.5.1.6. Address Collapsing

This consists of eliminating expensive address computations of multi-dimensional array elements in innermost loops by precomputing outside the loop the addresses of the elements referenced in the first iteration and updating their values by adding a constant in subsequent iterations. This optimization is based on the observation that in the majority of nested loops the sequence of machine addresses associated with a specific array reference form an arithmetic progression, which is completely determined by the first value and the increment.

4.5.1.7. Strength Reduction

This optimization is a generalization of address collapsing as it attempts to replace a time-consuming computation with an equivalent but faster one. One example is replacing an exponentiation having a small integer exponent which is known at compile time with a series of multiplications. On array references, the combination of strength reduction and code motion makes it possible to collapse address computations.

4.5.1.8. Subroutine Inlining

Subroutine and function calls are abstractions of computations repeatedly used in different part of the program. The use of calls introduce extra time overhead and hide potential optimizations present at the site of the call. By inlining leaf procedures after a proper instantiation of the formal arguments, it is possible to eliminate the call and uncover new optimizations. Although most optimizers claim that they do subroutine inlining, they tend to differ substantially in the amount of integration they perform.

4.5.1.9. Loop Unrolling

The idea here is to expand several iterations of the loop into a single basic block and hence expose new optimization opportunities. In loops when the amount of computation is very small, loop unrolling effectively reduces the impact of the loop overhead.

In this chapter we have concentrated on scalar optimizations. But in addition to the above optimizations, there are others program transformations which have been designed to exploit vector and parallel hardware. Some of these like, loop distribution, loop interchange, loop fusion, loop peeling, and stripmining are used to help compilers in recognizing hardware vector instructions [Paud86, Alle87, Hira91][6]. A description of a large test suite and evaluation of vectorizing Fortran compilers can be found in [Call88].

---

[6] *Loop distribution* separates independent statements inside a single loop into multiple loops which can be optimized independently [Hira91]. *Loop fusion* transforms two adjacent loops into a single loop with the goal of reducing the overhead of loops. *Loop collapsing* transforms two nested loops into a single one with the attempt of increasing the effective vector length. *Stripmining* transforms a single loop into two nested loops, when the number of iterations of the original loops is much larger than the number of elements in the vector registers [Paud86]. Loop Fusion and loop collapsing are, respectively, the inverse transformations of loop distribution and stripmining.

Machine	Compiler	Name/Location
VAX-11/785	BSD Unix F77 1.0	arpa.berkeley.edu
MIPS M/2000	MIPS F77 2.0	mammoth.berkeley.edu
Sparcstation 1+	Sun F77 1.3	heffal.berkeley.edu
VAX-11/785	Ultrix Fort 4.5	pioneer.arc.nasa.gov
Amdahl 5860	Amdahl F77 2.0	prandtl.nas.nasa.gov
CRAY Y-MP/8128	CRAY CFT77 4.0.1	reynolds.arc.nasa.gov
IBM RS/6000 530	IBM XL Fortran 1.1	coyote.berkeley.edu
Motorola M88K	Motorola F77 2.0b3	rumble.berkeley.edu

Table 4.8: List of machines with their respective Fortran compilers.

4.5.2. Optimization Results

We have run our experiments on several optimizing compilers and under different levels of optimization. In table 4.8 we give the list of machines along with their corresponding compilers. The complete results are presented in tables 4.24-4.29 in Appendix 4.C, while tables 4.9-4.11 summarize the same information. The Appendix's tables indicate for each optimization and different context (integer, float and mixed type expressions), whether the optimization was detected or not, while tables 4.9-4.11 give an overall rating based on the individual experiments. In our experiments we make a distinction between local and global optimizations. A local optimization is one in which the optimization and all the information needed for its detection is found within a single basic block. In contrast, detecting a global optimization requires the propagation of control and data flow information across basic block boundaries.

Local and global optimization summary results are given in table 4.9 and 4.10 respectively, while table 4.11 refers to optimizations where the distinction between local and global does not apply. On these tables, a 'yes' or 'no' entry indicates that the optimizer was able to detect all or none of the optimizations in the tests. The other two alternatives, two out of three and one out of three, correspond to entries 'partial' and 'marginal'.

The results give evidence to our assertion that some compilers are only able to apply optimizations under certain conditions and not on all cases. However, without a good knowledge about which optimizations are the most important, in terms of how much performance improvement they provide and how frequently they are found in applications, it is difficult to conclude whether, for example, implementing constant folding for floating-point expressions is something that all optimizers should do. The reader should keep this in mind in the following paragraphs where we compare the effectiveness of different optimizing compilers.

The optimization results on constant folding illustrates the difficulties in evaluating the effectiveness of an optimizer. While almost all the compilers are able to propagate integer constants inside a basic block, with the exception of the *f77* BSD Unix and Amdahl compilers, the situation is less clear for floating-point constant and global constant propagation. The Sun Fortran compiler does not apply constant propagation on floating-point or across basic blocks, while the *fort* Ultrix compiler from Digital implements constant propagation on all data types but only inside a basic block. On the MIPS compiler, constant propagation is applied in the local and global context only for

compiler	constant folding	common subexpr elim	code motion	copy propagation	dead code elimination
BSD Unix F77 1.0	no	partial	marginal	partial	no
Mips F77 2.0 -O2	partial	yes	yes	partial	yes
Mips F77 2.0 -O1	marginal	yes	no	marginal	no
Sun F77 1.3 -O3	marginal	yes	yes	no	yes
Sun F77 1.3 -O2	marginal	yes	yes	no	partial
Sun F77 1.3 -O1	no	no	no	no	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 4.9: Summary of local optimization. Each entry summarizes how well the optimizer detects the optimization using integer, floating-point, and mixed data types in arithmetic expressions. These optimizations do not extend beyond a single basic block.

compiler	constant folding	common subexpr elim	code motion	copy propagation	dead code elimination
BSD Unix F77 1.0	no	no	marginal	no	no
Mips F77 2.0 -O2	partial	yes	yes	marginal	yes
Mips F77 2.0 -O1	no	no	no	no	no
Sun F77 1.3 -O3	no	yes	partial	no	yes
Sun F77 1.3 -O2	no	yes	partial	no	partial
Sun F77 1.3 -O1	no	no	no	no	no
Ultrix Fort 4.5	no	yes	yes	partial	yes
Amdahl F77 2.0	no	no	no	no	no
CRAY CFT77 4.0.1	yes	partial	partial	no	yes
IBM XL Fortran 1.1	partial	partial	yes	marginal	yes
Motorola F77 2.0b3	no	partial	no	no	no

Table 4.10: Summary of global optimizations. Each entry summarizes how well the optimizer detects the optimization using integer, floating-point, and mixed expressions. These optimizations cover more than one basic block.

integers. For floating-point, constants are propagated only if they are assigned to a constant, but not if they are the result of a constant expression. For floating-point, the value of a variable known at compile time is propagated only if the variable is assigned a constant value, but not if it gets the constant as a result of evaluating an expression.

Common subexpression elimination is successfully detected by most compilers in all contexts. Although the IBM *XLF* compiler identified almost all common subexpressions, it missed a couple which involved floating-point adds and multiplies. The reason is that the RS/6000 series provides, in addition to the normal add and multiply operations, a combined multiply-add instruction. In our experiments the compiler generated

compiler	strength reduction	address calculation	inline substitution	loop unrolling
BSD Unix F77 1.0	partial	marginal	no	no
Mips F77 2.0 -O2	yes	yes	marginal	yes
Mips F77 2.0 -O1	no	yes	no	no
Sun F77 1.3 -O3	partial	marginal	no	yes
Sun F77 1.3 -O2	partial	no	no	yes
Sun F77 1.3 -O1	no	no	no	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 4.11: Additional optimizations. These optimizations are tested using a single data type, as their application is not affected by this kind of context. Here *partial* and *marginal* have a different meaning than on tables 8 and 9. Instead of summarizing the results of several experiments, they represent the effectiveness of the optimizers on a single test.

for two occurrences of the same subexpression, a multiply followed by an add in one case, but a single multiply-add on the other case. As a result of this, it did not recognize that the two expressions were identical. Missing an optimization as a result of applying another, however, is in many cases acceptable, if the first optimization provides a better improvement.

Table 4.11 shows that our tests detected that three compilers have some ability to inline procedures, but only the CRAY *CFT77* compiler takes full advantage of it. In the case of MIPS *f77* 2.0, 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 [Chow86]. In contrast, a real inline substitution is done by the IBM *XLF* 1.1 compiler [O'Br90], but here the insertion of unnecessary extra code obscures optimizations that inlining should have exposed. Only the CRAY's *CFT77* compiler was abled to detect all optimizations present after proper inlining.

4.5.3. Correlation Between Different Optimizing Compilers

An interesting question to consider is whether different compilers correlate in their ability to improve the execution time of individual programs. If indeed there is a strong correlation between the amounts of optimization obtained by different compilers, then knowing how much one optimizer reduces the execution time of a program would allow us to estimate the reduction on the other optimizer. Thus, this would give us alternative way of predicting execution times that would work, not only for invariant optimizations, but also for non-invariant ones.

We executed twice a suite of programs on many of the optimizers presented in table 4.8. One run was made without optimization, and the other with optimization, For each program-compiler combination we measured the reduction in execution time produced

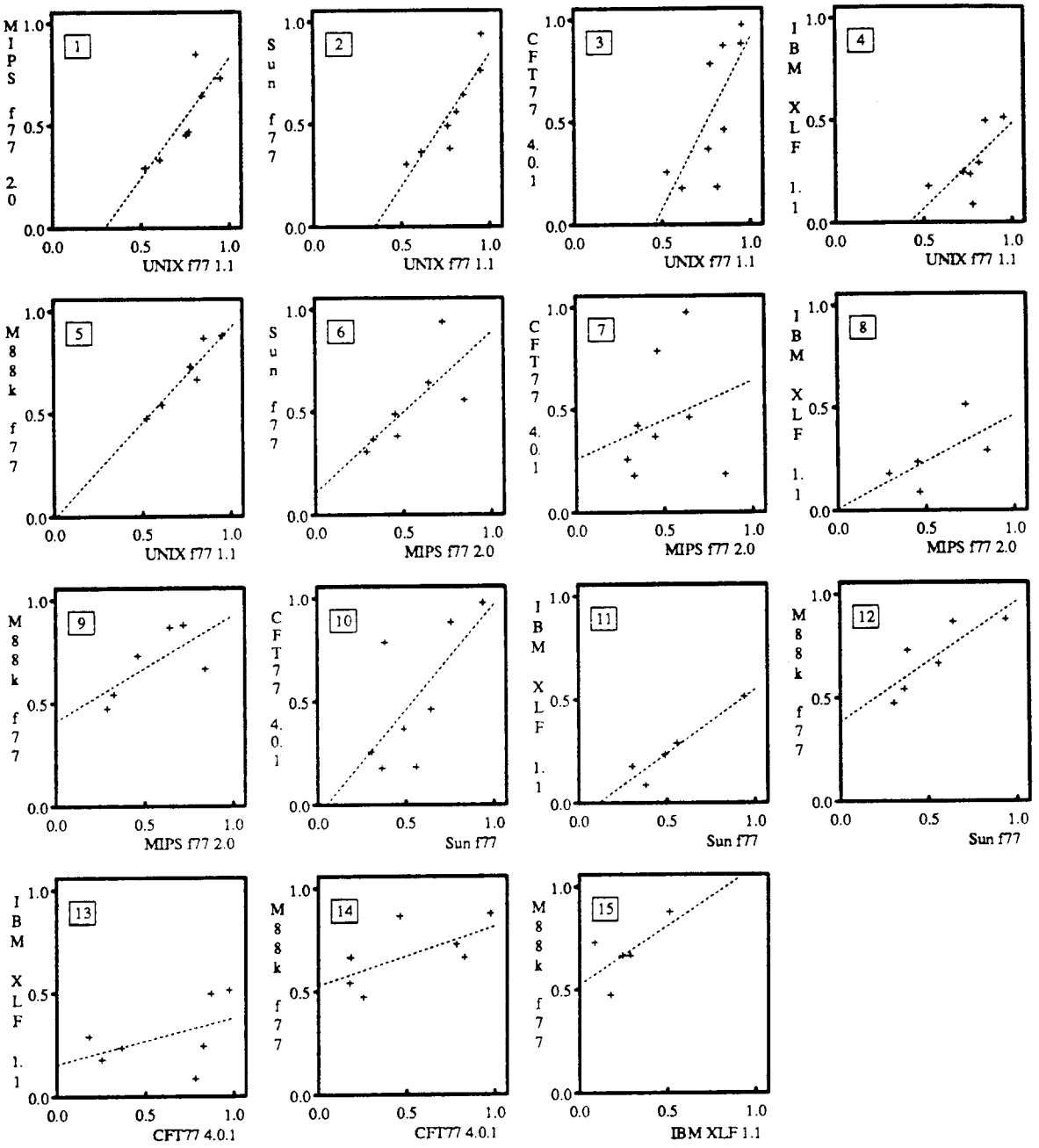


Figure 4.4: Correlation between execution time improvements of various optimizing compilers. Each graph includes the best linear fit.

by the optimizer. We then computed the coefficient of correlation between pairs of optimizers and the level of significance involved. In figure 4.4 we show the scattergrams and included on each one the best fit to the data. Table 4.12 gives the numerical values for the slope, y–intercept, correlation coefficient, and level of significance.

Compiler 1	Compiler 2	Obs.	Slope	y Intercept	Correlation coefficient	Level of significance
BSD Unix F77 1.0	MIPS F77 2.0	7	1.1883	–0.3600	0.8294	**0.020**
BSD Unix F77 1.0	Sun F77 1.3	8	1.2981	–0.4555	0.8987	**0.001**
BSD Unix F77 1.0	CRAY CFT77 4.01	9	1.6864	–0.7733	0.7376	**0.015**
BSD Unix F77 1.0	IBM XLF 1.1	7	0.8282	–0.3452	0.6892	0.040
BSD Unix F77 1.0	M88K F77 2.0b3	6	0.9912	–0.0547	0.9489	**0.001**
MIPS F77 2.0	Sun F77 1.3	7	0.7816	0.1064	0.7454	**0.025**
MIPS F77 2.0	CRAY CFT77 4.01	8	0.3829	0.2566	0.2699	0.300
MIPS F77 2.0	IBM XLF 1.1	5	0.4523	0.0086	0.6361	0.150
MIPS F77 2.0	M88K F77 2.0b3	6	0.5109	0.4106	0.6899	0.070
Sun F77 1.3	CRAY CFT77 4.01	8	1.0224	–0.0558	0.6864	0.040
Sun F77 1.3	IBM XLF 1.1	5	0.6152	–0.0686	0.9510	**0.007**
Sun F77 1.3	M88K F77 2.0b3	6	0.5805	0.3829	0.8325	**0.020**
CRAY CFT77 4.01	IBM XLF 1.1	7	0.2220	0.1555	0.4608	0.170
CRAY CFT77 4.01	M88K F77 2.0b3	7	0.2855	0.5375	0.6329	0.070
IBM XLF 1.1	M88K F77 2.0b3	5	0.5764	0.5298	0.6322	0.150

Table 4.12: Slope, y–intercept, correlation coefficient, and level of significance for pairs of compilers.

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. Unfortunately, the correlation across compilers does not appear to be strong enough to make this approach better than estimating the execution time using the concept of invariant optimizations.

4.6. Compiler Optimization and Benchmarks

In this section we discuss the main optimizations present in the SPEC benchmarks. We focus on the most time-consuming basic blocks as they provide most of the improvements observed in our measurements. In those programs where the amount of optimization is small, we describe some potential optimizations that future compilers might be able to apply as a result of current research on compiler technology. We also discuss some potential problems in the benchmarks that can be exploited by a clever compiler to artificially reduce the execution time.

4.6.1. Amount of Optimization in the SPEC Fortran Benchmarks

4.6.1.1. DODUC

This program along with *FPPPP* are the two SPEC program showing the least optimization improvement in the results of table 4.6. In contrast with *TRACK*, *MDG*, and *WHETSTONE*, the other three programs showing even less optimization potential, and where the main limitation is due to intrinsic functions, the limitations of *DODUC* and *FPPP* are of different nature. In these programs the most important basic blocks consist of a large sequence of scalar arithmetic expressions with very little reuse. In *DODUC* most of the computation does not reside in loops, so optimizations like strength reduction, address collapsing, code motion, and register allocation, which are effective in other

scientific programs, are not profitable here. Furthermore, the execution time of the program is not determined by a few basic blocks, so optimizers are forced to do a good job on the whole program to significantly reduce the execution time. Therefore *DODUC* can be considered a good challenge to any optimizer, and it is not surprising that its SPECratio on almost all machines is consistently lower than the overall SPECmark even on those machines with very good floating-point performance [SPEC90a, SPEC90b].

Even if *DODUC* is a difficult benchmark to optimize, it has some problems which could be easily exploited by a clever optimizer by extending standard optimizing techniques. Consider the following procedure which is one of the most time-consuming basic blocks and accounts on the average for almost 5 percent of the execution time[7]:

```
      SUBROUTINE X21Y21 (X,Y)
      DOUBLE PRECISION X(21), Y(21)
      REAL XX(21), YY(21)
      DATA XX /0.2, 200., 400., 600., 800., 1000., 1500., 2000.,
     .         2500., 3000., 3500., 4000., 6000., 8000., 10000.,
     .         15000., 20000., 25000., 30000., 40000., 50000./
      DATA YY /0., 69.31, 120.68, 166.92, 210.12, 251.19, 347.43,
     .         437.34, 522.82, 604.92, 684.31, 761.46, 1053.22,
     .         1325.78, 1584.89, 2192.16, 2759.46, 3298.77, 3816.78,
     .         4804.5,5743.49/

      DO 1 I = 1, 21
         X(I) = XX(I)
         Y(I) = YY(I)
1     CONTINUE
      RETURN
      END
```

The code clearly shows that *XX* and *YY* are constant vectors and that the purpose of this subroutine is to copy their values into arrays *X* and *Y*. This procedure is executed almost 150,000 times. What is surprising is that vectors *X* and *Y* remain constant for the whole execution, that is, their values are never redefined outside this procedure; obviously there is no need to call the procedure more than once. This can be easily detected by inlining the subroutine at its only call site, where data flow analysis will detect that *X* and *Y* can be safely replaced by *XX* and *YY*. Doing this will reduce the execution time by around 5 percent. To detect this optimization opportunity, in addition to procedure inlining, requires extending copy and constant propagation optimization to include vectors as well as simple variables.

4.6.1.2. FPPPP

As we mentioned above, optimizing *FPPPP* is quite difficult, because of the structure and size of its basic blocks. Its most time-consuming block contains 3000 floating-point operations without a branch. It alone accounts for approximately 40 percent of the execution time. The block uses 653 different variables, so achieving a good allocation of

[7] The contribution of a basic block to the total time varies from machine to machine, thus the 5 percent represents only an approximation.

variables to registers is the most important optimization problem here. Furthermore, on the average a variable is referenced only 6 times with around 161 intervening references between two consecutive uses of the same variable. This means that each variable, on the average, has a lifetime of almost 900 references. On this block, a register set with an unlimited number of registers would eliminate close to 82% of all loads and stores[8]. As a comparison, the MIPS compiler, at the maximum optimization level, is capable of eliminating only 22% of the loads and stores, while Sun's optimizer eliminates only 11%. In the program there are other basic blocks with similar characteristics.

Given that this program's performance is strongly determined by the scalar floating-point performance of the machine, the only way to significantly improve its performance is by reducing the execution time of floating-point operations, namely, add and multiply, by increasing the size of the register file, and by improving the register allocation algorithm [Chai82, Chow84][9].

4.6.1.3. TOMCATV

This is a good benchmark for testing an optimizer, as its most time consuming loop contains ample opportunities for optimization. Some of these optimizations cannot be detected by many of today's best optimizing compilers. The loop also contains some statements which serve no function at all, and which can be eliminated only if the optimizer implements the correct optimizations. The following loop is responsible for approximately 60% of the total execution time:

```
      DO 250 I = I1P, I2M
         IP = I + 1
         IM = I - 1
         XX = X(IP,J) - X(IM,J)
         YX = Y(IP,J) - Y(IM,J)
         XY = X(I,JP) - X(I,JM)
         YY = Y(I,JP) - Y(I,JM)
         A  = 0.250 * (XY * XY + YY * YY)
         B  = 0.250 * (XX * XX + YX * YX)
         C  = 0.125 * (XX * XY + YX * YY)
         QI = 0.0
         QJ = 0.0
C        QI = A * 0.5
C        QJ = B * 0.5
         AA(I,M) = - B
         DD(I,M) = B + B + A * REL
         PXX = X(IP,J) - 2. * X(I,J) + X(IM,J)
         QXX = Y(IP,J) - 2. * Y(I,J) + Y(IM,J)
         PYY = X(I,JP) - 2. * X(I,J) + X(I,JM)
         QYY = Y(I,JP) - 2. * Y(I,J) + Y(I,JM)
```

[8] It is evident that a register file having at least as many registers as there are variables in a basic block makes it possible to generate the minimum number of loads and stores.

[9] It may be possible to also speed up the program by finding some way to do loads faster than issuing successive load instructions. For example, load multiple.

```
          PXY = X(IP,JP) - X(IP,JM) - X(IM,JP) + X(IM,JM)
          QXY = Y(IP,JP) - Y(IP,JM) - Y(IM,JP) + Y(IM,JM)
          RX(I,M) = A * PXX + B * PYY - C * PXY + XX * QI + XY * QJ
          RY(I,M) = A * QXX + B * QYY - C * QXY + YX * QI + YY * QJ
250     CONTINUE
```

First, consider the two statements above the comments; not only can they be moved out of the loop, but if their constant values are propagated, the two rightmost subexpressions of the last two statements (`XX * QI + XY * QJ` and `YX * QI + YY * QJ`) can be eliminated, as they reduced to zero. This can be done, however, only if the optimizer implements floating-point constant propagation. The results of §4.5.2 show that the Sun's Fortran compiler cannot detect this optimization. Our measurements indicate that if the Sun compilers were capable of eliminating the useless computations, then execution time of the Sparcstation 1+ would improve by 9 percent.

The most obvious way of optimizing this loop, and what most compilers are able to do, is to eliminate the address calculation of array elements. This is accomplished by applying address collapsing, strength reduction, and code motion. Once this is done, most of the improvement comes by eliminating as many loads and stores as possible. First, most elements of arrays *X* and *Y* are used twice in the loop, so they need to be loaded only once. Second, a good compiler may notice that all scalar variables are temporaries whose values do not need to be stored for the duration of the loop. After this we are still left with 18 loads and 4 stores per iteration. However, few optimizers can achieve this because of the limited number of registers available to them. For example, the MIPS Fortran compiler, which is one of the best compilers, cannot keep all temporaries in registers and is forced to make 26 loads and 10 stores per iteration. This is because the R2010 coprocessor has only 16 floating-point registers.

If the machine has more than 16 floating-point registers it can further eliminate loads and stores by using a novel optimization technique called *predictive commoning* [O'Br90]. The idea here is to identify a sequence of values used in an iteration which contains a subsequence which is reused in the next iteration as a successor of the same sequence. An example of this is sequence `X(IP,J)`, `X(I,J)`, and `X(IM,J)`, whose first two elements are reused in the next iteration. The optimization consists of eliminating all the loads of the reused values by moving them to the registers where their successors reside at the end of each iteration. In the code there are six such sequences, so we can eliminate 12 of the 18 loads. Although this introduces 12 register to register move instructions, these can be eliminated by unrolling the loop.

The SPECratio of the IBM RS/6000 on this benchmark is much higher than that of the other benchmarks; a factor of three with respect to the overall SPECmark. This is due to the exceptional ability of the compiler to detect most of the optimizations we described and to the 32 floating-point registers in the machine. The IBM *XLF* compiler is capable of eliminating most of the loads by reusing registers and applying predictive commoning.

4.6.1.4. MATRIX300

It has been documented that this benchmark is completely dominated by a single basic block, which accounts for 99% of the execution time [Saav90]. This basic block implements the SAXPY vector to vector operation `Y[1,1:N] = Y[1,1:N] + A * X[1,1:N]`. This subroutine is used in the program to compute eight different variations of matrix multiplication each representing a particular combination between the three matrices and their transposes. Because the distance between two elements is different depending on whether the matrix is traversed by column or by row, this makes SAXPY difficult to optimize as each time it is called using different strides. Furthermore, the size of the matrices is greater than 2MB, so they do not fit in any of the caches of existing machines.

