Machine Characterization and Benchmark Performance Prediction[†]

Rafael H. Saavedra-Barrera

Department of Electrical Engineering and Computer Science
Computer Science Division
University of California
Berkeley, California 94720

and

Departamento de Ingenieria Eléctrica
División de Ciencias Básicas e Ingenieria
Universidad Autónoma Metropolitana, Iztapalapa
México D.F., México

ABSTRACT

From runs of standard benchmarks or benchmark suites, it is not possible to characterize the machine nor to predict the running time of other benchmarks which have not been run. In this paper, we report on a new approach to benchmarking and machine characterization. We describe the creation and use of a machine analyzer, which measures the performance of a given machine on Fortran source language constructs. The machine analyzer yields a set of parameters which characterize the machine and spotlight its strong and weak points. We also describe a program analyzer, which analyzes Fortran programs and determines the frequency of execution of each of the same set of source language operations. We then show that by combining a machine characterization and a program characterization, we are able to predict with good accuracy the running time of a given benchmark on a given machine. Characterizations are provided for the Cray X-MP/48, Cyber 205, IBM 3090/200, Amdahl 5840, Convex C-1, VAX 8600, VAX 11/785, VAX 11/780, SUN 3/50 and IBM RT-PC/125, and for the following benchmark programs or suites: Los Alamos (BMK8A1), Baskett, Linpack, Livermore Loops, Mandelbrot Set, NAS Kernels, Shell Sort, Smith, Whetstone and Sieve of Erathostenes.

June 30, 1988

† The material presented here is based on research supported in part by NASA under consortium agreement NCA2-128, the Mexican Council for Science and Technology (CONACYT) under contract 49992, the National Science Foundation under grants CCR-8202591 and MIP-8713274, by the State of California under the MICRO program, and by the International Business Machines Corporation, Digital Equipment Corporation, Tandem, Hewlett-Packard, and Signetics.

Table of Contents

List of Figures

List of Tables

1
Introduction

Machine performance is described by specifying, in the case of the CPU, the timings for all instructions, instruction interactions within the pipeline, storage delays and delay probabilities, etc. This approach for estimating performance is commonly used by computer architecture performance experts in the course of designing a new architecture or implementing an existing one [PEU77, MAC84, WIE82, EME84, CLA85]. This type of characterization makes it very difficult to compare machines with different instructions sets. The standard method of evaluating computers consists of selecting some "typical" existing programs and running them on the new machine(s), i.e. benchmarking. There are a number of known problems with this approach [DON87b, WOR84]:

(a) Unless the existing programs are modified, they may not take advantage of the new architecture.

(b) It is not reasonable to expect that a single figure of merit can meaningfully characterize a computer system.

(c) Each benchmark is itself a mixture of characteristics, and doesn't relate to a specific aspect of machine performance.

(d) It is very difficult to infer the performance of the N+1'st benchmark as a function of benchmarks 1,...,N.

(e) It is very difficult to predict the behavior of existing benchmarks on a new machine, even given the characteristics of the new machine, without running the benchmarks.

In this work we propose a new method to characterize the performance of computer systems at the level at which applications are written. This description can be used to predict the behavior of real workloads. The characterization is via experimental measurement of individual components of performance. We argue that by evaluating machines according to a number of (somewhat) independent parameters, it is possible to estimate performance for a wide range of workloads. This will permit much more valid comparisons between machines, and expose machine weaknesses or strong points for use by both the customer (in purchase, and in job assignment among various machines), and by the manufacturer who can then work to improve the next version of the product.

This report is organized as follows. In section 2 we discuss the limitations of existing benchmarks and identify the characteristics that must be taken into account in the design of programs that can be used in the future as industry standards for system characterization. Section 3 describes our model for system characterization and performance prediction. In sections 4-6 we present the main modules of our system: the system characterizer, the program analyzer, and the performance predictor, and describe their principal components. In Section 7 we make an analysis of our results, discuss future improvements, and give a summary of this report.

2
Limitations of Benchmarking

In the past few years there has been great interest in performance evaluation caused mainly by an increase in the number of different new architectures. The number of benchmarks currently used to evaluate these systems is growing day by day, and new studies of the performance of these machines appear either in technical journals or in popular magazines. Every new benchmark is created with the expectation that it will become the standard of the industry and that manufacturers and customers will use it as the definitive test to evaluate the performance of computer systems with a similar computer architecture. Sooner or later every major user of computer resources publishes its own benchmark that characterized the workload of that scientific institution [BAI85a, BUC85, CUR76, MCM86]. Most of the time these benchmarks provide useful information only to the particular set of users that are represented by the programs. The limitations of benchmarks mentioned in the introduction are well known, but those objections do not show the fundamental problems with benchmarking. To understand why the results obtained with these programs are inadequate for system characterization, we must discuss in more detail how to characterize a computer system, and its relation to performance evaluation.

2.1. System Characterization versus Performance Evaluation

Benchmarks, whether they are real programs, synthetic benchmarks, or kernels, have the problem that they confuse two different things: system characterization and performance evaluation. We define *system characterization* as an n-value vector where each of the components represent the performance of a particular primitive operation. This vector fully describes the whole system at some level of abstraction. From the designer's point of view the primitive operations could be the fetching of an instruction stream into the instruction buffer, a translation buffer miss, branching to a microinstruction, and so on. Users see the system at the level at which they write their applications, and for them, the primitive functions are the set of operations supported by the language they use. Here the execution time of each primitive function depends not only on the hardware but also on the code produced by the compiler, and sometimes on the libraries and the operating system.

The *performance evaluation* of a computer system is the measurement of some number of properties of the system during the execution of a particular workload. The properties measured may be the total execution time to complete some job steps, the memory used during each of the different steps, etc. The important thing to note here is that the evaluation depends on the set of programs executed. This means that the estimate is valid only for that particular suite. All of the existing benchmarks evaluate the performance of the system when this system is running that specific benchmark, and the results thus obtained cannot be extrapolated to other benchmarks or real workloads. This does not mean that benchmarks are not useful; they provide a good first approximation of the expected performance of the systems they measure.

Standard benchmarking can be described as an experimental evaluation of alternatives. Each experiment represents a point in the performance space of the system. Only by decomposing these measurements and relating them to the characteristics of the benchmarks can we use them to predict the performance of different workloads. We must start writing benchmarks that measure basic individual components that affect the performance of computer systems, and then use these results to evaluate their behavior. This implies that system characterization and performance evaluation must be seen as two independent activities. In this model, benchmarking is part of the characterization process. We will refer to this new kind of benchmarks as system characterizers to distinguish them from normal benchmarks. The results obtained with a system characterizer represent different aspects of the architecture and the software. It is clear that these parameters cannot be combined to produce a single figure of merit. With only one number it is not possible to isolate the effects of hardware and software components during the execution of a variety of applications.

2.2. The Role of Experimentation in System Characterization

Experimentation plays an important role in science and especially in fields like Physics, Biology and Chemistry. Practitioners of these fields use experiments to collect information about a phenomenon, and then analyze the data to sustain or refute a hypothesis about the phenomenon. Experimentation plays the role of predictor-corrector tool in the development of models and theories. Unfortunately the concept of experimentation in computer science is not a well defined activity, [DEN80, DEN81, MCC79, FEL79] especially in benchmarking where experimentation is confused with running programs and timing their execution times. A hypothesis or model to validate is almost never present.

An experimental performance evaluation of a computer system must satisfy several conditions to be considered experimentally sound. First, the results must be reproducible. This means that independent researchers must be able to produce the same results using different experiments. The results must also be consistent; the repetition of an experiment must produce the same results. The experiments must be performed in a controlled environment and the effect of extraneous variables must be quantified. And lastly, a model of the execution of the system must exist so the results of an experiment can be related to previous and future experiments.

The last condition is especially a weak point in benchmarking. We see experimentation as the only way in which disputes can be settled regarding the comparative performance of different systems and the effect of particular components on the performance of the system, as the only way of verifying system improvements, and as the only way of establishing a cumulative tradition in which improvements can be introduced and new architectures can be evaluated.

3
A Model for Performance Evaluation

Before we present our model for performance evaluation, we will first summarize our discussion of the last sections. We know that every performance evaluation is relative to the workload used to make the measurements, and also depends on the characteristics of the computer. It is not possible to evaluate a system without knowing the structure of the workload and the behavior of the different components of the system when the computer executes that workload. In an ideal world, we will expect that for any two machines there exists an order on their performance. If machine A executes program X faster than machine B, then if we replace X by any other program we obtain the same relationship. Unfortunately, this is almost never the case, and in some cases the difference can be significant. For this reason some of the goals of performance evaluation (prediction) should be to help us answer some of the following questions:

- For which set of programs will machine A execute faster than machine B (without having to run the programs on both machines)?
- What are the portions of the programs that will consume more resources in different machines?
- What are the components that have the potential of being the bottlenecks in the execution of some programs?

Looking at these three questions we can identify different subproblems that we must solve in order to answer them satisfactorily. The first involves the measurement of the performance of the individual components. This is what we called in the last section system characterization. Second, we need to decompose the workload using the same set of parameters. We should be able to make an analysis of the workload in terms of the parameters used to characterize the systems. Lastly, in order to solve the first question, we need to combine the characterization of the system with the analysis of the workload. This last phase we will call it execution prediction, and is part of the performance prediction of the system. In figure 3.1 we show the different stages of this process.

3.1. A Common Representation for Programs and Computer Systems

If we want to make predictions at the level of user programs and at the same time to quantify the behavior of the different components of the systems, we need to represent the systems and the programs using the same model. This model could be the machine code produced by the compiler, but this approach has several problems. First we need to know for every machine (in fact for every compiler) the code produced for each of the language constructs. Second the representation thus obtained is only valid for the machines that have the same instruction set. It becomes necessary for the program analyzer to know the inner workings of each possible compiler, or to compile each program in each machine and then to analyze the object code. What we want is a flexible common representation for all the systems that is independent of the architecture and the compiler.

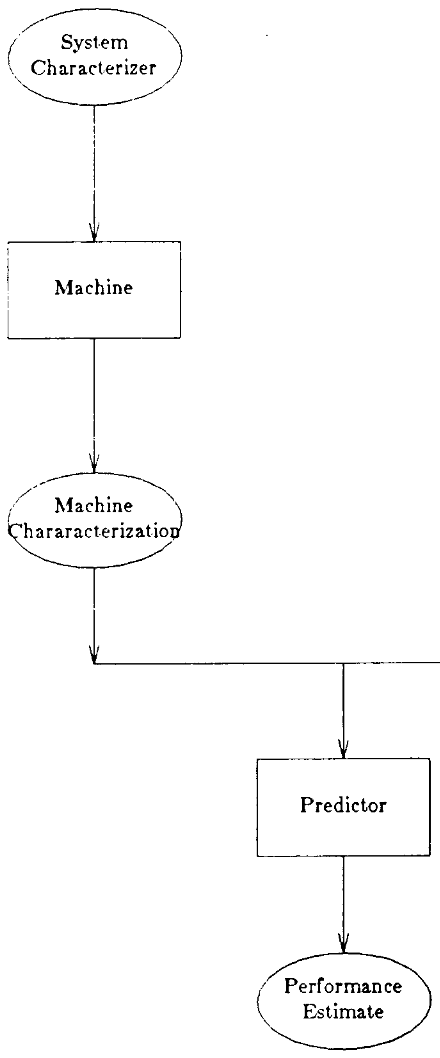


Figure 3.1: System characterization and performance analysis. On the upper left part we see the system characterization represented as a program (benchmark) executed in a computer and producing the characterization of the system. On the upper right side we have the program analyzer, which takes as input an application program and decomposes it (statically and dynamically) using the same set of parameters used in the system characterizer. The lower part represents the synthesis of both the characterization and the analysis producing the performance evaluation of the machine relative to the workload.

The execution time of a program depends on the code produced by the compiler, and this code is a function of the operations and control statements that a particular language provides. For this reason the set of parameters to use as a base in our decomposition of program and systems must be an abstraction of the operations supported by most of the machines, and these parameters must be identified with a particular operation or control statement of the high-level language.

It should be clear that the number of parameters depends not only on the number of operations and control statements that a programming language has, but also on the accuracy of the estimates that we want to obtain.

3.2. The Class of Systems Studied

A general model that produces estimates of the possible performance for all the different architectures and modes of computations is unlikely to exist in the near future. Nevertheless this does not mean that it is not possible to obtain a model of the performance for a significant number of the architectures, those which have a common mode of operation. Moreover once we have a good model for this class of system, it is possible to extend it to include more complex architectures. The approach that we use in this research is to choose a model of computation and obtain a model of the performance for the machines that share that mode of operation; we will later extend it to include more complex systems.

In this study we will restrict ourselves to a particular model of computation, in which we have in each system a single processor running in scalar mode, and the code generated by the compiler is not optimized. We will assume that the uniprocessor system does not support vector operations or, more precisely, that the compiler does not produce vector operations. We will also assume that the programs are compiled with the optimization switch turned off.

The machines described previously correspond to the SISD (single instruction stream/single data stream) model in the classification made by Flynn [FLY72]. Although there is only one processor executing a single stream of data, this does not imply that the processor does not have parallelism at the level of the execution of machine instructions. The execution of an instruction can occur at the same time that the next instruction is decoded and the last instruction is executed.

The characterization of systems with vector operations and/or including optimization has several problems not present in scalar processing without optimization. Although it is not difficult to extend the characterizer to include vector operations, using this information to predict the execution time of programs requires also the characterization of the compiler. Only by knowing which DO loops the compiler is capable of vectorizing can we make an acceptable prediction of the expected execution time. Even if it were possible to run experiments and detect when a particular compiler will generate vector code and which vector operations will be executed, we still have the problem of detecting in arbitrary programs the occurrence of possible vectorizations. This requires a program analyzer as 'smart' as any vectorizing compiler; in fact 'smarter', because it has to vectorize the same loops for any arbitrary compiler and program.

Optimization is even more difficult to handle. In addition to the problems mentioned in the last paragraph, we also find that there is not always a clear boundary on when to apply one optimization instead of another. In fact, applying one set of optimizations may prevent the compiler from detecting others. The decision of which optimization to try first normally depends on the order in which the optimizations are tried. In some cases optimization eliminates redundant or/and 'dead' code, especially inside DO loops and this not only affects user programs but also benchmarks, so validating the measurements is even more difficult. Lubeck et al. reported that vector optimization had to be disabled in order to obtain meaningful measurements on the Fujitsu VP-200 [LUB85]. If we add that most 'optimizing' compilers can only perform certain optimizations on some data types and not in others [LIN86a, LIN86b], we understand why it requires a 'super-optimizer' to know how a program will be modified in order to make accurate prediction of the expected execution time of optimized programs. It is outside the scope of this research to write such 'super-optimizer'; we will try to develop other techniques to characterize vectorization and optimization in the future.

3.3. A Linear Model for Program Execution

If we want to produce estimates of the time a program or set of programs will take to execute in some machines, we will need to produce a model of how the total execution time is obtained from the individual parameters. One approach used by machine designers is to obtain the mean execution rate of a system while executing a particular workload. To do this we decompose the mean instruction execution time I into the sum of three basic components [MAC84]

$$I = E + D + S \tag{3.1}$$

where D is the mean pipeline delay per instruction, caused by path conflicts, register dependencies, and taken branch delays; S is the mean storage access delay per instruction caused by a cache miss for instructions and operands; and E is the mean nominal execution time when there are no pipeline delay or storage access delays. In our model we do not deal with single machine instructions, but with the set of primitive operations supported by a particular programming language. Each of these primitive operations (parameters) is mapped into several machine instructions. Therefore the mean parameter execution time (P_i) is equal to the mean execution time of that sequence of machine instructions. We can decompose the mean execution time of each parameter as

$$P_i = E_i + D_i + S_i \tag{3.2}$$

where the three terms to the right of the equal sign have the same interpretation, but refer to a sequence of instructions instead of only one.

As we noted in the last paragraph the major difference between the two models is that hardware designers are interested in the mean execution time of each machine instruction, while in our model the set of parameters belong to a higher level of abstraction. The machine implementation (code produced by the compiler) of these parameters could be in its simplest case one machine instruction, but in most cases the compiler generates several machine instructions for each parameter.

The total execution time of a program is equal to the nominal total execution time (when no pipeline delays or storage delays occurs), plus the total pipeline delay time, and the total storage access delay time.

$$T = T_E + T_D + T_S \tag{3.3}$$

where

$$T_E = \sum_{i=1}^{n} C_i E_i \; ; \quad T_D = \sum_{i=1}^{n} C_i D_i \; ; \quad T_S = \sum_{i=1}^{n} C_i S_i \; ; \tag{3.4}$$

where C_i is the number of times parameter P_i is executed. We can use these equations to obtain the total execution time of the program as

$$T = \sum_{i=1}^{n} C_i (E_i + D_i + S_i) = \sum_{i=1}^{n} C_i P_i \tag{3.5}$$

Note that by using a system characterizer written in a high-level program it is neither possible to measure the mean nominal execution time of each parameter, nor the mean pipeline delay, nor the mean storage delay. What the system characterizer measures is the mean execution time (nominal execution plus pipeline delays and storage delays) of the set of machine instructions that implement each parameter.

3.4. Limitation of the Linear Model

The linear model for the execution time of applications proposed in the last section has some limitations. We assumed that the time it takes for the execution of n operations is just the sum of the individual execution times. In highly pipelined machines the execution time when there is a register dependency conflict may be several times greater than the execution time without this delay. As an example consider the CYBER 205 architecture. The scalar processor is derived from the CDC 7600. It has five arithmetic subunits within the Scalar Floating-point unit. All of them are pipelined and can accept a new pair of input operands at every clock cycle (20 ns). The Add/Subtract and Multiply Units each takes five clock periods to produce a result and return it to the input of another unit [IBB82]. Therefore it takes 100 ns from the beginning of the operation to the time the result is available. The execution time of n operations without any data dependency conflicts can take as little as $20n$ ns. On the other hand the execution of the same n operations can take $100n$ ns if each operation has a conflict with the next one. Consider the following two statements

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

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

if we compute their timing diagrams we find that for the first statement the execution of the RHS[1] takes approximately 360 ns (the four adds, that are leaves of the syntax tree, start execution in the first four cycles, the two multiplications in cycles 7 and 9, and the last add in cycle 14). For the second statement, the execution of the RHS takes 400 ns due to data dependencies. However the first statement executes seven operations, while the second only four. A simple linear model will not predict that the first statement will execute faster unless the model contains information about the behavior of the processor when data dependencies are present. Nevertheless we expect that in large programs discrepancies in different directions will cancel and therefore the total error will be small compared to the total execution time.

[1] Right Hand Side of the assignment statement.

4
The System Characterizer

System characterizers (to distinguish them from benchmarks) are a set of experiments that detect, isolate and measure hardware and software features. These features describe the system and determine its performance. The accuracy of the description depends on the number and detail of the experiments. A very coarse model would be one in which all floating point operations are represented by only one parameter. A better approximation will have as many parameters as there are floating point operations. An even better one will distinguish the length of the operand (number of bits) and their storage class. In some systems the time to access a variable depends on whether the variable is local or global.

Each parameter must be measured in a controlled way and, if repetition is used, the test must be run for a significant amount of time to reduce the experimental error due to clock resolution. If we view system characterization as an incremental process, we can build different system characterizers with different degrees of resolution. There are at least two possible benefits for doing this: (1) in the early phases of the evaluation process, it is appropriate and cost effective to use an approximation with not too many parameters. (2) If our system model does not detect some features for certain kinds of architectures, new experiments are incorporated or some of the existing ones are replaced with a minimum number of changes.

In figure 4.1 we present the process of characterization. On the left of the figure we have a single system characterizer run in several machines executing in uniprocessor scalar mode and without optimizing the code generated by the compiler. The output produced by the characterizer is the data base that we will use to produce performance estimates of the machines, with the help of the program analyzer and the execution predictor. Only one characterization per computer system is needed in order to estimates the performance of any program written in FORTRAN.

As an example, let us consider how characterization will work in the case of vector operations. Here we are interested in running some experiments to test the amount of vectorization that systems can do. Not only the vector operations that the hardware supports, but also the kind of language constructs that the compiler can detect as vectorizable. A possible way of characterizing this class of operations is by using two or more parameters. In the case of memory-to-memory vector machines only two parameters are needed, the startup time for the vector operation, and the asymptotic execution rate. This last parameter is the maximum rate at which the processor can execute a vector operation [HOC81, HWA84, HOC85, SHI87]. For register-to-register machines we need also the overhead time associated with the stripmining[1] process, and the length of the vector registers [BUC87, MAR87]. Machines with cache, the performance is also affected by

[1] When the number of elements in a vector operation is greater than the number of vector registers, the instruction must be treated as a sequence of vector operations. This technique is called stripmining, and is done at compile time. Because it takes some time to restart the next operation there is an overhead associated with stripmining and this overhead is normally less than the vector startup.

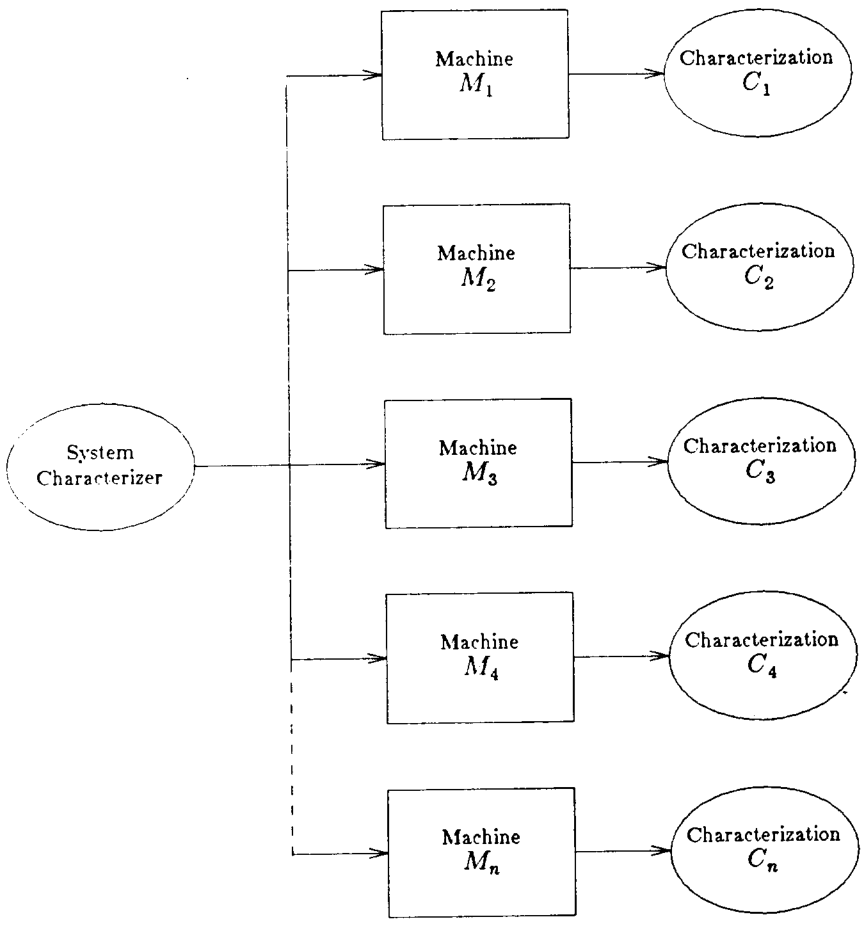


Figure 4.1: The same system characterizer is used in all systems

the size of cache; the asymptotic execution rate normally changes for vectors with a length greater than the size of the cache.

4.1. Machine Implementation of Data Types

To be useful the system characterizer must be easy to use and portable. Although FORTRAN has been standardized, there exist several differences between FORTRAN compilers and machines that makes complete portability difficult to achieve. One of these differences is the declaration of data types. Single precision real variables are implemented in CDC and CRAY machines using one word, which is equal to 64 bits, while double precision variables are assigned two words. The CYBER 205 supports another type named HALF PRECISION (32 bits). On the other hand, IBM 3090/200, VAX-11, and SUN 3 implement single precision with 32 bits and double precision with 64 bits. Also the *f77* compiler in UNIX systems implements single precision with 32 bits and double precision with 64 bits [FEL78]. In VAX machines running ULTRIX the *fort* compiler accepts quadruple precision, which is equivalent to double precision in CDC type machines. One way to avoid this problem is to specify explicitly the number of bits in the implementation of the different data types. In some compilers it is possible to say how many bytes to allocate to the variables by appending at the end of the type an asterisk follow by 2, 4, 8, or 16. But on the CYBER and CRAY the meaning of the first two options is different. These are changed to correspond to single precision on those machines. The problem is more difficult in the case of integer variables. The CDC-type machines support only 64-bit integers; on the CRAY, integers have 46 or 64 bits[2], while in the VAX and SUN machines integers are implemented either by 16 or 32 bits.

To make a fair comparison between different machines we need to make the evaluation under similar conditions in all systems. If all the machines are 32-bit microcomputers, the memory cell unit on all the systems is equal, and if the tests are run under the same conditions, it is not difficult to make a fair comparison. On the other hand if some machines are 64-bit mainframes, others 32-bit minicomputers and another subset is composed of 32-bit microcomputers, then is not clear how to make a fair comparison between the machines. Do we have to test the machines with all the data types implemented with 32 or 64 bits? How do we make a comparison if the machines do not have a common representation (same precision) of some data types? If some subset of the tests do not need more than 32-bit real numbers to execute correctly, why do we need to run these programs using 64-bit variables on the microcomputers? If a test gives erroneous results when run on a 32-bit machine, what is the point of saying that the machine runs at the same speed compared to a 64-bit mainframe?

The above discussion gives a hint of the difficulty of making a good comparison between machines even in the case when these have similar characteristics. The conditions in which comparisons are made should be decided case by case, depending on the machines and the objective of the study. The purpose of this report is not to make an evaluation of some computer systems, but to present a new methodology for performance evaluation and system characterization. It is for this reason that we run our system characterizer using the particular implementation of single and double precision on each machine. Table 4.1 gives for each machine the number of bits used in each of the data types.

4.2. Description of the System Characterizer's Parameters

Normally the characterization of computers at the architecture level is done using the instruction set of the machine. On the other hand, the decomposition and analysis of programs normally reflects the control structures and operators that a particular language supports [WEI84]. Because our representation of both computers and programs uses the same set of parameters, results may be combined.

To understand the set of parameters chosen in our system characterizer, we need to analyze the set of constructs that FORTRAN provides, and see how these features affect the execution time of a program. We will then create parameters for the different mechanisms that affect the execution and ignore those that are only used as aids either to the programmer or the compilers in the writing of correct and efficient programs. The constructs supported by imperative or statement-oriented languages, like FORTRAN, can be separated in four main categories: data type definition mechanisms, expressions,

[2] The default on the CRAY X-MP is 46 bits. There exists a compiler option that extends integers to 64 bits.

Table 4.1: **Characteristics of the machines (I)**

Machine	Name/Location	Operating System	Compiler version
CRAY X-MP/48	NASA Ames	COS 1.16	CFT 1.14
CYBER 205	NASA Ames	NOS	FTN200
IBM 3090/200	cmsa.berkeley.edu	VM/CMS r.4	FORTRAN v2
Amdahl 5840	ames-prandt.nasa	UTS V	F77
Convex C-1	convex.riacs.edu	UNIX C-1 v6	FC v2.2
VAX 8600	vangogh.berkeley.edu	UNIX 4.3 BSD	F77 v1.1
VAX-11/785	arpa.berkeley.edu	UNIX 4.3 BSD	F77 v1.1
VAX-11/780	ames-pioneer.nasa	Ultrix 2.0	F77 v1
SUN 3/50	orchid.berkeley.edu	UNIX 4.2 r.3.2	F77 v1
IBM RT-PC/125	jeff.berkeley.edu	AIX 4.3	F77 v1

Table 4.1: **Characteristics of the machines (II)**

Machine	Memory	Integer	Real	
		single	single	double
CRAY X-MP/48	8 Mwords	46	64	128
CYBER 205	8 Mwords	64	64	128
IBM 3090/200	32 Mbytes	32	32	64
Amdahl 5840	32 Mbytes	32	32	64
Convex C-1	100 Mbytes	32	32	64
VAX 8600	28 Mbytes	32	32	64
VAX-11/785	10 Mbytes	32	32	64
VAX-11/780	2 Mbytes	32	32	64
SUN 3/50	4 Mbytes	32	32	64
IBM RT-PC/125	4 Mbytes	32	32	64

Table 4.1: Characteristics of the machines. The size of the data type implementations are in number of bits.

statement-level and unit-level control structures, and simple statements.

4.2.1. Data Type Declarations

The data type declaration constructs in FORTRAN are used only as directives for the compiler, and the creation of data objects for the global (COMMON) and local variables is normally done before program execution. Therefore we do not need to create parameters for these statements. We will see in section 5.2 that the declarations of a FORTRAN program will also help the program analyzer in the decomposition of programs, in the same way as these statements help the compiler to generate correct machine code. The exception to this situation is the DATA statement, given that the initialization of the variables declared in the DATA must be done at each activation of a program unit. Normally the DATA statement is used for the initialization of global variables, and therefore its effect on the total execution time on scientific programs is small.

4.2.2. Expressions

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. Our system characterizer does not distinguished between the relational operators; all of these operators are grouped in the same class (.EQ., .LE., etc). This is because it takes the same time to compare two values independent of the relation. The same treatment is applied to the logical operators (.OR., .AND., and .NOT.). In contrast to the logical operators that can only take as arguments logical values, the arithmetic and relational operators are polymorphic. This means that, even when the semantic of the operation is different for different data types, the same name (symbol) is used. By looking at the arguments the compiler identifies the correct use of the operator and produces code accordingly. Because the execution time of a multiplication is different using integer arguments compared with real ones, we have to create a set of parameters that represent the execution using integer operands and another using real operands. In similar a way the execution time depends on the precision of the operands; it normally takes less time to execute an operation with single precision compared with double precision operands. Another classification is made with the storage class of the operands. Global variables in FORTRAN (variables defined in COMMONs) are sometimes treated differently from local variables. An example of this is the way the CYBER 205 deals with variables stored in COMMONs, when running without optimization. The compiler treats the COMMON as an array and allocates a base-descriptor pointing to the first element of the COMMON. An operand is loaded by first adding the offset (from the beginning of the COMMON block) 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.

The arithmetic operators defined in the system characterizer are: addition, multiplication, quotient, and exponentiation. The addition operator also includes subtraction. In the case of exponentiation with a real base, we distinguish two cases: one when the exponent is integer and the other when the exponent is real. This is because in each case the implementation is different. 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. When the exponent is real the result is computed using logarithms. If the base is an integer, we have two cases, one with the exponent equal to two and another with an exponent different from two. Because the number of exponentiations executed in most programs is small, these simplifications are enough for our purposes.

In tables 4.2 and 4.3 we present a description of the arithmetic parameters measured by the system characterizer. One table is for local operands and the other for variables allocated in COMMON blocks.

There are three different subsets of parameters in each table. The first subset is for single precision real variables; the second for double precision variables; and the last for integers. Two parameters require explanation. One is the set of parameters that measure the overhead of the store operation (SRSL, SRDL, SISL, SRSG, SRDG, and SISG), and the other, what we called memory transfer parameters (TRSL, TRDL, TISL, TRSG, TRDG, and TISG). In most high-level programming languages it is not possible to

Table 4.2: **Arithmetic operators with local operands**

Mnemonic	Operation	Data type	Precision	Storage class
SRSL	store	real	single	local
ARSL	addition	real	single	local
MRSL	multiplication	real	single	local
DRSL	division	real	single	local
ERSL	exp (X ** I)	real	single	local
XRSL	exp (X ** Y)	real	single	local
TRSL	memory transfer	real	single	local
SRDL	store	real	double	local
ARDL	addition	real	double	local
MRDL	multiplication	real	double	local
DRDL	division	real	double	local
ERDL	exp (X ** I)	real	double	local
XRDL	exp (X ** Y)	real	double	local
TRDL	memory transfer	real	double	local
SISL	store	integer	single	local
AISL	addition	integer	single	local
MISL	multiplication	integer	single	local
DISL	division	integer	single	local
EISL	exp (I ** 2)	integer	single	local
XISL	exp (I ** J)	integer	single	local
TISL	memory transfer	integer	single	local

Table 4.2: Arithmetic parameters with local operands.

execute a single load operation without executing at the same time another operation.

This is why we do not have a parameter that measures the time it takes to load a single operand. These times are included in the execution time of the operators. Another reason is that some compilers, even when optimization is turned off, load the variables in registers only once while evaluating an expression[3]. Even if we measure the time it takes to load an operand, we are left with the problem of deciding when the compiler will reload it or use the register that holds a copy of its value. On the other hand it is possible to run experiments that detect and measure the time it takes to store the result of the expression. In some cases the value of these parameters is negligible (the store operation overlaps with the execution of arithmetic operators), while on others the time can be significant. In an assignment where there are no operators on the right hand side of the equal sign, the execution time of these statements cannot be explained just by the store operation. This type of statements are characterized by the 'memory transfer' parameters.

In table 4.4 we give the set of parameters associated with compare and logical operations for local and global variables. As in the arithmetic case we distinguish the operands depending on their storage class, the data type and the precision. For the logical operations there is only one data type and one precision.

[3] The compiler does not attempt to eliminate redundant subexpressions; it only keeps a record of which variables were previously loaded. This information is not used in subsequent statements even when these are in the same basic block.

Table 4.3: **Arithmetic operators with global operands**				
Mnemonic	Operation	Data type	Precision	Storage class
SRSG	store	real	single	global
ARSG	addition	real	single	global
MRSG	multiplication	real	single	global
DRSG	division	real	single	global
ERSG	exp (X ** I)	real	single	global
XRSG	exp (X ** Y)	real	single	global
TRSG	memory transfer	real	single	global
SRDG	store	real	double	global
ARDG	addition	real	double	global
MRDG	multiplication	real	double	global
DRDG	division	real	double	global
ERDG	exp (X ** I)	real	double	global
XRDG	exp (X ** Y)	real	double	global
TRDG	memory transfer	real	double	global
SISG	store	integer	single	global
AISG	addition	integer	single	global
MISG	multiplication	integer	single	global
DISG	division	integer	single	global
EISG	exp (I ** 2)	integer	single	global
XISG	exp (I ** J)	integer	single	global
TISG	memory transfer	integer	single	global

Table 4.3: Arithmetic operations with global operands.

Table 4.4: **Conditional and logical parameters**				
Mnemonic	Operation	Data type	Precision	Storage class
ANDL	AND and OR	logical	single	local
CRSL	compare	real	single	local
CRDL	compare	real	double	local
CISL	compare	integer	single	local
ANDG	AND and OR	logical	single	global
CRSG	compare	real	single	global
CRDG	compare	real	double	global
CISG	compare	integer	single	global

Table 4.4: Conditional and logical parameters with local and global operands.

4.2.3. Statement-Level Control Structures

FORTRAN has eight different flow control statements that affect the execution of a program and only a few of them have an effect on the execution time. Here we will present each of the different types of statements and discuss their impact in the execution time of the programs when using these constructs. We will also indicate the parameters associated with these statements.

- **GO TO statements**: there are three different types of GO TO statements, the unconditional GO TO statement, the assigned GO TO statement, and the computed

GO TO statement. The unconditional GO TO is the most used of the three and also is the fastest to execute. In most machine this statement is implemented by a single machine instruction, but in pipeline architectures the cost of a pipeline stall can be significant if its target is not in the CPU prefetch buffer. We created one parameter (GOTO) to measure the mean execution time of an unconditional branch. We do not take into account the distance from the source of the branch to its target.

Branches 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[4]. All partially executed instructions in the pipeline must be discarded and the new stream of instructions must be fetched [LEE84]. A branch to an instruction that is not in the cache involves not only fetching the next instruction, but in addition the miss ratio is affected by changing the spatial locality of the execution [SMI82].

The computed GO TO statement is the equivalent of the **case** and **switch** statements in PASCAL and C respectively. The control of the transfer is the value of an integer expression. The implementation of this instruction uses a table and executes and indirect branch with the integer expression as an offset. This usually requires the execution of several machine instructions, like loading the value of the control variable from memory, selecting the branch displacement from the branch table, and branching to the new instruction. The execution time of this instruction is normally one order of magnitude greater than the unconditional branch. In fact for some machines the characterizer did not detect the execution time of an unconditional branch. We measure the execution time of this instruction with the parameter GCOM.

The assigned GO TO statement is an old and rarely used feature in FORTRAN; its purpose is to control the transfer to the value of an integer variable that was previously assigned the value of a label. In the system characterizer and the program analyzer this construct is treated in the same way as the computed GO TO.

- **DO loop statement**: this mechanism controls the repeated execution of a group of statements. The execution overhead associated with this statement may be significant in scientific applications running in scalar mode. Its implementation has two parts, the initialization of the control variable, limit and step, and the repetition control overhead. The first overhead is insignificant and in fact in the first versions of the characterizer there was no parameter associated with it. In programs where small loops (few iterations) are nested inside other loops, the initialization overhead may affect the total execution time. In the system characterizer we have four parameters that deal with DO statements due to implementation differences in most machines. In FORTRAN there is the possibility of omitting the step value of the header of the loop, and the compiler assigns one to the step by default[5]. The code produced by most compilers when the step is one is different than the code when the step is not one. In figures 4.2 and 4.3 we see the code produced by the CYBER 205 FTN200 FORTRAN compiler for these two cases.

[4] Even in machines that have some kind of branch prediction circuitry, a penalty must be paid when the prediction is incorrect.

[5] D. Knuth reports that 80% of the DO loops have a step value of 1 [KNU71].

```
                    DO 1 I = J, K
                        ...
                  1 CONTINUE


    RTOR #1,  C_4                   ; load #1 into register C_4 (step)
    RTOR K, C_5                     ; load K into register C_5 (limit)
    RTOR J,  I                      ; load J into variable I (control variable)
    IBXLE,BRF  C_5,, L2             ; is upper limit less than lower limit?
L1      ...
        ...                         ; body of loop
        ...
    IBXLE,BRB  I, C_4, L1, C_5, I   ; increment I by C_4, compare
L2                                  ; with C_5, branch to label L1
                                    ; if less equal, and store value in I
```

Figure 4.2: Assembler code of DO loop with step equal one

```
                    DO 1 I = J, K, L
                        ...
                  1 CONTINUE


    SUBX K, J, PR_3                 ; the statement DO 1 I = J, K, L is
    ADDX L, PR_3, PR_4              ; transformed to the equivalent statement
    DIVU PR_4, I, PR_5              ; DO 1 C_5 = 0, C_4 - 1, 1
    TRU  PR_5, PR_6                 ; where C_4 = ⌊(K - J + L) / L⌋
    RTOR PR_6, C_4
    RTOR #0, C_5                    ; load #0 into register C_5
    RTOR L, C_6                     ; load variable L into register C_6
    RTOR J, I                       ; load J into variable I
    IBXLE, BRF C_4,, L2             ; is upper limit less than lower limit?
L1      ...
        ...                         ; body of loop
        ...
    ADDX I, C_6, I                  ; increment I by register C_6
    IBXLE,BRB  C_5, #1, L1, C_4, C_5  ; increment, test, and branch inst
L2
```

Figure 4.3: Assembler code of DO loop with step different from one

As the two figures show, the initialization and iteration overhead are different, and in the second case can significantly affect the execution of the program when we have various DO loops nested. In the system characterizer there exist two parameters for each type of DO loop. For loops with step equal one the initialization and repetition overhead are called LOIN, and LOOV. In the other case the parameters are LOIX and LOOX. Making a good measurement of the overheads incurred by the DO loop statement can be difficult.

- **IF statements**: there are three different type of IF statements in FORTRAN, the block IF, the logical IF, and the arithmetic IF. The block IF statement is a new feature incorporated in the FORTRAN 77 standard and represents the **if-then-else** mechanisms of ALGOL-like languages. In fact the logical IF is a special case of the block IF, when no **else** clauses are used and the **then** part of the **if** contains only one executable statement. If we analyze the effect of a block IF statement in the execution time, we will notice that the only overhead incurred by this statement is the same as the one produced by a conditional branch instruction. We can see this by looking at the next two examples

  ```
  L = I .EQ. J .OR. I .NE. K
  ```

  where I, J, and K are integers and L is a logical variable, and

  ```
  IF (I .EQ. J .OR. I .NE. K)  GO TO 1
  ```

  The machine code produce by the compiler for the right hand side of the assignment and for the expression inside the parenthesis in the IF statement is the same, even for compilers that short-circuit expressions. The same situation occurs in the case of the **else** and **elseif** statements. The arithmetic IF is handled in a similar way. This statement has two parts; one is the evaluation of an arithmetic expression, and the other is a jump to one of three possible targets. The arithmetic expression is analyzed as any other expression and the branch is replaced by a computed GOTO statement.

- **CONTINUE statement**: this construct is not an executable statement and its occurrence in the source program should affect the execution time of an application.

- **CALL and RETURN statements**: the first statement transfers control from one unit to another and the second statement returns the control to the original caller. Also included in the CALL statement are the set of parameters that are passed from one subprogram to another. The overhead incurred by the execution of the CALL statement is considerable and can be divided in three parts: the overhead incurred in the passing of arguments, the prologue overhead and the epilogue overhead. This last part is the code executed by the RETURN statement. The amount of work that has to be done depends on the number and type of the arguments. In FORTRAN all the arguments are passed by reference including values computed by expressions. The characterizer has three parameters, these are: ARGS, ARGD, ARGI. These measure the time to load the corresponding pointer to single precision, double precision and integer variables either into the static environment of the callee subprogram or in the execution stack. Although FORTRAN uses static allocation, many machines use the execution stack to pass parameters and results between subprogram units. The addition of the prologue and epilogue execution time associated with the invocation of a unit program is characterized by the parameter PROC, even when each of them is executed in different subprogram units and at different moments.

- **STOP and PAUSE statement**: these instructions do not generate significant overhead, and therefore there are no parameters for these statements in the system characterizer.

Table 4.5: **Execution control and array access parameters**

Mnemonic	Operation	Data type	Precision
PROC	procedure call	na	na
AGRS	argument load	real	single
AGRD	argument load	real	double
AGIS	argument load	integer	single
ARR1	array with 1 dimension	na	na
ARR2	array with 2 dimensions	na	na
ARR3	array with 3 dimensions	na	na
ARR4	array with 4 dimensions	na	na
ARRN	array with $\geq$ 5 dimensions	na	na
IADD	addition in array index	integer	single
GOTO	simple goto	na	na
GCOM	computed goto	na	na
LOIN	do loop initialization step 1	na	na
LOOV	do loop overhead step 1	na	na
LOIX	do loop initialization step n	na	na
LOOX	do loop overhead step n	na	na

Table 4.5: Execution control and array access parameters.

4.2.4. Additional Parameters

In addition to the parameters presented in the last subsections, there are also other parameters that, although they cannot be associated to any particular statement or operation, have a significant execution time and should therefore be included in our model. The first subset deals with the overhead associated with the access of a value stored in an array. If the variable referenced by the program is stored in an n-dimensional array and the value of the indices that determine the particular element are not known at compile time, the compiler must generate code to compute the actual address at execution time. There are three parameters that measure this overhead: ARR1, ARR2, ARR3. Each measures the additional time it takes to access a variable in an array of one, two and three dimensions. The overhead for variables in four and five dimensions (ARR4, ARRN) is computed using a linear combination of the three basic parameters. We do not consider a more detailed characterization of array references, because, in our benchmarks, they were very few arrays with more than three dimensions and no examples of more than five.

Intrinsic functions form the last subset of parameters. Although the number of times these instructions are executed in a program is small, their execution time is normally very large compared with that of a single arithmetic operation. These parameters are shown in table 4.6 for single and double precision real arguments. 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 on how large is the argument, but also on 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 CRAY X-MP library reference manual [CRA84] contains a table with execution times for most of the intrinsic functions. In most cases the difference between the maximum and the minimum time is less than 20%[6]. However for programs with a large number of calls to these functions, a better characterization may be needed to obtain acceptable predictions.

Table 4.6: **Parameters for intrinsic functions**

Mnemonic	Operation	Data type	Precision	Storage class
EXPS	exponential	real	single	local
LOGS	logarithm	real	single	local
SINS	sine	real	single	local
TANS	tangent	real	single	local
SQRS	square root	real	single	local
EXPD	exponential	real	double	local
LOGD	logarithm	real	double	local
SIND	sine	real	double	local
TAND	tangent	real	double	local
SQRD	square root	real	double	local

Table 4.6: Parameters for intrinsic functions.

4.3. Experiment Design

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 produces an insignificant variance in the results. On the other hand, the measurement of the parameters of a system characterizer using a high level program is not easy due to a number of factors:

- The short execution time of most operations (20 nsec - 10 μsec)
- 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

The parameters in the system characterizer are composed of single or a small number of machine instructions; for this reason, the events we want to characterize have a duration of ten to thousands of nanoseconds. To achieve a meaningful measurement of these events using a high-level program and with the resolution of most system clocks

[6] The maximum difference reported is 100% for the arc cosine.

requires clever tests, especially when the characterizer is used in different machines, each with different machine instruction sets and architectures.

To isolate an operation for measurement normally requires robust tests to avoid optimizations[7] from the compiler that would eliminate the operation from the test and distort the results [CLA86]. 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 statement to control the execution of the test; initializing variables in external procedures to avoid constant folding. Separate compilation of variable initialization procedures, 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.

4.4. Test Structure and Measurement

The events that we want to measure and characterize have very small execution times. For this reason it is not possible to make a direct measurement of a single execution in most systems. The clock resolution in many machines is bigger than the execution time of a single operation. In machines with the UNIX operating system, the clock resolution is almost always 1/60'th of a second. This value is several orders of magnitude greater than the time it takes to execute almost any operation. In addition the overhead incurred by executing the clock routine affects our measurements. One way of reducing these factors is to repeat the test some number of times to obtain a measurement that is much greater than the errors produced by the clock resolution and the overhead of the timing routine combined. There are problems associated with this technique. In a machine with cache memory the value obtained for the execution time of a single operation using repetition is smaller than the execution time of a single operation when the arguments are not previously in the cache. The results obtained in this way will indicate that the system is faster than in the case when the arguments are not in the cache. Nevertheless there are at least two arguments that support using repetition. The first one is that the very idea of using cache memories in computer systems is because programs tend to satisfy the principle of locality. The second reason is that we expect that the error incurred by using repetition will be small compared with the experimental error, especially if we take into account that the cache hit ratio of typical applications is high.

Figure 4.4 shows the structure of the tests in our system characterizer. We can identify five parts in each experiment. The *initialization*, in which the number of iterations of the body of the test is computed. For each test we have to make sure that it will execute for a minimum amount of time in fast machines, but not for too long in slow systems. To control this we have three parameters, the SPEEDUP factor (> 0), that gives a crude approximation of the relative performance of the system compared to the CRAY X-MP/48, the number of iterations (LIMITO) it takes the CRAY X-MP/48 to execute the test for one second, and the duration of the test (TMAX). This last parameter will permit us to control the execution as a function of the resolution of the clock and the variance of the measurements. The *test* is the code included in the two lines with ellipsis, and is the instruction or group of instructions that we want to characterize. The additional code

[7] 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.