There are two ways to improve performance on this benchmark. The first is to increase the size of the cache to reduce the number of misses. This has the disadvantage that it doesn't improve the performance of existing machines, only of future designs. The second involves generating hand optimized code for the SAXPY loop. Our analysis of the object code produced by both the MIPS and Sun compilers show that the quality of the code can be improved. The best way to achieve this is to incorporate the same compiler technology developed for supercomputers to generate vectorized code. The idea is to apply data dependence analysis to identify loops that can either be decomposed into vector operations or parallel execution. The same technology, however, can also be used in scalar machines even if they do not have vector hardware. The idea here is for a preprocessor to generate, instead of vector instructions, subroutine calls to hand-coded routines. Because these routines can be highly optimized by taking into consideration the best scheduling and blocking factor, they achieve substantial reductions in the execution time. Some new high performance workstations are starting to use this approach: the new HP 700 series includes a preprocessor developed by Kuck and Associates to make a source to source transformation of the program which includes calls to library routines implementing vector operations. We expect most machine manufacturers to follow the same path, given that it is relatively easy to incorporate into existing compilers[10].

Although it is generally accepted that optimizations performed by optimizers on user programs are valid as long as they preserve the relation between the input and the output, it is not clear that arbitrary changes to the behavior of benchmarks should be regarded in the same way. Benchmarks, in contrast to user programs, should execute on all machines under the "same" conditions. This is necessary in order to make a fair comparison between different systems and the only way in which their results can be interpreted in a meaningful way. What is important on benchmarks is how their execution exercises different aspects of the system. Hence an optimizer which significantly alters the behavior of a program is in a way defeating the purpose of the benchmark. If one benchmark exercises the memory system on one machine, but not in another, as a result of applying an optimization to the algorithms, then the comparison is meaningless regardless of whether the results of the program on both machines are the same.

---

[10] The latest SPEC results [SPEC91a, SPEC91b] clearly indicate that manufacturers are now using preprocessors to dramatically improve the performance of *MATRIX300*.

One of the uses of *MATRIX300* has been to evaluate the the memory system of a machine in those situations when the amount data touched by the program is much larger than the size of the data cache and, in many cases, the address space covered by the TLB. The goal is to force the machine to generate a substantial number of memory reads and writes and to perform a large number of TLB loads. But changing the implementation of matrix multiply from non-blocking to blocking reduces dramatically the number of cache and TLB misses. This changes the nature of the benchmark from memory bound to CPU bound.

4.6.1.5. NASA7

This benchmark consists of several computation intensive kernels which are frequently found in scientific applications. As most of the work is localized in highly-nested loops, most of the optimization improvements come from eliminating computation from the innermost loops by using array addressing, code motion, and strength reduction optimizations.

As in other benchmarks based on kernels, this one has some characteristics which can be exploited by a clever compiler. For example, the following code implements matrix multiply by doing a 4-way unrolling of the outer loop and accounts for approximately 16 percent of the execution time:

```
      SUBROUTINE MXM (A, B, C, L, M, N)
      IMPLICIT DOUBLE PRECISION(A-H,O-Z)
      DIMENSION A(L,M), B(M,N), C(L,N)

      DO 100 K = 1, N
         DO 100 I = 1, L
            C(I,K) = 0.
100      CONTINUE
      DO 110 J = 1, M, 4
         DO 110 K = 1, N
            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) + A(I,J+2) * B(J+2,K)
     .                 + A(I,J+3) * B(J+3,K)
110   CONTINUE

      RETURN
      END
```

This subroutine is called 100 times by the following loop:

```
      DO 120 II = 1, IT
         CALL MXM (A, B, C, L, M, N)
120   CONTINUE
```

Now, it is possible for a good compiler to detect, after inlining the subroutine, that the code inside loop 120 is invariant with respect to the induction variable and hence it only has to be executed once.

4.6.1.6. SPICE2G6

This scalar code is similar to *DODUC* and *FPPPP* as it is a difficult program to optimize. The few opportunities for optimization involve finding small common subexpressions and allocating frequently used variables to registers. Some of the most executed blocks are very small and they do not contain much to be optimized. The following spaghetti-like code, for example, accounts for almost 43% of the total execution and contains few opportunities for optimization.

```
135   IF (J .LT. I) GO TO 145
      LOCIJ = LOCC
140   LOCIJ = NODPLC(IRPT+LOCIJ)
      IF (NODPLC(IROWNO+LOCIJ) .EQ. I) GO TO 155
      GO TO 140
145   LOCIJ = LOCR
150   LOCIJ = NODPLC(JCPT+LOCIJ)
      IF (NODPLC(JCOLNO+LOCIJ) .EQ. J) GO TO 155
      GO TO 150
155   VALUE(LVN+LOCIJ) = VALUE(LVN+LOCIJ) - VALUE(LVN+LOCC) *
     .                   VALUE(LVN+LOCR)
160   LOCC=NODPLC(JCPT+LOCC)
      GO TO 130
```

The small size of the basic blocks and the irregular way in which the array elements are accessed makes it difficult for the compiler to improve the code.

4.6.2. Explaining Benchmark Results

One of our main arguments in this research is that conventional benchmarking cannot explain or predict performance, but only observe it. We claim that, in order to explain benchmark results, we need not only a detailed performance model of the machine and the program, but also to isolate the performance contribution of different parts of the system. In this section we illustrate how, with our methodology, we can easily explain the wide variation in benchmark results. We show that in each program there are only a few parameters which explain most of the difference in performance. Unfortunately, each program has its own particular characteristics, so a small number of parameters are not enough to explain the results on all programs.

Table 4.13 shows on the left side the optimized execution times we presented in §4.4.5 (tables 4.21-4.23, Appendix 4.B). On the right side we give the relative performance between pairs of machines. In addition, the sorting of their performance ratios is indicated by a small number on the right. At the bottom of the table we include the geometric mean of all results and the geometric mean of only those benchmarks in the SPEC suite (under label Geom. Mean (SPEC)).

The results indicate that at the maximum level of optimization the MIPS M/2000 is twice as fast as the Sparcstation 1+, and the HP 720 is five times faster than the Sparcstation 1+. The distribution of ratios shows a large dispersion, however. For example, the ratio between the HP 720 and the Sparcstation 1+ ranges from 1.766 to 12.250.

The ratio of both the Sparcstation 1+ and the MIPS M/2000 with respect to the HP 720 on program *BDNA* is in both cases very high: 12.250 and 4.875. The main explanation lies in their relative speeds on the square root intrinsic function. This can be seen in

program	Sparc 1+ (sec)	MIPS M/2000 (sec)	HP 720 (sec)	I / II	I / III	II / III
Doduc	277	110	40	2.518 6	6.925 4	2.750 6
Fpppp	383	172	39	2.227 14	9.821 2	4.410 3
Tomcatv	299	151	52	1.980 16	5.750 11	2.904 5
Matrix300	309	330	175	0.936 25	1.766 25	1.886 21
Nasa7	1393	1033	387	1.348 22	3.599 21	2.669 7
Spice2g6	3659	2630	509	1.391 21	7.189 3	5.167 1
ADM	271	100	63	2.710 4	4.302 16	1.587 25
QCD	169	67	29	2.522 5	5.828 10	2.310 14
MDG	3079	1056	487	2.916 1	6.322 5	2.168 16
TRACK	77	34	14	2.265 13	5.500 13	2.429 12
BDNA	882	351	72	2.513 7	12.250 1	4.875 2
OCEAN	1209	483	196	2.503 8	6.168 7	2.464 11
DYFESM	117	90	44	1.300 23	2.659 24	2.045 20
ARC2D	1785	1014	439	1.760 17	4.066 18	2.310 15
TRFD	303	133	65	2.278 12	4.662 15	2.046 19
FLO52	288	120	51	2.400 11	5.647 12	2.353 13
Alamos	81	33	20	2.455 10	4.050 19	1.650 24
Baskett	0.50	0.39	0.15	1.282 24	3.333 23	2.600 9
Erathostenes	0.30	0.21	0.05	1.429 20	6.000 8	4.200 4
Linpack	11.0	3.8	2.3	2.895 2	4.783 14	1.652 23
Livermore	22.6	11.0	6.1	2.055 15	3.705 20	1.803 22
Mandelbrot	1.20	0.72	0.35	1.667 18	3.429 22	2.057 18
Shell	0.43	0.26	0.10	1.654 19	4.300 17	2.600 8
Smith	94	38	15	2.474 9	6.267 6	2.533 10
Whetstone	1.00	0.35	0.17	2.857 3	5.882 9	2.059 17
Maximum	–	–	–	2.916	12.250	5.167
Minimum	–	–	–	0.936	1.766	1.587
Max / Min	–	–	–	3.115	6.937	3.256
Geom. Mean	–	–	–	2.006	4.974	2.480
Geom. Mean (SPEC)	–	–	–	1.640	5.113	3.116

Table 4.13: The first three columns show the benchmark execution times under the maximum optimization levels. The last three columns give the respective ratios; these are sorted by magnitude using a small numeral. At the bottom of the table we show the geometric means for all programs and the geometric mean computed only for those benchmarks in the SPEC suite (Geom. Mean (SPEC)).

table 4.14 which shows the execution time distribution of the four most important parameters on *BDNA*. For each parameter we give the fraction of the total time that they represent and the execution time ratios between the machines.

That the contribution to the total execution time of the square root function on the HP 720 is less than the individual contributions of floating-point adds and multiplies does not mean that a single square root executes in less time than a single add or multiply. What the values on the tables indicate is the product of the number of times that each operation is executed in the program times its respective execution time.

While the square root function represents almost 46% of the execution time on the Sparcstation 1+, it only accounts for 8% on the HP 720. This is because a square root on the HP 720 is more than 50 faster than on the Sparcstation (showed in the right side of table 4.14). The other three parameters in table 4.14 have execution time ratios which

operation	Sparcstation 1+	MIPS M/2000	HP 720	I / II	I / III	II / III
square root	349.3 s (0.4597)	97.5 s (0.2970)	6.7 s (0.0776)	3.583	52.134	14.552
f.p. multiply	98.3 s (0.1295)	77.5 s (0.2360)	15.8 s (0.1814)	1.268	6.222	4.905
f.p. add	90.6 s (0.1193)	41.8 s (0.1274)	12.5 s (0.1441)	2.167	7.248	3.344
procedure call	62.2 s (0.0819)	11.2 s (0.0340)	10.3 s (0.1186)	5.554	6.039	1.087
partial sum	600.4 s (0.7904)	228.0 s (0.7007)	45.3 s (0.5217)	2.633	13.253	5.033

Table 4.14: Absolute and relative contribution of the four abstract operations that contribute most to the execution time on program *BDNA*. most important abstract operations on program *BDNA*. The last three columns show for each operation the execution time ratio between the machines. The main factor determining the relative performance on this program is the speed of the square root function.

are also higher than the overall geometric mean, indicating that floating-point performance and procedure calls are another two strong points of the HP 720. In benchmark *FPPPP*, it is the floating-point performance of the HP 720 which explains the high ratios of 9.821 and 4.410.

Another program in which the speed of intrinsic functions determines most of the execution time is *MDG*. The functions, exponential, square root, and absolute value consume close to 50% of the execution on the Sparcstation 1+. The reason why the ratio between the Sparcstation 1+ and the HP 720 is not as high as with *BDNA* is because the exponential function is only 4 times faster on the HP 720. On the other hand, the ratio between the MIPS M/2000 and the Sparcstation is very high because on the MIPS M/2000 the absolute value function is almost 24 times faster.

parameter	Sparc / MIPS	Sparc / HP	MIPS / HP
square root	3.582	51.832	14.468
exponential	2.887	3.990	1.381
absolute value	23.588	14.579	0.618

Table 4.15: Relative times of three intrinsic functions. These functions are important contributors to the execution time of program *MDG*.

Table 4.15 gives an idea of how much variation there is on the performance of intrinsic functions between the three machines, and how linear combinations of the three operations define a large space of potential benchmark results. Even when the HP 720 is consistently faster than the MIPS M/2000 on all programs, its performance on the absolute value function is 40% slower. Program *MDG* shows that even a single parameter like the absolute function can be an important determining factor in the execution time of a program, and this illustrates why it is not possible to reduce performance to a few number of parameters. Even when the execution time of the majority of programs is dominated by a small number of parameters, the union of all these parameters across all programs is significantly larger. By selecting only a few of them for the complete workload we run the risk of ignoring important factors which on some machines and program might be more important. Without relating benchmark results to the behavior of programs, the performance measurements on the machines, and without having a unified

program execution model it is not possible to explore this space in detail. Our abstract machine performance model and the tools we have developed in this context represent an attempt at identifying the sources of performance across many machines and on arbitrary large programs.

4.7. Conclusions

Evaluating and explaining the performance of a machine requires relating observed performance to the individual components of the system. Machine designers are able to do this by constructing detailed models and simulators of their machines. These machine models, however, usually contain so much machine-dependent information that in most cases it is impractical to use them when evaluating different machines. Three good studies which illustrate the difficulty of comparing machines at the instruction set level can be found in [Peut77, Shus78, Cmel91]. Our research has concentrated on developing a sound methodology which can be used to evaluate machines on equal terms without compromising our ability to make precise statements about them. We achieve this objective by: 1) using a machine-independent model of program execution which permits making predictions and thus validating the model; 2) understanding that performance is as much a function of the program and the compiler (optimizer) as it is of the machine; 3) modeling performance at various levels of detail: abstract operations, reduced parameters, performance shapes, total execution time, etc; 4) measuring experimentally every parameter in the model.

In this chapter we discussed how optimization can be incorporated in our methodology and show that it is possible to evaluate different optimizing compilers, not only by detecting the set of optimizations which they can perform, but also by predicting and explaining how much improvement they provide on large applications. By assuming that most of the optimizations are invariant with respect to the abstract decomposition of the program, we change the nature of the problem from one of detecting how a program could be changed by the compiler, to characterizing the performance of the 'optimized' machine defined by the optimizer. Using this approach we showed that it is possible to measure the contribution of optimization and predict the execution time of optimized programs, although not as well as in the nonoptimized case. For those programs where optimization actually affects the set of abstract operations executed, it is still possible to detect the optimizations and apply them to obtain equivalent programs which contain basically only invariant optimizations.

In order to understand the performance impact of optimizations it is necessary to: 1) detect the set of optimizations that a compiler can apply; 2) evaluate the performance impact of these optimizations; and 3) measure how much optimization opportunities exist in programs. Here we have addressed mainly point 1) and in lesser degree points 2) and 3). We wrote programs to detect local and global machine-independent optimizations and measured several optimizing compilers. We showed that optimizing compilers differ in the effectiveness in which they can apply the same optimizations. On some benchmarks these differences can affect the total execution time. We also evaluated the optimization improvement provided by several optimizers on the Fortran SPEC, Perfect Club, and other popular benchmarks. Finally, we discussed the main optimizations found in the benchmarks and discussed some of their characteristics which can be exploited by clever compilers.

One factor which needs to be addressed is characterizing how much improvement different optimization techniques provide over a large sample of programs. This requires obtaining the source code of a very good compiler, which in most cases is proprietary, and instrumenting it to collect statistics about the application of optimizations. One of the problems here is that it is not meaningful to measure optimizations in isolation, as in some cases they can be applied only when other optimizations are attempted first. In other words, optimizations in general are not commutative or associative. There is good understanding of what is the typical static and dynamic statistics of programs in many languages, but very little is known about the typical optimizations found on those languages. We believe this is an area that requires more attention.

4.8. References

[AhoA86] Aho, A.V., Sethi, R., and Ullman J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass., 1986.

[Alle87] Allen, R. and Kennedy, K., "Automatic Translation of FORTRAN Programs to Vector Form", *ACM Transactions on Programming Languages and Systems*, Vol.9, No.4, October 1987, pp. 491-542.

[Arno83] Arnold, C.N., "Vector Optimization on the CYBER 205", *Proc. of the 1983 Int. Conf. on Parallel Processing*, Columbus, Ohio, August 23-26 1983, pp. 530-536.

[BalH86] Bal, H.E. and Tanenbaum, A.S., "Language- and Machine-Independent Global Optimization on Intermediate Code", *Computer Languages*, Vol.11, No.2, 1986, pp. 105-121.

[Bane88] Banerjee, U., *Dependency Analysis for Supercomputing*, Kluwer Academic Publishers, Boston, 1988.

[Bras88] Braswell, R.N. and Keech, M.S., "An Evaluation of Vector FORTRAN 200 Generated by CYBER 205 and ETA-10 Pre-Compilation Tools", *Proc. of the Supercomputing '88 Conf.*", Orlando, Florida, November 14-18 1988, pp. 106-113.

[Call88] Callahan, D., Dongarra, J., and Levine, D., "Vectorizing Compilers: A Test Suite and Results", *Proc. of the Supercomputing '88 Conf.*, Orlando, Florida, November 14-18 1988, pp. 98-105.

[Call90] Callahan, D., Kennedy, K., and Carr, S., "Improving Register Allocation for Subscript Variables", *Proc. of the ACM SIGPLAN '90 Conf. on Prog. Lang. Design and Implementation*, White Plains, New York, June 1990, pp. 53-65.

[Call91] Callahan, D., Kennedy, K., and Porterfield, A., "Software Prefetching", *Proc. of the 2nd Int. Conf. on Arch. Support for Prog. Lang. and Oper. Sys. (ASPLOS III)*, Santa Clara, California, April 8-11 1991, pp. 40-52.

[Chai82] Chaitin, G.J., "Register Allocation and Spilling via Graph Coloring", *Proc. of the SIGPLAN '82 Symp. on Compiler Construction*, June 1982, pp. 98-105.

[Chow83] Chow, F., *A Portable Machine-Independent Global Optimizer*, Ph.D. Dissertation and Technical Report No. 83-254, Computer Systems

Laboratory, Stanford University, December 1983.

[Chow84] Chow, F. and Hennessy J.L., "Register Allocation by Priority-Based Coloring", *Proc. of the ACM SIGPLAN '84 Symp. on Compiler Construction*, Vol.19, No.6, June 1984, pp. 222-233.

[Chow86] Chow, F., Himelstein, M., Killian, E., and Weber, L., "Engineering a RISC Compiler System", *Proc. of the Compcon '86 Conf.*, San Francisco, California, March 4-6 1986, pp. 132-137.

[Cmel91] Cmelik, R.F., Kong., S.I., Ditzel, D.R., and Kelly, E.J., "An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks", *Proc. of the 2nd Int. Conf. on Arch. Support for Prog. Lang. and Oper. Sys. (ASPLOS III)*, Santa Clara, California, April 8-11 1991, pp. 290-302.

[Cock80] Cocke, J. and Markstein, P., "Measurement of Program Improvement Algorithms", Technical Report No. RC-8111 (#35193), IBM Yorktown Heights, February 7 1980.

[Cude89] Cuderman, K.J. and Flynn, M.J., "The Relative Effects of Optimization on Instruction Architecture Performance", Technical Report No. CSL-TR-89-398, Computer Systems Laboratory, Stanford University, October 1989.

[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 Technical Report 965, March 1990.

[Davi80] Davison, J.W. and Fraser, C.W., "The Design and Application of a Retargetable Peephole Optimizer", *ACM Transactions on Programming Languages and Systems*, Vol.2, No.2, April 1980, pp. 191-202.

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

[Ferr91] Ferrante, J., Sarkar, V., and Thrash, W., "On Estimating and Enhancing Cache Effectiveness", *Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, California, August 1991.

[Hira91] Hiranandani, S., Kennedy, K., and Tseng, C.W., "Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines", *Proc. of the Supercomputing '91 Conf.*, Albuquerque, New Mexico, November 18-22 1991, pp. 86-100.

[Jaza86] Jazayeri, M. and Haden, M., "Optimizing Compilers Are Here (mostly)", *SIGPLAN Notices*, Vol.21, No.5, May 1986, pp. 61-63.

[John86] Johnson, M.S. and Miller, T.C., "Effectiveness of a Machine-Level, Global Optimizer", *Proc. of the SIGPLAN '86 Symp. on Compiler Construction*, Palo Alto, June 25-27 1986, pp. 99-108.

[Kess86] Kessler, P.B., "Discovering Machine-Specific Code Improvements", *Proc. of the SIGPLAN '86 Symp. on Compiler Construction*, Palo Alto, California, June 25-27 1986, pp. 249-254.

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

[Lind86a] Lindsay, D.S and Bell, T.E., "Directed Benchmarks for CPU Architecture Evaluation", *Proc. of the CMG '86 Conf*, Las Vegas, Nevada, December 9-12 1986, pp. 379-385.

[Lind86b] Lindsay, D.S, "CPU Performance Comparison Among Vendors", *Proc. of the CMG '86 Conf*, Las Vegas, Nevada, December 9-12 1986, pp. 179-187.

[Lind86c] Lindsay, D.S, "Methodology for Determining the Effects of Optimizing Compilers", *Proc. of the CMG '86 Conf*, Las Vegas, Nevada, December 9-12 1986, pp. 366-373.

[Lind86d] Lindsay, D.S, "Comparison of the Optimization Produced by the IBM VS-FORTRAN Compiler and the DEC VAX/VMS FORTRAN Compiler", *Proc. of the CMG '86 Conf*, Las Vegas, Nevada, December 9-12 1986, pp. 374-378.

[Much86] Muchnick, S.S., "Here Are (Some of) the Optimizing Compilers", *SIGPLAN Notices*, Vol.21, No.2, February 1986, pp. 11-15.

[Noba89] Nobayashi, H. and Eoyang, C., "A Comparison Study of Automatically Vectorizing FORTRAN Compilers", *Proc. of the Supercomputing '89 Conf.*, Reno, Nevada, November 13-17 1989, pp. 820-825.

[O'Br90] O'Brien, K., Hay, B., Minisk, J., Schaffer, H., Schloss B., Shepherd, A., and Zaleski, M., "Advanced Compiler Technology for the RISC System/6000 Architecture", *IBM RISC System/6000 Technology*, SA23-2619, IBM Corporation, 1990, pp. 154-161.

[Patt85] Patterson, D.A., "Reduced Instructions Set Computers", *Comm of the ACM*, Vol.28, No.1, January 1985, pp. 8-21.

[Paud86] Pauda, D.A. and Wolfe, M.J., "Advanced Compiler Optimizations for Supercomputers", *Comm. of the ACM*, Vol.29, No.12, December 1986, pp. 1184-1201.

[Peut77] Peuto, B.L., and Shustek, L.J., "An Instruction Timing Model of CPU Performance", *The 4th Annual Symposium on Computer Architecture*, Vol.5, No.7, March 1977, pp. 165-178.

[Port89] Porterfield, A., *Software Methods for Improvement of Cache Performance on Supercomputers Applications*, Ph.D. Dissertation and Technical Report No. COMP-TR89-93, Rice University, 1989.

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

[Saav90] Saavedra-Barrera, R.H. "The SPEC and Pefect Club Benchmarks: Promises and Limitations", *Hot Chips Symposium 2*. Santa Clara, CA, August 1990.

[Shus78] Shustek, L.J., *Analysis and Performance of Instruction Sets*, Ph.D. Dissertation, Stanford University, May 1978.

[Sing91] Singh, J.P. and Hennessy J.L., "An Empirical Investigation of the Effectiveness and Limitations of Automatic Parallelization", *Int. Symp. on Shared Memory Multiprocessing*, Tokyo, Japan, April 1991, pp. 25-36.

[SPEC90a] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol.2, Issue 2, Spring 1990.

[SPEC90b] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol.2, Issue 3, Summer 1990.

[SPEC91a] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol.3, Issue 2, Spring 1991.

[SPEC91b] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol.3, Issue 3, Fall 1991.

[Tane82] Tenenbaum, A.S., van Staveren, H., and Stevenson, J.W., "Using Peephole Optimization on Intermediate Code", *ACM Transactions on Programming Languages and Systems*, Vol.4, No.1, January 1982, pp. 21-36.

[Wolf91] Wolf, M. and Lam, M., "A Data Locality Optimizing Algorithm", *Proc. of the ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, June 1991, pp. 30-44.

[Wolf85] Wolfe, M. and Macke, T., "Where are the Optimizing Compilers", *SIGPLAN Notices*, Vol.20, No.11, November 1985, pp. 64-77.

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

Appendix 4.A

Group 1: Floating-Point Arithmetic Operations (single, local)

machine	optim	SRSL	ARSL	MRSL	DRSL	ERSL	XRSL	TRSL
HP 720	-O0	40	89	117	273	84	10173	180
HP 720	-O1	40	47	108	198	75	10447	135
HP 720	-O2	7	40	31	167	58	10426	20
MIPS M/2000	-O0	40	159	258	583	316	26136	213
MIPS M/2000	-O1	< 1	101	169	498	229	26068	113
MIPS M/2000	-O2	< 1	95	145	395	219	26255	< 1
Sparcstation 1+	-O0	80	305	384	1856	2419	18414	510
Sparcstation 1+	-O2	75	298	322	1849	2398	18512	499
Sparcstation 1+	-O3	< 1	234	133	1444	2575	19603	35

Group 2: Floating-Point Arithmetic Operations (complex, local)

machine	optim	SCSL	ACSL	MCSL	DCSL	ECSL	XCSL	TCSL
HP 720	-O0	73	163	496	3603	1726	40659	276
HP 720	-O1	67	116	251	3574	1727	40470	288
HP 720	-O2	54	59	130	3528	1700	40525	224
MIPS M/2000	-O0	22	568	1162	5059	4696	37344	339
MIPS M/2000	-O1	16	305	795	5019	4672	36864	263
MIPS M/2000	-O2	12	227	713	4926	4352	37249	161
Sparcstation 1+	-O0	247	2772	5106	7471	5065	71209	1094
Sparcstation 1+	-O2	142	588	3287	8073	2423	75121	998
Sparcstation 1+	-O3	20	523	2698	8314	2841	72563	649

Group 3: Integer Arithmetic Operations (single, local)

machine	optim	SISL	AISL	MISL	DISL	EISL	XISL	TISL
HP 720	-O0	< 1	81	429	455	657	1603	162
HP 720	-O1	< 1	48	419	439	640	1589	108
HP 720	-O2	< 1	20	200	438	463	954	20
MIPS M/2000	-O0	< 1	134	564	1551	662	1270	198
MIPS M/2000	-O1	< 1	72	460	1506	715	1272	135
MIPS M/2000	-O2	< 1	37	489	1685	497	1038	47
Sparcstation 1+	-O0	< 1	216	1104	2625	4645	5967	519
Sparcstation 1+	-O2	< 1	188	1265	2755	4501	5832	556
Sparcstation 1+	-O3	< 1	41	949	2625	4559	5876	41

Table 4.16: Characterization results for groups 1-3 under different optimization levels. A value '< 1' indicates that the parameter was not detected by the experiment.

Group 4: Floating-Point Arithmetic Operations (double, local)

machine	optim	SRDL	ARDL	MRDL	DRDL	ERDL	XRDL	TRDL
HP 720	-O0	40	89	117	313	106	9051	180
HP 720	-O1	32	70	112	289	84	9118	140
HP 720	-O2	7	40	31	202	59	9325	20
MIPS M/2000	-O0	67	206	347	935	421	26040	305
MIPS M/2000	-O1	26	112	220	808	295	26208	274
MIPS M/2000	-O2	< 1	101	172	643	302	26292	27
Sparcstation 1+	-O0	232	440	757	3336	3797	34902	1045
Sparcstation 1+	-O2	226	413	469	3339	3839	35032	1047
Sparcstation 1+	-O3	< 1	234	253	2430	4243	37339	263

Group 5: Floating-Point Arithmetic Operations (single, global)

machine	optim	SRSG	ARSG	MRSG	DRSG	ERSG	XRSG	TRSG
HP 720	-O0	41	109	140	286	80	10202	167
HP 720	-O1	20	59	85	205	94	10469	170
HP 720	-O2	14	49	38	175	81	10384	32
MIPS M/2000	-O0	40	192	261	625	307	26134	203
MIPS M/2000	-O1	24	176	232	523	240	26036	108
MIPS M/2000	-O2	< 1	95	256	481	302	26650	19
Sparcstation 1+	-O0	95	333	427	1928	2391	18249	522
Sparcstation 1+	-O2	84	299	363	1850	2388	18511	419
Sparcstation 1+	-O3	< 1	233	133	1432	2563	19517	37

Group 6: Floating-Point Arithmetic Operations (complex, global)

machine	optim	SCSG	ACSG	MCSG	DCSG	ECSG	XCSG	TCSG
HP 720	-O0	81	209	609	3624	3305	47213	272
HP 720	-O1	67	120	293	3573	1719	41728	241
HP 720	-O2	53	73	152	3548	1710	41541	210
MIPS M/2000	-O0	185	652	1278	5172	4705	37529	430
MIPS M/2000	-O1	152	374	858	5190	4528	36802	270
MIPS M/2000	-O2	135	225	736	5129	4245	37490	166
Sparcstation 1+	-O0	241	2843	5217	7517	5083	71267	1143
Sparcstation 1+	-O2	190	612	3498	8095	2516	75239	989
Sparcstation 1+	-O3	19	586	2676	8397	2736	72942	977

Table 4.17: Characterization results for groups 4-6 under different optimization levels. A value '< 1' indicates that the parameter was not detected by the experiment.

Group 7: Integer Arithmetic Operations (single, global)

machine	optim	SISG	AISG	MISG	DISG	EISG	XISG	TISG
HP 720	-O0	< 1	109	454	482	669	1619	161
HP 720	-O1	< 1	72	419	450	670	1636	90
HP 720	-O2	< 1	37	139	449	408	864	50
MIPS M/2000	-O0	< 1	197	584	1608	676	1281	240
MIPS M/2000	-O1	< 1	106	466	1671	694	1230	152
MIPS M/2000	-O2	< 1	101	489	1747	761	1352	97
Sparcstation 1+	-O0	20	278	1160	2688	4664	5998	516
Sparcstation 1+	-O2	19	188	1265	2755	5609	5830	492
Sparcstation 1+	-O3	< 1	41	951	2611	4535	5864	40

Group 8: Floating-Point Arithmetic Operations (double, global)

machine	optim	SRDG	ARDG	MRDG	DRDG	ERDG	XRDG	TRDG
HP 720	-O0	40	110	141	326	80	9044	167
HP 720	-O1	27	76	121	246	92	9338	166
HP 720	-O2	13	49	39	210	83	9266	32
MIPS M/2000	-O0	67	240	350	978	410	25942	312
MIPS M/2000	-O1	58	221	339	846	341	26064	269
MIPS M/2000	-O2	< 1	133	381	782	376	27206	61
Sparcstation 1+	-O0	232	535	840	3475	3806	34717	1080
Sparcstation 1+	-O2	186	440	727	3332	6484	34990	1052
Sparcstation 1+	-O3	< 1	233	255	2427	4280	37251	267

Groups 9, 10: Conditional and Logical Parameters

machine	optim	ANDL	CRSL	CCSL	CISL	CRDL	ANDG	CRSG	CCSG	CISG	CRDG
HP 720	-O0	87	229	491	94	229	97	270	579	136	269
HP 720	-O1	72	168	336	61	175	81	175	337	68	175
HP 720	-O2	46	114	282	20	115	46	148	298	47	148
MIPS M/2000	-O0	109	283	281	202	365	135	365	366	283	447
MIPS M/2000	-O1	105	302	319	208	413	132	418	345	308	518
MIPS M/2000	-O2	70	206	207	95	204	97	231	340	95	221
Sparcstation 1+	-O0	224	582	1492	420	742	241	664	1572	503	914
Sparcstation 1+	-O2	182	595	1474	475	809	182	595	1470	476	807
Sparcstation 1+	-O3	79	365	1515	149	365	80	366	1518	149	366

Table 4.18: Characterization results for groups 7-10 under different optimization levels. A value '< 1' indicates that the parameter was not detected by the experiment.

Groups 11, 12: Function Call, Arguments and References to Array Elements

machine	optim	PROC	ARGU	ARR1	ARR2	ARR3	IADD
HP 720	-O0	81	153	64	217	320	6
HP 720	-O1	85	122	30	130	187	9
HP 720	-O2	79	50	47	28	50	< 1
MIPS M/2000	-O0	408	80	309	612	936	< 1
MIPS M/2000	-O1	283	145	150	380	583	< 1
MIPS M/2000	-O2	152	141	122	184	191	< 1
Sparcstation 1+	-O0	196	328	224	637	1007	< 1
Sparcstation 1+	-O2	81	58	124	409	705	< 1
Sparcstation 1+	-O3	77	41	133	194	281	< 1

Groups 13, 14: Branching and DO loop Parameters

machine	optim	GOTO	GCOM	LOIN	LOOV	LOIX	LOOX
HP 720	-O0	28	243	363	202	552	303
HP 720	-O1	20	263	242	121	447	161
HP 720	-O2	12	172	68	42	182	61
MIPS M/2000	-O0	81	609	580	322	838	604
MIPS M/2000	-O1	< 1	648	325	201	445	354
MIPS M/2000	-O2	< 1	486	1174	< 1	284	202
Sparcstation 1+	-O0	< 1	894	934	404	1458	662
Sparcstation 1+	-O2	< 1	853	220	114	467	280
Sparcstation 1+	-O3	< 1	650	161	81	406	244

Group 15: Intrinsic Functions (single precision)

machine	optim	EXPS	LOGS	SINS	TANS	SQRS	ABSS	MODS	MAXS
HP 720	-O0	2780	3457	1878	3037	292	61	3921	424
HP 720	-O1	2725	3396	1787	2927	283	61	3821	284
HP 720	-O2	2701	3376	1787	2913	304	61	3782	323
MIPS M/2000	-O0	4036	3323	3019	3657	3570	35	2232	434
MIPS M/2000	-O1	3937	3280	2966	3609	3567	81	2118	449
MIPS M/2000	-O2	4139	3482	3168	3793	3726	40	2120	407
Sparcstation 1+	-O0	5563	5996	9085	12441	7891	453	3673	3333
Sparcstation 1+	-O2	5728	6162	9266	12626	9083	539	3619	1204
Sparcstation 1+	-O3	6805	7183	10299	13777	9140	611	4206	1182

Table 4.19: Characterization results for groups 11-15 under different optimization levels. A value '< 1' indicates that the parameter was not detected by the experiment.

Group 16: Intrinsic Functions (double precision)

machine	optim	EXPD	LOGD	SIND	TAND	SQRD	ABSD	MODD	MAXD
HP 720	-O0	3142	3129	2418	3851	363	64	3813	429
HP 720	-O1	3085	3071	2359	3795	365	61	3697	284
HP 720	-O2	3066	3051	2340	3780	384	61	3652	324
MIPS M/2000	-O0	3586	4156	4327	4702	5519	41	2244	495
MIPS M/2000	-O1	3567	4205	4296	4718	5622	16	2263	531
MIPS M/2000	-O2	3607	4180	4336	4700	5537	35	2252	408
Sparcstation 1+	-O0	9048	10881	13025	16035	18787	624	4696	5373
Sparcstation 1+	-O2	9067	10944	13131	16005	18734	847	4610	1169
Sparcstation 1+	-O3	10408	12175	14106	17065	19885	892	5967	1115

Groups 17, 18: Intrinsic Functions (integer and complex)

machine	optim	ABSI	MODI	MAXI	EXPC	LOGC	SINC	SQRC	ABSC
HP 720	-O0	40	1668	161	17520	12060	30537	9434	5386
HP 720	-O1	< 1	1630	80	17488	12024	30514	9406	5400
HP 720	-O2	40	1649	81	17555	12059	30694	9434	5375
MIPS M/2000	-O0	91	1522	349	13835	10607	13559	16574	4253
MIPS M/2000	-O1	83	1400	203	13671	10178	13276	9580	4000
MIPS M/2000	-O2	94	1579	243	13774	10289	13401	9562	3765
Sparcstation 1+	-O0	982	2229	2956	41051	22567	59548	46154	23767
Sparcstation 1+	-O2	314	2220	657	40561	21951	59170	45925	22615
Sparcstation 1+	-O3	189	2189	405	40533	23827	59139	45487	22478

Group 19: Intrinsic Functions (type conversion)

machine	optim	CPLX	REAL	IMAG	CONJ
HP 720	-O0	< 1	20	20	61
HP 720	-O1	5	< 1	40	60
HP 720	-O2	40	< 1	< 1	62
MIPS M/2000	-O0	179	< 1	824	445
MIPS M/2000	-O1	164	< 1	918	711
MIPS M/2000	-O2	40	< 1	649	611
Sparcstation 1+	-O0	782	1522	2262	1482
Sparcstation 1+	-O2	161	< 1	< 1	111
Sparcstation 1+	-O3	567	< 1	< 1	144

Table 4.20: Characterization results for groups 16-19 under different optimization levels. A value '< 1' indicates that the parameter was not detected by the experiment.

Appendix 4.B

HP-9000/720

	Optimization level 0			Optimization level 1			Optimization level 2		
program	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)
Doduc	85	85	−0.12	65	66	+1.08	40	34	−13.67
Fpppp	78	90	+15.30	58	69	+19.52	39	35	−10.28
Tomcatv	182	157	−14.09	121	111	−8.42	52	35	−31.40
Matrix300	598	340	−43.13	434	159	−63.43	175	45	−74.44
Nasa7	–	–	–	1120	715	−36.15	387	318	−17.80
Spice2g6	–	–	–	–	–	–	509	727	+42.73
ADM	252	172	−31.92	193	116	−39.90	63	58	−8.08
QCD	81	108	+33.21	59	86	+46.59	29	41	+81.40
MDG	827	861	+4.04	681	631	−7.34	487	474	−2.69
TRACK	25	28	+14.63	19	21	+8.39	14	12	−13.13
BDNA	208	185	−11.32	147	152	+3.47	72	87	+21.37
OCEAN	764	791	+3.52	558	548	−1.84	196	320	+63.07
DYFESM	307	187	−39.09	209	101	−51.70	44	32	−25.92
MG3D	–	–	–	–	–	–	1088	1088	−0.03
ARC2D	2137	1267	−40.70	1594	593	−62.81	439	187	−57.47
TRFD	464	329	−28.98	279	202	−27.70	65	71	+9.61
FLO52	426	274	−35.70	292	142	−51.20	51	48	−7.23
Alamos	67	67	+0.45	48	37	−22.38	20	28	+38.73
Baskett	0.50	0.58	+16.00	0.38	0.41	+7.89	0.15	0.20	+33.33
Erathostenes	0.13	0.13	+1.54	0.10	0.08	−20.00	0.05	0.04	−26.00
Linpack	8.8	7.6	−13.67	5.5	5.1	−7.82	2.3	3.3	+41.30
Livermore	16.5	17.7	+7.27	11.2	11.4	+1.79	6.1	6.1	−0.49
Mandelbrot	0.89	0.89	0.00	0.66	0.69	+4.55	0.35	0.31	−11.43
Shell	0.49	0.42	−13.67	0.30	0.25	−16.67	0.10	0.15	+50.00
Smith	54	47	−12.15	40	32	−18.23	15	18	+25.17
Whetstone	0.28	0.26	−7.14	0.23	0.21	−8.70	0.17	0.14	−17.64

Table 4.21: Nonoptimized and optimized benchmark results on the HP 720. All times are reported in seconds and the errors are computed as $100\times(pred-real)/real$.

MIPS M/2000

	Optimization level 0			Optimization level 1			Optimization level 2		
program	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)
Doduc	187	208	+11.69	149	143	−4.03	110	107	−2.73
Fpppp	247	239	−8.14	202	174	−13.86	172	177	+2.91
Tomcatv	452	415	−3.21	313	251	−19.81	151	145	−3.97
Matrix300	816	614	−24.77	646	370	−42.72	330	182	−44.85
Nasa7	2906	2634	−9.36	2234	1703	−23.77	1033	980	−5.13
Spice2g6	4576	4539	−0.81	3660	2679	−26.80	2630	1937	−26.35
ADM	424	426	+0.47	309	259	−16.18	–	165	–
QCD	131	176	+34.48	97	124	+27.84	67	98	+46.27
MDG	1796	2254	+25.50	1436	1566	+9.05	1056	1281	+21.31
TRACK	58	71	+22.20	42	47	+10.49	34	31	−9.94
BDNA	733	582	−20.60	531	388	−26.93	351	328	−6.55
OCEAN	1618	1722	+6.47	1148	1063	−7.40	483	732	+51.55
DYFESM	407	370	−9.10	305	221	−27.54	90	119	+32.22
ARC2D	3465	2470	−28.70	2548	1155	−54.67	1014	799	−21.20
TRFD	577	566	−1.87	424	335	−20.99	133	184	−16.74
FLO52	721	853	−15.52	530	529	+0.19	120	208	+73.33
Alamos	118	139	+16.95	90	85	−5.56	33	48	+45.45
Baskett	1.00	1.13	+13.00	0.73	0.80	+9.59	0.39	0.46	+17.94
Erathostenes	0.47	0.31	−21.28	0.36	0.19	−47.22	0.21	0.10	−52.38
Linpack	12.7	14.5	+13.94	9.0	10.3	+14.44	3.80	5.13	+35.00
Livermore	30.0	38.6	+28.80	19.1	22.9	+19.90	11.0	16.8	+52.72
Mandelbrot	1.50	1.59	+6.00	1.05	1.04	−0.95	0.72	0.82	+13.88
Shell	1.64	1.59	−2.44	1.03	0.64	−37.86	0.26	0.38	+46.15
Smith	132	113	+15.05	103	68	−34.24	38	45	+17.80
Whetstone	0.48	0.48	+0.00	0.40	0.36	−10.00	0.35	0.31	−11.43

Table 4.22: Nonoptimized and optimized benchmark results on the MIPS M/2000 using the *f77* compiler version 1.21. All times are reported in seconds and the errors are computed as $100 \times (pred - real)/real$.

Sparcstation 1+

	Optimization level 1			Optimization level 2			Optimization level 3		
program	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)
Doduc	407	410	+0.74	283	348	+22.79	277	205	-25.79
Fpppp	590	481	+10.63	399	421	+5.62	383	188	-50.80
Tomcatv	576	637	-18.46	309	484	+56.48	299	239	-19.96
Matrix300	1502	815	-45.70	397	545	+38.26	309	251	-18.72
Nasa7	5516	4046	-26.66	1467	2826	+92.65	1393	1540	+10.52
Spice2g6	5348	6660	+24.54	4504	5669	-25.86	3659	2320	+36.60
ADM	716	551	-23.05	286	438	+53.11	271	253	-6.72
QCD	274	341	+24.47	190	265	+39.87	169	176	+4.27
MDG	3809	4072	+6.90	3153	3480	+10.37	3079	2804	-8.94
TRACK	108	122	+13.30	82	98	+19.54	77	60	-21.28
BDNA	1270	1173	-7.63	1027	914	-11.09	882	760	-13.87
OCEAN	3538	3551	+0.37	1713	2210	+29.00	1209	1485	+22.90
DYFESM	617	437.1	-29.10	141	296	+109.50	117	146	+24.91
ARC2D	6437	4091	-36.44	3039	3083	+1.45	1785	1757	-1.58
TRFD	1181	765	-35.26	320	525	+64.35	303	248	-18.18
FLO52	1132	1049	-7.28	314	785	+150.40	288	393	+36.27
Alamos	207	188	-9.26	152	121	-20.29	81	80	-1.12
Baskett	1.40	1.68	+20.00	0.60	1.13	+88.33	0.50	0.61	+22.00
Erathostenes	0.50	0.38	-24.00	0.30	0.27	-10.00	0.30	0.13	-56.67
Linpack	35.9	21.7	-19.03	13.9	15.0	+7.92	11.0	10.2	-7.27
Livermore	52.6	53.5	+1.17	25.7	42.6	+65.76	22.6	25.7	+13.72
Mandelbrot	2.40	2.70	+12.50	1.20	2.67	+122.50	1.20	1.23	+2.50
Shell	0.95	1.28	+34.74	0.70	1.07	+52.86	0.43	0.45	+4.65
Smith	198	144	-27.22	99	113	-14.23	94	57	-39.17
Whetstone	1.30	0.86	-28.33	1.00	0.77	-23.00	1.00	0.61	-39.00

Table 4.23: Nonoptimized and optimized benchmark results on the Sparcstation 1+ using the *f77* compiler version 1.3. All times are reported in seconds and the errors are computed as $100 \times (pred - real)/real$.

Appendix 4.C

compiler	Integer		Floating-Point		Mixed	
	local	global	local	global	local	global
BSD Unix F77 1.0	no	no	no	no	no	no
Mips F77 2.0 -O2	yes	yes	partial[1]	partial[1]	partial[1]	no
Mips F77 2.0 -O1	yes	no	no	no	no	no
Sun F77 1.3 -O3	yes	no	no	no	no	no
Sun F77 1.3 -O2	yes	no	no	no	no	no
Sun F77 1.3 -O1	no	no	no	no	no	no
Ultrix Fort 4.5	yes	no	yes	no	yes	no
Amdahl F77 2.0	no	no	no	no	no	no
CRAY CFT77 4.0.1	yes	yes	yes	yes	yes	yes
IBM XL Fortran 1.1	yes	yes	yes	no	yes	no
Motorola F77 2.0b3	yes	no	yes	no	yes	no

1 Variables assigned to constant expressions are not propagated.

Table 4.24: Optimization results for constant folding (local and global).

compiler	Integer		Floating-Point		Mixed	
	local	global	local	global	local	global
BSD Unix F77 1.0	yes	no	yes	no	no	no
Mips F77 2.0 -O2	yes	yes	yes	yes	yes	yes
Mips F77 2.0 -O1	yes	no	yes	no	partial[1]	no
Sun F77 1.3 -O3	yes	yes	yes	yes	yes	yes
Sun F77 1.3 -O2	yes	yes	yes	yes	yes	yes
Sun F77 1.3 -O1	no	no	no	no	no	no
Ultrix Fort 4.5	yes	yes	yes	yes	yes	yes
Amdahl F77 2.0	no	no	no	no	no	no
CRAY CFT77 4.0.1	yes	partial[2]	yes	partial[2]	yes	partial[2]
IBM XL Fortran 1.1	yes	yes	partial[3]	partial[3]	partial[3]	partial[3]
Motorola F77 2.0b3	yes	partial[2]	yes	partial[2]	yes	partial[2]

1 Not all common subexpressions are recognized.
2 Incomplete global analysis is not enough to detect all optimizations.
3 Transformations to the intermediate code destroy some common subexpressions.

Table 4.25: Optimization results for common subexpression elimination (local and global)."

compiler	Integer		Floating-Point		Mixed	
	local	global	local	global	local	global
BSD Unix F77 1.0	partial[1]	partial[1]	no	no	marginal[2]	marginal[2]
Mips F77 2.0 -O2	yes	yes	yes	yes	yes	yes
Mips F77 2.0 -O1	no	no	no	no	no	no
Sun F77 1.3 -O3	yes	yes	yes	partial[3]	yes	partial[3]
Sun F77 1.3 -O2	yes	yes	yes	partial[3]	yes	partial[3]
Sun F77 1.3 -O1	no	no	no	no	no	no
Ultrix Fort 4.5	yes	yes	yes	yes	yes	yes
Amdahl F77 2.0	no	no	no	no	no	no
CRAY CFT77 4.0.1	yes	partial[3]	yes	partial[3]	yes	partial[3]
IBM XL Fortran 1.1	yes	yes	yes	yes	yes	yes
Motorola F77 2.0b3	yes	no	yes	no	yes	no

1 Only simple expressions are moved (<var> <op> <var>).
2 Only simple integers expressions.
3 Blocks inside the loop are not considered.

Table 4.26: Optimization results for code motion (local and global).

compiler	Integer		Floating-Point		Mixed	
	local	global	local	global	local	global
BSD Unix F77 1.0	partial[1]	no	partial[1]	no	partial[1]	no
Mips F77 2.0 -O2	partial[2]	marginal[1]	partial[2]	marginal[1]	partial[2]	marginal[1]
Mips F77 2.0 -O1	marginal[1]	no	marginal[1]	no	marginal[1]	no
Sun F77 1.3 -O3	no	no	no	no	no	no
Sun F77 1.3 -O2	no	no	no	no	no	no
Sun F77 1.3 -O1	no	no	no	no	no	no
Ultrix Fort 4.5	yes	partial[3]	yes	partial[3]	yes	partial[3]
Amdahl F77 2.0	no	no	no	no	no	no
CRAY CFT77 4.0.1	yes	no	yes	no	yes	no
IBM XL Fortran 1.1	partial[4]	marginal[1]	partial[4]	marginal[1]	partial[4]	marginal[1]
Motorola F77 2.0b3	no	no	no	no	no	no

1 Propagates only simple assignments (<var> = <var>).
2 Compiler has a limited lookahead.
3 Incomplete global analysis is not enough to detect all optimizations.
4 Transformations to the intermediate code destroy some common subexpressions.

Table 4.27: Optimization results for copy propagation (local and global).

compiler	Integer		Floating-Point		Mixed	
	local	global	local	global	local	global
BSD Unix F77 1.0	no	no	no	no	no	no
Mips F77 2.0 -O2	yes	yes	yes	yes	yes	yes
Mips F77 2.0 -O1	no	no	no	no	no	no
Sun F77 1.3 -O3	yes	yes	yes	yes	yes	yes
Sun F77 1.3 -O2	yes	yes	no	no	no	no
Sun F77 1.3 -O1	no	no	no	no	no	no
Ultrix Fort 4.5	yes	yes	yes	yes	yes	yes
Amdahl F77 2.0	no	no	no	no	no	no
CRAY CFT77 4.0.1	yes	yes	yes	yes	yes	yes
IBM XL Fortran 1.1	yes	yes	yes	yes	yes	yes
Motorola F77 2.0b3	no	no	no	no	no	no

Table 4.28: Optimization results for dead code elimination (local and global).

compiler	strength reduction	address calculation	inline substitution		loop unrolling
			apply	affects	
BSD Unix F77 1.0	partial[1]	marginal[2]	no	no	no
Mips F77 2.0 -O2	yes	yes	partial[2]	no	yes[4]
Mips F77 2.0 -O1	no	yes	no	no	no
Sun F77 1.3 -O3	partial[1]	marginal[1]	no	no	yes[4]
Sun F77 1.3 -O2	partial[1]	marginal[1]	no	no	yes[4]
Sun F77 1.3 -O1	partial[1]	no	no	no	yes[4]
Ultrix Fort 4.5	yes	yes	no	no	no
Amdahl F77 2.0	no	no	no	no	no
CRAY CFT77 4.0.1	yes	yes	yes	yes	yes[4]
IBM XL Fortran 1.1	yes	yes	yes	no[3]	no
Motorola F77 2.0b3	partial[1]	no	no	no	no

1 Optimization is partially applied.
2 There is no code substitution; caller and callee use same stack frame.
3 There is code substitution; extra code obscures optimization.
4 Arbitrary unrolling of loops.