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

Figure 4.4: The basic structure of an experiment.

delimited by the two invocations to the SECOND function (timing function) is what we called an *observation*. Here we control the number of times the body of the test is executed, so we can obtain a meaningful observation. The additional code delimited by the DO loop represents the *experiment*. This consists of a set of observations (controlled by the variable REPEAT). The last part is the computation of the *measurement*. In this part we compute the mean value of our observations and the variance. To control the error in our measurements we have two possibilities; one is to run each test for a significant amount of time; the other is to increase the number of observations inside the experiment. In the first case we increase the number of iterations that the body of the test is executed (increasing the value of LIMIT0 and LIMIT in figure 4.4). In the second case we execute the body of the test the same number of times, but increase the size of the sample statistic (increasing the value of variable REPEAT). In section 4.5.1 we discuss the effect of each one of these possibilities.

4.4.1. Direct Tests, Composite Tests and Indirect Tests

To understand the possible sources of experimental error, and how to compute them, we need the concepts of a 'direct test', 'composite test' and 'indirect test'. As we explained in the last paragraph, inside of the 'if-loop' construct we have the test. Now in a ***direct test*** the body of the test consists of N occurrences of the operation we want to characterize and nothing more. In a ***composite test*** in addition to the N operations there are several other operations of different type inside the body. This is necessary because in most of the cases it is not possible to make a direct measurement of the parameters, and we have to include some additional operations. In an ***indirect test*** the execution time for the operation we are measuring (P_i) is obtained by running two different tests. Some parameters of the system characterizer are coupled; it is not possible to execute one without executing the other, and therefore the way to isolate one of the parameters is to run two tests with different number of operations for each of the parameters. The body of the second test is the same as the body of the first test, plus some additional work. The difference in the execution time between the two tests gives us the value of one of the coupled parameters. An example of this is the DO loop initialization and overhead. Every time we have a DO loop in a FORTRAN program, the compiler generates code that includes the initialization of the loop and also the overhead to control the iteration. By changing the number of times the loop is executed, in two tests, we can obtain a pair of linear equations to compute the values of the initialization and the overhead. In the next subsection we will see that the variance of our measurements depends on whether the observations are done using direct, composite, or indirect tests.

4.5. Experimental Errors and Confidence Intervals

As we pointed out in the last section, one of the important parts of the characterizer is to control the accuracy and exactitude of our measurements. In order to make an evaluation of the quality of our measurements, we need to quantify the sources of error in our experiments. Currah gives a long list of the causes in the variability in CPU time as measured by the system clock [CUR75, MER83]. Some of these factors are: (a) Timer resolution of CPU clock. (b) Improper allocation of CPU time for I/O interrupt handling. (c) Changes in cache hits due to interference with concurrent tasks. (d) Cycle stealing while another component is sharing a resource with the CPU. (e) Number of context switches; the time spend by the dispatcher and timer routine before dequeuing or after enqueing a process. Some of these events have a length of time far greater than the phenomena that we are measuring.

We also have to subtract the execution time of the code that controls our test and the overhead incurred by the timer routine. These measurements have their own variance and the subtraction of these overheads increases the variance of our measurements. All the factors combined can be significant compared to the magnitude of our results. We will now proceed to quantify the sources of variability and obtain expressions for the variance for the different types of experiments.

We denote the factors affecting our measurements as follows:

Table 4.7: **Definitions of terms used in the time 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
$\hat{O}$	::=	sample mean of each observation (measurement)
$\hat{B}$	::=	sample mean execution time of the test
$\hat{P}_i$	::=	sample mean of parameter i
σ^2	::=	variance operator

We know that each observation O_j is equal to

$$O_j = T_{j_1} - T_{j_0} \tag{4.1}$$

then the mean value ($\hat{O}$) of these observations is

$$\hat{O} = \frac{1}{N_{repeat}} \sum_{j=1}^{N_{repeat}} O_j \tag{4.2}$$

and its variance

$$\sigma^2 O = \frac{1}{N_{repeat} - 1} \sum_{j=1}^{N_{repeat}} (O_j - \hat{O})^2 \tag{4.3}$$

Now the mean value of each experiment is equal to the time it takes to execute the body of the test N_{limit} times, plus the overhead of the timing function

$$\hat{O} = N_{limit} \left(\hat{B} + IF_{overhead}\right) + C_{overhead} \tag{4.4}$$

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

$$\hat{B} = \frac{\hat{O} - C_{overhead}}{N_{limit}} - IF_{overhead} \tag{4.5}$$

and

$$\sigma^2 B = \frac{\sigma^2 O + \sigma^2 C_{overhead}}{{N_{limit}}^2} + \sigma^2 IF_{overhead} \tag{4.6}$$

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

$$\hat{P}_i = \frac{\hat{B}}{N}; \qquad \sigma^2 P_i = \frac{\sigma^2 B}{N^2} \tag{4.7}$$

In a composite and indirect test we have

$$\hat{P}_i = \frac{\hat{B} - \hat{W}_{extra}}{N}; \quad \sigma^2 P_i = \frac{\sigma^2 B + \sigma^2 W_{extra}}{N^2} \tag{4.8}$$

where W_{extra} is the additional work inside the body of the test or in the second test.

Looking at the above equations we can see that there are four factors affecting the magnitude of the variance in a direct test and five for composite and indirect tests. These factors are: the resolution of the timing function; the variance of our observations; the variance of the execution time of the timing function; the variance of the IF control statements; and the variance of the additional work executed inside the body of the test or by the second test. If the execution time of each observation is such that we have

$$O_j = T_{j_1} - T_{j_0} \gg C_{resolution} + C_{overhead} + IF_{overhead} \tag{4.9}$$

then the only factors that affect our measurements are the dispersion of our observations, affected by concurrent activity on the system, and the variance in the execution time of the extra work present in the composite and indirect tests.

Table 4.8 gives the experimental values for $C_{resolution}$, $C_{overhead}$, $IF_{overhead}$, and the minimum duration of one observation ($T_{\min}$) such that the magnitude of the right hand side of equation 4.9 is less than five percent the magnitude of O_j in a direct test.

Table 4.8: **Sources of Experimental Error**

System	$C_{resolution}$	$C_{overhead}$	$IF_{overhead}$	$T_{\min}$
CRAY X-MP/48	1.0 μs	2.3 μs	0.47 μs	112 μs
CYBER 205	1.0 μs	2.6 μs	0.64 μs	124 μs
Amdahl 5840	1.0 μs	475. μs	0.95 μs	10 ms
IBM 3090/200	10.0 ms	376. μs	0.27 μs	200 ms
Convex C-1	10.0 ms	276. μs	1.04 μs	205 ms
VAX 8600	16.$\bar{6}$ ms	175. μs	1.31 μs	338 ms
VAX-11/785	16.$\bar{6}$ ms	585. μs	4.42 μs	346 ms
VAX-11/780	16.$\bar{6}$ ms	825. μs	5.86 μs	351 ms
Sun 3/50	20.0 ms	713. μs	1.49 μs	414 ms
IBM RT-PC/125	16.$\bar{6}$ ms	507. μs	2.79 μs	344 ms

Table 4.8: Sources of experimental error. $T_{\min}$ gives the minimum time that a test must be run to reduce the error due to the resolution and call overhead of the clock, and overhead of the test to less than 5 percent in a direct test.

4.5.1. Reducing the Variance

We have two ways of reducing the variance of our results and therefore the size of the confidence intervals. The first is by increasing the length of the test by augmenting the value of N_{limit}. But the problem is that by doing this the probability of a context switch increases and also the possibility of a cache flush that will be reflected in higher cache misses. The second possibility is to increase the number of observations in each experiment (N_{repeat}). Because each of our observations is an independent and identically distributed random variable we can apply classic statistics and therefore the confidence intervals for our measurements will be reduced by the square root of the number of observations made in each experiment. On the other hand by increasing the number of observations, the probability that an event in the system occurs increases (e.g. swapping, the update of the superblock in UNIX every 30 seconds, etc) and this will increase the variance.

In some measurements using indirect tests, the variance obtained can be significant compared to the actual measure. We can see this by considering the following case. To measure the overhead and initialization of the DO loop statement we run three experiments. In the first case the test consists of a DO loop with some extra statements inside the loop (to prevent elimination by the compiler) that is executed N times. In the second test the loop is executed $2N$ times. For the third test the loop executes N times, but the extra work inside the loop is twice as much as in the other tests. We can express the above conditions in terms of the mean execution time of the body of each test (B_i).

$$\hat{B}_1 = DO_{initialization} + N\,(DO_{overhead} + W_{extra})$$

$$\hat{B}_2 = DO_{initialization} + 2 \cdot N\,(DO_{overhead} + W_{extra})$$
$$\hat{B}_3 = DO_{initialization} + N\,(DO_{overhead} + 2 \cdot W_{extra}) \tag{4.10}$$

it is easy to see that we can obtain values for $DO_{initialization}$ and $DO_{overhead}$ and their variance in terms of the B_is.

$$DO_{initialization} = 2 \cdot \hat{B}_1 - \hat{B}_2\,; \qquad \sigma^2 DO_{initialization} = 4 \cdot \sigma^2 B_1 + \sigma^2 B_2 \tag{4.11}$$

and

$$DO_{overhead} = \frac{\hat{B}_2 - \hat{B}_3}{N}\,; \qquad \sigma^2 DO_{overhead} = \frac{\sigma^2 B_2 + \sigma^2 B_3}{N^2} \tag{4.12}$$

In table 4.9 we give results obtained on a VAX-11/785 for a sample statistic of size five and each of length of one second. We can see that even when the sample standard deviation is small ($< 5\%$) for the B_is, in the case of the DO loop parameters the standard deviation is very large.

Table 4.9: **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
$DO_{initialization}$	12.5 μs	9.11 μs	72.9
$DO_{overhead}$	1.99 μs	0.73 μs	36.8

Table 4.9: Mean and standard deviation. Relative magnitude of the standard deviation compared to the sample mean for $DO_{initialization}$ and $DO_{overhead}$. Each test consists of 5 observations executed for 1 second on a VAX-11/785.

It is therefore important to know what are the values for N_{limit} and N_{repeat} that will give a small standard deviation in our measurements. These values are system dependent and are affected by the resolution of the clock, the concurrent activity on the system, etc. In figure 4.5 we show 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 obtain measurements for N_{repeat} equal to 5, 10 and 20 observations. The confidence intervals for P_i are obtained using the Student's t distribution and the standard error of $\hat{P}_i$ as follows

$$\left[\hat{P}_i - t_{.95} \left[\frac{\sigma^2 P_i}{N_{repeat}} \right]^{1/2},\ \hat{P}_i + t_{.95} \left[\frac{\sigma^2 P_i}{N_{repeat}} \right]^{1/2} \right] \tag{4.13}$$

and the normalized confidence intervals are

$$\left[-\frac{t_{.95}}{\hat{P}_i} \left[\frac{\sigma^2 P_i}{N_{repeat}} \right]^{1/2},\ \frac{t_{.95}}{\hat{P}_i} \left[\frac{\sigma^2 P_i}{N_{repeat}} \right]^{1/2} \right] \tag{4.14}$$

We can see that for a fixed value of N_{repeat} 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 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 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 duration of the test. In machines with good clock resolution acceptable results are obtained for 10 observations and .2 seconds for each test.

In our system characterizer each test executes for at least 2 seconds on a CRAY X-MP. A potential problem with this is that a test that runs for 2 seconds on a CRAY X-MP usually takes much longer on most systems. A system characterizer constructed in this way will have an excessive execution time, and therefore will be unsuitable for benchmarking. To avoid this problem we calibrated each test to execute for 2.2 seconds in the CRAY X-MP/48 and adjusted each particular test according to a 'speed-up' factor that approximates the ratio of performance between the CRAY X-MP/48 and the system we are characterizing. Even with this approximation the execution time of each test will not be equal to 2.2. If the actual running time is greater than 2.2 seconds we keep the measurement, because this value reduces the experimental error even more. In the other case, if the time is less than 2 seconds, the system characterizer computes a new approximation and runs the test again. The gap between 2 and 2.2 reduces the possibility of unnecessarily repeating the test. At the beginning of the execution of the system characterizer, the system runs four tests that measure the clock resolution, the clock routine overhead, the test control overhead, and the speed-up factor. With these four quantities the system characterizer computes the execution times needed in each test to run for 2 seconds.

4.6. Is the Minimum Better than the Average?

In the previous section we mentioned that to obtain the expected execution time per parameter we have to compute the average of a number of observations. The 'noise' in our measurements is the result of concurrent activity in the system and the resolution of our measuring tools. In most systems an increase in the execution load produces an increase in the real and CPU time of programs. The time we measure for the execution of a basic operation is always greater or equal to the 'real' execution time when there is no other activity. Therefore it should be better to take the minimum instead of the average, given that the minimum is always less or equal than the average. By doing this we reduce the discrepancy between our measurement and the 'real' execution time.

The main objective of this research is to characterize the actual performance of systems, and when these systems are used in everyday situations there is always some degree of concurrency present. We expect that if we filter the extra time due to this concurrency our predictions will tend to be less than the actual running time of programs. However the only way that we can be sure that this is the case is to characterize some systems using each of these techniques and see which of them produces better estimates.

We ran the system characterizer twice in the Convex C-1 taking first the average and then the minimum, and used these results to estimate the execution time of a workload composed of ten program. The difference in the value of the parameters was

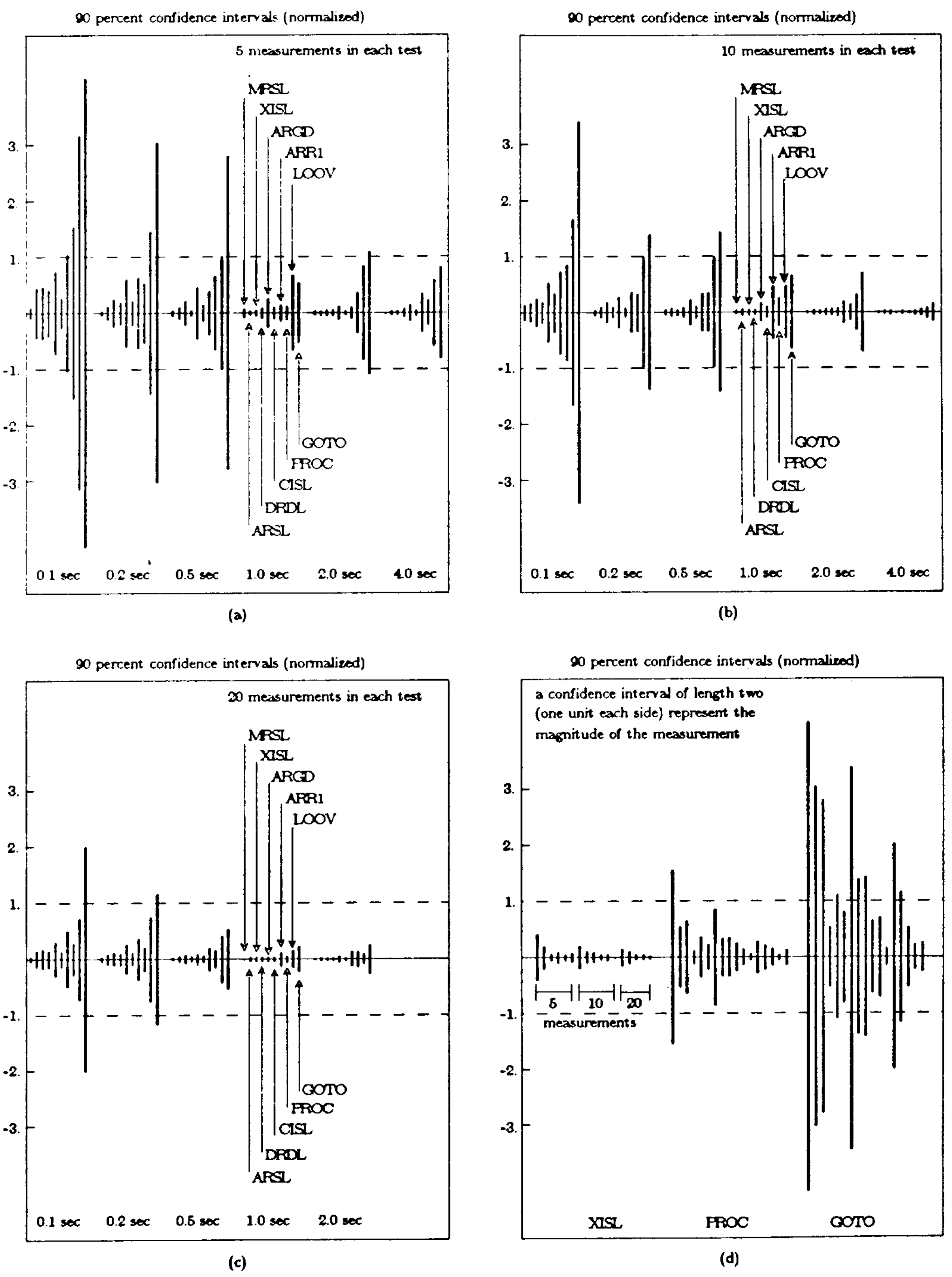


Figure 4.5: Confidence intervals for ten different parameters. In (a), (b), and (c) we show how the length of the test and the number of observations affect the confidence interval of the measurements. For a fixed number of observations an increase in the execution time of the test tends to reduce the length of the confidence interval. Figure (d) shows all the confidence intervals for three of the ten parameters. All confidence intervals are normalized with respect to parameter P_i.

between 2-15%. Taking the average of the measurements produced the smallest error in the total execution time of the workload as we can see in table 4.10.

Table 4.10: **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 %

Table 4.10: Estimates taking the average and the minimum. The execution time of a workload of ten programs was predicted with the set of parameters obtained using the average and the minimum of the measurements.

4.7. Results Obtained with the System Characterizer

We executed the system characterizer in the ten machines shown in table 4.1, and in figures 4.6-4.9 we present the experimental values that we obtained. In the appendix (section 9) we show the results in tabular form. Table 4.11 explains the meaning of each of the twelve regions in which the set of parameters have been grouped. The number of each region is printed at the top of the first graph in each of the three figures. Each point in the graphs represents the execution time for a single operation in nanoseconds. Some parameters have value zero, as for example, the execution time for the GOTO operation in the Convex C-1, or the addition of a constant to an index, if the index is a component of an array. This happens when the execution time of a parameter is small and its execution overlaps with other operations; the total execution time does not depend in the occurrence of the parameter.

In figures 4.10-4.11 we show the results for all the machines normalized with respect to the VAX-11/780. There are several interesting patterns in these figures that give information about the characteristics of the machines. First, we can see that the execution times for operations accessing local variables are similar to the times obtained for global variables in all the cases except for the CYBER 205. In this machine the operations using global variables take longer time to execute, and can take as much as ten times longer as in the case of the integer add operation and the AND operation. We corroborate this observation by looking at the same parameters compared with the ones obtained for the CRAY X-MP. We can also see that to execute floating point operations with single precision arguments, the CRAY X-MP has better times than the CYBER 205, and the IBM 3090. But if we look at the same operations with double precision operands, we find that the times for the CRAY X-MP are greater than the ones obtained for the CYBER 205, IBM 3090, and Amdahl 5840. Moreover, in the case of addition, multiplication and division, the Convex C-1 and the VAX 8600 have smaller times than the CRAY X-MP. We can see from the graphs and tables that the high performance of the CRAY X-MP when running in scalar mode and without optimizations lies in the fast execution of the arithmetic floating point operations with single precision.

It is important to point out that double precision on the Convex, the various VAX machines and the Sun is 64 bits against 128 bits on the other machines. The purpose of this research is to present a new methodology of performance characterization and not to compare different machines. A serious evaluation of their performance must address the problem of data type representation carefully in order to make a fair comparison.

Table 4.11: **Parameter regions in figures 4.6-4.11**

Region	Set of Parameters		Region	Set of Parameters	
1	real operations (single), local operands		4	real operations (single), global operands	
	01 SRSL	store		22 SRSG	store
	02 ARSL	addition		23 ARSG	addition
	03 MRSL	multiplication		24 MRSG	multiplication
	04 DRSL	division		25 DRSG	division
	05 ERSL	exp (X ** I)		26 ERSG	exp (X ** I)
	06 XRSL	exp (X ** Y)		27 XRSG	exp (X ** Y)
	07 TRSL	memory transfer		28 TRSG	memory transfer
2	real operations (double), local operands		5	real operations (double), global operands	
	08 SRDL	store		29 SRDG	store
	09 ARDL	addition		30 ARDG	addition
	10 MRDL	multiplication		31 MRDG	multiplication
	11 DRDL	division		32 DRDG	division
	12 ERDL	exp (X ** I)		33 ERDG	exp (X ** I)
	13 XRDL	exp (X ** Y)		34 XRDG	exp (X ** Y)
	14 TRDL	memory transfer		35 TRDG	memory transfer
3	integer operations, local operands		6	integer operations, global operands	
	15 SISL	store		36 SISG	store
	16 AISL	addition		37 AISG	addition
	17 MISL	multiplication		38 MISG	multiplication
	18 DISL	division		39 DISG	division
	19 EISL	exp (I ** 2)		40 EISG	exp (I ** 2)
	20 XISL	exp (I ** J)		41 XISG	exp (I ** J)
	21 TISL	memory transfer		42 TISG	memory transfer
7a	logical operations with local operands		7b	logical operations with local operands	
	43 ANDL	AND & OR		47 ANDG	AND & OR
	44 CRSL	compare, real, single		48 CRSG	compare, real, single
	45 CRDL	compare, real, double		49 CRDG	compare, real, double
	46 CISL	compare, integer, single		50 CISG	compare, integer, single
8	function call and arguments		9	References to array elements	
	51 PROC	procedure call		55 ARR1	array 1 dimension
	52 AGRS	argument load, real, single		56 ARR2	array 2 dimensions
	53 AGRD	argument load, real, double		57 ARR3	array 3 dimensions
	54 AGIS	argument load, integer, single		58 IADD	array index addition
10a	branching parameters		10b	DO loop parameters	
	59 GOTO	simple goto		61 LOIN	loop initialization (step 1)
	60 GCOM	computed goto		62 LOOV	loop overhead (step 1)
				63 LOIX	loop initialization (step n)
				64 LOOX	loop initialization (step n)
11	intrinsic functions (single precision)		12	intrinsic functions (double precision)	
	65 EXPS	exponential		70 EXPD	exponential
	66 LOGS	logarithm		71 LOGD	logarithm
	67 SINS	sine		72 SIND	sine
	68 TANS	tangent		73 TAND	tangent
	69 SQRS	square root		74 SQRD	square root

Table 4.11: The regions in the graphs represent different aspects of the characterization of the machines. Parameters ARR4 and ARRN are not included, because these are not measured directly by the system characterizer.