Table 4.29: Optimization results for strength reduction, address calculation, inline substitution, and loop unrolling.

5 Locality Effects and Characterization of the Memory Hierarchy

5.1. Summary

In this chapter we deal with the issue of locality and incorporate this factor in our performance model. The model presented in the last chapters ignores how the stream of memory references generated by a program affects the contents of the cache and the TLB, which is tantamount to assuming that the amount of locality in the program does not affect its execution time. All machines, however, attempt to exploit in many ways the spatial and temporal locality of programs in order to improve performance, and the amount of locality present is a function of how the instructions are executed and how the data is accessed. Because our measuring tools are based primarily on timing a small sequence of instructions, which are executed many times in order to get a significant statistic, these measurements tend to reflect what happens when locality is high. This means that, for a program with bad locality, our prediction will tend to underestimate the actual execution time. Our results on the SPEC and Perfect benchmarks in Chapter 3 do not show large errors because the locality on most of these programs is relatively high [Pnev90, GeeJ91].

Here we show that our basic model can be extended to include a term which accounts for the time delay experienced by a program as a result of bringing data to the processor from different levels of the memory hierarchy. We focus on characterizing the cache and TLB units by running experiments which measure their most relevant parameters. We present cache and TLB measurements for a variety of computers. We then combine these measurements with results obtained by other studies of the cache and TLB miss ratios for the SPEC benchmarks to compute the delay experienced by these programs as a result of the cache and TLB misses. These new results are then used to evaluate how much our execution time predictions for the SPEC benchmarks improve when we incorporate these memory delays. We show that the prediction errors decrease in most of the programs, although the improvement is modest. We also consider the SPEC benchmarks as being part of a single workload and use them to evaluate the impact of

memory delay in the overall performance of different machines.

Finally, we discuss in some detail the performance differences between the caches and TLBs of four machines based on the same family of processors. We show that the SPEC benchmark results on these machines can be explained by the differences in their memory systems.

5.2. Introduction

The new generation of computers achieve high-performance by maintaining a balance between the performance of individual components: integer, branch, and floating-point units, caches, bus, memory system, and I/O units. Cache memories, which are present in almost all of today's computers, are an essential component in reducing the increasing performance gap between the CPU and main memory [Smit82]. This has allowed using faster and faster microprocessors without requiring very complex and far more expensive memory systems in order to maintain a small memory access time.

The reason why caches are effective is because programs have the property of distributing their memory references nonuniformly over their address space; in other words, the sequence of memory references exhibit temporal and spatial locality [Smit82]. Because caches try to keep those locations which have a higher probability of being reuse in the near future, arbitrary permutations of the sequence of memory references produce significantly different execution times. These variations in the execution time due to changes in locality are not captured by our performance model, which ignores how the stream of references affects the content of both the cache and the TLB. This is a direct consequence of using a linear model, and it is clearly expressed in the following equation

$$T_{A,M} = \sum_{i=1}^{n} C_{i,A} P_{i,M} \tag{5.1}$$

where $T_{A,M}$ is the total execution time of the program, $C_{i,A}$ is the number of times operation i is executed by program A, and $P_{i,M}$ is the execution time of parameter i on machine M.

Although equation (5.1) does not capture the variation in the execution time due to locality, it represents a good approximation that allows us to predict quite well the execution time of programs, as we have shown in previous chapters. This is the case because most of the programs we have used in these study exhibit a high degree of locality, so our predictions do not show a bias due to underestimating the effect of cache misses. The only exception is *MATRIX300*, where the predictions are consistently lower than the actual values.

5.3. Locality and the Abstract Machine Performance Model

We can modify equation (5.1) to include the performance penalty due to high miss ratios at every level of the memory hierarchy, by introducing a term that models the execution time lost in bringing blocks of data down the hierarchy. This new equation is

$$T_{A,M} = \sum_{i=1}^{n} C_{i,A} P_{i,M} + \sum_{i=1}^{m} H_{i,A} D_{i,M}, \tag{5.2}$$

where $H_{i,A}$ is the number of misses at the level i of the memory hierarchy, and $D_{i,M}$ is the penalty paid by the respective miss. How many levels of the memory hierarchy exist

varies between machines, but in most machines there are one or two levels of caches, a TLB, main memory, and disk[1]. In order to use equation (5.2) we need: 1) to measure the number of misses at each level of hierarchy, or at least on those levels which significantly affect the execution time, and 2) to measure the set of penalties due to different types of misses.

Measuring, or at least approximating, the number of misses of a program on a particular machine is relatively straightforward. All we need is a trace of all memory references and a cache simulator. The trace is passed through the cache simulator which is set to the primary or secondary cache or TLB parameters found on the machine. This technique, known as *trace-driven simulation*, is the most common way of evaluating the effectiveness of caches and TLBs [Smit82, Smit85, Hill89, Borg90].

In this paper we focus more on characterizing the performance impact of miss penalties than on measuring the number of misses. Furthermore, we are interested mainly in the cache and TLB, because these two structures are the most influential on the performance of a program. Nevertheless, our approach extends trivially to the other levels of the memory hierarchy. We also assume that the numbers of cache and/or TLB misses of a program can be obtained using traditional techniques.

5.4. Characterizing the Performance of the Cache and TLB

In order to characterize and compare the cache and TLB organizations of different computers, we have written a benchmark that measures the main performance parameters of the cache and TLB. Basically, our program consists of making two hundred observations of a single test covering all the combinations of: 1) the size of the address space touched by the experiment and 2) the distance between two consecutive addresses sent to the cache. By varying these two dimensions we can measure: a) the size of the cache and TLB; b) the size of a cache line and the granularity of a TLB entry; c) the execution time needed to satisfy a cache or TLB miss, or/and the time to load a cache line in the case of a cache with a wraparound load; 4) the cache and TLB associativity; and 5) the performance effect of write buffers. Other parameters like the number of sets in the cache or entries in the TLB are obtained easily from the above parameters.

Previous studies have also characterized the cache miss penalty by observing the changes in execution time of a test program as a function of the number of cache misses. In particular, Peuto and Shustek [Peut77] used this technique to evaluate the effect of cache misses in the total execution time on the IBM 370/168 and the Amdahl 470 V/6. They found that even when the CPU performance of the Amdahl was greater than that of the IBM, cache effects accounted for only 1% to 5% of the execution time on the IBM, and 3% to 9% on the Amdahl.

---

[1] The TLB is not a level in the memory hierarchy, but it is a high-speed buffer which maintains recently used virtual and real memory address pairs [Smit82]. However, to simplify our discussion in the rest of the paper we consider it as part of the memory hierarchy. Doing this does not affect in any way our methodology or conclusions.

5.4.1. Experimental Methodology

We will explain how different parameters of the cache are measured by first assuming that there is no TLB present, so the only effect observed is due to the cache. In what follows we assume the existence of separate instruction and data caches, although this is done only to simplify the discussion. Assume that a machine has a cache capable of holding D 4-byte words, a line size of b words, and an associativity a. The number of sets in the cache is given by D/ab. We also assume that the replacement algorithm is LRU, and that the lowest available address bits are used to select the cache set.

Each of our experiments consists of computing many times a simple floating-point function on a subset of elements taken from a one-dimensional array of N 4-byte elements. This subset is given by the following sequence: $1, s+1, 2s+1, \ldots, N-s+1$. Thus, each experiment is characterized by a particular value of N and s. The stride s allows us to change the rate at which misses are generated, by controlling the number of consecutive accesses to the same cache line, cache set, etc. The magnitude of s varies from 1 to $N/2$ in powers of two. In addition, the main loop which computes the function is executed many times in order to eliminate start-up effects and improve the accuracy of the measurement.

Computing a new value on a particular element involves reading first the element into the CPU, computing the new value using a simple recursive equation, and writing the result back into the cache. Thus, on each iteration the cache gets two consecutive requests, one read and one write, having both the same address. Of these two requests only the read can generate a cache miss, and it is the time needed to fetch the value for the read that our experiments measure.

Depending on the magnitudes of N and s in a particular experiment, with respect to the size of the cache (D), the line size (b), and the associativity (a), there are four possible regimes of operations; each of these is characterized by the rate at which misses occur in the cache. A summary of the characteristics of the four regimes is given in table 5.1.

Regime	Size of Array	Stride	Frequency of Misses	Time per Iteration
1	$1 \le N \le D$	$1 \le s \le N/2$	no misses	$T_{no-miss}$
2.a	$D < N$	$1 \le s < b$	one miss every b/s elements	$T_{no-miss} + Ms/b$
2.b	$D < N$	$b \le s < N/a$	one miss every element	$T_{no-miss} + M$
2.c	$D < N$	$N/a \le s \le N/2$	no misses	$T_{no-miss}$

Table 5.1: Cache miss patterns as a function of N and s. No misses are generated when $N \le D$. When $N > D$, the rate of misses is determined by the stride between consecutive elements.

Regime 1: $N \le D$.

The complete array fits into the cache and thus, independently of the stride s, once the array is loaded for the first time, there are no more misses. The execution time per iteration ($T_{no-misses}$) includes the time to read one element from the cache, compute its new value, and store the result back into the cache. In a cache where the update policy is *write-through*, in which write operations to data are also done simultaneously to main memory, $T_{no-miss}$, may also include the time that the

processor is forced to wait if the write buffer happens to overflows.

Regime 2.a: $N > D$ and $1 \leq s < b$.

The array is bigger than the cache, and there are b/s consecutive accesses to the same cache line. The first access to the line always generates a miss, because every cache line is displaced from the cache before it can be re-used in subsequent computations of the function. This follows from condition $N > D$. Therefore, the execution time per iteration is $T_{no-miss} + Ms/b$, where M is the miss penalty and represents the time that it takes to read the data from main memory and resume execution.

Regime 2.b: $N > D$ and $b \leq s < N/a$.

The array is bigger than the cache and there is a cache miss every iteration, as each element of the array maps to a different line. Again, every cache line is displaced from the cache before it can be re-used. The execution time per iteration is $T_{no-miss} + M$.

Regime 2.c: $N > D$ and $N/a \leq s \leq N/2$.

The array is bigger than the cache, but the number of addresses mapping to a single set is less than the set associativity; thus, once the array is loaded, there are no more misses. Even when the array has N elements, only $N/s < a$ of these are touched by the experiment, and all of them can fit in a single set. This follows from the fact that $N/a \leq s$. The execution time per iteration is $T_{no-miss}$.

Figure 5.1 illustrates the state of the cache in each of the four regimes. In these examples we assume that the cache size is large enough to hold 32 4-byte elements, the cache line is 4 elements long, and the associativity is 2. We also assume that the replacement policy is LRU, and that the first element of the array maps to the first element of the first line of the cache. On each of the cache configurations we highlight those elements that are read and generate a miss, those that are read but do not generate a miss, and those that are loaded into the cache as a result of accessing other elements in the same line, but are not touched by the experiment. The four diagrams in upper part of the figure corresponds to regime 1. Here the size of the array is equal to the cache size, so, independently of the value of s, no misses occur. If we double N, which is represented by the lower half of the figure, then cache misses will occur at a rate which depends on the value of s. The leftmost diagram represents regime 2.a, the middle two diagrams regime 2.b, and the rightmost diagram regime 2.c.

5.4.2. Measuring the Characteristics of the Cache

By making a plot of the value of the execution time per iteration as a function of N and s, we can identify where our experiments make a transition from one regime to the next, and using this information we can obtain the values of the parameters that affect the performance of the cache and the TLB. In what follows we explain how these parameters are obtained.

5.4.2.1. Cache Size

Measuring the size of the cache is achieved by increasing the value of N until cache misses start to occur. When this happens the time per iteration becomes significantly larger than $T_{no-miss}$. The cache size is given by the largest N such that the average time

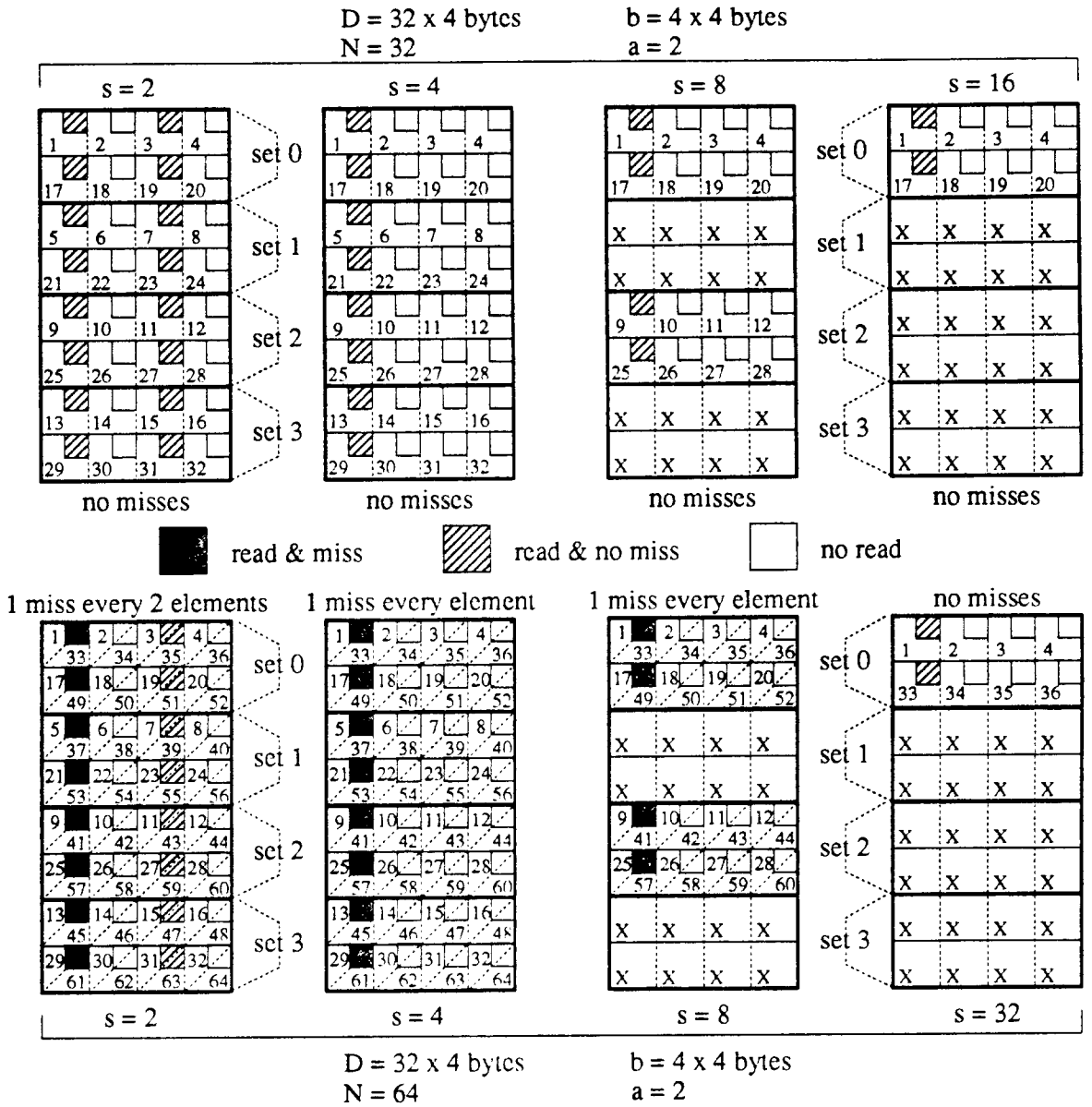


Figure 5.1: The figure illustrates the four different regimes of cache accesses produce by a particular combination of N and s. Each diagram shows the mapping of elements to cache entries, assuming that the first element of the array maps to the first entry of the first cache line in the cache. The replacement policy is LRU. The four diagrams on the upper part of the figure correspond to regime 1. For the diagram in the lower half of the figure, the leftmost diagram corresponds to regime 2.a, the two in the middle to regime 2.b, and the rightmost to regime 2.c. The sequence of elements reference by an experiment is: $1, s+1, 2s+1, \ldots, N-s+1$.

iteration is equal to $T_{no-miss}$.

5.4.2.2. Average Miss Delay

An experiment executing in regime 2.b generates a miss every iteration, while one in regime 1 does not generate misses, so the difference between their respective times gives the memory delay per miss. An alternative way is to measure the difference in the iteration time between regime 2.a and regime 1, and then multiply this difference by b/s, which is the number of references per miss.

5.4.2.3. Cache Line Size

In regime 2.a, the rate at which misses occur is one every b/s iterations. This rate increases with s, and achieves its maximum when $s \geq b$, when there is a miss on every iteration (regime 2.b). The value of s when the transition between regimes 2.a and 2.b happens gives the cache line size.

5.4.2.4. Associativity

The associativity of the cache (for $a \geq 2$) is given by the value N/s, which is the number of different elements referenced by the experiments, if it is the case that there is a transition from regime 2.b to regime 2.c. As we mentioned before, if $N/a \leq s$, then $a \geq N/s$, which means that this regime is easily identified because the time per iteration drops from $T_{no-miss} + M$ to $T_{no-miss}$. In a direct-mapped cache, however, there is no transition because the maximum value of s for our experiments is $N/2$, which corresponds to an associativity of 2. However, we can identify a direct-mapped cache when we observe that the time per iteration does not decrease when s equals $N/2$.

5.4.2.5. Write Buffers

A cache, where the update policy is write-through, normally contains a small buffer of pending writes which are sent to main memory only when the memory bus is not being used to satisfy fetches. The existence of a write buffer allows giving priority to reads over writes. This helps in reducing the amount of time that the CPU has to stall waiting for writes to complete. Furthermore, if the processor needs a datum which is still in the write buffer, it can be read immediately from there without waiting for the write to occur. The existence of write buffers and their depth can be detected by observing how the time per iteration changes as s gets closer to $N/2$.

We know that the number of different elements touched by a particular experiment is N/s. This number decreases as s increases, which means that the time between two accesses to the same element also decreases. In a cache with a write buffer it may happen that if s is very close to $N/2$, then the time from the moment an element is written until it is read again can become smaller than the time it takes for the write to occur, so the fetch can be retrieved from the buffer. When this occurs the time per iteration will decrease by the difference in time between fetching the data from the write buffer and fetching it from memory.

5.4.3. Measuring Parameters of the TLB

The phenomena we observe when we consider the TLB is similar to what happens with the cache, the only difference resides in the particular values of N and s where the changes in behavior occur. These changes are proportional to the size of the address space covered by the TLB, the granularity of the TLB entries, and the magnitude of the miss delay. The measurements we present in the next section show the behavior of both the cache and the TLB when both are active, and in some regions their effects overlap. However, in all cases it is relatively straightforward to isolate the effects of one from the other.

5.5. Experimental Results for Caches and TLBs

We ran our benchmark on several computers, and we show the results in figures 5.2-5.4 and in table 5.2. The graphs shown in the figures depict the average time per iteration as a function of the size of the array and the stride, while table 5.2 summarizes the cache and TLB parameters extracted from the profiles. The units used in the graphs are: bytes for measures of size, and nanoseconds for time related magnitudes. Each curve on each of the graphs correspond to a particular array size (N), while the horizontal axis represents different stride values (s). We only shown curves for array sizes that are greater or equal to the size of the cache.

The four basic regimes for the different cache and TLB miss patterns can be seen clearly in most of the figures. The best example of this are the results for the IBM RS/6000 530 (fig. 5.2, lower-right graph). On this machine, regime 1 is represented by the curve labeled 64K. The other three regimes of cache misses are in the three curves with labels 128K, 256K, and 512K. The first segment on each curve, where the value of the average time per iteration increases proportionally to the magnitude of s, corresponds to regime 2.a. The next portion of the curve, when the time per iteration is almost constant corresponds to regime 2.b, and the sudden drop in the time per iteration at the end is where regime 2.c starts. In the same graph, curves for array sizes of 1M, 2M, and 4M show the same 2.b regimes but overlapped with the effects of both the cache and the TLB.

The results in table 5.2 for the DEC 3100, DEC 5400, MIPS M/2000, and DEC 5500 show the differences in their cache organizations. These four machines use the R2000/R2001 or R3000/R3001 processors from MIPS Co. All have a 64KB, direct mapped cache, and a fully-associative TLB, and with 64 entries with an entry granularity of 4096 bytes. The main difference between their respective caches are the line size and the miss penalty. The DEC 3100 has the smallest line size having only 4 bytes [Furl90]; the DEC 5400 and 5500 have line sizes of 16 bytes, and the MIPS M/2000 has the largest line size of 64 bytes. The miss penalty per line also shows a wide range of values, from 540 ns for the DEC 3100 to 1680 ns for the DEC 5400.

It is interesting to compare the ratios between the cache and TLB penalty misses and the execution time of a single iteration with no misses, which are given in table 5.2. Although the no-miss time of the test is not a good measure of the true speed of the processor, it at least gives an indication of the basic floating-point performance (add and multiply) and helps putting in perspective the miss penalties. The results show a large variation in the normalized cache penalties, ranging from 0.50 on the Sparcstation 1 to 4.32 on the VAX 9000. In the VAX 9000 loading a cache line takes 5.3 times longer

than the no-miss iteration time. With respect to TLB misses the range of values goes from 0.53 to 6.35, with the highest value corresponding to the IBM RS/6000 530.

In comparing these results we have to keep in mind that the effectiveness of a cache and TLB depends on many factors and that the miss penalty is only one of these. For example, doubling the cache's line size tends to significantly reduce the miss ratio, but, other things being equal, it increases the line fetch time [Smit87]. The benefits of a larger line size are more evident on scientific programs, where traversing large matrices is done in regular strides.

The fact that the miss penalty on the DEC 3100 is only 540 ns does not in itself represent a well designed cache. Memory structures, in order to support good performance on many different workloads, should maintain certain aspect ratios, i.e., the ratio between the size of a block (line) and number of blocks (sets) [Alpe90]. Increasing only the size of a cache tends to decrease the number of capacity misses, but does not decrease the number of compulsory misses [Hill89]. *Capacity misses* are those that are caused by the size of the cache, and by attempting to map a larger region of memory into a smaller region in the cache. *Compulsory misses* are those that are caused by changes in the loci of execution; the program starts to reference data not previously loaded in the cache. The number of compulsory misses can be reduced by increasing the line size, but on a small cache doing this can result in an increase in the number of capacity misses. In fact, a line size of 4 bytes is too small for a cache of 64 Kbytes [Smit87]. Most high performance workstation have line sizes ranging from 16 bytes to 128 bytes (IBM RS/6000 series).

5.5.1. Effective Prefetching

An interesting characteristic on the IBM RS/6000 which can be observed in our measurements is what we call *effective prefetching*. The cache does not have hardware support to do prefetching [O'Bri90], but it can produce the same effect, that is, fetching cache lines before they are needed by the computation, thus preventing the processor from stalling. This is accomplish by having independent integer, branch, and floating-point units. The integer and branch unit can execute several instructions ahead of the floating-point unit in floating-point intensive code and generate loads to the cache that even in the presence of misses arrive before the floating-point unit requires the values [O'Bri90]. Because the execution of our tests is dominated by floating-point operations, the illusion of prefetching is present in our measurements. This is evident on the left side of the RS/6000 curves (regime 2.a), independent of the address space region; as long as the stride is less or equal to 16 bytes (4 words) there is no miss penalty.