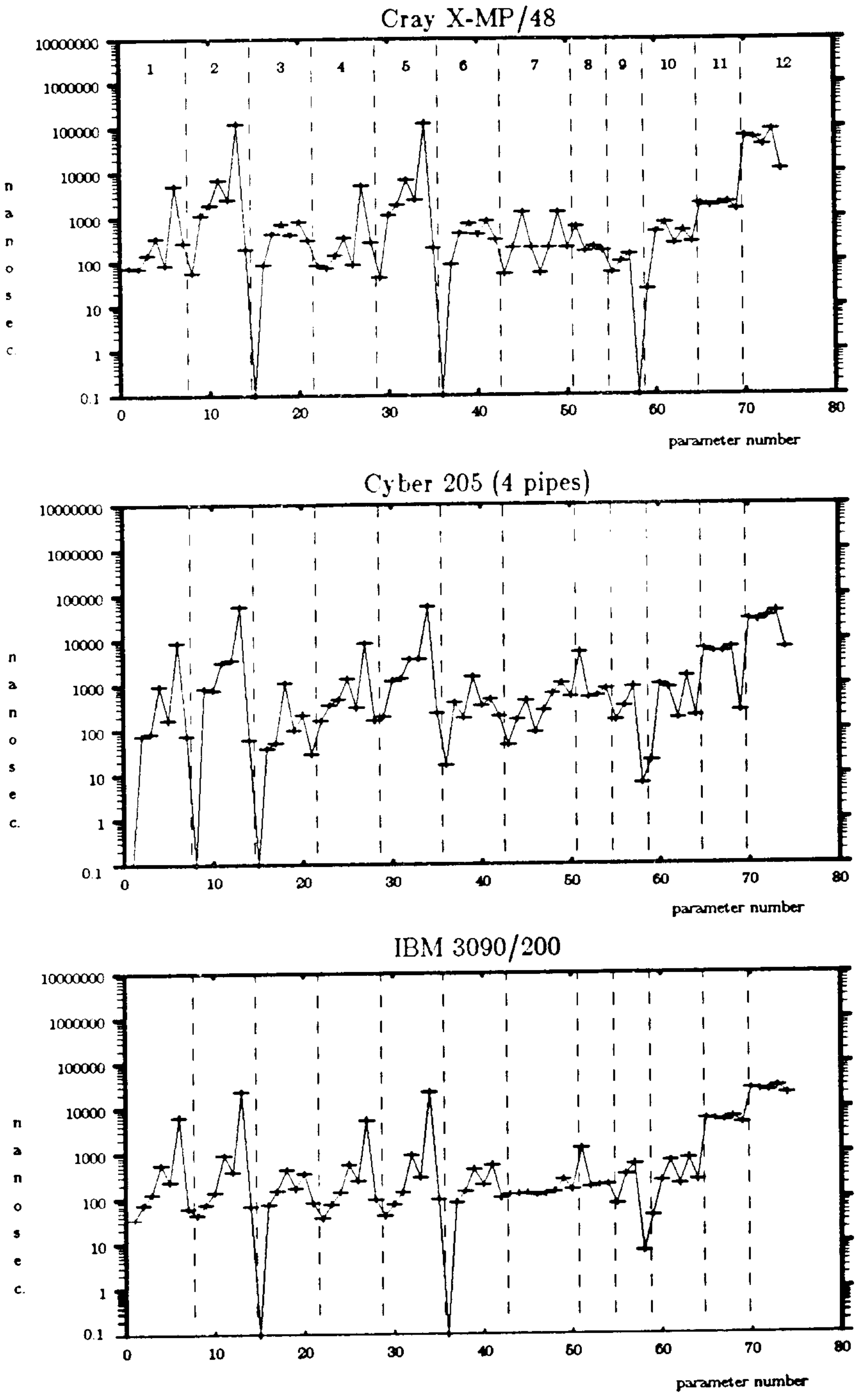
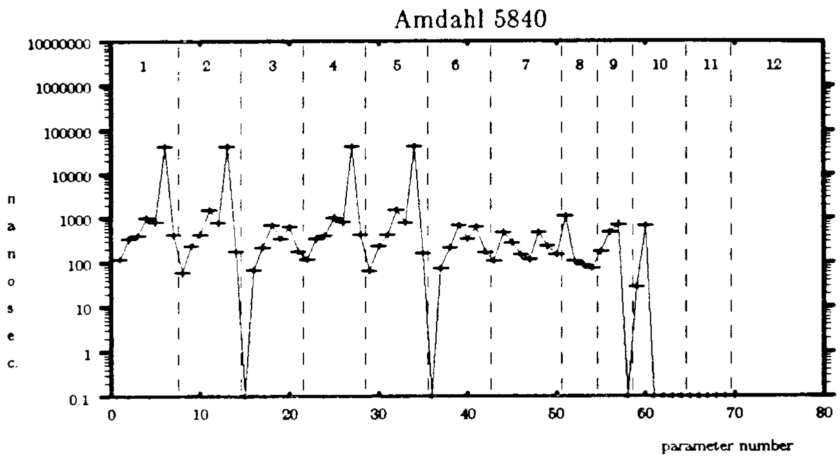
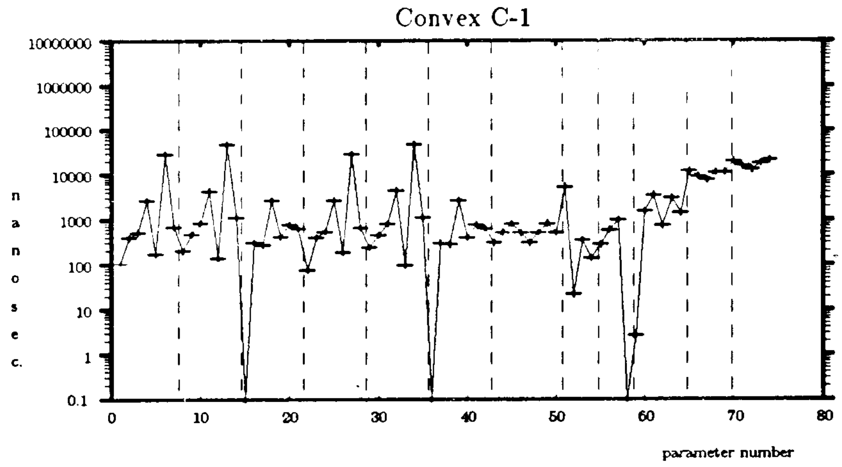


Figure 4.8: Characterization results (Cray, Cyber, and IBM 3090). The graphs show the value of each parameter in nanoseconds. The twelve regions represent different aspects of the characterization.





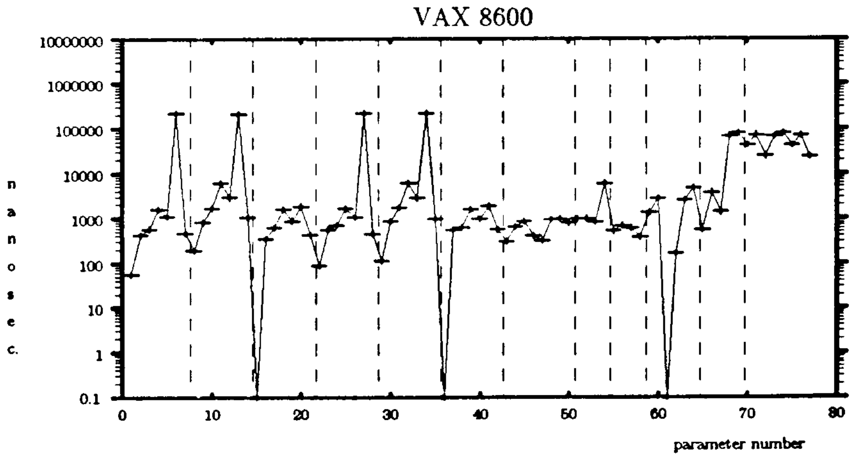


Figure 4.7: Characterization results (Amdahl, Convex, and Vax 8600). The graphs show the value of each parameter in nanoseconds. The twelve regions represent different aspects of the characterization.

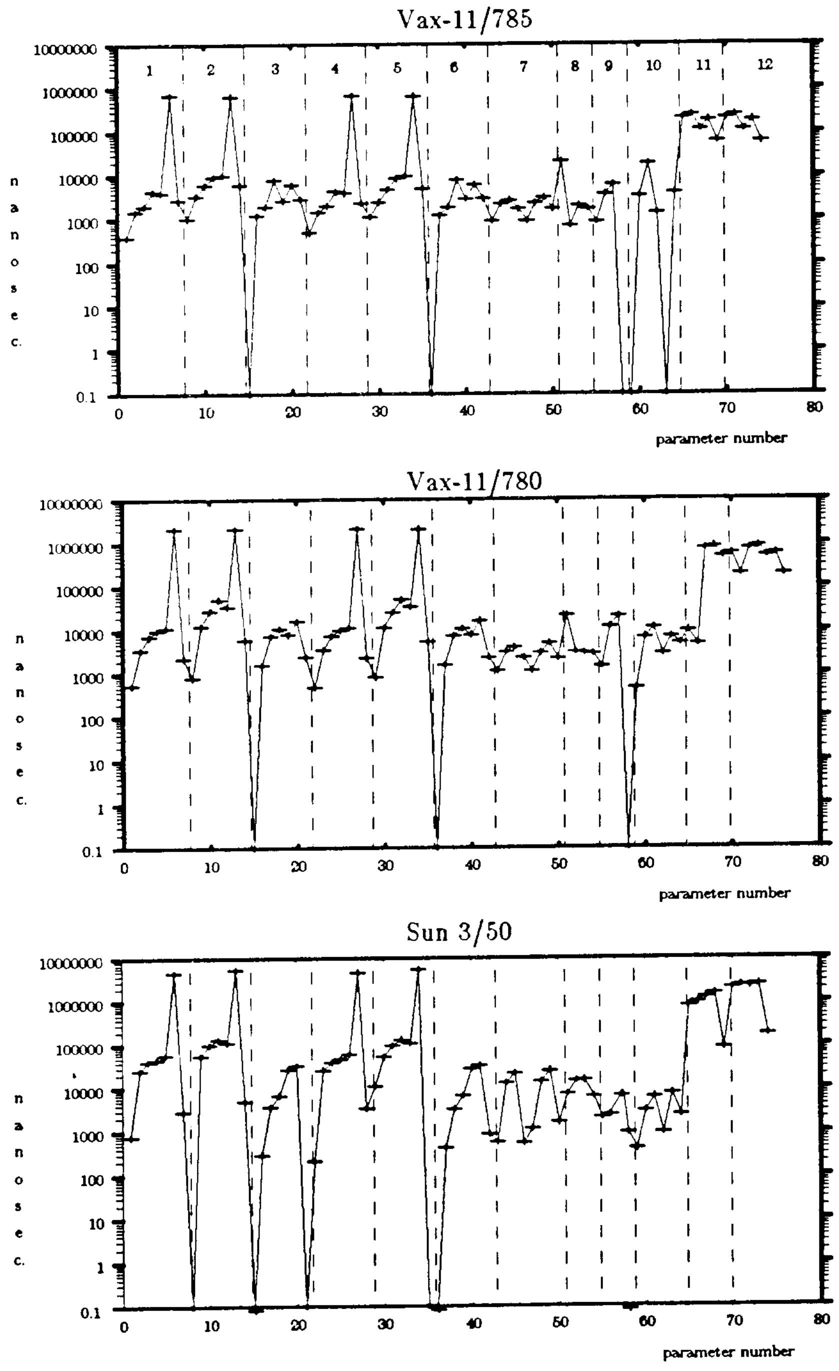


Figure 4.8: Characterization results (Vax 785, Vax 780, and Sun 3/50). The graphs show the value of each parameter in nanoseconds. The twelve regions represent different aspects of the characterization.

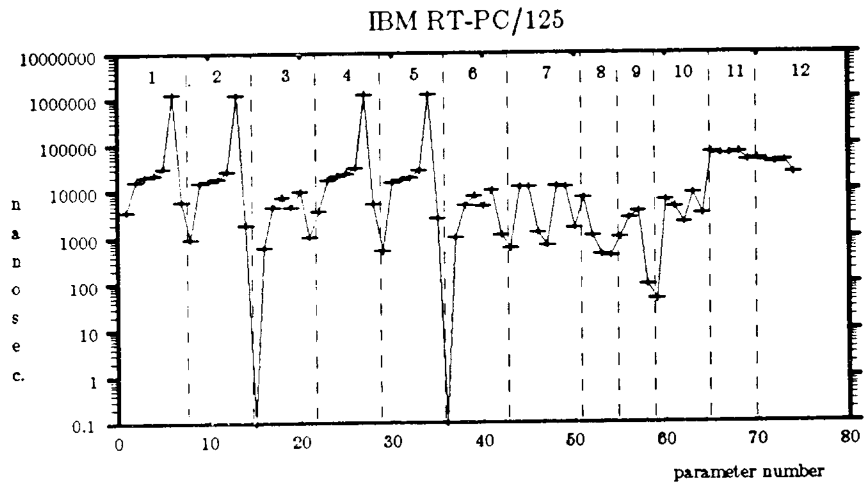


Figure 4.9: Characterization results (IBM RT-PC/125). The graph show the value of each parameter in nanoseconds. The twelve regions represent different aspects of the characterization.

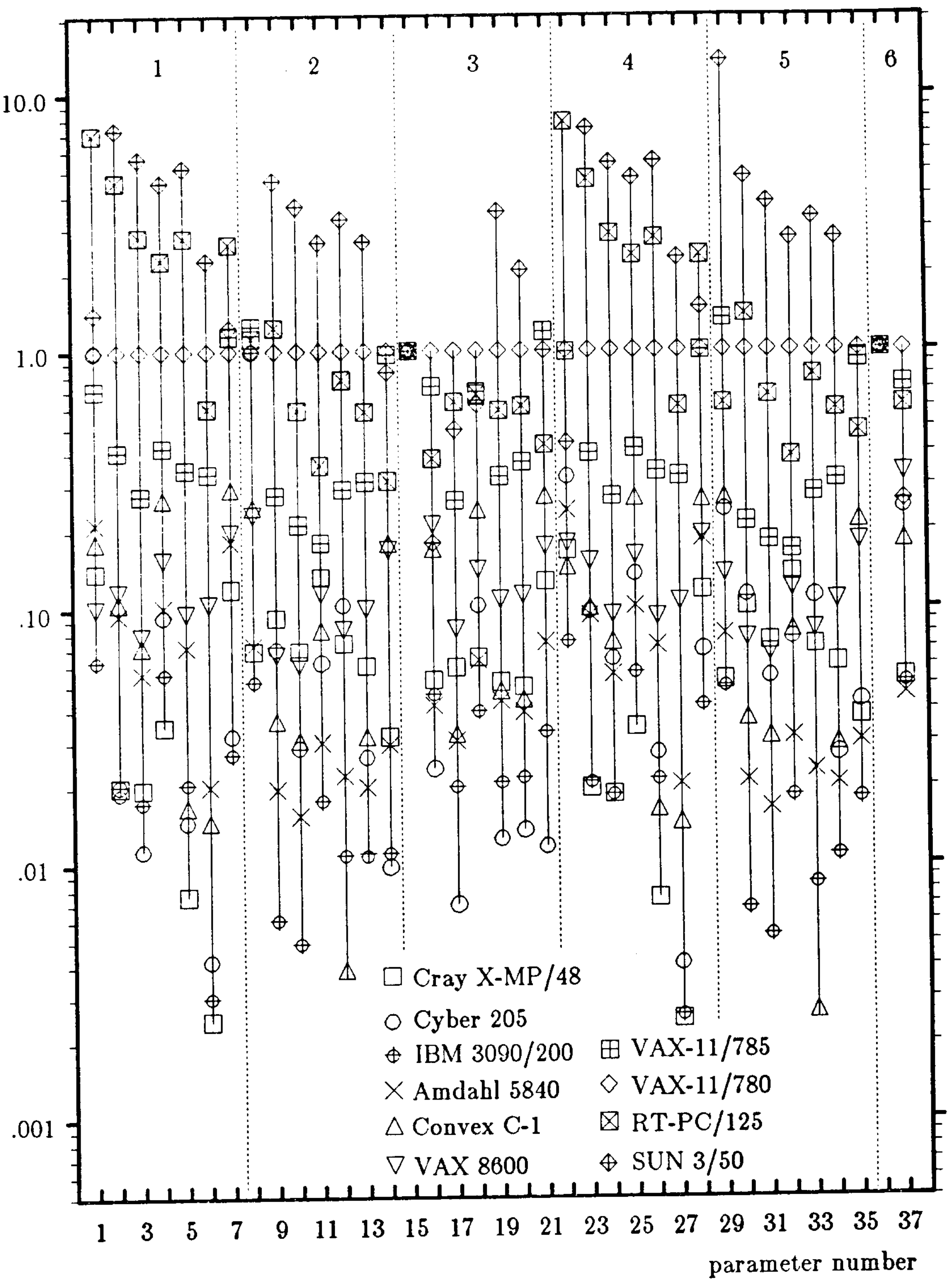


Figure 4.10: Characterization results for all machines normalized to the VAX-11/780 (I). Regions 1-6 represent different aspects of the characterization (see table 4.11).

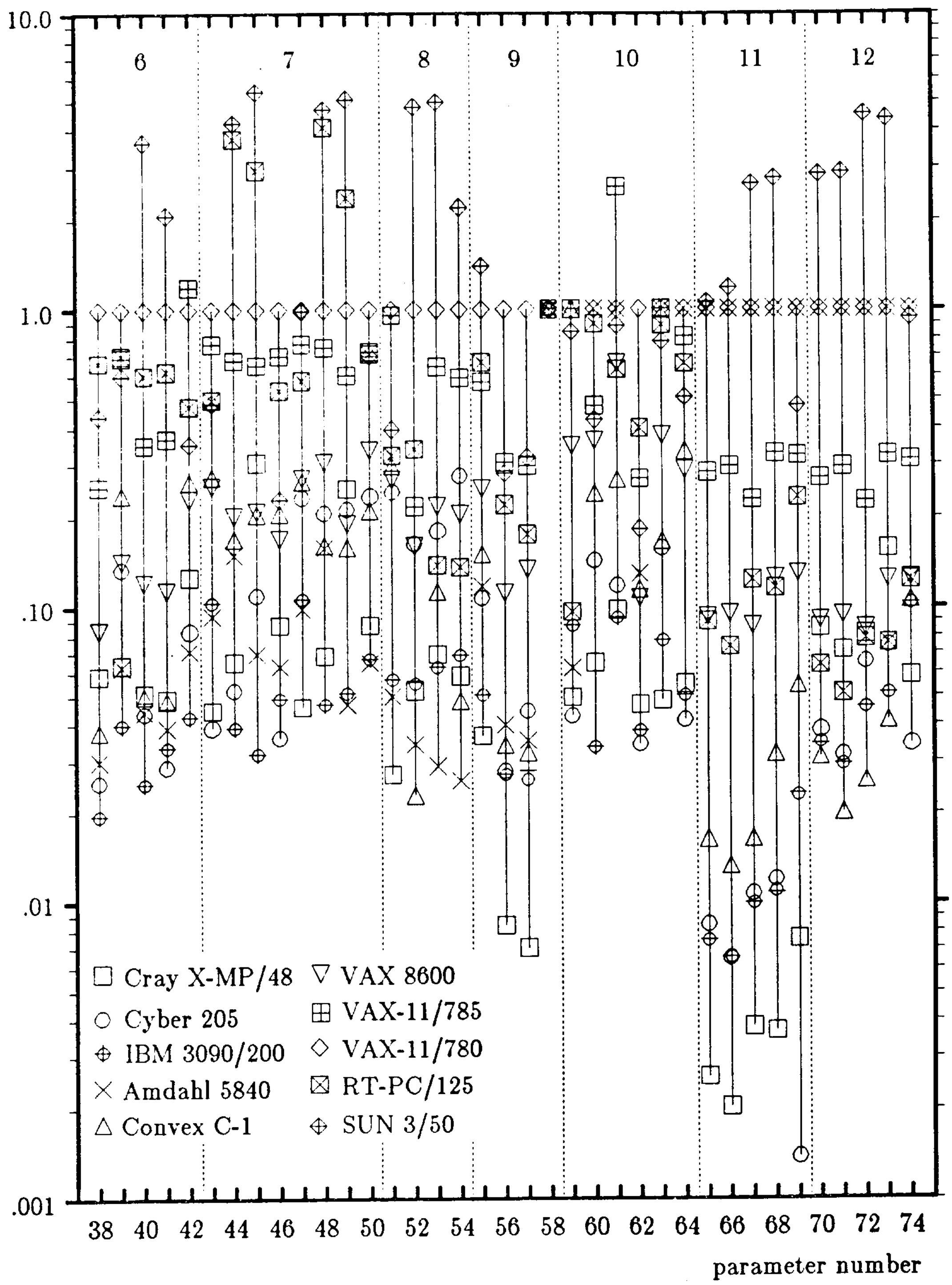


Figure 4.11: Characterization results for all machines normalized to the VAX-11/780 (II). Regions 6-12 represent different aspects of the characterization (see table 4.11).

5
The Program Analyzer

The system characterizer allows us to represent the performance of individual operations of different architectures using a single unified model. On the other hand, the program analyzer decomposes applications in terms of the same group of parameters. The program analyzer is a tool for measuring static and dynamic properties of programs. These properties determine how the application will be executed by the system we are evaluating. The parameters chosen for this decomposition are exactly the set of operations supported by the programming language. It is for this reason that to implement a program analyzer we only need to modify the compiler to obtain the static properties of the application. Also we need to instrument the source code or the object code to produce dynamic statistics at run time. Using the static and dynamic statistics it is possible to obtain the dynamic behavior of the application.

5.1. Execution Profilers

Most of the computer systems currently in use have utilities to produce execution profiles using additional information generated by the compiler [POW83]. As an example, in UNIX 4.3BSD, the C, Pascal, and FORTRAN compilers have two options to obtain reports about the program's execution profile. The first option (-p switch), instruments the object code to record information about the number of times each function is executed and the amount of CPU each function consumes. When the program finishes execution, it produces a file called mon.out that contains the values of the counters and timers. A utility program called 'prof' takes this information and using the table of symbols located at the end of the object file, produces a detailed report about how many times each function was executed and the amount of time it spent in each function. A more useful tool is gprof [GRA82]. The profile information stored in the monitor file (gmon.out) also contains the call graph of the execution and the report generated by gprof gives specific information of who invoked each particular function and how many times.

The SUN workstation also provides information about the number of times each line in the source code executes. The utility program 'tcov' prints the original source program along with the number of times each lines was executed. As in the case of 'prof' and 'gprof' the compiler instruments the object code by including counters for each basic block of the source code.

The information that our program analyzer produces is similar to the one produce by 'tcov'. However, in addition to the number of times each basic block executes, we need to count for each line, or more specifically for each statement, how many times each operation appears in the statement. As we saw in the last section, it is not possible to access at the high level of a programming language the primitive operations like load or store. It is for this reason that we need to distinguish for each accessible operation at the high level the type of the operands, their storage class, and the number of bits used in their representation. Because this is also the kind of information that the compiler needs in order to produce correct object code, it is the compiler the best place to obtain this information.

Figure 5.1 shows how static and dynamic statistics are measured by the program analyzer. As we can see, this process is very similar compared to how execution analyzers work. In our system we are using the front end of a FORTRAN compiler to instrument and collect static statistics for each block, and to instrument the source code to produce the dynamic statistics as well. The statistics of an application produced by the program analyzer depend only on the application itself and not on the computer systems in which it runs. The significance of this is that if we have M applications that we are going to use to evaluate N computer systems, we only need one description for each program and one for each system ($N+M$). To make a performance evaluation using normal benchmarks, we need to make $N \cdot M$ runs.

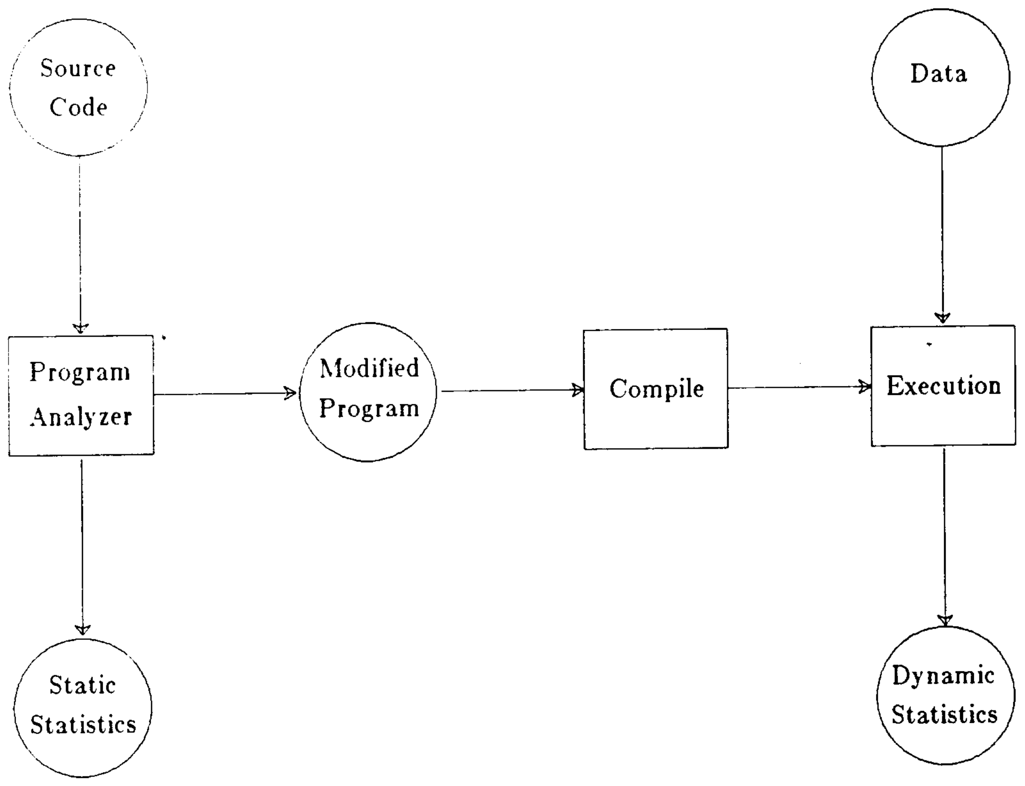


Figure 5.1: Static and dynamic analysis of programs.

5.2. Static and Dynamic Statistics

We can see how the program analyzer works using as an example a particular statement in one of the kernels of the Livermore Loops [MCM86]. These collection of loops represent the type of computational kernels found in the codes normally executed at Lawrence Livermore National Laboratory. These loops contains mathematical operations, such as inner product and matrix multiplication, and more complicated algorithms like the Monte Carlo Search Loop and 2-D Particle in a Cell Loop. John Feo [FEO87] has investigated the computational and parallel complexity of the loops, in an attempt to use the loops to evaluate the performance of MIMD machines. In the next few lines we can see a single statement taken from the eighth kernel of the Livermore Loops:

```
     U1 (KX,KY,NL2) = U1 (KX,KY,NL1) + A11 * DU1 (KY) +
   1                           A12 * DU2 (KX) + A13 * DU3 (KY) +
   2                           SIG * (U1 (KX+1,KY,NL1) - 2. * U1 (KX,KY,NL1) +
   3                           U1 (KX-1,KY,NL1))
```

Table 5.1: **Static and dynamic statistics of kernel**

param.	operation	type	precision	class	static	dynamic
SRSL	store	real	single	local	1	62280
ARSL	addition	real	single	local	6	373680
MRSL	multiplication	real	single	local	5	311400
ARR1	array with 1 dim.	na	na	na	3	186840
ARR3	array with 3 dims.	na	na	na	5	311400
IADD	index addition	integer	single	local	2	124560

Table 5.1: Static and dynamic statistics of kernel (Livermore Loops).

The program analyzer decomposes this statement not only in the number and type of operations involved, but also makes the distinction that the integer addition (2 operations) executes in the context of an index array. Normally operations between indexes are handled different from other arithmetic operations. The program analyzer keeps separate counters to distinguish between conventional arithmetic operations in expressions and arithmetic operations using indexes.

5.2.1. Description of the Test Programs

We ran the program analyzer for the ten programs in table 5.2. There are three group of programs: small integer oriented tests like the Baskett puzzle, Shell, and Erathostenes. There are some floating point computational intensive programs like Los Alamos benchmark, The Livermore Loops, the NAS benchmark, Whetstone, the Mandelbrot set, and the Linpack benchmark. The last group is represented by the Smith benchmark that contains intensive integer, floating point and logical computations. The execution time for the programs varies from .1 of a second to approximately 600 seconds on a CRAY X-MP/48.

- **Los Alamos**: this is one of the benchmarks used by LANL Computing and Communication Division [BRI86, BUC85, GRI84, SIM87] to evaluate the performance of supercomputers. This code is known as BMK8A1 an consists of a series of simple vector calculations (run in scalar mode in this study) to test the rates of vector operations as a function of vector length. The vectors are stored in contiguous memory locations. Typically one million floating-point operations are timed.
- **Conway–Baskett puzzle**: This benchmark is a program developed by Forest Baskett [BEE84] and normally used to evaluate the performance of microcomputers and RISC-based machines [PAT82]. The program is a depth-first, recursive, backtracking tree search algorithm to find a solution to a particular puzzle invented by John Conway. The puzzle consists in placing 18 tri-dimensional pieces to form a cube of five units on each side.

Table 5.2: **Characteristics of the test programs**

Name	Description of the program
Alamos	Los Alamos benchmark for vector operations execution rates
Baskett	A backtrack algorithm to solve the Conway-Baskett puzzle
Erathostenes	The sieve of Erathostenes on 60000 numbers
Linpack	The standard linear equations software of Argonne Nat. Labs.
Livermore	The twenty four Livermore Loops
Mandelbrot	Compute the Mandelbrot set on a grid of 200 by 100 points
NAS Kernels	NASA Numerical Aerodynamic Simulation benchmark
Shell	Sorts and array of 10000 random number using the Shell sort
Smith	A collection of tests similar to our System Characterizer
Whetstone	The Whetstone benchmark

Table 5.2: Characteristics of the test programs.

- **Erathostenes sieve**: This program is a simple search for prime numbers using the centuries-old sieve method. The computation of arithmetic expressions is minimal and most of the time is spent in doing comparisons.
- **Linpack benchmark**: This is one of the most popular benchmark used in performance evaluation for floating point computations. The program consist of two routines: the first computes the decomposition of a matrix, and the second routine solves a system of linear equations represented by the above matrix. The program was originally designed to give the users of the Linpack software package information about the possible execution times for solving linear equations. Nowadays, there are over two hundred machines reported in the list collected by J. Dongarra at the Argonne National Laboratory [DON85, DON87a, DON87b, DON88]. Because Linpack running in single precision fits completely in a 64K cache, the performance reported by this benchmark may be higher than the actual performance obtained by solving linear equations in real problems. In some machines a small memory-cache bandwidth can slow down considerable the execution if the matrix does not fit entirely in the cache [MIP87].
- **The Livermore Loops**: This benchmark is a set of 24 kernels that measure FORTRAN numerical computation rates [MCM86]. The loops (originally fourteen) were written by Fred McMahon in the early seventies and represent the kind of computations found in Livermore codes. The benchmarks gives the computational rates for each of the loops and for different vector lengths. It also computes a sensitivity analysis of the harmonic mean for seven work distributions giving a total of forty nine possible CPU workloads. The benchmark is a good test of the capabilities of the compiler to produce efficient (vectorizable) code. The range of performance for vector machines can vary up to two order of magnitude in the different loops. The twenty four loops are shown in table 5.3.
- **Mandelbrot set**: This program computes for a window of 200 X 100 points on the complex plane the mapping $Z_n \leftarrow Z_{n-1}^2 + C$, until the norm of Z_n is greater than 2, or the number of iterations is equal to one hundred. This program (variations of it) is used to benchmark graphic engines. All the computations are scalar and with floating point variables.

Table 5.3: **Livermore Loops kernels**

Number	Kernel Description
1	hydro fragment
2	incomplete Cholesky - conjugate gradient
3	inner product
4	banded linear equations
5	tri-diagonal elimination
6	general linear recurrence equations
7	equation of state
8	A.D.I. integration
9	integrate predictors
10	difference predictors
11	sum of two vector elements
12	difference of two vector elements
13	particle in cell (2 dimensions)
14	particle in cell (1 dimension)
15	casual FORTRAN (development version)
16	Monte Carlo search loop
17	conditional computation
18	2 dimensions explicit hydrodynamics fragment
19	general linear recurrence equations
20	discrete ordinates transport
21	matrix product
22	Planckian distribution
23	2 dimensions implicit hydrodynamics fragment
24	finds first minimum in an array

Table 5.3: Livermore Loops kernels.

- **NAS benchmark**: The NAS kernel benchmark was developed by D. Bailey and J. Barton to assist in supercomputer performance evaluation [BAI85a]. The program consists of seven kernels that represent calculations typical of NASA Ames supercomputing. The kernels perform the following calculations [BAI85b]: "outer product" matrix multiplication (MXM), two dimensional complex Fast Fourier Transform (CFFT2D), vector Cholesky decomposition (CHOLSKY), vector block tridiagonal matrix solution (BTRIX), sets up an array for a vortex method solution and performs Gaussian elimination (GMTRY), creates new vortices according to certain boundary conditions (EMIT), and inverts three pentdiagonal matrices (VPENTA). The program executes approximately 2 billion floating point operations and has extensive calculations with multidimensional arrays with different loop memory strides. Testing this benchmark can use four different levels of tuning depending on the number of lines changed, deleted or inserted. This program is normally run only in supercomputers given that it may take several hours to run in a machine without vector operations.

- **Shell sort**: This is a small program that sorts ten thousand random numbers using the Shell sort [KNU73]. The algorithm was proposed by Donald L. Shell in 1959 and is also called sort by diminishing increments. The number of operations executed by the program is $O(N^{3/2})$. The operations executed are comparisons and memory transfers.

```
PROGRAM STATISTICS
 Lines processed              -> from 1 to 37 [37]

 mnem  operation          occurrences  fraction
[srsl] store  (01)        occur:     8 (0.1481)
[arsl] add    (02)        occur:     8 (0.1481)
[mrsl] mult   (03)        occur:     4 (0.0741)
[drsl] divide (04)        occur:     2 (0.0370)
[trsl] trans  (07)        occur:    12 (0.2222)
[sisl] store  (15)        occur:     1 (0.0185)
[sisl] add    (16)        occur:     3 (0.0556)
[sisl] trans  (21)        occur:     3 (0.0556)
[andl] and-or (43)        occur:     1 (0.0185)
[crsl] r-sin  (44)        occur:     1 (0.0185)
[cisl] i-sin  (46)        occur:     1 (0.0185)
[proc] proc   (51)        occur:     2 (0.0370)
[argr] r-sin  (52)        occur:     2 (0.0370)
[goto] goto_s (61)        occur:     2 (0.0370)
[loin] do-ini (63)        occur:     2 (0.0370)
[loov] do-lop (64)        occur:     2 (0.0370)
```

Figure 5.2: Static statistics for the Mandelbrot set. The fraction column gives the static distribution of the occurrence of the parameters in the source code.

- **Smith benchmark**: It is a FORTRAN program which consists of 77 individual timed loops, each of which measures some aspect of machine performance. It contains tests of branch code, numeric code, procedure calls, and data movement, and has samples of other computations, such as matrix multiplies and bubble sorts [SMI88]. This benchmark is designed to measure various aspects of system performance; the user can then weight the various performance factors as he sees fit. More than seventy machines have been measured, ranging from microcomputers to multiprocessors and supercomputers. The benchmark has been designed to prevent most optimizations performed by compilers.
- **Whetstone benchmark**: this is a synthetic benchmark based on the statistics of 949 programs written in ALGOL 60 at the National Physical Laboratory and Oxford University during the late sixties [CUR76]. The results using this benchmark are still quoted by some manufactures but few accept the validity of this benchmark as a real measure of floating point intensive calculations. The most important reason is that the distribution of statements in programs has changed as a results of improvements in the programming methodologies, programming languages and most important machine architectures. The benchmark produces a single figure of merit for scalar processing. This number represents the performance for the execution of its ten modules (tests), and combines the performance of floating point operations with the performance of trigonometric and integer operations. It is important to note that most of the time is spent on module 7 that makes extensive use of trigonometric functions.

5.3. Output from the Program Analyzer

In this subsection we present an actual example of how the program analyzer works. In figure 5.2 we can see the output for the Mandelbrot program. We can see that for each parameter we report the number of static occurrences in the code and the static distribution of parameters. Although in this research the static distribution does not give us any additional information, these statistics are sometimes by software engineers to compute software complexity metrics.

It is possible to insert 'compiler' directives in the source code to instruct the program analyzer to produce partial reports for some number of source lines. This gives the user the opportunity of knowing the static and dynamic statistics of that portion of the code, and also to produce, with the aid of the system characterizer and the execution predictor, execution time estimates of subparts of the program.

5.4. Programs Statistics

Tables 5.5 and 5.6 present the dynamic statistics of each program sorted by value, and also their cumulative distribution. The table shows the most costly parameters for each program in terms of the number of times each parameter is executed. These not necessary represent the most time consuming parameters, which depend on the characteristics of the machines. That type of information will be generated using the program predictor and the parameters of each machine. We see in the distributions that there is a small number of operations that account for almost all the execution time. In all the programs except the Whetstone benchmark between five and seven operations represent almost ninety percent of the total number of operations executed.

The number of times each operation executes depends on the input given to the program. Executing the program for a different data input will almost always produce a different distribution in the dynamic statistics. The parameters that correspond to the access of 1 dimensional arrays (ARR1), the loop overhead time (LOOV), the arithmetic operations -add and multiply- are the most executed operations for all the programs. But the distribution of these parameters varies considerably from program to program. The reason for this variation is that some programs (Mandelbrot, Shell, Erathostenes) have a small number of lines, between 40 and 100. In fact, these program represent only a very small fraction of the total execution time of our workload. Five of the benchmarks execute in less than 2 seconds on the CRAY X-MP, while the other five programs take between 10 and 600 seconds. Because these programs use a small number of different operations normal errors in our measurements are not balanced out by other errors on other operations. An example of this is the Mandelbrot program that executes mainly scalar floating point arithmetic without using array elements. In the case of the Los Alamos benchmark the code is a repetition of small loops of the form:

```
          CALL JOBTIM (T1)
          DO 20 J = 1,LOOPS
             DO 20 I = 1,LEN
                R(I) = V1(I) * S1
       20 CONTINUE
          CALL JOBTIM (T2)
```

these kind of constructs are not representative of real applications. If we compare the statistics for the Livermore Loops with a program length of 1900 lines, against the results reported by Knuth [KNU71, WEI84] we find a fairly good agreement between the two distributions; see table 5.4.

Distribution of Statements

Statement	Knuth		Livermore	
	sta	dyn	sta	dyn
assignment	51	67	55.5	66.3
call user	5	3	7.4	4.9
call standard	7	1	9.2	1.6
return	4	3	2.3	4.9
if	10	11	6.5	14.1
do loop	9	3	11.1	3.2
goto	9	9	5.7	2.9
other	5	7	2.3	2.0

Table 5.4: Static and dynamic statistics at the statement level (all quantities in percentages).

Alamos			
param	dyn	frac	cum
ARR1	322006015	.4607	.4607
LOOV	93493112	.1338	.5945
SRSG	91003000	.1302	.7247
MRSG	84000000	.1202	.8449
ARSG	49000000	.0701	.9150
IADD	35000000	.0501	.9651
ARSL	21000197	.0300	.9951
LOIN	2314108	.0033	.9984
AGRS	712296	.0010	.9994
AGIS	180000	.0003	.9997
PROC	179592	.0003	1.000
AISL	4000	.0000	1.000
EISL	2000	.0000	1.000
TRSL	1445	.0000	1.000
DRSL	1105	.0000	1.000
XRSG	1000	.0000	1.000
TRSG	1000	.0000	1.000
SISG	1000	.0000	1.000
CISL	686	.0000	1.000
ANDL	392	.0000	1.000

Livermore			
param	dyn	frac	cum
ARSL	42082283	.2447	.2447
ARR1	24874691	.1446	.3893
SRSL	23755236	.1381	.5274
MRSL	15721656	.0914	.6188
LOOV	13014396	.0757	.6945
ARR2	10725888	.0624	.7569
ARSG	8765400	.0510	.8079
MRSG	6963630	.0405	.8484
IADD	6576587	.0383	.8867
SRSG	6576586	.0382	.9249
TRSG	1888908	.0110	.9359
ARR3	1307880	.0076	.9435
AISL	1121655	.0065	.9500
AGRS	979662	.0057	.9557
PROC	975927	.0057	.9614
SISL	878034	.0051	.9665
TISL	867738	.0050	.9715
AGIS	791125	.0046	.9761
CRSL	710287	.0041	.9802
AISG	451542	.0026	.9828

Baskett			
param	dyn	frac	cum
CISG	1365170	.2439	.2439
IADD	1133154	.2024	.4463
LOOV	790495	.1412	.5875
ARR2	766720	.1370	.7245
ARR1	685213	.1224	.8469
ANDL	540905	.0966	.9435
GOTO	57795	.0103	.9538
TISL	43390	.0078	.9616
AGIS	38663	.0069	.9685
CISL	31913	.0057	.9742
TISG	30855	.0055	.9797
AISL	30152	.0054	.9851
SISL	29909	.0053	.9904
LOIN	21468	.0038	.9942
PROC	19336	.0035	.9977
SISG	6009	.0011	.9988
AISG	5996	.0011	.9999
MISL	445	.0001	1.000
TRSL	4	.0000	1.000
SRSL	1	.0000	1.000

NAS kernels			
param	dyn	frac	cum
ARR2	1300189061	.2273	.2273
IADD	685238000	.1198	.3471
ARSL	566581256	.0991	.4462
ARR3	551229173	.0964	.5426
MRSL	452430305	.0791	.6217
LOOV	402504958	.0704	.6921
SRSL	363214880	.0635	.7556
MRSG	314803900	.0550	.8106
ARR1	258770855	.0452	.8558
ARR4	246288901	.0431	.8989
ARSG	183914506	.0322	.9311
SRSG	173863550	.0304	.9615
TRSL	83067591	.0145	.9760
AISL	62408350	.0109	.9869
ERSL	18740872	.0033	.9902
AGRS	14561611	.0025	.9927
PROC	13931792	.0024	.9951
DRSL	12000684	.0021	.9972
LOGS	5495002	.0010	.9982
TRSG	3413494	.0006	.9988

Linpack			
param	dyn	frac	cum
ARR1	27356020	.3767	.3767
LOOV	9735148	.1341	.5108
SRSL	9469508	.1304	.6412
ARSL	9344931	.1287	.7699
MRSL	8930874	.1230	.8929
ARR2	1604752	.0221	.9150
AGRS	1089573	.0150	.9300
AGIS	952191	.0131	.9431
PROC	819045	.0113	.9544
TRSL	674414	.0093	.9637
CISL	548318	.0076	.9713
TISL	288574	.0040	.9753
IADD	283722	.0039	.9792
DRSL	275205	.0038	.9830
MISL	270000	.0037	.9867
GOTO	267488	.0037	.9904
CRSL	265149	.0037	.9941
AISL	154571	.0021	.9962
LOIN	147117	.0020	.9982
ANDL	133822	.0018	.9999

Erathostenes			
param	dyn	frac	cum
ARR1	324880	.3252	.3252
CISL	216994	.2172	.5424
LOOX	150938	.1511	.6935
LOOV	119999	.1201	.8136
TISL	113943	.1141	.9277
GOTO	56754	.0568	.9845
SISL	6057	.0061	.9906
AISL	6057	.0061	.9967
LOIX	3245	.0032	.9999
TRSL	2	.0000	1.000
PROC	2	.0000	1.000
LOIN	2	.0000	1.000
AGRS	2	.0000	1.000
SRSL	1	.0000	1.000
ARSL	1	.0000	1.000
–	–	–	–
–	–	–	–
–	–	–	–
–	–	–	–
–	–	–	–

Table 5.5: Dynamic statistics of test programs (I). In the third and fourth columns we report the fraction of the total execution time that each parameter represents and the cumulative distribution .

Mandelbrot			
param	dyn	frac	cum
ARSL	2348726	.2206	.2206
SRSL	2328726	.2187	.4393
MRSL	2308524	.2168	.6561
GOTO	597131	.0561	.7122
CRSL	597131	.0561	.7683
CISL	597131	.0561	.8244
ANDL	597131	.0561	.8805
AISL	577133	.0542	.9347
SISL	577131	.0542	.9889
TRSL	80207	.0075	.9964
LOOV	20200	.0019	.9983
TISL	20002	.0019	1.000
LOIN	201	.0000	1.000
PROC	2	.0000	1.000
DRSL	2	.0000	1.000
AGRS	2	.0000	1.000

Shell			
param	dyn	frac	cum
ARR1	2021588	.3730	.3730
TISL	739869	.1365	.5095
SISL	721809	.1332	.6427
AISL	721794	.1332	.7759
CISL	721682	.1332	.9090
GOTO	263303	.0486	.9576
LOOV	230018	.0424	1.000
LOIN	16	.0000	1.000
DISL	14	.0000	1.000
TRSL	2	.0000	1.000
PROC	2	.0000	1.000
AGRS	2	.0000	1.000
MISL	1	.0000	1.000
–	–	–	–
–	–	–	–
–	–	–	–

Smith			
param	dyn	frac	cum
ARR1	166383224	.3535	.3535
TISl	58968760	.1253	.4788
AISL	52178266	.1109	.5897
SISL	49496467	.1052	.6949
LOOV	43184695	.0918	.7867
GOTO	22826813	.0485	.8352
GCOM	19325095	.0411	.8763
TRSL	12053242	.0256	.9019
CISL	8077412	.0172	.9191
IADD	6707232	.0143	.9334
ANDL	5756582	.0122	.9456
MISL	5585525	.0119	.9575
AGIS	4200312	.0089	.9664
LOIN	3346721	.0071	.9735
ARSL	3025742	.0064	.9799
SRSL	2815742	.0060	.9859
SRDG	2457705	.0052	.9911
ARR2	1420727	.0030	.9941
PROC	900312	.0019	.9960
DRSL	472761	.0010	.9970

Whetstone			
param	dyn	frac	cum
ARR1	301825	.1801	.1801
ARSL	210650	.1257	.3058
SRSL	161900	.0966	.4024
AGRS	135592	.0809	.4833
MRSG	113700	.0678	.5511
LOOV	111650	.0666	.6177
TRSG	92409	.0551	.6728
AISG	84000	.0501	.7229
GOTO	55250	.0330	.7559
MISG	52500	.0313	.7872
TISG	51758	.0309	.8181
CISG	51750	.0309	.8490
DRSG	49150	.0293	.8783
PROC	45662	.0272	.9055
SISG	31500	.0188	.9243
SRSG	23400	.0140	.9383
MISL	21009	.0125	.9508
IADD	21000	.0125	.9633
SINS	12800	.0076	.9709
DRSL	7850	.0047	.9756

Table 5.6: Dynamic statistics of test programs (II). In the third and fourth columns we report the fraction of the total execution time that each parameter represents and the cumulative distribution .

6
The Execution Predictor

As we explained in the last two sections, the system characterizer and the program analyzer are the only tools that we need to produce estimates of the execution time of programs running in different architectures. characterizer and the program analyzer to obtain an estimate of the expected execution time of the application. It is clear that these estimates have meaning only for the data used in the dynamic analysis of the programs. In this section we will obtain prediction for our benchmarks and compare these results to the actual running times.

6.1. Computing Execution Estimates and Experimental Errors

In section 3.3 we proposed a model of execution in which the total time is a linear combination of the number of times each operation executes in the program and the times it takes to execute these operations. We gave expressions to compute for each kind of test the variance involved in the measurements. The variance in the total execution time for an application is

$$\sigma^2 T = \sum_{i=1}^{n} C_i^2 \cdot \sigma^2 P_i$$

where C_i is the number of operations of type i executed in the program. If the experimental errors are small compared to our measurements, the total variance in our predictions will tend to be small. Programs that execute many arithmetic operations tend to produce predictions with small intervals of uncertainty. This also applies for systems where the clock resolution is fine.

Figure 6.1 presents a sample output from the execution predictor. Each line contains the number of times the operation was executed, and the fraction of the total that that number represents. The output also includes the expected execution time, the fraction of the total time and the standard deviation.