5.5.2. TLB Entries with Different Granularities

The results for the Sparcstation 1 and 1+ show that their respective TLB entry granularities are 128 Kbytes and at least more than 2 Mbytes. The reason for these large numbers is that the TLB entries can map memory regions using four different levels of granularity. Furthermore, entries with different granularities can coexist in the TLB. In addition, the Sparcstation has a virtual cache. In a virtual cache there is no need to translate addresses on every cache reference, but only after a cache miss or when the first write to unmodified page occurs. This eliminates the critical path between the TLB and cache, while at the same time it introduces the problem of synonyms across different

Cache Parameters					
	DEC 3100	**DEC 5400**	**MIPS M/2000**	**VAX 9000**	**RS/6000 530**
cache size	64 KB	64 KB	64 KB	128 KB	64 KB
associativity	1-way	1-way	1-way	2-way	4-way
line size	4 bytes	16 bytes	64 bytes	64 bytes	128 bytes
miss penalty (word)	540 ns	1680 ns	800 ns	740 ns	350 ns
normalized penalty	0.6490	2.2400	1.2389	4.0000	2.0588
miss penalty (line)	540 ns	1680 ns	1440 ns	980 ns	700 ns
normalized penalty	0.6490	2.2400	2.5477	5.2973	4.1176
miss penalty / line	540 ns	420 ns	90 ns	61 ns	22 ns
normalized penalty	0.6490	0.5600	0.1592	0.3297	0.1294
virtual prefetching	no	no	no	no	yes
TLB Parameters					
	DEC 3100	**DEC 5400**	**MIPS M/2000**	**VAX 9000**	**RS/6000 530**
region covered	256 KB	256 KB	256 KB	8 MB	512 KB
num. of entries	64	64	64	1024	128
associativity	64-way	64-way	64-way	2-way	2-way
entry granularity	4096 bytes	4096 bytes	4096 bytes	8192 bytes	4096 bytes
miss penalty (entry)	480 ns	400 ns	350 ns	280 ns	1080 ns
normalized penalty	0.5769	0.5333	0.6194	1.5135	6.3529
page size	4096 bytes	4096 bytes	4096 bytes	8192 bytes	4096 bytes

Cache Parameters				
	HP 9000/720	**Sparc 1**	**Sparc 1+**	**DEC 5500**
cache size	256 KB	128 KB	64 KB	64 KB
associativity	1-way	1-way	1-way	1-way
line size	32 bytes	16 bytes	16 bytes	16 bytes
miss penalty (word)	360 ns	780 ns	560 ns	750 ns
normalized penalty	1.6744	0.5652	0.5091	1.875
miss penalty (line)	480 ns	780 ns	560 ns	750 ns
normalized penalty	2.2326	0.5652	0.5091	1.875
miss penalty / line	60 ns	195 ns	140 ns	188 ns
normalized penalty	0.2791	0.1413	0.1273	0.4700
virtual prefetching	no	no	no	no
TLB Parameters				
	HP 9000/720	**Sparc 1**	**Sparc 1+**	**DEC 5500**
region covered	512 KB	8 MB	1 GB	256 KB
num. of entries	64	64	64	64
associativity	64-way	64-way	64-way	64-way
entry granularity	8192 bytes	128 Kbytes	16 Mbytes	4096 bytes
miss penalty (entry)	940 ns	880 ns	n.a.	260 ns
normalized penalty	4.3721	0.6377	n.a.	0.6500
page size	8192 bytes	4096 bytes	4096 bytes	4096 bytes

Table 5.2: Cache and TLB parameters measured using the memory hierarchy benchmark. The normalized penalty is the ratio between the cache penalty time and the no-miss execution time per iteration. We define virtual prefetching as the ability of the machine to satisfy cache misses before the subunit that consumes the data needs the values, which is manifested in the execution time as a zero-cycle miss delay.

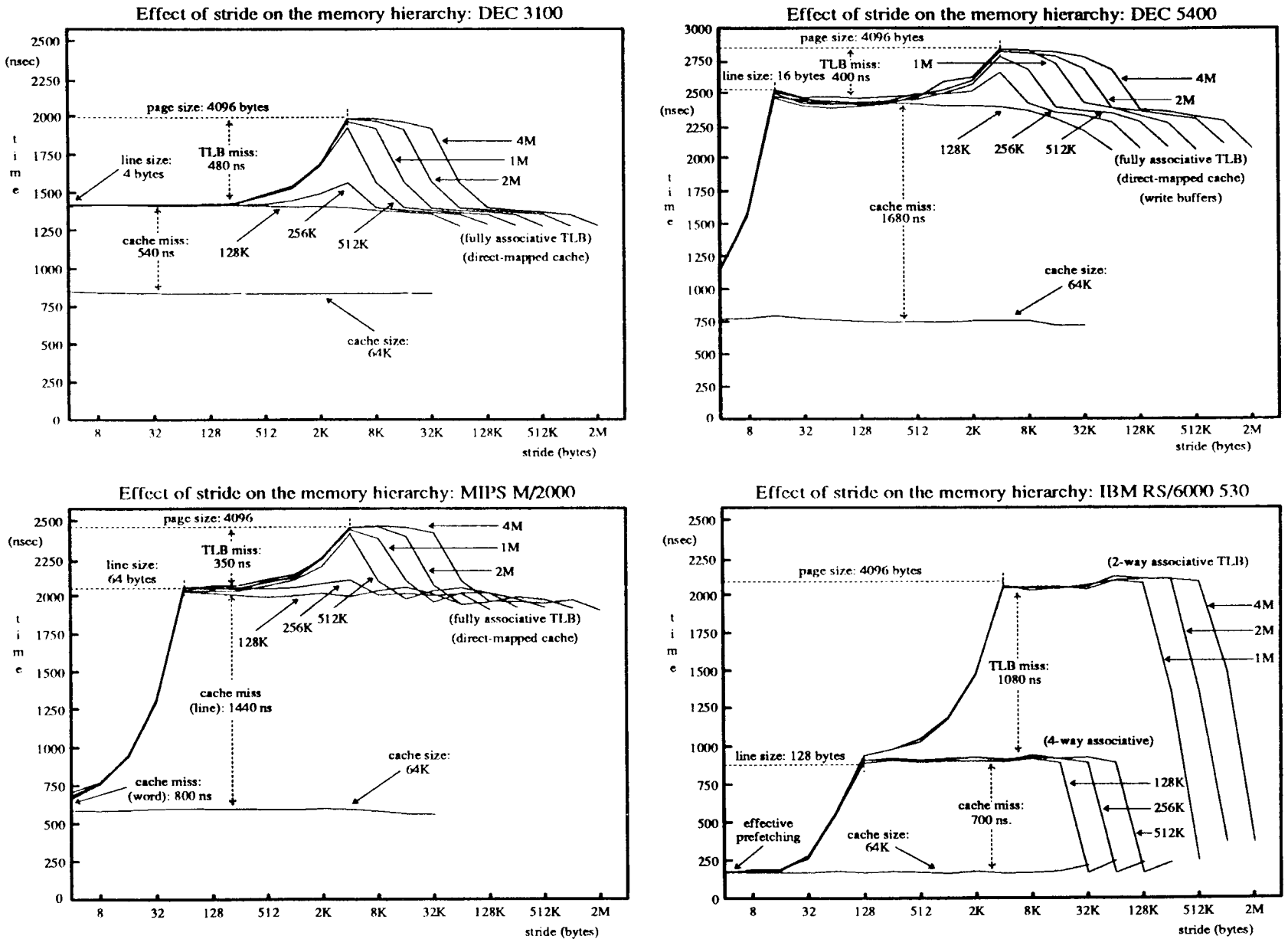


Figure 5.2: Profile of the performance of the memory hierarchy (cache and TLB) on the DECstation 3100, MIPS M/2000, Decstation 5400, and IBM RS/6000 530. Each curve indicates the amount of address space touched by the experiment and the stride represents the distance between two consecutive addresses.

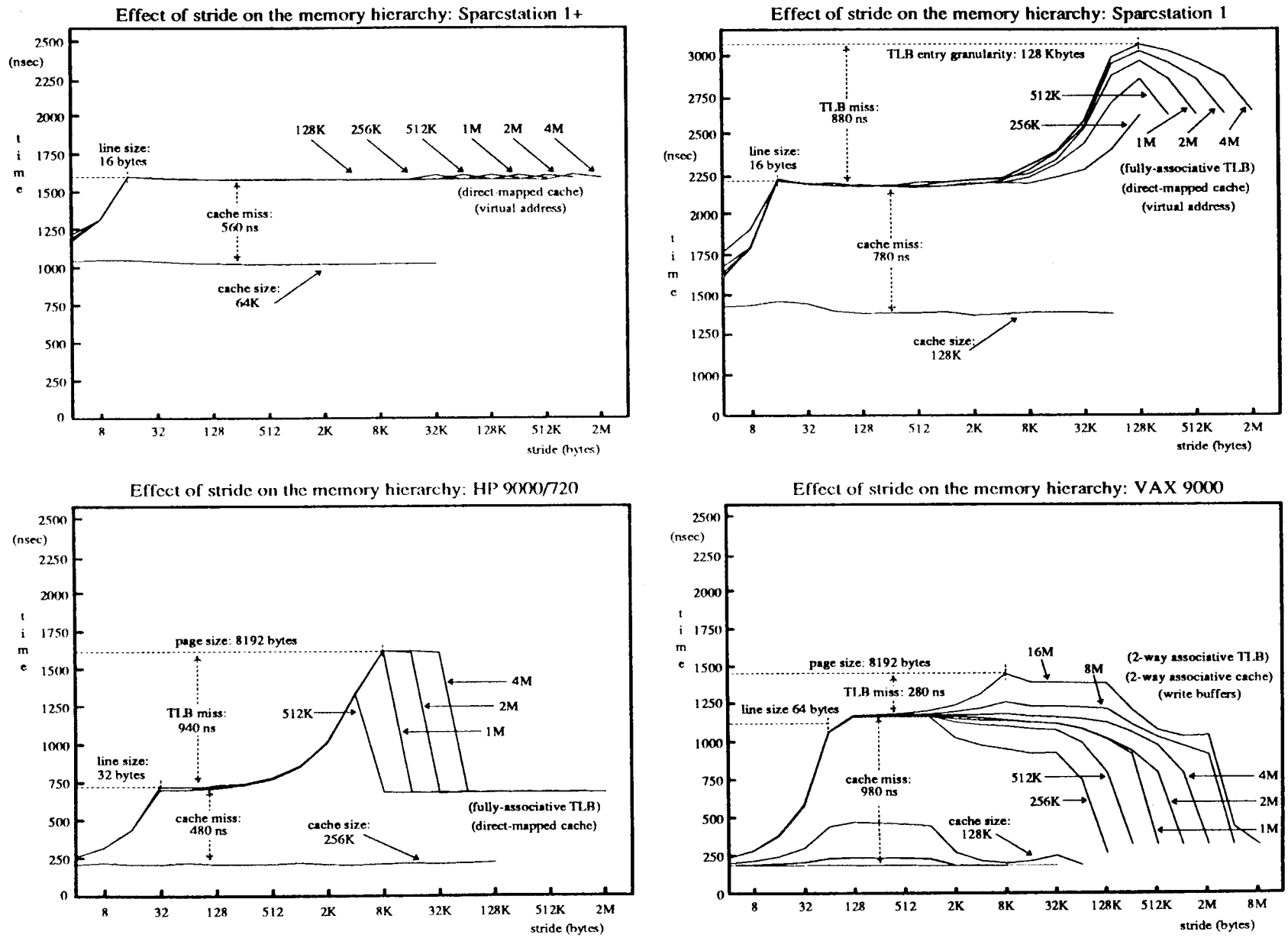


Figure 5.3: Profile of the performance of the memory hierarchy (cache and TLB) on the Sparcstation 1, Sparcstation 1+, HP 9000/720, and VAX 9000. Each curve indicates the amount of address space touched by the experiment and the stride represents the distance between two consecutive addresses.

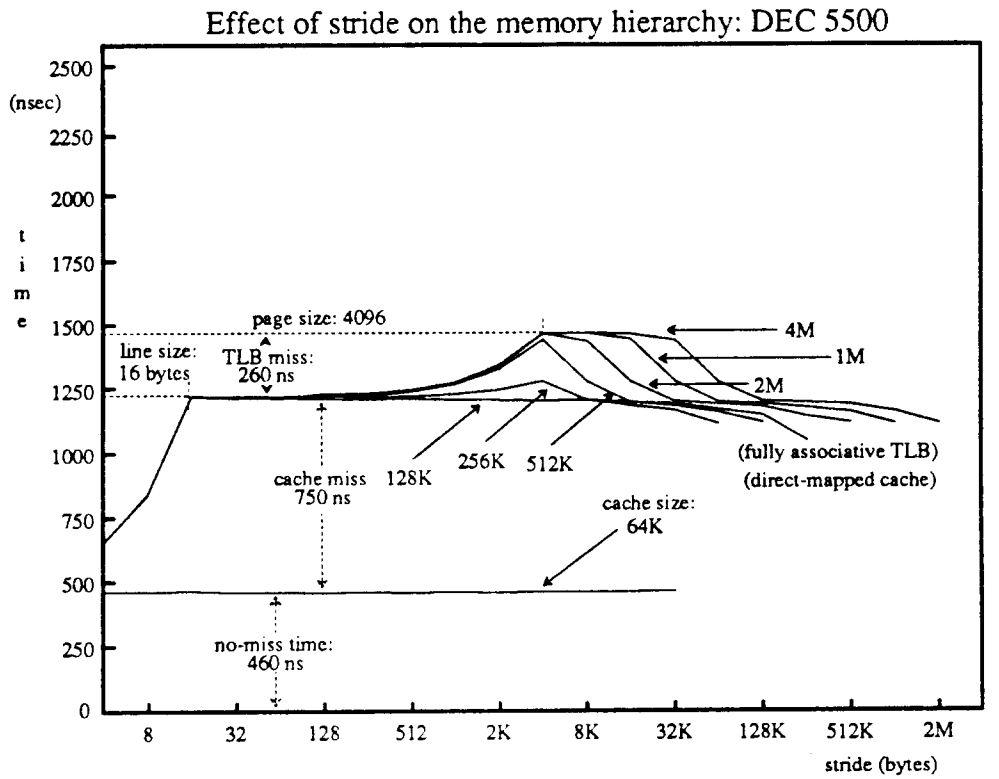


Figure 5.4: Profile of the performance of the memory hierarchy (cache and TLB) on the DECstation 5500. Each curve indicates the amount of address space touched by the experiment and the stride represents the distance between two consecutive addresses.

virtual address spaces.

We can understand why the TLB can contain entries with different granularities by considering how page table entries are stored in the page tables and how they are found when a TLB miss occurs [CYPR90]. After a TLB miss, the MMU starts a table walk through a 4-level table structure. Page tables at any level in the tree walk can contain both *page table entries* (PTEs) and *page table pointers* (PTPs). A PTE points to a region of physical memory, while a PTP points to another page table lower in the walk. An entry type field indicates whether the value found in the page table should be interpreted as a PTE or a PTP. The search done by the MMU ends when a PTE is found, and this may happen in any of the four levels. It is this level which determines the size of the memory region mapped. Thus, a page table entry in the Sparcstation can point to regions of sizes: 4GB (level 0), 16MB (level 1), 256KB (level 2), and 4KB (level 3). Once the PTE is found, the physical address and the mapping granularity, along with other access information are stored in the TLB.

When a virtual address is submitted to the TLB for translation, it is compared against all entries in the TLB. A virtual address in the Sparcstation contains four index fields, and how many of these fields are used for matching against a particular entry is

determined by the granularity of the entry. This granularity also determines the number of bits from the virtual address that are concatenated to the PTE to form the physical address. We can see that TLB entries with different granularities can coexist as long as none of the regions covered by the TLB overlap. Normally, the determination of how large is an address mapping covered by a PTE is normally done when the region is first allocated by the operating system.

The are several advantages in using page tables entries with different granularities. First, by using this scheme the TLB can cover a larger region of memory, and, all other things being equal, doing this reduces the number of TLB misses. Second, the operating system can make decisions about how many PTEs to use when mapping a large region of address space, based on the availability and demand of memory. This helps to improve the effectiveness of the TLB when the load of the system is low, without decreasing its performance when more processes are competing for memory. Finally, a TLB with LRU replacement policy will tend not to discard entries covering large regions of memory, because the number of references falling within those regions increase in proportion to their sizes.

5.6. The Effect of Locality in the SPEC Benchmarks

In this section we combine the experimental cache and TLB results obtained in the last section with the cache and TLB miss ratios for the Fortran SPEC benchmarks to compute the memory delay caused by misses. We then use these results to evaluate: 1) whether our execution time predictions improve when we incorporate the memory delay experience by the programs; and 2) how much impact does each cache and TLB configuration have on the overall performance of their respective machines.

5.6.1. The SPEC Benchmarks Cache and TLB Miss Ratios

The experimental cache and TLB measurements of the memory hierarchy obtained in the last section can be combined with previously computed miss ratios on SPEC Fortran benchmarks to compute the specific miss ratios that each machine experiences.

Gee et al. [GeeJ91] and Gee and Smith [GeeJ92] have measured the cache and TLB miss ratios for the entire suite of SPEC benchmarks and have compared their results against other measurements based on hardware monitors, very long address traces, and those which include operating system and multiprogramming behavior. They found that the instruction cache miss ratios on all the SPEC benchmarks are very low, while data cache miss ratios for the integer benchmarks are consistently lower than published results. The data miss ratios for the floating-point benchmarks, however, are significantly higher and they appear to be in agreement with previous measurements.

In this section we use their results on the Fortran SPEC benchmarks to obtain the approximate miss ratios for the different cache and TLB configurations characterized in the previous sections. We have to keep in mind when looking at the following results that the miss ratios reported in [GeeJ91, GeeJ92] were obtained under specific conditions which are not necessarily valid on all systems. The measurements were obtained in the following way: 1) by using traces taken from the DECstation 3100 which contains the MIPS 2000 microprocessor; 2) by using specific Fortran and C compilers; and 3) by ignoring the effect of multiprogramming. For these and other reasons specific to their cache simulator, their miss ratios are only approximations of the true miss ratios on

Cache Miss Ratios

machine	DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	SPICE2G6	Average
DECstation 3100	0.0280	0.0814	0.2218	0.1860	0.2470	0.1758	0.1566
DECstation 5400	0.0140	0.0407	0.1109	0.0930	0.1235	0.0879	0.0783
MIPS M/2000	0.0107	0.0277	0.0501	0.0763	0.0977	0.0648	0.0546
VAX 9000	0.0004	**0.0001**	0.0188	**0.0292**	**0.0589**	**0.0317**	**0.0232**
IBM RS/6000 530	0.0003	**0.0001**	**0.0094**	0.0670	0.0703	0.0380	0.0309
HP 9000/720	**0.0001**	0.0342	0.0691	0.0679	0.0703	0.0371	0.0465
Sparcstation 1	0.0071	0.0405	0.1101	0.0881	0.1100	0.0698	0.0709
Sparcstation 1+	0.0140	0.0407	0.1109	0.0930	0.1235	0.0879	0.0783
DECstation 5500	0.0140	0.0407	0.1109	0.0930	0.1235	0.0879	0.0783
average	0.0098	0.0340	0.0902	0.0882	0.1139	0.0757	0.0686

TLB Miss Ratios

machine	DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	SPICE2G6	Average
DECstation 3100	0.0000	0.0000	0.0003	0.0993	0.0409	0.0048	0.0242
DECstation 5400	0.0000	0.0000	0.0003	0.0993	0.0409	0.0048	0.0242
MIPS M/2000	0.0000	0.0000	0.0003	0.0993	0.0409	0.0048	0.0242
VAX 9000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
IBM RS/6000 530	0.0000	0.0000	0.0019	0.0919	0.0410	0.0038	0.0231
HP 9000/720	0.0000	0.0000	0.0001	0.0503	0.0266	0.0010	0.0130
Sparcstation 1	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Sparcstation 1+	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
DECstation 5500	0.0000	0.0000	0.0003	0.0993	0.0409	0.0048	0.0242
average	0.0000	0.0000	0.0004	0.0599	0.0257	0.0027	0.0148

Table 5.3: Cache miss ratios and TLB miss ratios. These results are based on the miss ratios reported reported by Gee, et. al. [GeeJ91a and GeeJ91b] using trace-driven simulations.

machines not sharing the characteristics of the DECstation 3100 or where multiprogramming and operating activity are significant. However, we believe that the differences between the machines and their compilers are not significantly large as to invalidate the use of these results.

In table 5.3 we present the cache and TLB miss ratios for the different machines. All the results, except those for the DECstation 3100 were obtained by using the parameters of tables 5.2 and using the results published by Gee et al. The cache miss ratios for the DECstation 3100 were not obtained directly from their tables; the block size on this machine is only 4 bytes, while the cache miss ratios published in [GeeJ91] were computed for block sizes ranging from 16 to 256 bytes. However, we have made a crude approximation of the miss ratios on the DECstation 3100 by doubling the results computed for a line size of 16 bytes. We did this based on the observation that the precision of floating-point numbers used in the Fortran SPEC benchmarks is 8 bytes, and hence on a machine with a 32-bit memory interface, reading or writing the second part of a floating-point number never generates a miss, if the line size is at least 8 bytes long. On the other hand, when the line size is only 4 bytes long, if the first part of the floating-point number misses, then the second part also generates a miss.

On table 5.3, the smallest cache miss ratios for each of the programs are highlighted. The effect of associativity and a large block size can be seen in the miss ratios of the IBM RS/6000 530. The average miss ratio on this machine which has a

64KB, 4-way set associative cache with a 128-byte block size is 0.0309; this value is smaller than the 0.0465 miss ratio on the HP 9000/720 which has a larger 256KB, direct mapped cache with a 32-byte block size. Thus, this means that at least with respect to the SPEC Fortran benchmarks, increasing only the cache size does not decrease the miss ratio as fast as increasing both the associativity and the block size. Furthermore, it is the VAX 9000 which has the lowest average cache miss ratio of all machines. This machine has a 128KB, 2-way set associative cache with a 64-byte line. Hence, at least with respect to this particular workload it is not the machine having the largest cache size, or the longest cache line, or the highest degree of associativity the one with the smallest miss ratio, but the one which combines the three factors in a more balanced way. However, the effectiveness of the cache configurations may change for different workload.

With respect to the TLB, only three of the six programs exhibit TLB miss ratios that affect the execution time of the programs on the DECstations, MIPS M/2000, IBM RS/6000 530, and HP 9000/720. On these machines the TLB miss ratio for *MATRIX300* is almost 0.10, and for *NASA7* it is close to 0.04. Furthermore, the degree of associativity appears not to affect the TLB miss ratios on these two programs. The results in [GeeJ92], however indicate that a TLB with 256 entries, 2-way set associative and with an entry granularity of 8KB will have miss ratios of less than 0.0001 on all benchmarks (including the C programs). Thus, we expect that the current SPEC benchmark suite will not test the performance of the TLB in new machines by the middle of this decade.

5.6.2. Execution Time Delay Due to Cache and TLB Misses

In table 5.4 we combine the cache and TLB miss ratios of the SPEC Fortran programs with the memory delays measured on each of the machines to compute the execution time penalty due to misses. We give both the individual delays for the cache and TLB, plus the sum of both delays. We also give the total amount of execution time delay on each of the machines. The results show that the delay due to TLB misses on benchmarks *MATRIX300* and *NASA7* is as large as the delay due to cache misses. Moreover, on the IBM RS/6000 530 the total delay due to TLB misses (187.57 sec) is larger than the delay due to cache misses (130.13).

5.6.3. Execution Prediction with Locality Delay

We can now use the execution time penalties due to cache and TLB misses computed in the last section to test if our execution time predictions can be improved by including the effect of locality. We do this by computing the overall machine and program prediction errors with and without the locality delay. Because our original predictions on the *SPICE2G6* (see table 3.33 in Appendix 3.C) were significantly larger than the actual execution times, indicated by a positive error, adding the latency overhead can only increase the prediction error. For this reason we have computed the overall prediction on all machines and programs twice: in one case we ignore the errors of *SPICE2G6*, and on the other case we include them. In this way we can evaluate if the execution time delay overhead actually improves our predictions on the other programs. A summary of the prediction errors is given in tables 5.5 and 5.6. The complete results, including the individual execution time predictions and prediction errors are given in table 5.13 in Appendix 5.A. In the tables, a negative (positive) average error means that our prediction was lower (greater) than the actual execution time.