6.2. Execution Prediction and System Characterizers

As mentioned in the introduction, one of the problems with benchmarks is relating their results to the actual characteristics of the machine and the application programs. Knowing that a computer system runs the Dhrystone benchmark at a certain rate does not give us sufficient information about what will be the expected execution time of other programs. Obviously if machine A has a Whetstone rate that is several times greater than the rate for machine B, we can expect that scientific applications will run faster on machine A, but this does not give us precise information about how fast the programs will actually run. Also the machines we need to evaluate are normally comparable in their overall performance, and knowing that one has better results for a couple of benchmarks does not imply that it is going to run our applications faster; especially when these applications execute only a small set of operations that run faster in the machine that did not get the best results using a particular set of benchmarks. The situation gets complicated when the performance evaluation index produced using some group of benchmarks differs

```
PROGRAM STATISTICS FOR THE VAX-11/785
 Lines processed            -> from 1 to 37 [37]

mnem  operation     times-executed  fraction  execution-time  fraction  std.dev.
[srsl] store  (01)  exec:   2328726 (0.2187)  time: 0.900751 (0.0703) 0.452488
[arsl] add    (02)  exec:   2348726 (0.2206)  time: 3.453567 (0.2695) 0.128027
[mrsl] mult   (03)  exec:   2308524 (0.2168)  time: 4.666220 (0.3641) 0.438389
[drsl] divide (04)  exec:         2 (0.0000)  time: 0.000009 (0.0000) 0.000000
[trsl] trans  (07)  exec:     80207 (0.0075)  time: 0.212172 (0.0166) 0.022616
[sisl] store  (15)  exec:    577131 (0.0542)  time: 0.000000 (0.0000) 0.000000
[aisl] add    (16)  exec:    577133 (0.0542)  time: 0.682171 (0.0532) 0.027356
[tisl] trans  (21)  exec:     20002 (0.0019)  time: 0.057306 (0.0045) 0.004174
[andl] and-or (43)  exec:    597131 (0.0561)  time: 0.536582 (0.0419) 0.027528
[crsl] r-sin  (44)  exec:    597131 (0.0561)  time: 1.276666 (0.0996) 0.141042
[cisl] i-sin  (46)  exec:    597131 (0.0561)  time: 0.997388 (0.0778) 0.066700
[proc] proc   (51)  exec:         2 (0.0000)  time: 0.000042 (0.0000) 0.000003
[agrs] r-sin  (52)  exec:         2 (0.0000)  time: 0.000001 (0.0000) 0.000001
[goto] goto_s (61)  exec:    597131 (0.0561)  time: 0.000000 (0.0000) 0.000000
[loin] do-ini (63)  exec:       201 (0.0000)  time: 0.003643 (0.0003) 0.000492
[loov] do-lop (64)  exec:     20200 (0.0019)  time: 0.027965 (0.0022) 0.006056

Estimate execution time = 12.814481 sec.  Standard Deviation = 0.663125
```

Figure 6.1: Execution time estimate for the Mandelbrot program run on a Vax-11/785.

from the actual evaluation obtained by running the real codes. In these cases it is extremely difficult to find the causes or to modify the benchmarks to better represent the characteristics of our workload. The major flaw in the benchmark approach is that we lack a model for the system that we are trying to characterize and this makes very difficult to correlate our benchmark results with application programs.

6.3. Model Validation

One of the most important tests for a computer model is the experimental validation of the accuracy and sensitivity of the model. This validation is important for several reasons: if the execution estimates agree with the experimental validation, we can have confidence that the model really characterizes the system. This provides evidence that the set of parameters used in the program analyzer are adequate to decompose applications. It also increases our confidence that the estimates produced by the execution predictor are acceptable (within a confidence interval) with the real execution time of actual codes. On the other hand, if for some applications the execution estimates do not agree with the experimental validation, we can conclude that they are some characteristics in the computer system that our model is missing or fails to capture. In this situation it is possible to isolate in the application program the operation or set of operations that cause the problem and to include them in a new more general model. This is possible because the system characterizer, the program analyzer, and the execution predictor use a machine model that is common to all machines that execute programs written in FORTRAN. We may build a new model by incorporating some additional parameters to our linear equation. To improve the model we need to write new tests to detect and measure these parameters in the system characterizer, modify the program analyzer to count the occurrence of these operations in the source codes, and produce new estimates using the results obtained with the system characterizer and the program analyzer. Isolating the portions of the codes that cause the erroneous prediction makes it possible to redefine some parameters to detect machine features not previously detected with the model. Continuing this process will lead to a more complete model of computer systems and to more accurate predictions of execution times. Here the term 'complete' refers to our ability to predict, using the characterization of a computer system to obtain the expected execution time of some set of applications. This way of approaching system characterization and performance evaluation agrees with the premises we mentioned earlier about experimentation and incremental model refinement.

We can illustrate the point made in the last paragraph with the following example. In the first version of our model arithmetic operations were classified according to the characteristics of the operands, independent of where the operation appeared in the text. The first predictions that we made for the Livermore Loops running on a the VAX-11/785 were not very far from the actual running times for some loops, but for a couple of them the actual running times were almost three times smaller than the execution estimates. When we examined the source code we found that loops 1, 4, 7, 8, and 18 have the characteristic of adding or subtracting a constant to most of their array indexes. In our model an integer arithmetic operation inside of one of the dimensions of the array was considered identical to the same operation executed between two variables of the same type, size and class storage. In almost all the existing compilers, arithmetic operations between indexes inside a loop use registers instead of making reference to memory locations, and in other cases, the constant is added to the base-descriptor of the array at compile time eliminating the unnecessary operation.

These are not an optimization but a standard features in most compilers. We improved our model to make a distinction between an integer arithmetic operation executed in the context of making a reference to an array element, and a normal operation between integer variables in expressions. In addition we wrote a small set of tests to measure the execution time of these new operations in the system characterizer. The new predictions obtained using this new approximation were as good as the best obtained previously.

6.4. Execution Predictions and Actual Running Times

We obtained execution estimates for the programs in table 5.2 and for each of the machines in table 4.1. Aside from this, we also executed each of the programs in the same systems, and measured the actual execution times. We tried to reproduce the same conditions in these tests as when we ran the system characterizers. Only in this way we can guarantee that the execution estimates obtained using the results of the system characterizer correspond to the same systems in which the programs were run. In tables 6.1-6.2 and in figures 6.2-6.4 we present the measurements along with the estimates. All the results are plotted together in figure 6.5. We also show the difference between the real measurements and the predictions.

Table 6.1: **Execution estimates and actual running times (I)**

	Alamos			Baskett			Erathostenes		
System	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
CYBER 205	91.1	83.9	−7.90	1.45	1.16	−19.85	0.154	0.145	−5.84
IBM 3090/200	80.5	73.4	−8.82	0.66	0.78	+18.18	0.130	0.114	−12.31
Amdahl 5840	345.9	327.2	−5.41	2.23	2.67	+19.73	0.463	0.408	−11.88
Convex C-1	236.1	243.6	+3.18	2.75	2.32	−15.64	0.650	0.580	−10.77
VAX 8600	265.3	266.7	+0.53	2.82	3.24	+14.89	0.750	0.603	−19.64
VAX-11/785	701.7	758.3	+8.07	7.38	8.27	+12.06	1.733	1.726	−0.40
VAX-11/780	1581.7	1702.7	+7.65	14.85	16.17	+8.89	2.766	2.462	−10.99
Sun 3/50	6273.2	5795.8	−7.61	7.06	8.315	+17.78	0.900	0.916	+1.78
IBM RT-PC/125	3881.9	3810.0	−1.85	6.20	7.40	+19.35	1.100	1.354	+23.09
average			−2.02			+6.95			−3.89
root mean sq.			6.55			15.90			12.45

	Linpack			Livermore			Mandelbrot		
System	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)
CRAY X-MP/98	8.05	8.29	+2.98	15.3	16.9	+10.46	1.002	1.057	+5.49
CYBER 205	13.51	14.31	+5.92	32.1	31.7	−1.25	0.676	0.588	−13.02
IBM 3090/200	–	–	–	19.5	18.5	−5.13	0.220	0.226	+2.73
Amdahl 5840	–	–	–	–	–	–	3.344	3.546	+6.04
Convex C-1	35.4	31.48	−11.07	67.9	69.9	+2.96	3.948	3.380	−14.39
VAX 8600	41.6	35.43	−14.83	88.2	88.7	+0.57	3.490	3.614	+3.55
VAX-11/785	99.7	106.15	+6.47	223.3	255.9	+14.60	11.36	12.82	+12.85
VAX-11/780	220.1	227.53	+3.38	611.0	653.5	+6.96	33.42	32.13	−3.86
Sun 3/50	763.7	752.96	−1.41	2457.0	2583.7	+5.16	163.94	165.81	+1.14
IBM RT-PC/125	473.9	448.47	−5.37	1610.1	1573.8	−2.25	105.43	104.09	−1.27
average			−1.74			+3.56			−0.07
root mean sq			7.67			6.99			8.04

Table 6.1: Execution estimates and actual running times (I). All real times and predictions in seconds; errors in percentage.

The result for the Livermore Loops on the Amdahl 5840 is missing because the compiler complained of an error in the program when the tests were run[1]. The NAS kernels and the linpack program were not available when the test program were run on the Amdahl 5840 and the IBM 3090/200. On the Convex C-1, VAX 8600, VAX-11/785, and VAX-11/780 the NAS kernels does not run in single precision[2].

[1] The code generator detected an error in the code produced by the first pass module.

[2] The program divides by zero on these machines if the benchmark is executed using 32-bit floating point numbers. The random number generator needs 64-bit numbers to execute correctly [BAI87]. However on the SUN 3/50 and IBM RT-PC/125 the program executed without errors with single precision.

Table 6.2: **Execution estimates and actual running times (II)**

	NAS kernels			Shell		
System	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)
CRAY X-MP/48	533.8	605.2	+13.40	0.683	0.593	−13.18
CYBER 205	1456.7	1672.7	+14.83	0.555	0.481	−13.33
IBM 3090/200	–	–	–	0.440	0.395	−10.23
Amdahl 5840	–	–	–	1.893	1.965	+3.80
Convex C-1	–	–	–	1.828	1.770	−3.17
VAX 8600	–	–	–	2.233	2.140	−4.16
VAX-11/785	–	–	–	5.800	6.110	+5.34
VAX-11/780	–	–	–	9.183	8.803	−4.14
Sun 3/50	89800.	77053.	−14.19	3.140	3.522	+12.17
IBM RT-PC/125	50863.	42216.	−17.00	4.68	4.61	−1.50
average			−0.74			−2.84
root mean sq.			14.92			8.33

	Smith			Whetstone			average	rms
System	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	error (%)	error (%)
CRAY X-MP/48	66.7	65.77	−1.39	0.302	0.296	−1.99	+1.01	8.15
CYBER 205	138.0	92.9	−32.68	1.129	0.934	−17.27	−9.04	10.85
IBM 3090/200	53.2	45.3	−14.85	0.350	0.335	−4.29	−4.34	9.47
Amdahl 5840	198.0	185.4	−6.36	1.697	1.942	+14.44	+2.91	9.43
Convex C-1	193.1	197.2	+2.12	1.111	1.170	+5.31	−4.61	9.08
VAX 8600	238.7	230.0	−3.64	2.870	2.631	−8.33	−3.45	9.88
VAX-11/785	683.9	691.6	+1.13	7.95	7.385	−7.11	+5.89	9.17
VAX-11/780	1087.5	1018.8	−6.32	21.57	21.74	+0.79	+0.26	9.52
Sun 3/50	914.8	877.4	−4.09	34.24	39.5	+15.36	+2.61	13.91
IBM RT-PC/125	545.1	675.3	+23.89	12.05	11.95	−0.82	+3.63	11.82
average			−4.22			−0.39		
root mean sq.			14.06			4.45		

Table 6.2: Execution estimates and actual running times (II). All real times and predictions in seconds; errors in percentage.

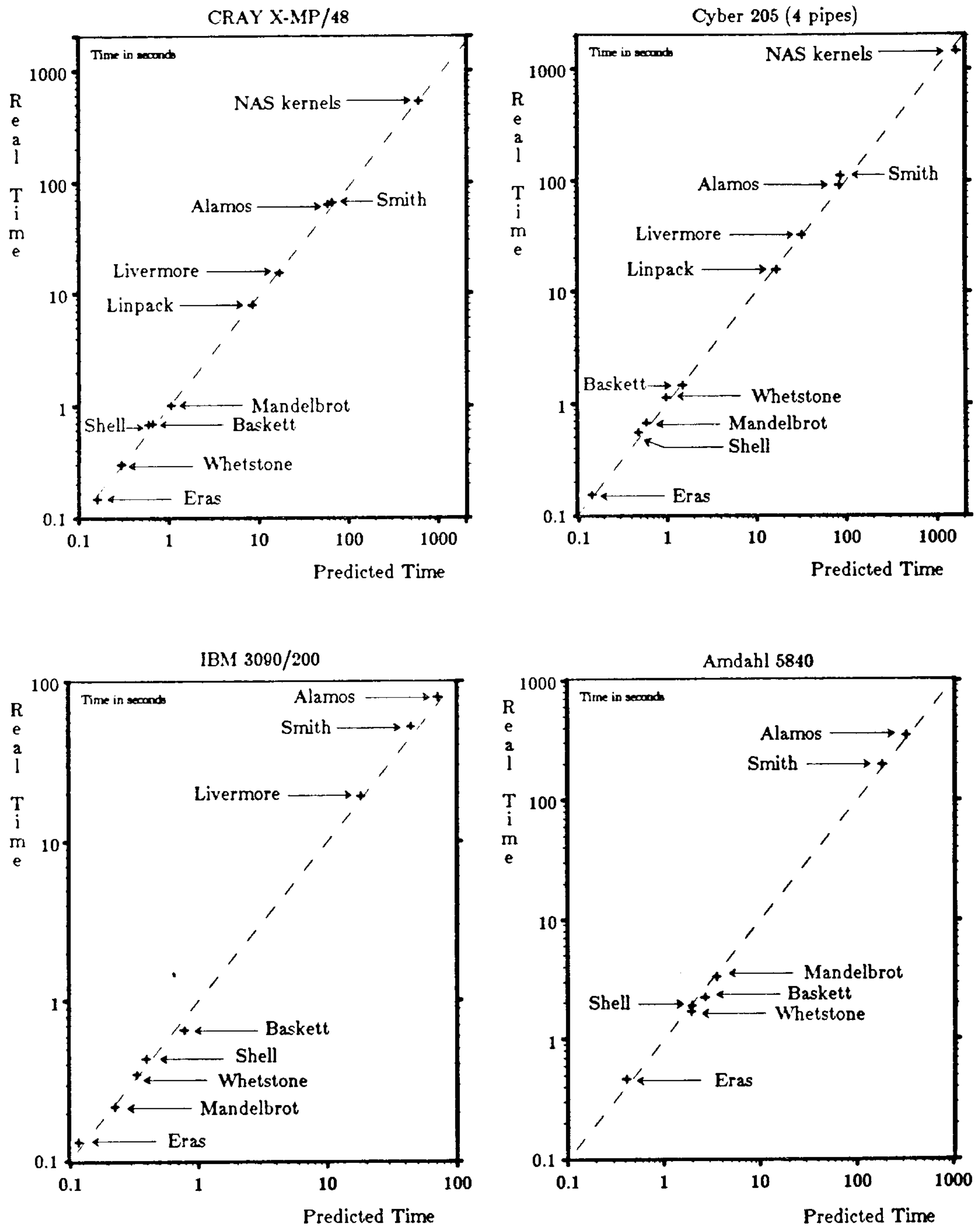


Figure 6.2: Predicted times versus real execution times (I). Results for the CRAY X-MP/48, the CYBER 205 (4 pipes), the IBM 3090/200, and the Amdahl 5840. Scales are logarithmic and values are reported in seconds.

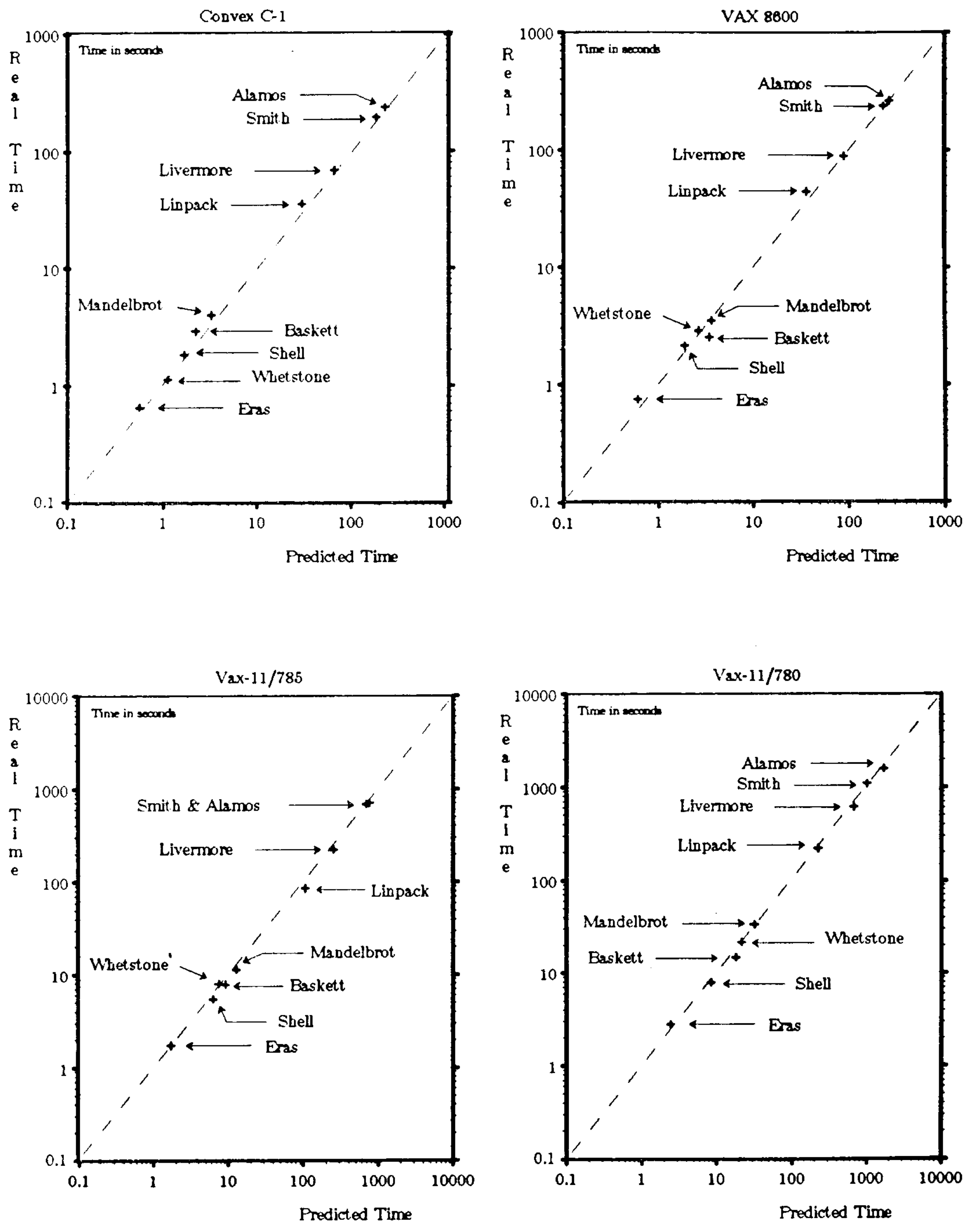
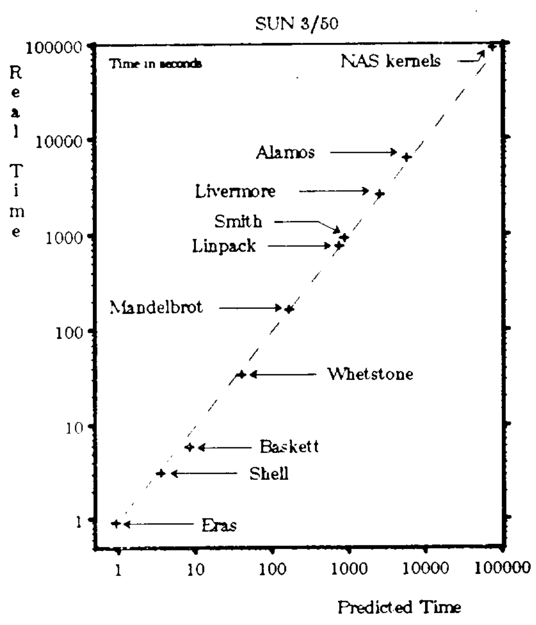


Figure 6.3: Predicted times versus real execution times (II). Results for the Convex C-1, the VAX 8600, the VAX-11/785, and the VAX-11/780. Scales are logarithmic and values are reported in seconds.



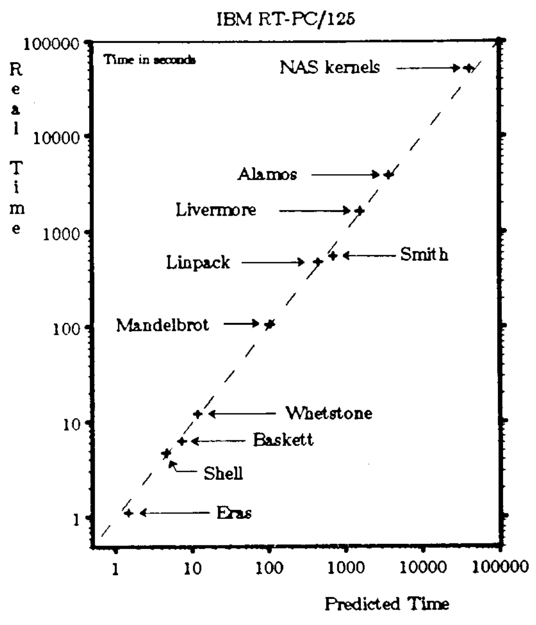


Figure 6.4: Predicted times versus real execution times (III). Results for the SUN 3/50, and the IBM RT-PC/125. Scales are logarithmic and values are reported in seconds.