Execution Time Penalty: Cache

machine	DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	SPICE2G6	Total
DECstation 3100	4.41	33.29	54.69	65.89	314.81	393.41	866.50
DECstation 5400	6.86	51.78	85.07	102.49	489.70	611.97	1347.87
MIPS M/2000	2.50	16.78	18.30	40.04	184.47	214.83	476.93
VAX 9000	0.09	0.06	6.35	14.17	102.87	97.21	220.75
IBM RS/6000 530	0.03	0.03	1.50	15.38	58.07	55.12	130.13
HP 9000/720	0.01	9.32	11.36	16.03	59.73	55.35	151.81
SPARCstation 1	1.62	23.92	39.21	45.08	202.51	225.62	537.96
SPARCstation 1+	2.29	17.26	28.36	34.16	163.23	203.99	449.29
DECstation 5500	3.06	23.12	37.98	45.75	218.61	273.20	601.73
average	2.32	19.51	31.42	42.11	199.33	236.74	531.44

Execution Time Penalty: TLB

machine	DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	SPICE2G6	Total
DECstation 3100	0.00	0.00	0.07	31.27	46.34	9.55	87.23
DECstation 5400	0.00	0.00	0.05	26.05	38.61	7.96	72.67
MIPS M/2000	0.00	0.00	0.05	22.80	33.79	6.96	63.60
VAX 9000	0.00	0.00	0.00	0.00	0.00	0.00	0.00
IBM RS/6000 530	0.00	0.00	0.94	65.11	104.51	17.01	187.57
HP 9000/720	0.00	0.00	0.04	31.02	59.02	3.90	93.98
Sparcstation 1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Sparcstation 1+	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DECstation 5500	0.00	0.00	0.04	16.94	25.10	5.17	47.25
average	0.00	0.00	0.13	21.47	34.15	5.62	61.37

Execution Time Penalty: Cache + TLB

machine	DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	SPICE2G6	Total
DECstation 3100	4.41	33.29	54.76	97.16	361.15	402.96	953.73
DECstation 5400	6.86	51.78	85.13	128.54	528.31	619.93	1420.55
MIPS M/2000	2.50	16.78	18.35	62.84	218.26	221.79	540.52
VAX 9000	0.09	0.06	6.35	14.17	102.87	97.21	220.75
IBM RS/6000 530	0.03	0.03	2.44	80.49	162.58	72.13	317.70
HP 9000/720	0.01	9.32	11.40	47.05	118.75	59.25	245.78
Sparcstation 1	1.62	23.93	39.21	45.08	202.51	225.62	537.97
Sparcstation 1+	2.29	17.26	28.36	34.16	163.23	203.99	449.29
DECstation 5500	3.06	23.12	38.01	62.69	243.71	278.37	648.96
average	2.32	19.51	31.56	63.58	233.49	242.36	592.81

Table 5.4: Total Execution time penalty due to cache misses, TLB misses, and the sum of both cache and TLB misses. All times are given in seconds. The delay for each program and machine combination is computed from the miss ratios and the average memory delay measurements.

The results in table 5.5 indicate that if the results for benchmark *SPICE2G6* are not considered, then the root mean square (rms) error consistently decreases on all machines, except on the Sparcstation 1. In fact, the overall average error decreases from −9.58% to −3.54, while the rms error decreases from 20.42% to 18.06%. The overall error on the Sparcstation 1 increases, because our predictions on *FPPPP* and *TOMCATV* have a positive error even when locality effects are ignored. The skewness in the distribution of average errors appears to decrease when locality is taken into account. When locality is ignored, the errors range from −18.84 to +0.64, but with the delay factor the errors range

machine	Spice2g6 Excluded				Spice2g6 Included			
	Without Latency		With Latency		Without Latency		With Latency	
	average error	root mean square	average error	root mean square	average error	root mean square	average error	root mean square
DECstation 3100	−9.56 %	14.34 %	−3.30 %	11.47 %	−1.47 %	20.62 %	5.39 %	22.52 %
DECstation 5400	−8.44 %	12.77 %	1.79 %	11.01 %	−0.90 %	19.00 %	10.21 %	23.59 %
MIPS M/2000	−9.91 %	16.65 %	−4.51 %	13.24 %	−8.39 %	15.20 %	−3.08 %	12.20 %
VAX 9000	−6.44 %	24.09 %	−0.24 %	20.57 %	−8.07 %	22.96 %	−1.91 %	19.24 %
IBM RS/6000 530	0.64 %	20.95 %	5.42 %	19.35 %	7.00 %	24.84 %	11.48 %	24.56 %
HP 9000/720	−11.40 %	22.48 %	−4.41 %	20.32 %	−6.55 %	21.76 %	−0.11 %	20.51 %
Sparcstation 1	−6.73 %	22.84 %	−2.43 %	23.08 %	0.50 %	25.66 %	5.13 %	27.41 %
Sparcstation 1+	−15.56 %	23.26 %	−12.83 %	21.78 %	−6.57 %	22.31 %	−9.57 %	22.81 %
DECstation 5500	−18.84 %	22.81 %	−11.36 %	17.02 %	−7.29 %	29.29 %	0.79 %	29.55 %
overall	−9.58 %	20.42 %	−3.54 %	18.06 %	−3.86 %	22.9278 %	2.37 %	22.78 %

Table 5.5: Summary of prediction errors by machine. The prediction errors under the label "Spice2g6 Excluded" are computed on five of the six Fortran SPEC benchmarks, while those under the label "Spice2g6 Included" are computed over the six benchmarks.

program	Without Latency		With Latency	
	average error	root mean square	average error	root mean square
DODUC	−0.26 %	5.30 %	0.58 %	5.47 %
FPPPP	−3.82 %	23.22 %	1.51 %	22.51 %
TOMCATV	−3.67 %	14.20 %	3.00 %	13.27 %
MATRIX300	−31.77 %	33.21 %	−23.80 %	26.55 %
NASA7	−8.39 %	14.62 %	1.00 %	14.58 %
overall (1)	−9.58 %	20.42 %	−3.54 %	18.06 %
SPICE2G6	28.37 %	35.98 %	35.68 %	42.81 %
overall (2)	−3.86 %	22.78 %	2.37 %	22.92 %

Table 5.6: Summary of prediction errors by program. Even when the rms error of *DODUC* increases when the cache and TLB miss delay is included, on eight of the nine machines the prediction error decreases.

from −12.83 to +5.42.

If we include *SPICE2G6*, then we see that the overall rms error decreases very little, from 22.92% 22.78%, while the average error changes from −3.86 to +2.37. The distribution of average errors, however, also presents less skewness. In fact, when locality is ignored, the number of machines with negative and positive average errors are 7 and 2 respectively. The corresponding numbers when the our predictions take into account locality are 4 and 5.

With respect to the programs (table 5.6), the results show that the overall rms errors improve in four out of the six benchmarks. The only benchmark where the rms error increases are *DODUC* and *SPICE2G6*. As we already mentioned, our original predictions for *SPICE2G6* showed consistently positive errors, these errors cannot decrease by

adding an additional positive term. Although the overall rms error on *DODUC* increases, the individual predictions show that on eight of the nine machines the prediction error decreases or remains constant (see table 5.13, Appendix 5.A). The reason why the overall error increases is because the error on the MIPS M/2000, which is the one that increases, is much larger than the other eight errors[2]. Of all the programs the one which experiences the largest improvement is *MATRIX300*, where the rms error decreases from 33.21% to 26.55%.

SPECratios: With and Without Locality Effects

machine		DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	SPICE2G6	SPECfp
DECstation 3100	orig.	11.31	12.51	9.88	9.84	13.18	9.49	10.95
	modi.	11.62	14.49	12.42	12.48	17.26	11.30	13.11
	diff.	2.68 %	13.71 %	20.43 %	21.13 %	23.68 %	15.97 %	16.54 %
DECstation 5400	orig.	12.80	13.37	9.88	10.43	12.81	9.10	11.27
	modi.	13.43	17.32	14.47	14.81	19.31	11.90	15.02
	diff.	4.71 %	22.79 %	31.74 %	29.62 %	33.68 %	23.55 %	24.91 %
MIPS M/2000	orig.	17.58	20.39	17.66	13.31	18.37	12.07	16.29
	modi.	18.00	22.98	20.12	16.33	22.94	13.59	18.67
	diff.	2.36 %	11.26 %	12.23 %	18.48 %	19.95 %	11.18 %	12.76 %
VAX 9000	orig.	46.81	69.52	40.32	43.64	46.00	46.00	47.92
	modi.	46.92	69.62	44.63	50.54	60.17	56.56	54.10
	diff.	0.23 %	0.14 %	9.67 %	13.66 %	23.55 %	18.67 %	11.42 %
IBM RS/6000 530	orig.	27.68	54.74	75.69	21.80	35.48	27.59	36.71
	modi.	27.69	54.77	81.36	35.60	49.76	30.09	43.29
	diff.	0.05 %	0.05 %	6.97 %	38.77 %	28.70 %	8.31 %	15.20 %
HP 9000/720	orig.	47.17	78.10	51.34	25.81	51.88	75.22	51.68
	modi.	47.18	102.71	65.90	35.28	74.82	92.42	65.35
	diff.	0.03 %	23.96 %	22.09 %	26.84 %	30.66 %	18.61 %	20.92 %
Sparcstation 1	orig.	5.05	7.82	5.96	11.04	10.21	8.22	7.76
	modi.	5.07	8.33	6.53	12.40	11.38	8.91	8.38
	diff.	0.44 %	6.16 %	8.82 %	11.00 %	10.29 %	7.74 %	7.47 %
Sparcstation 1+	orig.	8.07	11.42	9.17	16.35	15.60	10.27	11.42
	modi.	8.15	12.21	10.16	18.66	17.86	11.26	12.49
	diff.	0.99 %	6.49 %	9.81 %	12.35 %	12.67 %	8.75 %	8.59 %
DECstation 5500	orig.	21.15	25.72	19.59	19.61	26.05	16.41	21.14
	modi.	21.91	31.99	27.26	26.93	38.08	20.27	27.11
	diff.	3.47 %	19.58 %	28.11 %	27.17 %	31.60 %	19.07 %	22.01 %
average		2.13 %	14.88 %	21.41 %	28.43 %	30.68 %	18.84 %	19.97 %

Table 5.7: The effect of memory delay in the overall machine performance using as design target miss ratios the cache and TLB miss ratios of the Fortran SPEC benchmarks. The results labeled *orig.* include the memory delay due to cache and TLB misses, while those labeled *modi.* are computed by subtracting the respective memory delay penalty. The SPECfp is obtained by taking the geometric mean of the individual SPECratios.

5.6.4. The Effect of the Memory System on Performance

In this section we will use the results of §5.6.2 to evaluate the impact of the different memory systems in the overall performance of the machines. We use the Fortran

[2] The rms error is a non linear function which assigned more weight to the largest values.

SPEC benchmarks as a single workload and use their original SPECratios as our baseline to compute a new set of SPECratios representing the amount of improvement that the programs would experience when we ignore the memory delay caused by cache and TLB misses. The SPECfp, which is the geometric mean of the individual SPECratios of the SPEC floating-point benchmarks, will give us the overall impact of the memory system. In this respect the cache and TLB miss ratios of the SPEC benchmarks correspond to what Smith has defined as *design target miss ratios* [Smith85, Smith87]. The design target miss ratios represent the expected miss ratios of a large set of programs, and could be used by machine designers in evaluating different cache organizations, in a similar way as we do here.

The baseline SPECratios we use here have been taken from the original SPEC reports [SPEC90a SPEC90b SPEC91a, SPEC91b], except for the VAX 9000, which we benchmarked ourselves. We also changed a few of the original SPEC numbers, in particular, we ignored the lastest SPECratios for *MATRIX300* on the HP-9000/720, the IBM RS/6000 series, and the Sparcstations. Here we decided to use older results or re-executed the benchmark without using the machines' optimizing preprocessors. The reason for this is that these preprocessors change the original matrix multiply algorithm, which is based on the SAXPY routine, and replace it by a blocking algorithm. These blocking algorithms exhibit significantly lower miss ratios than the ones computed by Gee et al. Therefore, using these optimized results is meaningless in conjunction with the miss ratios computed for the SAXPY algorithm.

In table 5.7 we give for each machine and program combination the original SPECratios (orig.), the modified SPECratios assuming a memory delay of zero cycles (modi.), and their respective difference. In addition, on the rightmost column we show the SPECfp computed over their original and modified results. As expected, the impact of the memory system varies significantly from program to program. For example, *DODUC* exhibits the smallest effect with the maximum performance degradation of less than 5% (DEC 5400), and an average of only 2.13%. On the other side of the spectrum, the largest average impact is observed on *MATRIX300* and *NASA7* benchmarks with 28.43% and 30.68%.

On the machines, we find that the largest performance impact of the memory system is on the DEC 5400 and DEC 5500 with improvements of 24.91% and 22.01%. The lowest impact is observed on the Sparcstation 1 which has a value of only 7.47%. It is important to remember that the impact of the memory system is a function of three factors: a) the miss ratios of the benchmarks, b) the delays in loading the cache and TLB when misses occurred, c) and the raw performance of the CPU. The reason why the impact is lower on the Sparcstations is mainly because the performance gap between memory and the CPU is smaller on the Sparcstations than on the other machines, In other words, as we increase the raw performance of a CPU it is more difficult to build a memory system to match this performance. For example, the results from table 5.3 show that the Sparcstation 1 has an average cache miss ratio of .0783 which is larger than the .0309 and .0465 on the IBM RS/6000 530 and HP-9000/720. The total delay on the SPEC benchmarks due to cache misses is also much larger on the Sparcstation 1, with 538 seconds, than the 130 and 151 seconds exhibit by the other two machines (table 5.8). But because the Sparcstation has a SPECfp of only 7.76, the memory delay represent only 7.47% decrease in performance.

Machine Characteristics

Characteristics	DEC 3100	DEC 5400	MIPS M/2000	DEC 5500
CPU	R2000	R3000	R3000	R3000
FPU	R2010	R3010	R3010	R3010
Frequency	16.67 MHz	20 MHz	25 MHz	30 MHz
Freq. ratio	0.834	1.000	1.250	1.500
Cache (instr)	64 KB	64 KB	64 KB	64 KB
Cache (data)	64 KB	64 KB	64 KB	64 KB
Main memory	24 MB	64 MB	64 MB	32 MB
CC compiler	MIPS 1.31	MIPS 2.1	MIPS 2.1	MIPS 2.1
F77 compiler	MIPS 2.1	MIPS 2.1	MIPS 2.1	MIPS 2.1

SPEC Benchmark Results

program	DEC 3100	DEC 5400	MIPS M/2000	DEC 5500
Gcc	10.9 (0.991)	11.0 (1.000)	19.0 (1.727)	20.3 (1.845)
Espresso	12.0 (0.851)	14.1 (1.000)	18.3 (1.298)	21.7 (1.539)
Spice 2g6	9.5 (1.044)	9.1 (1.000)	12.1 (1.330)	16.4 (1.802)
Doduc	11.3 (0.883)	12.8 (1.000)	17.6 (1.375)	21.1 (1.648)
Nasa7	13.2 (1.031)	12.8 (1.000)	18.4 (1.438)	26.1 (2.039)
Li	13.1 (1.073)	12.2 (1.000)	23.8 (1.951)	23.4 (1.918)
Eqntott	11.2 (0.824)	13.6 (1.000)	18.4 (1.353)	22.4 (1.647)
Matrix300	9.8 (0.942)	10.4 (1.000)	13.3 (1.279)	19.6 (1.885)
Fpppp	12.5 (0.933)	13.4 (1.000)	20.4 (1.522)	25.7 (1.918)
Tomcatv	9.9 (1.000)	9.9 (1.000)	17.7 (1.788)	19.6 (1.980)
SPECint	11.8 (0.929)	12.7 (1.000)	19.8 (1.555))	21.9 (1.724)
SPECfp	10.9 (0.965)	11.3 (1.000)	16.3 (1.443)	21.1 (1.867)
SPECMark	11.3 (0.958)	11.8 (1.000)	17.6 (1.492)	21.5 (1.822)

Table 5.8: SPEC benchmark results for machines based on the MIPS Co. processors. The results are given in terms of the SPECratio, while the numbers inside parenthesis are normalized with respect to the DEC 5400. The performance for the DEC 5400 is lower than the corresponding difference in the clock rates.

5.7. Discussion

In this section we will show how we can use our methodology and tools to explain the performance difference observed in the SPEC benchmarks on some of the machines having the same instruction set architecture. In table 5.8 we show the main machine characteristics of four different machines based on MIPS processors: DEC 3100, DEC 5400, MIPS M/2000, and DEC 5500. Note that the main difference between the machines is their clock rates. Although the DEC 3100 uses the R2000/R2010 processor pair instead of the R3000/R3010, the performance difference between them is too small to have a significant effect. In the same table we give the SPECratios of the machines on the SPEC programs as quoted in the SPEC Newsletter [SPEC90, SPEC91]. Alongside each SPECratio we indicate, in parenthesis, the relative performance with respect to the DEC 5400. In the previous sections we showed that it is possible to evaluate the effect of the memory hierarchy on the observed performance of a benchmark.

Traditional benchmarking focuses mainly in reporting execution time observations similar to those given in table 5.8. With only this information and without knowing anything about what it is that makes each program a unique experiment and what are the basic performance characteristics of machines, it is not possible to explain or predict program execution times.

The most interesting observation here is that the SPEC results on the DEC 5400 compared to the other machines cannot be explained only by the relative differences in their clock rates. The SPEC results indicate that, with respect to the DEC 5400, the DEC 3100 is 15% (0.958 vs. 0.834) faster than we would expect it to be. Similarly, the MIPS M/2000 is around 19% faster (1.492 vs. 1.250), while the DEC 5500 is 21% faster (1.822 vs. 1.500) than what their clock rate ratios indicate. Notwithstanding the small statistical variation, this situation appears to be consistent across all benchmarks.

The results in table 5.8 clearly illustrates the main limitation of traditional benchmarking. Because there is very limited information about the performance of individual machine components and the characteristics of the programs, it is not possible to provide an explanation for the lower than expected performance of the DEC 5400. In contrast, with our tools and methodology we can make a much better analysis of the results and discover the source of the discrepancy. Given that all of our predictions and performance metrics are synthesized from basic machine and program measurements, we can proceed backwards and explain them in terms of more simple measurements. The basic measurements in our model correspond to the performance of the abstract machine operations. Here, however, instead of comparing the 109 parameters, we use the reduced model containing only thirteen parameters, where each dimension represent the performance of some functional unit. This reduced model is enough to explain the SPEC results. In figure 5.5 we show the normalized reduced parameters for the DEC 5500, MIPS M/200, and the DEC 3100. Each reduced parameter is normalized with respect to the corresponding performance on the DEC 5400.

We see in figure 5.5 that the performance distribution of twelve of the thirteen parameters lies within the ratio of the clock rates. In fact, the average relative performance of these parameters is 1.500 for the DEC 5500, 1.262 for the MIPS M/2000, and 0.803 for the DEC 3100. These numbers correspond to the expected values. The performance of memory operations, however, is significantly higher in the three machines than it is on the DEC 5400. This performance limitation is what explains the lower performance observed of the SPEC suite on the DEC 5500.

We can proceed even further by comparing the memory hierarchies of the machines. In figures 5.2 and 5.4 we see that the basic structure of their caches and TLBs are similar, i.e., the four machines have a direct-mapped caches of 64 KB, and fully associative TLBs with 64 entries with an entry granularity of 4096 bytes. Furthermore, the ratio between the TLB misses (480 ns for the DEC 3100, 400 ns for the DEC 5400, 350 ns for the MIPS M/2000, and 260 ns for the DEC 5500) are very close to the clock rate ratios (0.8333, 1.143, and 1.538). The main physical difference between the caches is that the line size of the DEC 3100 is only 4 bytes, instead of 16 bytes on the DEC 5400 and DEC 5500, and 64 bytes on the MIPS M/2000. However, this difference is not the source of the discrepancy. The reason behind the DEC 5400 worse than expected performance is the excessive penalty of cache misses. A cache miss on the DEC 5400 takes approximately 1680 ns compared to 750 ns on the DEC 5500, even when both machines

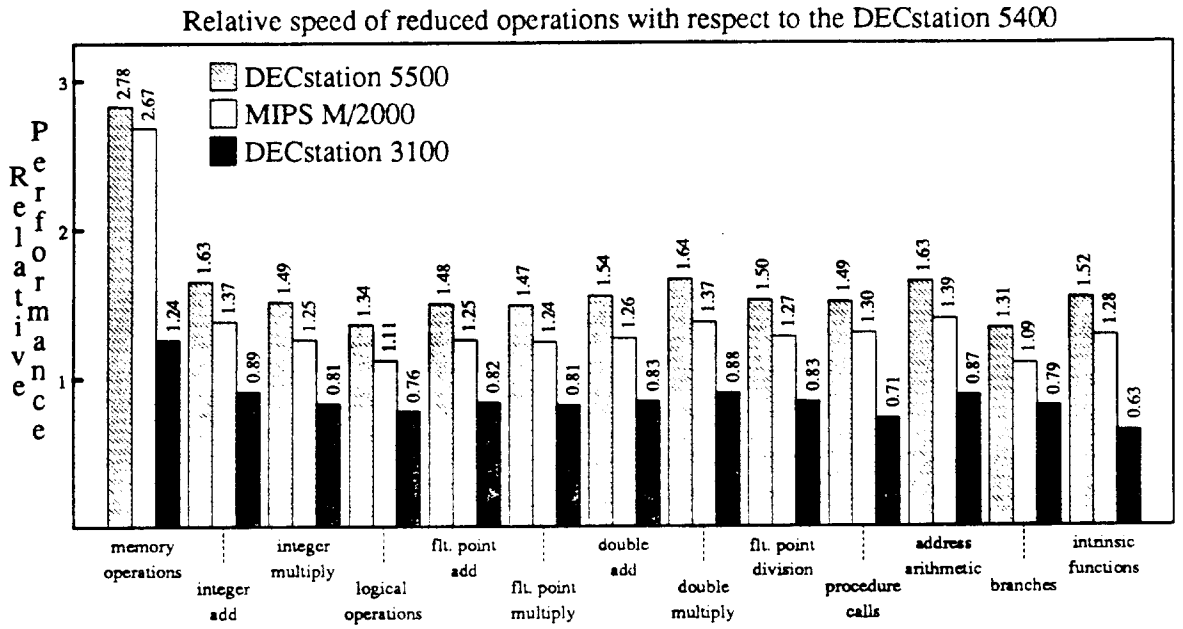


Figure 5.5: Normalized performance of the abstract reduced parameters. The results are normalized with respect to those of the DEC 5400. The ratio of all dimensions, except memory operations, is close to the relative clock rate ratios.

have a 16-byte line size. This gives a ratio of 2.24 which is much higher than the 1.5 clock rate ratio.

Comparing the miss penalties between the DEC 3100 and MIPS M/2000 with the DEC 5400 is not as straightforward, because of the difference in line sizes. Here the line size, from the 3100 to the 5400, increases by a factor of 4, at the same time the miss penalty increases only by a factor of 3.111. However, the reduction in miss ratios due to a larger line size is for most programs smaller than the corresponding line size increase. For example, an acceptable rule of thumb for caches with parameters similar to the DECstations, and over a large sample of programs, is that doubling the line size will not decrease the miss ratio by more than a factor of 1.5 [Smit87]. Therefore, the decrease in the miss ratio when we increase the line size by a factor of four should be only around 2.25, which is much lower than the 3.111 increase in the miss penalty of the DEC 5400.

Now, if we look at the cache parameters of the MIPS M/2000 we see that the line size is a factor of 4 larger with respect to the DEC 5500 and a factor of 16 with respect to the DEC 3100, but the respective increases in the the miss penalties are only 2.667 and 1.920. Even if we assume that the decrease in miss ratio is only 1.4, as a result of doubling the line size, the corresponding decreases in miss ratios should be 3.842 and 1.960 which are larger than the corresponding line miss penalties increases. The more aggressive cache design on the MIPS M/2000 is effectively reducing the miss ratio without overly increasing the penalty as is the case with the DEC 5400. This complemented with wraparound loads are the main reasons why the performance of the MIPS M/2000 on the

SPEC benchmarks is higher than other machines based on the R3000/R3010 chips and with comparable clock rates.

5.8. Cache and TLB Behavior on Matrix Multiply

In §5.6 we computed the amount of memory delay produced by the SPEC scientific programs. We then incorporated these results in our predictions and found that there was little improvement with respect to the previous estimates which ignored memory delays. The main reason for not observing a larger improvement is because the miss ratios of the SPEC programs is so small that the memory delays involved are smaller than our experimental errors.

In this section we make an analysis of two algorithms of matrix multiply: DOT and SAXPY, and obtain exact formulas for the number of cache and TLB misses they generate over a large space of input matrices. In §5.8.4 we use these results to show that it is possible to predict how much time is spent as a result of misses. This is because these programs are almost identical. Their only difference resides in the respective amounts of locality.

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

a) DOT Algorithm

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

b) SAXPY Algorithm

Figure 5.6: Two non-blocking algorithms for matrix multiply. DOT uses the dot product between `A` and `B` as the basic building block, while SAXPY uses sums vector `C` to the product of constant `B(I,J)` to vector `A`.

5.8.1. The DOT and SAXPY Algorithms

First, consider the two algorithms for matrix multiply shown in figure 5.6. Algorithm DOT (fig. 5.6 a)) is based on the standard algorithm that takes the dot product between a row of matrix A and a column of B, while algorithm SAXPY (fig. 5.6 b)) avoids doing a reduction and moving across rows at the innermost loop by multiplying a column of A by a constant (`B(I,J)`) and adding this to a column of C. Of the six

different algorithms obtained by the permutations of the i, j, and k indices, DOT and SAXPY are the worst and best algorithms in terms of the cache and TLB misses they generate.

We make several simplifying assumptions in order to obtain exact formulas for the number of misses. The most important assumption is that the order of the matrices (N) is a power of two. Obtaining a closed formula for arbitrary matrix sizes does not seem to be possible, because of the complex interactions of matrix size, cache and line size, and associativity. Reference [LamM91] is a good example of the difficulties present in the analysis of cache behavior for matrices of arbitrary size even on a direct-mapped cache. By restricting our analysis to powers of two we can obtain upper bounds on the cache and TLB miss ratios, because self-interference is highest on these matrices. In the discussion that follows we concentrate on the behavior of the cache; our results, however, can also be used to compute the number of TLB misses.