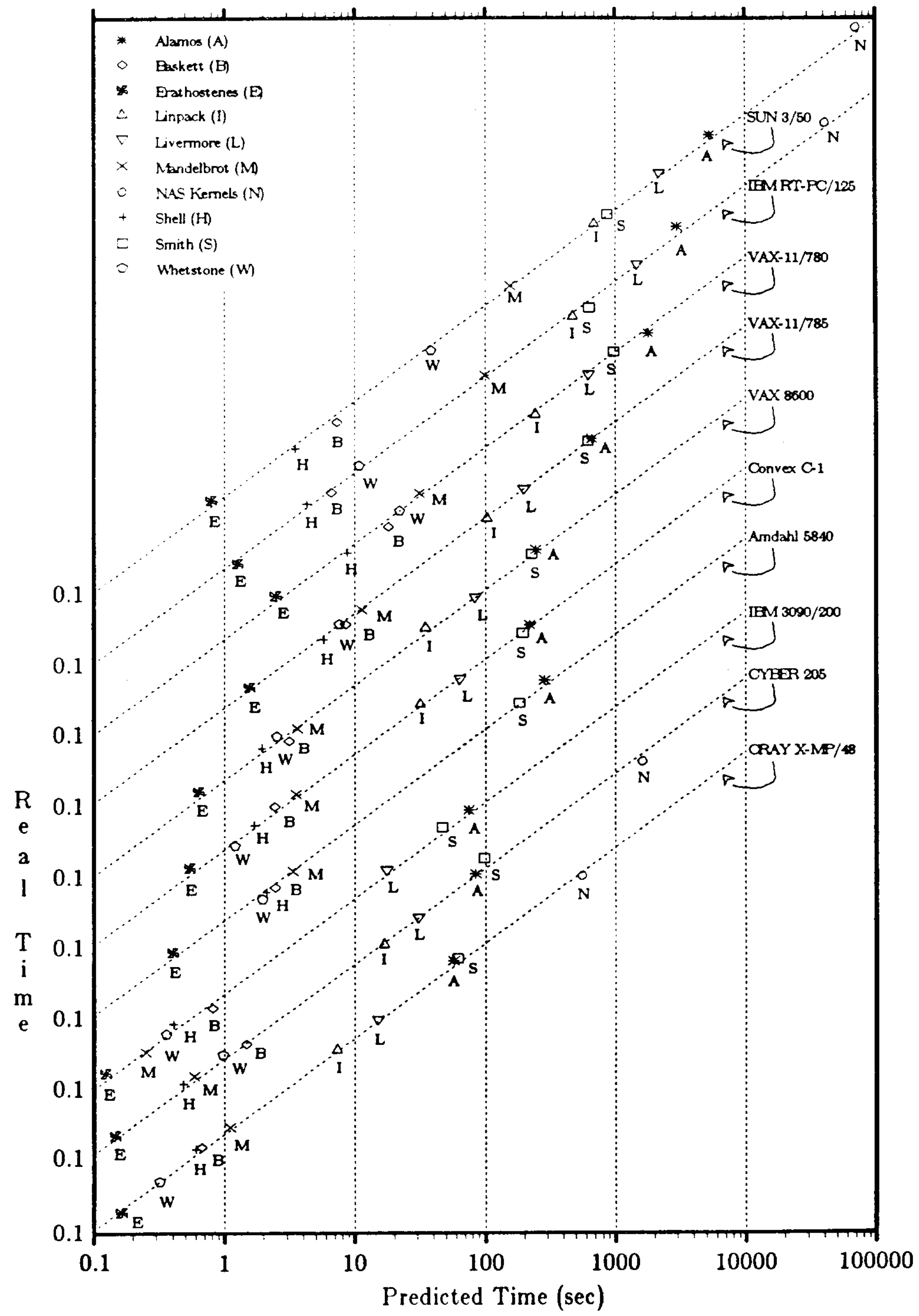


Figure 6.5: Predicted times versus real execution times (IV). Each diagonal line represents one graph from figures 6.2-6.4.

7
Analysis of Results and Summary

In this section we make an analysis of the data obtained with the system characterizer, the program analyzer and the execution predictor and show how these results can be combined to identify the strong and weak features of the systems with respect to the workload used. In section 7.2 we discuss some of the factors that must be addressed in order to improve the accuracy of our execution estimates. We finish this report by giving a summary in section 7.3.

7.1. Analysis of Results

A comparison of the execution times between our predictions and real measurements show several interesting patterns (figures 6.2-6.4 and tables 6.1-6.2). First we can see that the relative performance of the systems is not the same in all programs. For example if we consider the behavior of the three fastest machines used in these study we find the following. The CYBER was the fastest to run the Mandelbrot program; The CRAY X-MP has the shortest times for Los Alamos, The Livermore Loops, the Linpack, the NAS kernels and the Whetstone benchmarks[1]; while the IBM was the fastest on the Smith benchmark, Shell sort, and the Baskett puzzle. If we look at the codes of these programs we find that in the Mandelbrot program almost 80 percent of the dynamic statistics correspond to scalar arithmetic and logic operations. On the other hand the programs that the CRAY runs faster have intensive floating point arithmetic operations with arrays. While for the Baskett puzzle, the Shell sort and the Smith benchmark the predominant characteristic is the execution of integer operations with arrays. Except in the case of the Erathostenes sieve, our execution estimates correspond closely to the results obtained in the real executions. With the Erathostenes program the predicted times and the real times are almost identical for the three machines (the value between the minimum and the maximum execution time is six percent). This difference is less than the experimental error due to clock resolution for the IBM 3090/200.

Relative differences in performance is clearer in the case of the VAX 785, the VAX 780, the IBM RT-PC and the Sun 3/50. For the Livermore Loops, the Mandelbrot program, the Linpack benchmark, and Los Alamos, the real measurements and the predictions indicate a relative performance that varies from 8:5:2:1 to 14:9:3:1[2]. On the other hand, the result of the Shell sort and the Erathostenes sieve indicate that the Sun 3/50 and the IBM RT-PC are faster than both the VAX 785, and the VAX 780; this agrees with the real measurements and our estimates. In this case, their relative performance is around .5:.3:.75:1.

In table 7.1 we present the real and estimated relative performance between the SUN 3/50 and the IBM RT-PC/125. We see that the estimates agree with the real times in predicting which machine will execute faster each of the programs. Except for the Smith

[1] The Linpack and the NAS kernels were not run on the IBM 3090/200.

[2] The order is VAX 785, VAX 780, IBM RT-PC, and Sun 3/50.

Table 7.1: **Relative performance**			
	SUN 3/50 : IBM RT-PC/125		
program	real time	prediction	error (%)
Los Alamos	1.616	1.679	+3.90
Baskett	1.139	1.124	−1.32
Erathostenes	0.818	0.677	−7.24
Linpack	1.611	1.679	+4.20
Livermore	1.526	1.642	+7.60
Mandelbrot	1.555	1.593	+2.44
NAS kernels	1.765	1.743	−0.25
Shell	0.671	0.764	+12.17
Smith	1.678	1.299	−22.59
Whetstone	2.841	3.305	+16.33
average	1.522	1.551	+0.42
geometric	1.416	1.411	−
root mean sq.	−	−	10.37

Table 7.1: Relative performance between the SUN 3/50 and the IBM RT-PC/125. A value greater than one indicates that the IBM RT-PC executes faster than the SUN. The first two columns are dimensionless and quantities on the third column are in percentages.

benchmark, the absolute differences between the real and predicted relative performance were less than 20 percent. For this program the predicted time on the IBM RT-PC was almost 24 percent greater than the real time.

The results also indicate that our model works better for programs with long execution times and arithmetic operations. In table 7.2 we see how the predictions agree with the real execution times for each machine and for different intervals of error. We observe that approximately 60 percent of all predictions are within a distance of 10 percent from the real execution times.

Table 7.2: **Accuracy of the prediction estimates**				
System	< 5 %	< 10 %	< 15 %	< 20 %
CRAY X-MP/48	3 (30.0)	7 (70.0)	10 (100.)	10 (100.)
CYBER 205	1 (10.0)	4 (40.0)	7 (70.0)	9 (90.0)
IBM 3090/200	2 (25.0)	4 (50.0)	7 (87.5)	8 (100.)
Amdahl 5840	1 (14.3)	4 (57.1)	6 (85.7)	7 (100.)
Convex C-1	4 (44.4)	5 (55.5)	8 (88.8)	9 (100.)
VAX 8600	5 (55.5)	6 (66.6)	8 (88.8)	9 (100.)
VAX-11/785	2 (22.2)	6 (66.6)	9 (100.)	9 (100.)
VAX-11/780	4 (44.4)	8 (88.8)	9 (100.)	9 (100.)
Sun 3/50	4 (40.0)	6 (60.0)	8 (80.0)	10 (100.)
IBM RT-PC/125	5 (50.0)	6 (60.0)	6 (60.0)	8 (80.0)
Total	31 (34.1)	56 (61.5)	78 (85.7)	88 (96.7)

Table 7.2: Accuracy of the model for different intervals. The numbers inside the parenthesis show the proportion of the programs that are inside the error interval.

Table 7.3: **Predicted Distribution of Execution Time by Operation (Los Alamos)**

parameter	dyn	CRAY X-MP	CYBER	IBM 3090	Amdahl	Convex
array reference (1 dim)	.4607	.2658	.3565	**.3915**	**.4871**	.3650
loop overhead (step 1)	.1338	**.3390**	.1138	**.2842**	.2015	**.2771**
store-real-single-global	.1302	**.1142**	**.1009**	.0732	.0848	.0275
multiply-real-single-global	.1202	.1764	.2731	.1531	.1303	.1804
add-real-single-global	.0701	.0533	.1198	.0519	.0508	.0797
addition in index array	.0501	.0000	.0015	.0039	.0000	.0000
add-real-single-local	.0300	.0231	.0107	.0232	.0309	.0332
loop initialization	.0033	.0242	.0134	.0132	.0090	**.0330**
argument-real-single	.0010	.0018	.0026	.0015	.0003	.0001
argument-integer-single	.0003	.0005	.0010	.0005	.0002	.0000

parameter	dyn	VAX 8600	VAX 785	VAX 780	SUN 3/50	IBM RT
array reference (1 dim)	.4607	**.4187**	.3578	.2810	.1113	.0622
loop overhead (step 1)	.1338	.1796	.1693	.1594	.0146	.0517
store-real-single-global	.1302	.0279	.0578	.0263	.0032	.0652
multiply-real-single-global	.1202	.2045	.2201	**.3650**	**.5529**	**.4907**
add-real-single-global	.0701	.0946	.0926	.1042	**.2155**	**.2282**
addition in index array	.0501	.0000	.0000	.0000	.0054	.0041
add-real-single-local	.0300	.0309	.0404	.0443	**.0918**	**.0924**
loop initialization	.0033	**.0377**	**.0549**	.0155	.0024	.0045
argument-real-single	.0010	.0013	.0006	.0013	.0018	.0002
argument-integer-single	.0003	.0004	.0004	.0003	.0002	.0000

Table 7.3: Distribution of time for the ten most common operations in Los Alamos benchmark. The numbers in bold have a magnitude that is 50% higher than the geometric mean taking the distributions of the ten machines as sample.

Table 7.3 shows the distribution of the execution times per operations using the estimates for the Los Alamos benchmark. As we expect on different systems the distribution of the operations is different, and some operations affect the total execution time more strongly than others. Although the add and multiply operations represent only 20 percent of the total, for the IBM and the Amdahl they amount to 18 percent of the execution time, but in the case of the Sun 3/50 and the IBM RT-PC/125, this quantity is more than 70 percent. For the Amdahl 5840 its distribution is quite similar to the dynamic distribution. We cannot conclude from this table that the Amdahl is a more balanced system, because the execution time of the parameters must be proportional to the complexity of each operation in addition to how many times the operation is executed. We can see this more clearly if we compare the Convex C-1 against the Amdahl 5840. In three of the seven programs the Amdahl has better execution times than the Convex (in the predictions the Amdahl has better times in only two). For the Los Alamos benchmark, the Convex has a completely different distribution compared to the Amdahl, but because the access time of an array element is 3 times faster in the Convex, the total execution time is approximately 30 percent less for the Convex. The numbers in bold type in the table are 50% above the value of the geometric mean of the same parameter when we take as sample all the distributions.

Figures 7.1-7.3 show the systems characterization from a different perspective to help us explain the relative performance of the systems. In these figures the value of each parameter is normalized with respect to the execution of the VAX-11/780. Instead of showing all the parameters we chose a representative subset of the most executed operations. In particular, arithmetic operations with global variables were omitted given that on most systems, except the CYBER 205, the execution times are almost the same with local and global operands.

The CRAY X-MP executes faster for almost every parameter, especially floating point arithmetic operations with single precision, references to array elements, procedure calls, and intrinsic functions. The first two groups represent the most frequently executed operations in scientific programs and for this reason the CRAY executes faster the floating point intensive benchmarks. On the other hand, the scalar floating point arithmetic operations with double precision operands are executed faster on the IBM 3090, and even the VAX 8600 has better results than the CRAY X-MP. However as we pointed out in section 4.1, our benchmarks executed using 64-bit floating point numbers on the CRAY and CYBER 205.

In figure 7.2 we see that on the Convex C-1 the execution time of almost all arithmetic parameters is smaller than the VAX 8600, with the exception of the divide operation. This parameter executes slower for single precision floating point and integer data types. The normalized results for the VAX 8600 show that for almost every parameter the execution on the VAX 8600 is between 4.5 and 7 times faster than the VAX-11/780. In the case of the VAX-11/785, arithmetic operations and intrinsic functions are between 3 and 4 times faster with respect to the VAX-11/780, but the difference is less for other parameters. The SUN 3/50 executes faster integer operations, access to array elements, branching and loops, but arithmetic operations take more time to execute. The reason for this is that on the SUN the benchmarks were executed using software emulation of floating point operations.

The most interesting aspect of these figures is that the relative performance is not uniform; some architectures execute faster for some operations but are slower in others. Again, this tells us that a single figure of merit cannot show all the dimensions of the system's performance.

7.2. Future Improvements to Our System

In last section we showed that most of our predictions are within 15% of the real execution times and all but three within 20%. The discrepancy between real and predicted times is greater on small programs that use a small number of operations, like the Erathostenes sieve, and better on computationally intensive programs. There are still some factors that affect our predictions and they must be taken into account in a new version of the system if we desire to produce better estimates. The following paragraphs present a discussion of these factors.

i) Locality and Cache Memory. The code we use to measure individual parameters has a degree of locality in the reference of variables. For this reason, estimates for a program that exhibit less locality than our tests will tend to produce a larger discrepancy with respect to its actual execution time. Although scientific programs normally spend most of their time in a small number of DO loops, the amount of memory 'touched' by these loops tends to be very large. Therefore the hit ratio for

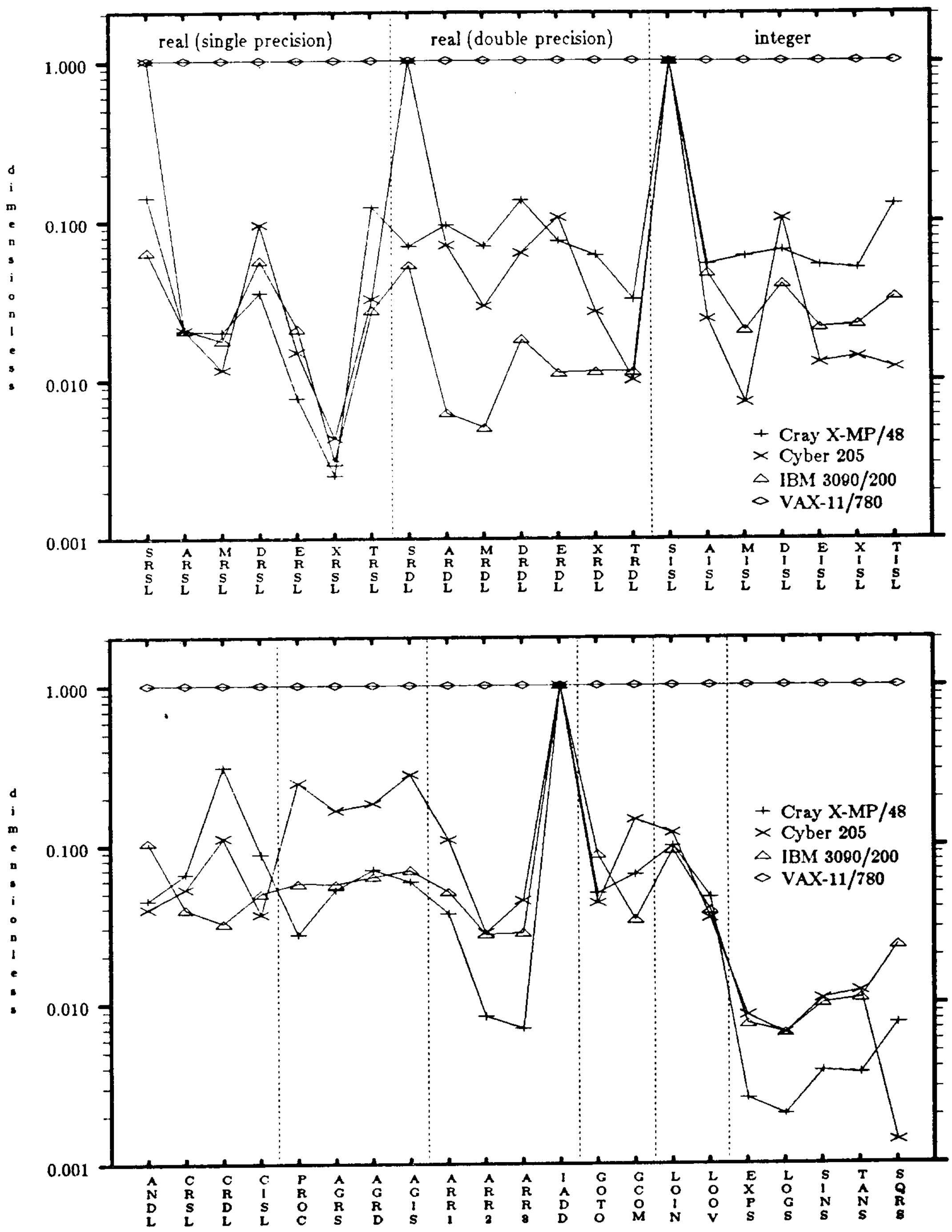
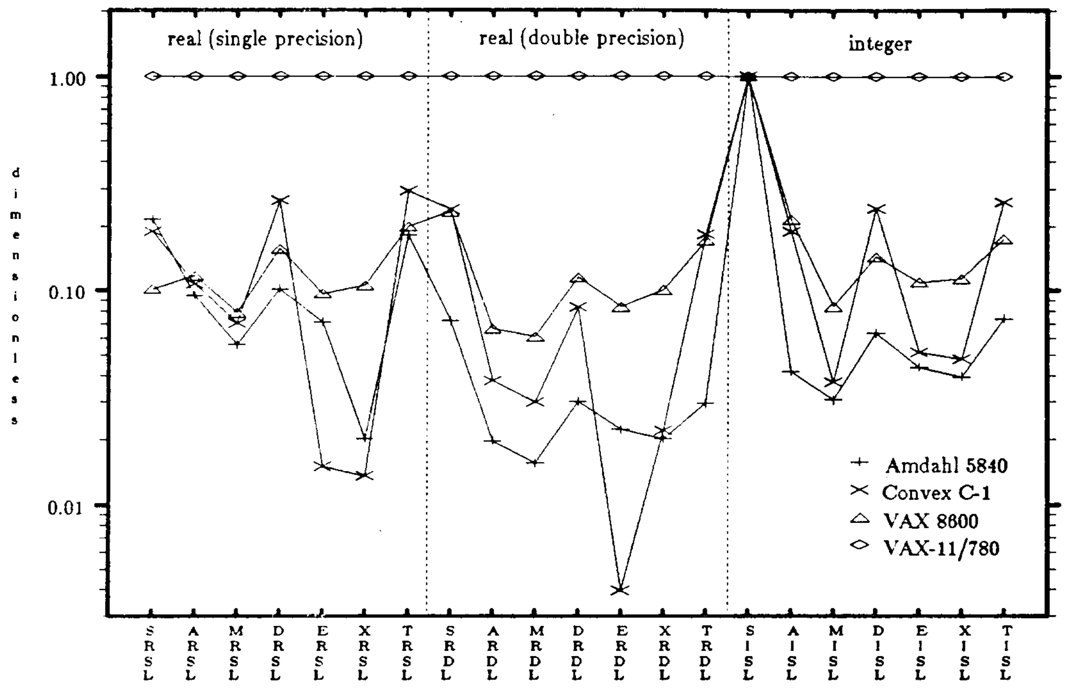


Figure 7.1: Parameters normalized against the VAX-11/780 (I). The CRAY executes faster floating point arithmetic operations (single precision) and has the shortest access time for array elements. These are the most frequently executed operations in the ten benchmarks.



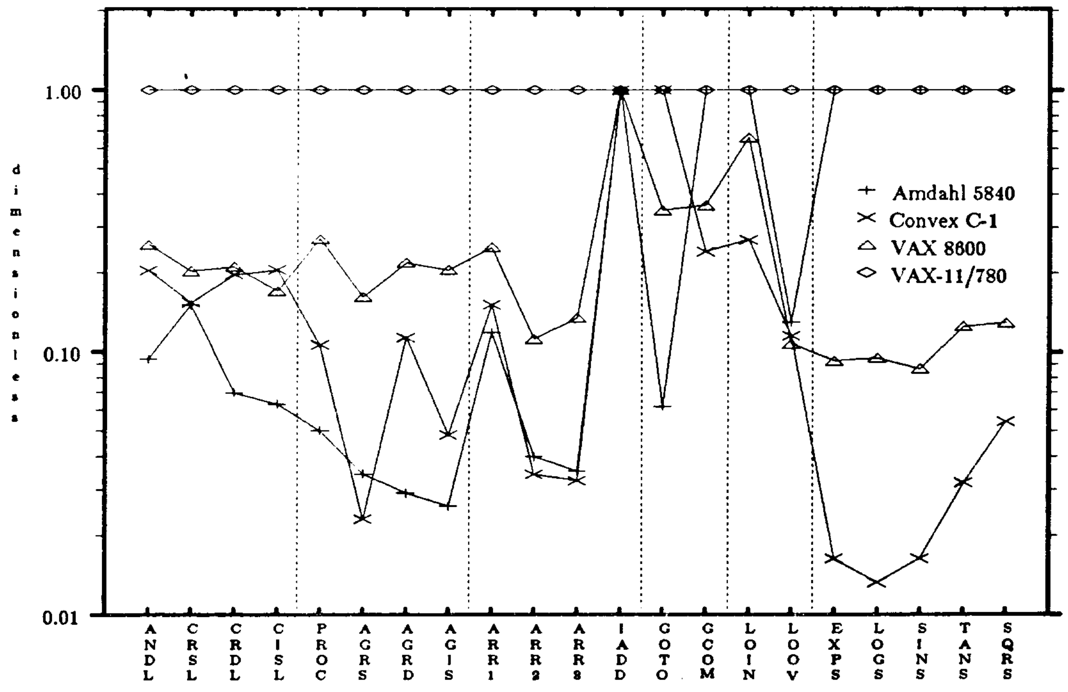
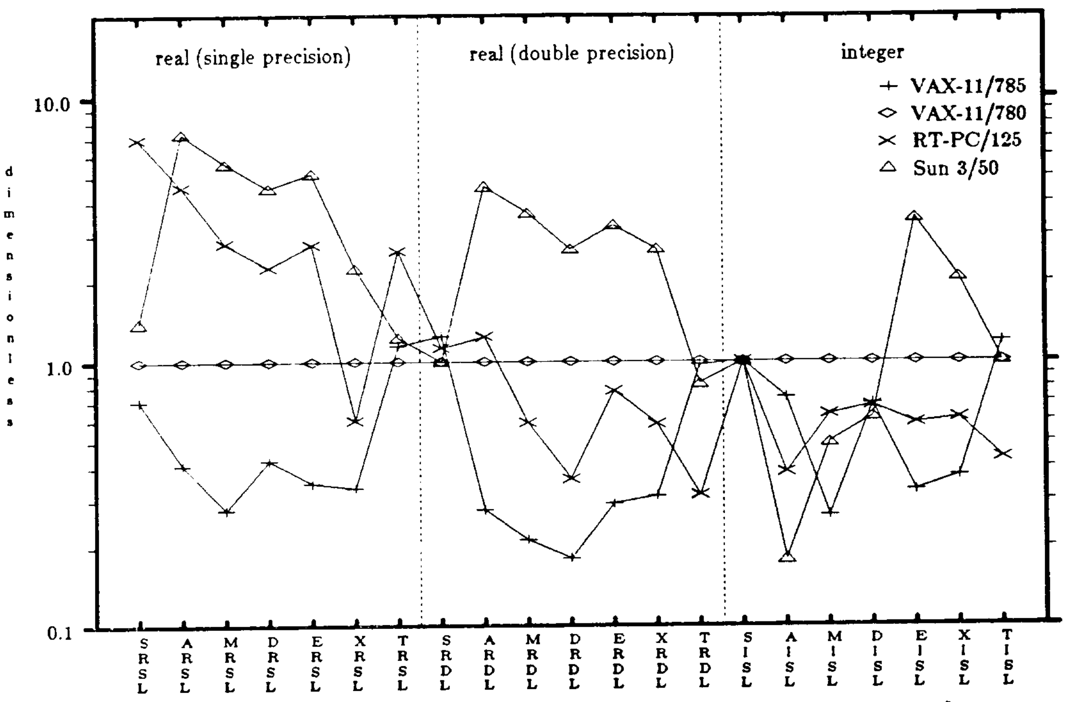


Figure 7.2: Parameters normalized against the VAX-11/780 (II).