5.8.2. Reusability on Non-Blocking Matrix Multiply Algorithms

Let D, b, and a be the cache size, line size, and associativity of the cache. We assume that the order of matrices **A**, **B**, and **C** is N. Let the *occupancy ratio* (R) be the ratio between the number of elements in each matrix (N^2) and D ($R = N^2/D$). It is obvious that R as well as N and D is also a power of two. We restrict our analysis for values of R in the region: $1 < R \leq D$, as this represents the most interesting region in terms of cache and TLB behavior. When $R = 1$ a whole matrix fits in the cache, i.e., $N = D^{1/2}$, and it is not difficult to prove that the number of misses is $O(N^2)$ which is insignificant compared to the total number of references. Moreover, if $R > D$, then it is not possible for a single column to fit in the cache without producing self-interference. On existing caches, where D is of the order of 2^{16} elements, the size of a matrix has to be greater than 2^{17} or higher for R to be greater than D. Matrix multiplication on matrices of this size take hundreds of days to execute in high-performance workstations like the HP 9000/720 or the IBM RS/6000 530. Therefore our analysis covers the most interesting region with respect to cache and TLB misses and execution time.

It is clear from both algorithms that each element of the three matrices is re-used N times; however, the interval between re-uses is different on each matrix. Let the *element re-use distance* (ERUD) be the number of references to elements of the same matrix between two re-uses of the same element. The ERUD of a matrix is proportional to the time its elements must stay in the cache before they are re-used again. In the same way, we can define the *line re-use distance* (LRUD) as the number of references between two re-uses of the same line, but with the restriction that the two re-uses are to different elements in the same line. The ERUD and LRUD values represent the amount of temporal and spatial locality that can be exploited on each matrix. Another important aspect of matrix multiply is how each matrix is traversed at the innermost loop level. Two matrices can have similar ERUDs, but if one matrix is traversed by columns and the other by rows, the number of misses may be different in each case, because the number of lines touched on their innermost loops are different.

Most of the misses generated by the two algorithms are the result of self-interference, which represent conflicts between elements of the same matrix. Condition $1 < R \leq D$ guarantees that a complete column of N elements fits in the cache without having self-interference, but a whole matrix cannot, so visiting the whole matrix before re-

using its elements will displace most of them from the cache.

In addition, it is not possible to load a row without having self-interference. The main reason is that both N and D are power of two, and even when a small number of sets are occupied by one row, the number of lines which map to each set is greater than the associativity, as the elements of a row will map only to a small subset of cache sets. This can be proved using the following argument. The number of sets in a cache is D/ab. Now, the number of lines needed to cover a column of a matrix is N/b. In contrast a row requires N different lines. If a single column does not cause self-interference in the cache, then the N/b lines are mapped to N/b different sets. But traversing the matrix by row touches one and only one element of the N/b consecutive sets. Hence the N elements of a row are mapped to only D/Na sets, which is obtained by dividing the total number of sets D/ab by the distance, measured in sets, between consecutive elements of a row N/b. Therefore, each set touched by the row must accommodate N^2a/D elements. We can now substitute the occupancy ratio in this equation and get the following inequality: $N^2a/D = Ra > a$. This clearly shows that loading a single row always causes self-interference. This analysis implies that re-using a column N times generates only N/b misses, which the minimum necessary to load the column once, while re-using a row N times generates N^2 misses.

Using the above analysis we can now compute the total number of cache misses for each algorithm with the aid of the following observations: 1) If the value of ERUD is 1 for a particular matrix, then only one of the N re-uses of each element causes a miss; 2) if ERUD is N^2, then at least the first element of each line misses on every reference; 3) if ERUD is N and the innermost direction of traversal is by column, then, unless there is cross-interference, only one of the N re-uses causes a miss; and 4) if the ERUD is N and the innermost direction of traversal is by row, then every re-use causes a miss, as a result of self-interference. In addition, there are similar observations in terms of spatial locality: 1) if LRUD is 1, then, without considering cross-interference, only the first element of the line experiences misses; and 2) if LRUD is N^2, then each element of a line generates as many misses as the first element.

	DOT algorithm				**SAXPY algorithm**			
	re-use distance		traversal	number of	re-use distance		traversal	number of
matrix	element	line	direction	misses	element	line	direction	misses
A	N	N^2	row	N^3	N	1	column	N^2/b
B	N^2	1	column	N^3/b	1	N^2	row	N^2
C	1	N^2	row	N^2	N^2	1	column	N^3/b

Table 5.9: Re-use information for the DOT and SAXPY algorithms. The number of misses reported here correspond only to self-interference misses.

In table 5.9 we give, for each matrix, the ERUD and LRUD values, the direction of traversal, and the basic number of misses, caused by self-interference. From these results we can compute the exact number of misses. For DOT the number of misses are:

$$M(\text{DOT}) = \left\lceil 1 + \frac{1}{b} \right\rceil N^3 + 2N^2 + (Rb - R)N\ \delta(a, 1) \qquad \text{for } 1 < R \leq N, \tag{5.3}$$

where $\delta(a,1)$ is Kronnerker's delta which takes the value 1 when a equals 1, and 0 in all other cases, and this term represent misses cause by cross-interference.

The first two terms associated with N^3 correspond to the self-interference misses generated by matrices A and B. The $2N^2$ represents the misses caused by C: one as a result of reading the value at the beginning of the innermost loop and another when written the result back to memory. During the duration of the loop the value of C is kept in a register. Equation (5.3) indicates that there is only cross-interference in a direct-mapped cache ($\delta(a,1)$). In fact, increasing the associativity higher than 2 does not affect the number of misses. The reason for this is twofold. First, the only cross-interference is between matrices A and B, not between C and the other matrices. Second, the number of misses is not a function of R, the ratio between the size of the matrix and the cache size. This implies that doubling the size of the matrix or reducing the size of the cache, as long as R remains in the region of validity, does not affect the number of interference misses. Hence, we can see why there are no cross-interference misses when $a > 1$ by considering a 2-way associative cache as two independent direct-mapped caches of size $D/2$, one for matrix A and the other for B. In this new cache configuration it is clear that there cannot be cross-interference misses, but only self-interference ones and by the first argument the total number of misses remains the same.

The cross-interference misses occur when index i and j are almost equal, and a row of A interferes with a column of B. This happens N times in the algorithm, when R elements of A remove lines of B from the cache. The duration of each cross-interference spans for b iterations of I, thus the total number of misses is RbN. However, we have to subtract from this number RN misses which were already counted as self-interference misses.

In an analogous way we can obtain an exact formula for the misses which occur on SAXPY

$$M(\text{SAXPY}) = \frac{1}{b}N^3 + \left[1+\frac{1}{b}\right]N^2 + M_R\,\delta(a,1) \qquad \text{for } 1 < R < N/2. \tag{5.4}$$

where

$$M_R = \left[2R - \frac{R}{b}\right]N^2 + \left[R - \frac{R}{b}\right]N - R\,.$$

As with DOT, here there are no cross-interference misses when $a > 1$. When $a = 1$, however, the number of cross-interference misses is significantly larger, because there is more chance that a line will be removed by other matrix if it was left untouched by self-interference conflicts. An important observation here is that by doubling the size of the matrices, which increases the total number of references by a factor of 8, results in a factor of 16 increase in the number of cross-interference misses. This number comes from the product of N^2 and R, where both increase by a factor of 4. This means that as R increases, when we maintain the cache size constant, the cross-interference misses contribute more to the miss ratio.

Equation (5.4) is only valid when R is less than $N/2$. When R reaches this threshold, there is another source of misses, which is a function of the associativity, and it happens when two columns, one from A and another from C, can no longer fit in the cache without interfering with each other. Moreover, when $R = N$ the number of misses

increases significantly by a factor of two. The formula for cache misses when $R = N/2$ is

$$M(\text{SAXPY}) = \begin{cases} \frac{1}{b}N^3 + \left[1+\frac{1}{b}\right]N^2 + \left[2R - \frac{R}{b}\right]N^2 + \left[R - \frac{R}{b}\right]N - R & \text{for } a = 1 \\ \frac{1}{b}N^3 + \left[1+\frac{1}{b}\right]N^2 + \frac{a}{2}N(N-1) & \text{for } a > 1, \end{cases} \tag{5.5}$$

and when $R = N$ is

$$M(\text{SAXPY}) = \begin{cases} 2N^3 + N^2 & \text{for } a = 1 \\ \frac{2}{b}N^3 + N^2 & \text{for } a > 1, \end{cases} \tag{5.6}$$

The number of references generated by each algorithm is different when both programs are compiled in real machines with optimization enabled. DOT makes $2N^3+2N^2$ references (N^3 for A, N^3 for B, and $2N^2$ for C), while SAXPY makes $3N^3+N^2$ (N^3 for A, $2N^3$ for C, and N^2 for B). In SAXPY, however, the maximum number of misses that can occur is $2N^3+N^2$, because the write to C, which happens immediately after reading its value, can never miss. Therefore, in what follows, for comparison purposes, we compute miss ratios in terms of the potential misses instead of all the references. The justification is based on the observation that the difference in the delay penalty contribution caused by misses in the two programs is proportional to the total number of misses, not the total number of references.

In figure 5.7 we show the miss ratios for DOT and SAXPY. Even when the miss ratios for programs are independent on the cache sizes, we assume that the size of the cache is 64 K elements (256 Kbytes), which is the size of the cache on the HP 9000/700 series, because the maximum value that R can take depends in the cache size. For comparison we also show the miss ratios when $R = 1$, which generates in the order of $O(N^2)$ misses, and that we present here without discussing the actual derivation

$$M(\text{DOT}) = \begin{cases} \left[2a+4+\frac{3}{b}\right]N^2 - \left[2+\frac{2a}{b}\right]N + (b-1)N & \text{for } a = 1 \\ \left[2a+4+\frac{3}{b}\right]N^2 - \left[2+\frac{2a}{b}\right]N & \text{for } 1 < a < \frac{N}{b} \\ \frac{1}{b}N^3 + 2\left[1+\frac{2}{b}\right]N^2 - 2N & \text{for } a = \frac{N}{b}, \end{cases} \tag{5.7}$$

and

$$M(\text{SAXPY}) = \begin{cases} \left[4+\frac{3}{b}\right]N^2 - \left[2+\frac{3}{b}\right]N + 1 & \text{for } a = 1 \\ \left[2+\frac{2a^2+3a-1}{ab}\right]N^2 - \left[\frac{2a^2+2a-1}{a} + \frac{5a-3}{b}\right]N + (5a-3) & \text{for } 1 < a < \frac{N}{b} \\ \frac{2}{b}N^3 + N^2 & \text{for } a = \frac{N}{b}. \end{cases} \tag{5.8}$$

On a cache size of 64K elements, $R = 1$ represents a matrix of order 256, and $R = 65536$ corresponds to a matrix of order 65536.

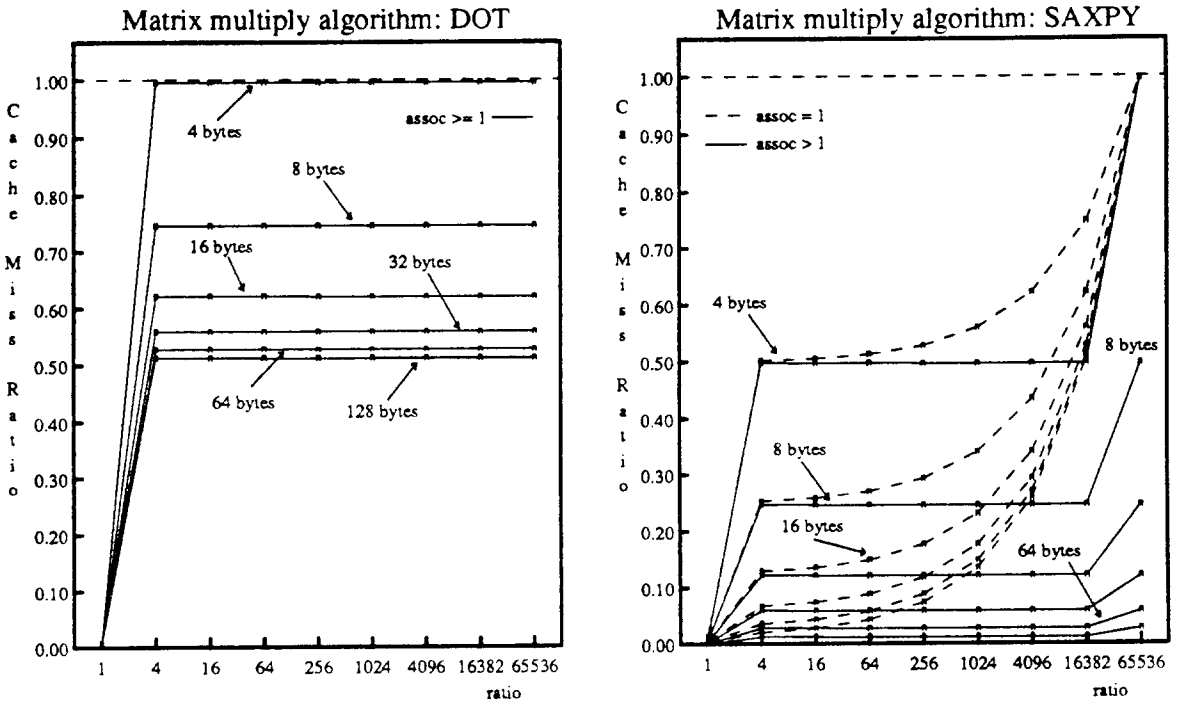


Figure 5.7: Analytical miss ratios for DOT and SAXPY algorithms. The results on DOT for different associativities are so similar that they are plotted as a single curve. SAXPY shows significant lower miss ratios on most cache configuration except on direct-mapped caches when R is close to N.

In figure 5.7 a) the extra misses when $a = 1$ are so few that their effect is not perceivable, so all curves all are shown as one. We see that on both graphs that the number of misses is very high when the line size is only 4 bytes. Thus machines like the DEC 3100 pay a high penalty because of their small line. It is clear that SAXPY consistently generates less misses than DOT for most cache configurations. The only exception is when R approaches N for direct-mapped caches and long line sizes. In this situation SAXPY's miss ratios approaches 1 very fast and are even larger than DOT's miss ratios. This clearly illustrate how difficult it is to develop algorithms that will perform well over a large space of cache configurations and inputs. From these graphs, however, we can conclude that an a-way associative cache, for $a \geq 2$, with a line size of 64 bytes or more, produce very low miss ratios which are barely affected by large values of R.

5.8.3. TLB Misses

Although the previous analysis is valid when we replace the cache for a TLB, the parameters of actual TLBs fall outside the validity region of the equations. Equations (5.3)-(5.6) were obtained under the assumption that $b \leq N$, as it is the case on actual caches and large matrices. The granularity of a TLB entry, however, is large enough that

several columns fit in a single entry. Hence we need to derive an expression for TLB under this new condition. Given that all the TLBs in this study have associativities which are greater than or equal to 2, we compute the number of TLB misses for this situation. We are also assuming that the size of each matrix is greater than the region covered by all TLB entries.

5.8.3.1. TLB Misses for the DOT Algorithm

The TLB misses for DOT can be easily computed by counting the misses generated by each of the three matrices. From table 5.9 we know that matrix A is traversed by row, so loading a single row of A requires touching all of its pages, which means that the memory region represented by these pages is larger than what the TLB can cover. Hence, all the entries of the TLB present before the row is loaded will be subsequently discarded (assuming none is referenced during the interval the row is loaded). Therefore, the number of TLB misses generated by one of A's rows is N/B, where B is the number of columns that can fit in a page ($B = b/N$). Moreover, loading matrix A once generates N^2/B misses, and doing this N times gives a total of N^3/B for this matrix.

The behavior of matrix B is similar to that of A with the only difference that B is traversed by column, so instead of one miss every B elements there is one miss every b elements. Therefore, the total number of misses for B is $N^3/b = N^2/B$.

Finally, we can compute the TLB misses for matrix C by observing that every write to an element of C misses, as a results of loading one of A's rows; there are no entries left in the TLB belonging to C. Hence, the number of misses due to writes is N^2. With respect to reads, the situation is more complex. After each write, there is an entry in the TLB to one of C's pages. If the next element of C happens to be in the same page, then there is no miss; otherwise a miss will occur. Because C is traversed by row, a new page will be touched every B references, so there are N^2/B misses.

The following equation gives the sum of the number of misses for the three matrices

$$M\_TLB(\text{DOT}) = \frac{1}{B}N^3 + \left[1 + \frac{2}{B}\right]N^2 \tag{5.9}$$

5.8.3.2. TLB Misses for the SAXPY Algorithm

On SAXPY each of A's columns is re-used N times, so the only misses are those due to the first iteration when the column is first loaded. Thus the number of misses caused by A is only $N^2/b = N/B$.

Elements from matrix B are read at the beginning of the innermost loop, and are kept on a register during the duration of the loop. Thus, there are only N^2 memory reads that can cause TLB misses, and because matrix B is traversed by row, only one of every B references causes a miss, giving a total of N^2/B misses.

Because matrix C is traversed by column, loading one of these columns for the first time generates N/b misses, and loading the matrix causes N^2/b misses. However, loading the complete matrix will generate self-interference, so the N visits to the matrix produce $N^3/b = N^2/B$ misses.

The following equation gives the sum of the individual subtotals for the three matrices

$$M\_TLB(\text{SAXPY}) = \frac{2}{B}N^2 + \frac{1}{B}N \tag{5.10}$$

We have to keep in mind that the above two equations were derived assuming an LRU replacement policy. Because many fully-associative TLBs use random replacement of entries, the above numbers represent only approximations, albeit good ones.

machine	matrix size: 256x256			matrix size: 512x512			SPEC mark
	DOT	SAXPY	DOT-SAXPY	DOT	SAXPY	DOT-SAXPY	
DECstation 3100	25.4 s	14.7 s	10.7 s	228.9 s	116.3 s	112.6 s	11.3
DECstation 5400	41.0 s	13.8 s	27.2 s	330.5 s	113.4 s	217.1 s	11.8
MIPS M/2000	31.0 s	6.2 s	24.8 s	264.4 s	53.5 s	210.9 s	18.3
VAX 9000	13.0 s	4.5 s	8.5 s	124.2 s	33.1 s	91.1 s	–
IBM RS/6000 530	18.3 s	5.9 s	12.4 s	175.6 s	45.6 s	129.9 s	28.9
HP 9000/720	7.9 s	6.4 s	1.5 s	160.4 s	52.9 s	107.5 s	59.5
Sparcstation I+	26.8 s	14.9 s	11.9 s	213.6 s	118.9 s	94.7 s	11.8
Sparcstation I	45.0 s	29.3 s	15.7 s	358.1 s	233.6 s	124.5 s	8.4
Decstation 5500	18.0 s	6.4 s	11.6 s	156.8 s	52.5 s	104.3 s	21.5

Table 5.10: Execution times for matrix multiply using algorithms DOT and SAXPY. The difference in execution time between the two algorithms is due to differences in locality as both program execute the same number of arithmetic operations. For comparison purposes we indicate the SPECmark on some of the machines.

5.8.4. Predicting the Penalty Due to Poor Locality

We can now combine the results of the previous two sections and compute, using equation (5.2), the amount of delay experienced by different machines on algorithms DOT and SAXPY. We wrote two programs for matrix multiply that apart from a small initialization code are identical to the two algorithms shown in figure 5.6. The programs were executed on several machines and using two different problem sizes: 256x256 and 512x512. The execution times results are given in table 5.10. In addition to the execution times, we show the time represented by their difference in locality, and also give their respective SPECmarks [SPEC90a, SPEC90b, SPEC91].

The results in table 5.10 are quite interesting. First, the execution times on SAXPY correlate much better with the SPECmark than those on DOT. This is evident in the results for the DEC 3100, DEC 5400, and Sparcstation 1+, which show similar SPECmarks and execution times on SAXPY. The SAXPY results for the MIPS M/2000 and the DEC 5500 also match with the SPECmark numbers of these machines. That SAXPY and the SPECmark results correlate is not surprising. Gee et al., [GeeJ91] have shown that the cache and TLB miss ratios on the SPEC benchmarks are very low, which indicates that these programs measure mainly CPU performance. Given that SAXPY also experiences significantly less misses than DOT and this respect it behaves as a benchmark for CPU performance.

The effect of the memory hierarchy is evident in the results for the MIPS M/2000. The execution time on DOT on this machine are worse than those of the DEC 3100 and the Sparcstation 1+, while its SPECmark is 65% higher. The reasons for this are the number of misses on the DOT and the large cache line size on the MIPS M/2000. While

Program DOT: matrix 256x256									
	DEC 3100	**DEC 5400**	**MIPS M/2000**	**VAX 9000**	**RS/6000 530**	**HP 9000/720**	**Sparc I**	**Sparc I+**	**DEC 5500**
data references	33,619,968	33,619,968	33,619,968	33,619,968	33,619,968	33,619,968	33,619,968	33,619,968	33,619,968
occupancy ratio	4	4	4	2	4	1	2	4	4
cache misses	33,619,968	21,040,128	17,906,688	17,891,328	17,891,328	255,968	21,038,592	21,040,128	21,040,128
miss ratio	1.0000	0.6258	0.5326	0.5322	0.5322	0.0076	0.6258	0.6258	0.6258
ex. time (min)	18.16 s	35.35 s	14.33 s	13.24 s	6.26 s	0.09 s	16.41 s	11.78 s	15.78 s
ex. time (max)	18.16 s	35.25 s	25.79 s	17.53 s	12.52 s	0.12 s	16.41 s	11.78 s	15.78 s
occupancy ratio	1	1	1	1/32	1/2	1/2	1/32	1/4096	1
TLB misses	4,276,224	4,276,224	4,276,224	96	148,287	24,576	6	1	4,276,224
miss ratio	0.1272	0.1272	0.1272	0.0000	0.0044	0.0000	0.0000	0.0000	0.1272
execution time	2.05 s	1.70 s	1.49 s	0.00 s	0.16 s	0.02 s	0.00 s	0.00 s	1.11 s
TLB + cache (min)	20.21 s	36.95 s	15.82 s	13.24 s	6.42 s	0.11 s	16.41 s	11.78 s	16.89 s
TLB + cache (max)	20.21 s	36.95 s	27.28 s	17.53 s	12.68 s	0.14 s	16.41 s	11.78 s	16.89 s

Program SAXPY: matrix 256x256									
	DEC 3100	**DEC 5400**	**MIPS M/2000**	**VAX 9000**	**RS/6000 530**	**HP 9000/720**	**Sparc I**	**Sparc I+**	**DEC 5500**
data references	50,397,184	50,397,184	50,397,184	50,397,184	50,397,184	50,397,184	50,397,184	50,397,184	50,397,184
occupancy ratio	4	4	4	2	4	1	2	4	4
cache misses	17,170,428	4,735,740	1,627,068	1,118,208	1,118,208	286,113	4,505,982	4,735,740	4,735,740
miss ratio	0.3407	0.0940	0.0323	0.0222	0.0222	0.0057	0.0894	0.0940	0.0940
ex. time (min)	9.27 s	7.96 s	1.30 s	0.83 s	0.39 s	0.10 s	3.51 s	2.65 s	3.55 s
ex. time (max)	9.27 s	7.96 s	2.34 s	1.10 s	0.78 s	0.14 s	3.51 s	2.65 s	3.55 s
occupancy ratio	1	1	1	1/32	1/2	1/2	1/32	1/4096	1
TLB misses	32,832	32,832	32,832	96	3,071	16,416	6	1	32,832
miss ratio	0.0007	0.0007	0.0007	0.0000	0.0000	0.0003	0.0000	0.0000	0.0007
execution time	0.02 s	0.01 s	0.01 s	0.00 s	0.00 s	0.02 s	0.00 s	0.00 s	0.01 s
TLB + cache (min)	9.29 s	7.97 s	1.31 s	0.83 s	0.39 s	0.12 s	3.51 s	2.65 s	3.56 s
TLB + cache (max)	9.29 s	7.97 s	2.35 s	1.10 s	0.78 s	0.16 s	3.51 s	2.65 s	3.56 s

DOT – SAXPY: matrix 256x256									
	DEC 3100	**DEC 5400**	**MIPS M/2000**	**VAX 9000**	**RS/6000 530**	**HP 9000/720**	**Sparc I**	**Sparc I+**	**DEC 5500**
cache misses	16,449,540	16,304,388	16,279,620	16,773,210	16,773,210	-30145	16,532,610	16,304,388	16,304,388
ex. time (min)	8.88 s	27.39 s	13.04 s	12.21 s	5.87 s	0.01 s	12.90 s	9.13 s	12.23 s
ex. time (max)	8.88 s	27.39 s	23.44 s	16.44 s	11.74 s	0.01 s	12.90	9.13 s	12.23 s
TLB misses	4,177,856	4,177,856	4,177,856	0	145,216	8160	0	0	4,177,856
execution time	2.01 s	1.67 s	1.46 s	0.00 s	0.16 s	0.01 s	0.00 s	0.00 s	1.09 s
TLB + cache (min)	**10.89 s**	**29.06 s**	**14.50 s**	**12.41 s**	**6.03 s**	**0.02 s**	**12.90 s**	**9.13 s**	**13.32 s**
TLB + cache (max)	**10.89 s**	**29.06 s**	**24.90 s**	**16.44 s**	**11.90 s**	**0.02 s**	**12.90 s**	**9.13 s**	**13.32 s**
actual difference	**10.7 s**	27.2 s	**24.8 s**	**8.5 s**	**12.4 s**	**1.5 s**	**15.7 s**	**11.9 s**	**11.6 s**

Table 5.11: Cache and TLB statistics of DOT and SAXPY algorithms for a 256x256 size matrix multiply problem. The **min** and **max** times represent the delay computed using the miss penalty per miss and miss penalty per line respectively. The lowest portion of the table compares the delay prediction against the real execution time difference between the two algorithms.

Program DOT: matrix 512x512									
	DEC 3100	**DEC 5400**	**MIPS M/2000**	**VAX 9000**	**RS/6000 530**	**HP 9000/720**	**Sparc I**	**Sparc I+**	**DEC 5500**
data references	268,697,600	268,697,600	268,697,600	268,697,600	268,697,600	268,697,600	268,697,600	268,697,600	268,697,600
occupancy ratio	16	16	16	8	16	4	8	16	16
cache misses	268,697,600	168,058,880	142,899,200	142,868,480	142,868,480	151,271,424	168,046,592	168,058,880	168,058,880
miss ratio	1.0000	0.6255	0.5319	0.5318	0.5318	0.5630	0.6254	0.6255	0.6255
ex. time (min)	145.10 s	282.27 s	114.32 s	105.72 s	50.01 s	54.46 s	131.08 s	94.11 s	126.04 s
ex. time (max)	145.10 s	282.37 s	205.77 s	140.03 s	100.02 s	72.62 s	131.08 s	94.11 s	126.04 s
occupancy ratio	4	4	4	1/8	2	2	1/8	1/1024	4
TLB misses	67,502,080	67,502,080	67,502,080	384	67,241,984	33,882,112	24	1	67,502,080
miss ratio	0.2512	0.2512	0.2512	0.0000	0.2503	0.1261	0.0000	0.0000	0.2512
execution time	32.40 s	27.00 s	23.63 s	0.00 s	72.62 s	31.85 s	0.00 s	0.00 s	17.55 s
TLB + cache (min)	177.50 s	309.37 s	137.95 s	105.72 s	122.63 s	86.51 s	131.08 s	94.11 s	144.59 s
TLB + cache (max)	177.50 s	309.37 s	229.41 s	140.03 s	172.64 s	104.47 s	131.08 s	94.11 s	144.59 s

Program SAXPY: matrix 512x512									
	DEC 3100	**DEC 5400**	**MIPS M/2000**	**VAX 9000**	**RS/6000 530**	**HP 9000/720**	**Sparc I**	**Sparc I+**	**DEC 5500**
data references	402,915,328	402,915,328	402,915,328	402,915,328	402,915,328	402,915,328	402,915,328	402,915,328	402,915,328
occupancy ratio	16	16	16	8	16	4	8	16	16
cache misses	138,936,294	41,228,272	16,801,264	8,667,136	8,667,136	19,052,540	37,555,192	41,228,272	41,228,272
miss ratio	0.3448	0.1023	0.0417	0.0215	0.0215	0.0473	0.0932	0.1023	0.1023
ex. time (min)	75.03 s	69.26 s	13.44 s	6.41 s	3.03 s	6.86 s	29.29 s	23.09 s	30.92 s
ex. time (max)	75.03 s	69.26 s	24.19 s	8.49 s	6.07 s	9.15 s	29.29 s	23.09 s	30.92 s
occupancy ratio	4	4	4	1/8	2	2	1/8	1/1024	4
TLB misses	262,400	262,400	262,400	384	268,289	131,200	24	1	262,400
miss ratio	0.0007	0.0007	0.0007	0.0000	0.0007	0.0003	0.0000	0.0000	0.0007
execution time	0.13 s	0.10 s	0.09 s	0.00 s	0.29 s	0.12 s	0.00 s	0.00 s	0.07 s
TLB + cache (min)	75.16 s	69.36 s	13.53 s	6.41 s	3.32 s	6.98 s	29.29 s	23.09 s	30.99 s
TLB + cache (max)	75.16 s	69.36 s	24.28 s	8.49 s	6.36 s	9.27 s	29.29 s	23.09 s	30.99 s

DOT – SAXPY: matrix 512x512									
	DEC 3100	**DEC 5400**	**MIPS M/2000**	**VAX 9000**	**RS/6000 530**	**HP 9000/720**	**Sparc I**	**Sparc I+**	**DEC 5500**
cache misses	129,761,296	126,833,680	126,190,096	134,201,344	134,201,344	132,218,884	130,491,400	126,833,680	126,833,680
ex. time (min)	70.07 s	213.08 s	100.95 s	99.31 s	46.97 s	45.60 s	101.79 s	71.02 s	95.13 s
ex. time (max)	70.07 s	213.08 s	181.71 s	131.52 s	93.94 s	63.47 s	101.79 s	71.02 s	95.13 s
TLB misses	67,239,680	67,239,680	67,239,680	0	66,973,695	33,750,912	0	0	67,239,680
execution time	32.28 s	29.90 s	23.53 s	0.00 s	72.33 s	32.73 s	0.00 s	0.00 s	17.48 s
TLB + cache (min)	102.35 s	242.98 s	124.48 s	99.31 s	119.30 s	78.33 s	101.79 s	71.02 s	112.61 s
TLB + cache (max)	102.35 s	242.98 s	205.24 s	131.52 s	166.27 s	96.20 s	101.79 s	71.02 s	112.61 s
actual ex. time	112.6 s	217.1 s	210.9 s	91.1 s	129.9 s	107.5 s	124.5 s	94.7 s	104.3 s

Table 5.12: Cache and TLB statistics of DOT and SAXPY algorithms for a 512x512 size matrix multiply problem. The **min** and **max** times represent the delay computed using the miss penalty per miss and miss penalty per line respectively. The lowest portion of the table compares the delay prediction against the real execution time difference between the two algorithms.

a large line size is normally an advantage on most benchmarks, the extra delay paid to load a complete line causes problem on DOT.

On tables 5.11-5.12 we show the delay predictions obtained using the cache and TLB experimental measurements of §5.5 and the analysis of DOT and SAXPY made on §§5.8.2 and 5.8.3. For each program and problem size, we give the number of data references, cache misses, miss ratio, and the respective execution time delay due to misses. We also compute the difference in execution time between the two algorithms and compare these values against the actual differences. This are given under the heading "DOT – SAXPY". Note that for those machines with wraparound loads we quote two numbers: one assuming that the miss penalty equals only the time needed to satisfy a word miss, and the other where the miss penalty equals the time to load a line.

The correct prediction numbers for wraparound load caches should be those corresponding to line misses. The reason is that the intergap miss distribution of algorithm DOT is skewed to very short distances. We can obtain the intergap miss distribution in the following way. First, the stream of references on DOT is characterized by the regular expression: $(C\ (A\ B)^N\ C)^{N^2}$. This pattern and the fact that every reference to A produces a miss as indicated by equation (5.3), means that intergap miss distance of two consecutive misses cannot be greater than two. More precisely, the intergap miss distribution for DOT is given by:

$$Dist(\text{DOT}) = \begin{cases} \dfrac{2}{b} N^3 + 2N^2 + 2(Rb - R) N\, \delta(a, 1) & \text{at distance 1} \\ \left(1 - \dfrac{1}{b}\right) N^3 - (Rb - R)\, \delta(a, 1) & \text{at distance 2} \end{cases} \tag{5.11}$$

Because of this high rate of misses and the small amount of computation between them, the penalty delay per miss on DOT should be equal to the penalty on a line miss instead of a single word miss. In contrast, on SAXPY, $N^3/b - RN^2/b$ of the misses have an intergap miss distance of at least $3b - 1$. Moreover, this intergap distance increases as b grows. The results on table 5.11 support this analysis, as the actual delay differences between the two algorithms appear to correlate better with the the line miss penalties.

5.9. Conclusions

In this chapter we have shown that we can extend our basic abstract machine model to incorporate the effects of program locality and the characteristics of the memory hierarchy to compute the delay due to the misses that occur at some level of the memory hierarchy. Our predictions are reasonably good with respect to actual observations. An important aspect of our methodology, and something which is illustrated in this chapter, is that we can construct relatively simple machine-independent tools for making good observations about the behavior of different units on many machines using programs written in a high-level language. These measurements are accurate enough to make predictions and at the same time can be used to compare machines with different instructions sets or memory structures. In §5.7 we showed how our cache and TLB measurements can be used to explain, in conjunction with the machine characterizations, the performance differences observed on machines with similar characteristics.

5.10. References

[Alpe90] Alpern, B., Carter, L., and Feig, E., "Uniform Memory Hierarchies", *Proc. of the Symp. on Foundations of Computer Science*, October 1990.

[Borg90] Brog, A.m Kesslet, R.E., and Wall, D.W., "Generation and Analysis of Very Long Address Traces", *Proc. of the 17th Int. Symp. on Comp. Arch.*, Seattle, Washington, May 1990, pp. 270-279.

[CYPR90] Cypress Semiconductors, *SPARC Reference Manual*, Cypress Semiconductors, 1990.

[Furl90] Furlong, T.C., Nielsen, M.J.K., Wilhelm, N.C., "Development of the DECstation 3100", *Digital Technical Journal*, Vol.2, No.2, Spring 1990, pp. 84-88.

[GeeJ91] Gee, J., Hill, M.D., Pnevmatikatos, D.N., and Smith A.J., "Cache Performance of the SPEC Benchmark Suite", *submitted for publication*, also University of California, Berkeley, Technical Report No. UCB/CSD 91/648, October 1991.

[GeeJ92] Gee, J. and Smith, A.J., "TLB Performance of the SPEC Benchmark Suite", *paper in preparation*, January 1992.

[Hill87] Hill, M.D., *Aspects of Cache Memory and Instruction Buffer Performance*, Ph.D. Thesis, U.C. Berkeley, Computer Science Division Tech. Rept. UCB/CSD 87/381, November 1987.

[Hill89] Hill, M.D. and Smith, A.J., "Evaluating Associativity in CPU Caches", *IEEE Trans. on Computers*, Vol.38, No.12, December 1989, pp. 1612-1630.

[LamM91] Lam, M., Rothberg, E.E., and Wolf, M.E., "The Cache Performance and Optimizations of Blocked Algorithms", *Proc. of the Fourth Int. Conf. on Arch. Support for Prog, Lang. and Op. Sys. (ASPLOS-IV)*, Santa Clara, California, April 8-11 1991, pp. 63-74.

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

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

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

[Smit82] Smith, A.J., "Cache Memories", *ACM Computing Surveys*, Vol.14, No.3, September 1982, pp. 473-530.

[Smit85] Smith, A.J., "Cache Evaluation and the Impact of Workload Choice", *Proc. of the 12'th Int. Symp. on Comp. Arch.*, Boston, Massachusetts, June 17-19 1985, pp. 64-75.

[Smit87] Smith, A.J., "Line (Block) Size Choice for CPU Caches", *IEEE Trans. on Computers*, Vol. **C-36**, No.9, September 1987, pp. 1063-1075.

[SPEC90a] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol.2, Issue 2, Spring 1990.

[SPEC90b] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol.2, Issue 3, Summer 1990.

[SPEC91] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol.3, Issue 1, Winter 1991.

Appendix 5.A

machine		DODUC	FPPPP	TOMCATV	MATRIX300	NASA7	SPICE2G6	Average	RMS
DECstation 3100	real	352 s	664 s	674 s	1176 s	4103 s	4102 s	n.a.	n.a.
	pred 1	346 s	510 s	648 s	922 s	4207 s	5702 s	n.a.	n.a.
	error	-1.70%	-23.19%	-3.86%	-21.60%	2.53%	39.01%	-1.47%	20.62%
	pred 2	350 s	543 s	703 s	1019 s	4568 s	6105 s	n.a.	n.a.
	error	-0.57%	-18.22%	4.30%	-13.35%	11.33%	48.83%	5.39%	22.52%
DECsystem 5400	real	330 s	625 s	619 s	1017 s	3695 s	3994 s	n.a.	n.a.
	pred 1	325 s	480 s	583 s	863 s	3824 s	5462 s	n.a.	n.a.
	error	-1.52%	-23.20%	-5.82%	-15.14%	3.49%	36.76%	-0.90%	19.00%
	pred 2	332 s	532 s	668 s	992 s	4352 s	6082 s	n.a.	n.a.
	error	0.61%	-14.88%	7.92%	-2.46%	17.78%	52.28%	10.21%	23.59%
MIPS M/2000	real	187 s	247 s	542 s	816 s	2906 s	4576 s	n.a.	n.a.
	pred 1	208 s	239 s	415 s	614 s	2634 s	4539 s	n.a.	n.a.
	error	11.23%	-3.24%	-23.43%	-24.75%	-9.36%	-0.81%	-8.39%	15.20%
	pred 2	211 s	256 s	433 s	677 s	2852 s	4761 s	n.a.	n.a.
	error	12.83%	3.64%	-20.11%	-17.03%	-1.86%	4.04%	-3.08%	12.20%
VAX 9000	real	54 s	42 s	93 s	191 s	599 s	1620 s	n.a.	n.a.
	pred 1	57 s	54 s	81 s	111 s	530 s	1357 s	n.a.	n.a.
	error	5.56%	28.57%	-12.90%	-41.88%	-11.52%	-16.23%	-8.07%	22.96%
	pred 2	57 s	54 s	87 s	125 s	633 s	1454 s	n.a.	n.a.
	error	5.56%	28.57%	-6.45%	-34.55%	5.68%	-10.25%	-1.91%	19.24%
IBM RS/6000 530	real	135 s	93 s	196 s	630 s	1601 s	2438 s	n.a.	n.a.
	pred 1	125 s	101 s	244 s	404 s	1815 s	3385 s	n.a.	n.a.
	error	-7.41%	8.60%	24.49%	-35.87%	13.37%	38.84%	7.00%	24.84%
	pred 2	125 s	101 s	246 s	484 s	1978 s	3457 s	n.a.	n.a.
	error	-7.41%	8.60%	25.51%	-23.17%	23.55%	41.80%	11.48%	24.56%
HP 9000/720	real	85 s	78 s	182 s	598 s	1250 s	1588 s	n.a.	n.a.
	pred 1	85 s	90 s	157 s	340 s	1056 s	1869 s	n.a.	n.a.
	error	0.00%	15.38%	-13.74%	-43.14%	-15.52%	17.70%	-6.55%	21.76%
	pred 2	85 s	99 s	168 s	387 s	1175 s	1928 s	n.a.	n.a.
	error	0.00%	26.92%	-7.69%	-35.28%	-6.00%	21.41%	-0.11%	20.51%
Sparcstation 1	real	344 s	361 s	571 s	1300 s	5118 s	3594 s	n.a.	n.a.
	pred 1	341 s	446 s	603 s	803 s	3906 s	4911 s	n.a.	n.a.
	error	-0.87%	23.55%	5.60%	-38.23%	-23.68%	36.64%	0.50%	25.66%
	pred 2	343 s	470 s	642 s	848 s	4109 s	5137 s	n.a.	n.a.
	error	-0.29%	30.19%	12.43%	-34.77%	-19.71%	42.93%	5.13%	27.41%
Sparcstation 1+	real	419 s	599 s	585 s	1369 s	5350 s	4700 s	n.a.	n.a.
	pred 1	410 s	481 s	637 s	815 s	4045 s	5660 s	n.a.	n.a.
	error	-2.15%	-19.70%	8.89%	-40.47%	-24.39%	20.43%	-9.57%	22.81%
	pred 2	412 s	498 s	665 s	849 s	4209 s	5864 s	n.a.	n.a.
	error	-1.67%	-16.86%	13.68%	-37.98%	-21.33%	24.77%	-6.57%	22.31%
DECsystem 5500	real	182 s	340 s	391 s	680 s	2442 s	2500 s	n.a.	n.a.
	pred 1	172 s	200 s	343 s	511 s	2188 s	3761 s	n.a.	n.a.
	error	-5.49%	-41.18%	-12.28%	-24.85%	-10.40%	50.44%	-7.29%	29.29%
	pred 2	175 s	223 s	381 s	574 s	2432 s	4039 s	n.a.	n.a.
	error	-3.85%	-34.41%	-2.56%	-15.59%	-0.41%	61.56%	0.79%	29.55%
average	pred 1	-0.26%	-3.82%	-3.67%	-31.77%	-8.39%	24.75%	-3.86%	n.a.
RMS	pred 1	5.30%	23.22%	14.20%	33.21%	14.62%	32.08%	n.a.	22.78%
average	pred 2	0.58%	1.51%	3.00%	-23.80%	1.00%	31.93%	2.37%	n.a.
RMS	pred 2	5.47%	22.51%	13.27%	26.55%	14.58%	39.00%	n.a.	22.92%

Table 5.13: This table shows the real and predicted execution times with their corresponding errors. Predictions labeled *pred 1* ignore the amount of delay due to misses, while those labeled *pred 2* include the delay.

6 Conclusions and Future Directions

6.1. An Abstract Machine Performance Model

The main goal of this dissertation has consisted in showing that it is possible to construct a realistic performance model using the concept of an abstract machine, and that this model can be used to characterize machines with different instructions sets in a way that we can predict the execution time of arbitrary programs and at the same time explain the results in terms of machine and program characteristics. Furthermore, the basic model can be extended to take into account the ability of optimizing compilers to improve the execution time, and the delay experienced by programs due to the memory system.

One hypothesis of this research was that it is possible to use benchmarking techniques to make detailed experimental measurements on machines covering a large range of performance with enough accuracy so as to allow us to make reasonable predictions. The fact that it is possible to characterize machines using a machine-independent model was shown in Chapter 2. That these measurement coupled with program dynamic statistics can be combined to produce execution time prediction was shown in Chapter 3.

In Chapter 4 we extended the performance model to include the effects of optimizing compilers. We did this by first identifying a large set of optimizations which are invariant with respect to an abstract decomposition of the program, that is, these optimizations only affect the object code while preserving the original computation. Because our characterization of the program focuses on the source code and not on the object code, our dynamic statistics are valid for optimized and nonoptimized compilations. We show that when using a machine characterization of the 'optimized' system it is possible to predict the execution time for most programs independently of their execution time or apparent complexity.

Chapter 5 finalized our presentation by incorporating the effects of locality of reference in our model. In it we described how to measure experimentally the most important parameter of the memory system and how these affect programs. We used these results to compare the cache and TLB organizations of many systems and identify their main

differences. We concluded the chapter by evaluating the performance differences of four different systems all based on the R3000/R3010 microprocessors and whose only difference is the organization of their memory systems. We showed that variations in performance are mainly the results of their different memory organizations.

6.2. Narrow Spectrum Benchmarking

Every parameter in our performance model is obtained either by running an experiment on the machine or by making an analysis of the source code. The experimental measurement of machine parameters is done using narrow spectrum benchmarking, which allow us to isolate each abstract operation and measure it under the "typical" conditions it is used in most programs. However, the different contexts in which an operation is executed vary substantially within a program and from program to program; so, there are intrinsic limitations by using only the measurements corresponding to the most frequent use. In fact, this is the main source of error in our predictions. Furthermore, because all our tests are done from a program written in a high level language and using poor resolution timing functions, this makes difficult to measure a phenomena which lasts from tenths to thousand of nanoseconds. In Chapter 2 and 3 we showed, however, that it is possible to make good measurements using narrow spectrum benchmarking and that predictions obtained from them match very well actual execution times.

6.2.1. Machine and Program Similarity

In Chapter 2 we introduced the concept of *performance shape* which depicts graphically for each machine its performance distribution on a set of reduced parameters representing the most important CPU characteristics. These reduced parameters are synthesized from the original 109 abstract parameters. We then presented a similarity measure, which we called *pershape distance*, that clusters machines according to their respective performance shapes. We presented evidence that the pershape distance is related to the amount of dispersion found in the distribution of the relative performance between two machines found when using a large number of benchmarks. In addition, the clustering of machines helps in identifying patterns corresponding to the way that the performance of machines evolves over time.

6.3. Future Work

6.3.1. Operating Systems and Input/Output

In this dissertation we attacked the problem of characterizing the performance of the CPU, the memory system, and the compiler. The natural extension is to include the effects of input/output activity and the operating system. One way of doing this is by considering the system calls of the operating system as the set of abstract operations of a system defined by the functionality of the operating system itself. Because the execution time of some of these system calls depends on the size of the input and the level of activity in the system, a single performance number is not sufficient. Accounting for the size of the input, however, is not difficult. In many cases a simple scheme can be used in which the execution time of a system call is equal to a constant startup time, plus a term which depends on its input's size. If it is the case that, depending on the size of the input, there is more than one execution domain, then the last subterm can comprise several

subterms, one for each domain.

What is significantly more difficult to characterize is the effect of higher levels of activity in the system. In this case, it is not clear how to control this activity in order to: a) make meaningful measurements; b) relate these measurements to an arbitrary state of the system; and c) model the unpredictability of the system and incorporate this into the predictions. One example of this is the amount of paging generated by the active processes. The number of active processes that are competing for main memory at a given moment determines the time it takes to satisfy a page fault or whether a process will be swapped out of the system while waiting on the ready queue, and this information cannot be known in advance.

6.3.2. Architecture Specific Abstract Machine

Abstracting the characteristics of machines that have different instruction sets into a single performance model is what allows us to compare them and make predictions for arbitrary programs. It also makes it possible to consider the execution of a program as machine independent, and hence use the same set of dynamic statistics on all machines without considering in detail how each compiler works. However, we pay a price in doing this, which manifests itself in our inability to make precise statements about how machine dependent characteristics account for the behavior we observe; all we can do here is to identify the abstract operations which account for the observed behavior. Machine designers, however, need to identify the specific machine operations and functional units which are the source of any performance imbalance.

One possible solution to this problem is to construct an abstract machine model based on a particular instruction set, and then use it to evaluate different implementations of the architecture. Because the same compiler is used on all the implementations, there is only one set of dynamic statistics to consider. Machine designers normally build very detailed simulators of their systems, and use these to execute real programs and collect performance measurements. This approach is very time consuming and limits the number of programs that can be effectively simulated. We believe that our approach could be used to speed-up substantially the time it takes now using simulation to obtain the execution time of a program.

6.3.3. Parallel Architectures

It is clear that in the future substantial improvements in performance are going to come from executing programs on massively parallel machines based on hundreds or thousands of commodity microprocessors instead of on sequential machines based on very powerful special purpose processors. Characterizing the performance of a parallel machine, however, is much more difficult than that of any vector or a scalar machine, mainly because the size of the performance space is substantially larger. For example, the variation in performance of sequential machines on different programs is rarely larger than a factor of 10. On vector machines, like the CRAY Y-MP, the performance difference between scalar and vector operations can be as high as 100. On a parallel machine having thousands of processors, the range of potential amounts of variation can be as large as 1000 or more. Furthermore, other factors not present in sequential execution makes it much more difficult to characterize the behavior of parallel programs. Some of these factors are: process communication and synchronization, process scheduling, and

resource allocation.

Parallel architectures, some based on the shared memory model and others on the message passing model, have proliferated in the last decade and will continue to grow in the years ahead. Likewise, the number of different topologies and interconnection networks have followed the same trend. For these and other reasons, it is unlikely that a uniform performance model for parallel computation can be developed that would make it possible to predict reasonably well the execution of arbitrary programs. Any abstract model of parallel computation would be oblivious to most of the underlying hardware characteristics which at least now have a direct impact on the performance of parallel programs. More specifically, the performance observed on different patterns of communication between processes is directly related to how these patterns map to the underlying topology, and there is no way to model this unless there is specific information in the model. However, we believe that it is possible to apply the techniques presented in this dissertation to particular topologies and produce reasonably good models whose execution time estimates will correlate well with actual times.

Two machines which appear to be good candidates for applying our performance evaluation methodology are the nCUBE and the Intel iPSC/860. Both of these machines use the hypercube as their interconnection network, and communication and synchronization between processes is done by message passing and is explicit. This means that user programs written for these machines contain system calls to send and receive information. Because communication primitives are exposed in the source code, we can use a modified version of our analyzer to characterize the distance and the length of messages exchanged at different points in the program. We believe that this information, coupled with a model of communication, can be used to make reasonable predictions on this class of machines.