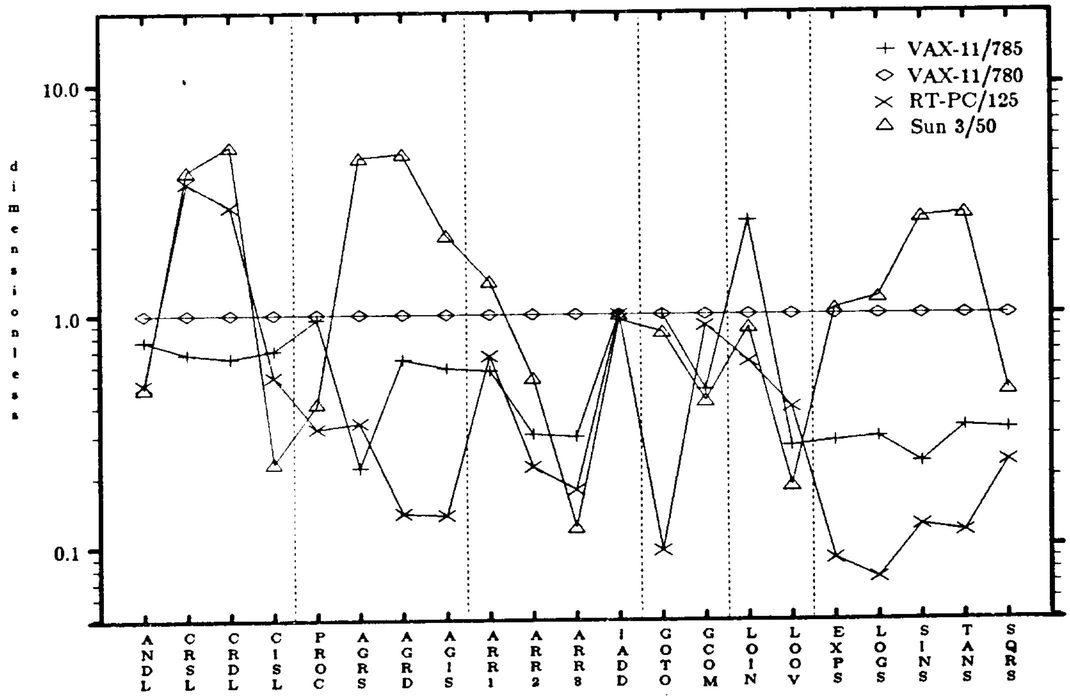


Figure 7.3: Parameters normalized against the VAX-11/780 (III).

the code is high, but for the data is low. We ran some tests increasing the number of different variables inside the body of the test and also increasing the time between successive reference to the same variable. We found that the measurements obtained in this way were larger by four to ten percent.

ii) Change of Environment in Branches. When a branch is taken or a subroutine call is executed, there is normally a change in the set of variables that are referenced. This increases the number of cache misses and also the total execution time of the program. If the branch jumps to a new page this may cause a page fault along with a context switch. A context switch normally involves flushing the cache and this has the effect of increasing the execution time of the program. Several parameters that characterize the 'size' of the branch will help to measure the penalty that we pay as a function of the distance between the branch and its target.

iii) Hardware and/or Software Interlocks. In pipelined machines the time it takes to produce the next result for a particular operation depends on the context in which this operation executes. This time normally depends on the functional and data dependencies with respect to the previously scheduled instructions. The data dependencies are a function of the source code, the code produced by the compiler, and the hardware. We discussed this problem on section 3.4.

iv) Missing Parameters. In our model there are some simplifications that may increase the discrepancy between our predictions and the real execution times for some programs. An example of this is the access of array elements. We assumed that the overhead in accessing an element is constant for different data types and also that this overhead is independent of the context in which the access occur. The traversing of a multidimensional array inside a loop is normally done in a regular way (fixed stride). and the compiler may detect that some dimensions remain constant during the whole execution of the loop. With this information the compiler may compute the address for the next element using less operations than it will require if we reference the element outside the loop. Several new parameters are needed to represent all the different variation in the reference of an array element.

v) Limitations of the Linear Model. The assumption that the cost of executing an operation is independent of the adjacent operations, data dependencies, etc, does not remain valid if we want to reduce the error in our predictions. Although it is possible to create new parameters that characterize pair of instructions and with this keep the linear model hypothesis, this will create an explosion in the number of parameters. An additional disadvantage is that these 'compound' parameters lose their natural interpretation and it is more difficult to identify weak features in the systems.

vi) Machine Idioms. Some architectures implement special cases of some instructions very efficiently. On the VAX architecture it is possible to multiply an integer by two, four or sixteen, then add another integer and use this result as an address during the execution of the same instruction. Unless we know the architecture and how the compiler works it is not possible for us to detect which are the idioms of a given architecture.

vii) Random Noise Produced by Concurrent Activity. Although we discused this in section 4.1, there is still some potential problem when we run in a loaded system.

If there is a peak of activity during the execution of an experiment, our measurements will be slightly affected by this 'unusual' high activity. In programs where these parameters are the most executed the 'noise' will increase our figures in a significant way.

7.3. Summary

In this report, we have presented a new paradigm for system characterization and performance evaluation. The principal attribute of this model is that the set of parameters used in the characterization of systems are the same set of parameters used to estimate the expected execution time of programs. The characterization is achieved by running a set of software experiments that identify, isolate and measure hardware and software features. We exposed the disadvantages and limitations of using current benchmarks to characterize systems and infer their performance on workloads different from themselves. We think that our approach will enrich the area of performance evaluation in several ways.

(1) A uniform 'high level' model of the performance of computer systems allow us to make a better comparison 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) We can study the sensitivity of the system to changes in the workload, and in this way detect imbalances in the architectures.

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

(5) 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 hasn't 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.

7.4. Acknowledgements

I want to give special thanks to my research adviser Prof. Alan J. Smith for suggesting this problem and for his patience and continuous support; his valuable comments improved the quality of this report enormously. Thanks are also due to Eugene Miya and Ken G. Stevens Jr. for their comments on earlier drafts of this document and for the provision of resources at NASA Ames, as well as to José A. Ambros-Ingerson and Prof. Domenico Ferrari who made many useful suggestions during discussions.

8
Bibliography

[BAI85a] Bailey, D.H., Barton, J.T., "The NAS Kernel Benchmark Program", NASA Technical Memorandum 86711, August 1985.

[BAI85b] Bailey, D.H., "NAS Kernel Benchmark Results", *Proc. First Int. Conf. on Supercomputing*, St. Petersburg, Florida, December 16-20, 1985, pp. 341-345.

[BAI87] Bailey, D.H., personal (electronic mail) communication.

[BRA86] Bratten, C., Clark, R., Dorn, P., and Grant, R., "IBM 3090: Engineering/Scientific Performance", IBM's Technical Report No. GG66-0245, June, 1986.

[BEE84] Beeler, M., "Beyond the Baskett Benchmark", *Computer Architecture News,* Vol. 1, No. 1, March 1986.

[BRI86] Brickner, R.G., Wasserman, H.J., Hayes, A.H., and Moore, J.W., "Benchmarking the IBM 3090 with Vector Facility", Los Alamos Technical Report No. LA-UR-86-3300, 1986.

[BUC85] Bucher, I.Y., Simmons, M.L., "Performance Assestment of Supercomputers", *Vector and Parallel Processors: Architecture, applications, and Performance Evaluation,* Editor: M. Ginsberg, to be published by North Holland.

[BUC87] Bucher, I.Y., and Simmons, L.M. "A Close Look at Vector Performance of Register-to-Register Vector Computers and a New Model". *ACM Sigmetrics Conference on Modeling and Measurement of Computer Systems*, Banff, Canada, May 1987.

[CLA85] Clark, D.W., and Emer, J.S. "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement", *Transactions on Computer Systems,* Vol. 3, No. 1, February 1985, pp. 31-62.

[CLA86] 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.

[CRA84] *CRAY X-MP and CRAY-1 Library Reference Manual,* SR-0014, December 1984.

[CUR75] Currah B., "Some Causes of Variability in CPU Time", *Computer Measurement and Evaluation,* SHARE project, Vol. 3, 1975, pp. 389-392.

[CUR76] Curnow, H.J., Wichmann, B.A., "A Synthetic Benchmark", *The Computer Journal,* Vol. 19, No.1, February 1976, pp. 43-49.

[DEN80] Denning, P.J., "What is Experimental Computer Science", *Communications of the ACM,* Vol. 23, No. 10, October 1980, pp. 543-544.

[DEN81] Denning, P.J., "Performance Analysis: Experimental Computer Science at Its Best", *Communications of the ACM,* Vol. 24, No. 11, November 1981, pp. 725-727.

[DON85] 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.

[DON87a] Dongarra, J.J., "The Linpack Benchmark: An Explanation", *Supercomputing First International Conference Proceedings, Athens 1987, Lecture Notes in Computer Science 297*, pp. 456-473.

[DON87b] Dongarra, J.J., Martin, J., and Worlton J., "Computer Benchmarking: paths and pitfalls", *Computer*, Vol. 24, No. 7, July 1987, pp. 38-43.

[DON88] 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.

[EME84] 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.

[FEL78] Feldman, S.J., and Wienberger, P.J., "A Portable Fortran 77 Compiler", UNIX 2.2.10 (1981).

[FEL79] Feldman, J.A., and Sutherland, W.R., "Rejuvenating Experimental Computer Science", *Communications of the ACM*, Vol. 24, No. 11, November 1981, pp. 497-502.

[FEO87] Feo, J.T., "An Analysis of the Computational and Parallel Complexity of the Livermore Loops", to appear *Parallel Computing*, 1987.

[FLY72] Flynn, M.J., "Some Computer Organizations and their Effectiveness", *IEEE Transactions on Computers*, **C-21** pp. 948-960 (1972).

[GRA82] 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, pp 120-126, June 1982.

[GRI84] Griffin, J.H., Simmons, M.L., "Los Alamos National Laboratory Computer Benchmarking 1983", Los Alamos Technical Report No. LA-10151-MS, June 1984.

[HOC81] Hockney, R.W. and Jesshope, C.R., *Parallel Computers* (Adam Hilger, Bristol, 1981).

[HOC85] Hockney, R.W., "(r_∞,$n_{1/2}$,$s_{1/2}$) measurements on the 2-CPU CRAY X–MP", *Parallel Computing*, Vol. 2, pp. 1-14 (1985).

[HWA84] Hwang, K. and Briggs, F.A., *Computer architecture and Parallel Processing*, McGraw Hill, New York, 1984.

[IBB82] 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.

[KNU71] Knuth, D.E., "An Empirical Study of FORTRAN Programs", *Software-Practice and Experience*, Vol. 1, pp. 105-133 (1971).

[KNU73] Knuth, D.E., *The Art of Computer Programming:* Vol. 3, *Sorting and Searching*, Addison-Wesley, Reading, Mass, 1973.

[LEE84] Lee, J.K.F., Smith, A.J., "Branch Prediction Strategies and Branch Target Buffer Design", *Computer*, Vol. 17, No. 1, January 1984, pp. 6-22.

[LIN86a] Lindsay, D.S., "Methodology for Determining the Effects of Optimizing Compilers", *CMG 1986 Conference Proceedings*, Las Vegas, Nevada, December 9-12, 1986, pp. 366-373.

[LIN86b] Lindsay, D.S., "Do FORTRAN Compilers Really Optimize", *CMG Transactions*, Spring 1986, pp. 23-27.

[LUB85] Lubeck, O., Moore, J., and Mendez, R., "A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S810.20, and CRAY X-MP/12", *Proceedings of the First International Conference on Supercomputing Systems*, St. Petersburg, Florida, December 16-20, 1985, pp. 320-329.

[MAC84] MacDougall, M.H., "Instruction-Level Program and Processor Modeling", *Computer*, Vol. 7 No. 14, July 1982, pp. 14-24.

[MAR87] Martin, J.L., "Performance Evaluation: Applications and Architectures", *Proc. Second Int. Conf. on Supercomputing*, Vol. III, pp. 369-373.

[MCC79] McCraken, D.D., Denning, P.J., Grandin, D.H., "An ACM Executive Committee Position on the Crisis in Experimental Computer Science", *Communications of the ACM*, Vol. 24, No. 11, November 1981, pp. 503-504.

[MCM86] McMahon, F.H., "The Livermore Fortran Kernels: A Computer Test of the Floating-Point Performance Range", Lawrence Livermore National Laboratory, UCRL-53745, December 1986.

[MER83] Merrill, H.W., "Repeatability and Variability of CPU timing in Large IBM Systems", *CMG Transactions*, Vol. 39, March 1983.

[MIP87] MIPS Computer Systems, "A Sun-4 Benchmark Analysis", July 1987.

[PAT82] Patterson D., "A Performance Evaluation of the Intel 80286", *Computer Architecture News*, Vol. 10, No. 5, September 1982, pp. 16-18.

[PEU77] 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.

[POW83] Power, L.R., "Design and Use of a Program Execution Analyzer", *IBM Systems Journal*, Vol. 22, No.3, pp. 271-292, 1983.

[SHI87] Shimasaki, M., "Performance Analysis of Vector Supercomputers by Hockney's Model", *Proc. Second Int. Conf. on Supercomputing*, Vol. III, pp. 359-368.

[SIM87] Simmons, M.L. and Wasserman H.J., "Los Alamos National Laboratory Computer Benchmarking 1986", Los Alamos National Laboratory, LA-10898-MS, January 1987.

[SMI82] Smith, A.J., "CPU Cache Memories", *ACM Computing Surveys*, Vol. 14, No. 3, September 1982, pp. 473-530.

[SMI88] Smith, A.J., paper in preparation.

[WEI84] Weicker, R.,P., "Dhrystone: A Synthetic Systems Programming Benchmark", *Communications of the ACM*, Vol. 27, No. 10, October 1984.

[WIE82] Wiecek, A.C., "A case Study of VAX-11 Instruction Set Usage for Compiler Execution", *Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, March 1-3, 1982, pp. 177-184.

[WOR84] Worlton, J., "Understanding Supercomputer Benchmarks", *Datamation*, September 1, 1984, pp. 121-130.

9
Appendix

Region 1: Floating Point Arithmetic Operations (single, local)

machine	SRSL 1	ARSL 2	MRSL 3	DRSL 4	ERSL 5	XRSL 6	TRSL 7
CRAY X-MP/48	76	73	145	352	86	5155	276
CYBER 205	1<	73	83	947	168	8844	74
IBM 3090/200	34	74	128	564	236	6358	62
Amdahl 5840	118	341	407	1028	812	42516	420
Convex C1	104	385	514	2668	172	28621	676
VAX 8600	55	421	575	1583	1101	221337	459
VAX-11/785	387	1470	2021	4290	3976	703419	2645
VAX-11/780	547	3601	7310	10135	11422	2104654	2307
IBM RT-PC/125	3821	16480	20525	22974	31633	1265778	6034
SUN 3/50	767	26364	41204	46172	59030	4745964	2843

Region 2: Floating Point Arithmetic Operations (double, local)

machine	SRDL 8	ARDL 9	MRDL 10	DRDL 11	ERDL 12	XRDL 13	TRDL 14
CRAY X-MP/48	57	1128	1900	6821	2575	125534	193
CYBER 205	1<	847	797	3179	3629	55359	60
IBM 3090/200	43	74	137	914	383	23390	68
Amdahl 5840	60	240	432	1541	782	42170	179
Convex C1	199	461	829	4263	139	46303	1110
VAX 8600	193	805	1681	5885	2923	210304	1035
VAX-11/785	1038	3374	5918	9312	10177	651684	5921
VAX-11/780	831	12188	27778	51291	35004	2097689	6103
IBM RT-PC/125	933	15102	16324	18487	27030	1217774	1910
SUN 3/50	1<	56163	101565	136108	114533	5569884	5024

Region 3: Integer Arithmetic Operations (single, local)

machine	SISL 15	AISL 16	MISL 17	DISL 18	EISL 19	XISL 20	TISL 21
CRAY X-MP/48	1<	86	432	713	411	799	307
CYBER 205	1<	39	52	1139	101	221	29
IBM 3090/200	1<	75	148	439	167	354	80
Amdahl 5840	1<	68	223	694	347	636	178
Convex C1	1<	303	272	2661	407	766	627
VAX 8600	1<	345	612	1577	865	1821	422
VAX-11/785	1<	1182	1925	7634	2587	5958	2865
VAX-11/780	1<	1624	7336	11083	7969	16151	2418
IBM RT-PC/125	1<	618	4617	7487	4649	9804	1039
SUN 3/50	1<	292	3620	6825	27699	33418	1<

Table 9.1: Characterization results for regions 1-3. A value 1< indicates that the parameter was not detected by the experiment.

Region 4: Floating Point Arithmetic Operations (single, global)

	SRSG	ARSG	MRSG	DRSG	ERSG	XRSG	TRSG
machine	22	23	24	25	26	27	28
CRAY X-MP/48	83	72	139	348	85	5169	273
CYBER 205	160	354	470	1382	312	8539	160
IBM 3090/200	37	76	139	568	246	5370	97
Amdahl 5840	118	339	409	1033	818	43108	431
Convex C1	74	396	523	2659	186	27985	648
VAX 8600	88	553	698	1629	1060	220643	445
VAX-11/785	486	1445	2004	4205	3871	675706	2311
VAX-11/780	493	3628	7415	10148	11443	2091213	2322
IBM RT-PC/125	3820	16776	21058	23794	31329	1260805	5420
SUN 3/50	215	26519	39694	47488	61871	4778238	3401

Region 5: Floating Point Arithmetic Operations (double, global)

	SRDG	ARDG	MRDG	DRDG	ERDG	XRDG	TRDG
machine	29	30	31	32	33	34	35
CRAY X-MP/48	44	1133	1902	6777	2440	125841	204
CYBER 205	201	1252	1381	3785	3798	55516	235
IBM 3090/200	41	76	136	914	288	22404	98
Amdahl 5840	66	238	427	1552	795	42703	164
Convex C1	236	441	798	4319	94	46083	1091
VAX 8600	113	851	1714	5887	2800	220449	974
VAX-11/785	1112	2419	4709	8309	9575	648899	4942
VAX-11/780	837	11198	25783	49574	34300	2067708	5422
IBM RT-PC/125	516	15400	17084	19113	27205	1216349	2608
SUN 3/50	11312	52740	96362	134009	112031	5625248	1<

Region 6: Integer Arithmetic Operations (single, global)

	SISG	AISG	MISG	DISG	EISG	XISG	TISG
machine	36	37	38	39	40	41	42
CRAY X-MP/48	1<	86	432	709	407	795	307
CYBER 205	1<	397	188	1492	346	467	201
IBM 3090/200	1<	82	145	442	200	545	103
Amdahl 5840	1<	74	221	697	350	632	173
Convex C1	1<	299	276	2651	397	768	624
VAX 8600	1<	535	615	1585	960	1851	558
VAX-11/785	1<	1185	1871	7653	2780	5984	2882
VAX-11/780	1<	1622	7343	11063	7912	16205	2415
IBM RT-PC/125	1<	987	4894	7801	4767	10063	1145
SUN 3/50	1<	420	3220	6642	28881	33728	854

Region 7: Conditional and Logical Parameters

	ANDL	CRSL	CRDL	CISL	ANDG	CRSG	CRDG	CISG
machine	43	44	45	46	47	48	49	50
CRAY X-MP/48	52	207	1247	208	54	206	1257	202
CYBER 205	46	167	447	87	273	628	1074	548
IBM 3090/200	122	124	130	118	125	141	258	155
Amdahl 5840	110	477	284	151	116	478	236	150
Convex C1	302	515	800	499	300	502	806	507
VAX 8600	300	644	860	407	318	921	966	779
VAX-11/785	899	2138	2641	1670	895	2249	3025	1670
VAX-11/780	1170	3161	4068	2388	1167	3003	5023	2305
IBM RT-PC/125	583	11820	11950	1281	674	12292	11920	1643
SUN 3/50	568	13372	22004	549	1158	14083	25556	1615

Table 9.2: Characterization results for regions 4-7. A value 1< indicates that the parameter was not detected by the experiment.

Regions 8, 9: Function Call, Arguments and References to Array Elements

machine	PROC 51	AGRS 52	AGRD 53	AGIS 54	ARR1 55	ARR2 56	ARR3 57	IADD 58
CRAY X-MP/48	590	165	214	170	55	99	142	1<
CYBER 205	5322	521	560	802	160	330	901	6
IBM 3090/200	1242	179	193	201	75	322	565	7
Amdahl 5840	1089	109	89	75	176	471	715	5
Convex C1	5270	74	348	140	276	579	969	1<
VAX 8600	5849	512	674	597	373	1322	2726	1<
VAX-11/785	20808	692	1978	1709	850	3613	6046	1<
VAX-11/780	21716	3165	3079	2907	1489	11829	20263	1<
IBM RT-PC/125	7014	1078	428	397	987	2612	3566	96
SUN 3/50	8588	15164	15326	6402	2085	2452	6328	937

Region 10: Branching and DO loop Parameters

machine	GOTO 59	GCOM 60	LOIN 61	LOOV 62	LOIX 63	LOOX 64
CRAY X-MP/48	23	451	693	240	473	265
CYBER 205	20	1000	839	176	1542	200
IBM 3090/200	41	234	654	197	754	243
Amdahl 5840	29	—	—	666	—	—
Convex C1	1<	1531	3470	722	2979	1402
VAX 8600	163	2523	4669	550	3682	1386
VAX-11/785	1<	3302	18123	1384	1<	3894
VAX-11/780	469	6974	7085	5139	9793	4825
IBM RT-PC/125	45	6230	4444	2039	8641	3154
SUN 3/50	395	2957	6237	940	7599	2429

Region 11: Intrinsic Functions (single precision)

machine	EXPS 65	LOGS 66	SINS 67	TANS 68	SQRS 69
CRAY X-MP/48	1864	1647	1849	2046	1404
CYBER 205	6116	5249	5212	6651	259
IBM 3090/200	5434	5287	4865	5987	4323
Amdahl 5840	—	—	—	—	—
Convex C1	11693	9068	7455	11106	11166
VAX 8600	67253	76579	42078	69678	24109
VAX-11/785	205647	239165	111996	181389	59729
VAX-11/780	728008	809876	487725	556329	186099
IBM RT-PC/125	65016	59481	60181	64385	43299
SUN 3/50	769844	955845	1278529	1525150	87532

Region 12: Intrinsic Functions (double precision)

machine	EXPD 70	LOGD 71	SIND 72	TAND 73	SQRD 74
CRAY X-MP/48	62927	58226	39906	88232	11013
CYBER 205	28372	25696	32122	41739	6484
IBM 3090/200	25433	23989	22542	28711	19432
Amdahl 5840	—	—	—	—	—
Convex C1	20007	15684	12801	17866	21023
VAX 8600	66552	76417	41673	69408	23608
VAX-11/785	200483	240539	111825	181420	58633
VAX-11/780	739768	816049	492781	562209	188379
IBM RT-PC/125	46947	41634	38524	42439	23228
SUN 3/50	2099322	2355612	2225962	2447243	176096

Table 9.3: Characterization results for regions 8-12. A value 1< indicates that the parameter was not detected by the experiment. The results for the Amdahl 5840 were obtained using a simpler model.