Prediction of Software Execution Time Based on Software Metrics

By

Ying N. Li

B.A.Sc., University of British Columbia, 1995

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

In

The Faculty of Graduate Studies

Department of Electrical and Computer Engineering

We accept this thesis as conforming to the required standard:

The University of British Columbia

December 1998

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electronic and Computer Engineering

The University of British Columbia
Vancouver, Canada

Date Dec 18/97

Abstract

A method to estimate the execution time of software based on static metrics is proposed in this thesis. The ability to produce an accurate estimate of execution times as early as possible in the development phase is highly desirable. For hard real-time systems, an extremely slow function may require an entire system to be redesigned. In the proposed method, principal components and linear regression modeling are used to formulate a model from a given set of representative functions. It is assumed that all functions are programmed in a structured manner. The final result is a model that can be used to generate decent first approximations of execution times. Once the model is established, it is used to predict the execution times of other test functions. The major problem encountered in the modeling is the indeterminate nature of loops in a function. The number of times a loop structure is executed is dependent on the input parameters. It is a dynamic characteristic and is impossible to measure with static metrics. Our solution is to expand the source code by the number of times the loop is expected to execute. Then, the metrics are taken from the expanded code. Extremely high correlations were observed between the actual and the predicted execution times with the exception of fast functions. It appears that the metrics used were insufficient for fast functions. This method seems to work even across different execution platforms and languages. Though, this claim is requires further investigation.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I like to thank my father who started me on the long road of learning. Also, I wish to express my appreciation to my professors at the University of British Columbia for their patience and thought provoking ideas. In researching this thesis, I have received invaluable assistance from Dr. H. Jin throughout the past few years. Additionally, special thanks is due to Dr. G. Bond for his constant advice and guidance.

Financial support from Nortel and the BC Advanced Systems Institute is acknowledged with gratitude.

Chapter 1: Introduction

The objective of this work is to estimate the execution time of a structured function. The capability of predicting the execution time prior to compiling and running a program is a valuable tool in software development. It helps software developers to identify the slow modules of a program. The performance of these bottleneck modules can then be improved. This is particularly important in real time systems where timing is critical. The proposed technique can be used at a very early stage of development. Not only will the results pinpoint problem modules, it can also aid in determining the hardware needs of the system.

1.1 Background and Motivation

Prior research has shown that efforts directed at preventing, detecting, and correcting software defects could significantly reduce the total software development costs [Basili 84, Boehm 81]. This is not surprising as the earlier an error can be found and corrected, the less effect it has on the rest of the system. This applies equally to real-time systems. In addition to the above, in real-time systems, timing is an important component of the requirements. For example, failure to meet timing requirements in hard real-time systems may result in the failure of a project or even be life threatening. As companies are striving to improve product performance while reducing the development cost and time to market, there have been great interest and research activities in the general area of software

performance evaluation. This explains the interest in time estimation, which is a part of performance evaluation.

1.2 Review of Existing Work

Many techniques exist for predicting the execution time of programs at the assembly or object code level. However, very few techniques are available at the structured source code level. In many cases, having a high level technique is important and useful since compilers and/or platforms may not be accessible or available when needed. However, once source code is available, it is possible to extract information from the code using software metrics. Then, this information can be used to estimate the execution time of the source code. Although various software code metrics have been applied in areas such as quality control and error detection, with the exception of Tempus Project [Tempus 97], we are not aware of any other work in progress attempting to relate metrics to the running time of programs. This work proposes to use software code metrics to predict the execution time of functions.

1.2.1 Low Level Techniques

Low level techniques are prediction methods based on the machine or object code. Almost all the techniques are based on the idea that each operation takes a certain amount of time and the execution time for a function is calculated by summing up the execution time of each instructions in that function. The question of which instruction will be executed is answered either by static annotations [Puschner 89] or by dynamic profiling of the code

[Gupta 94]. Also, there are many papers, such as [Arnold 94], that examine the effect of pipelining and caches on the prediction.

There are both advantages and disadvantages with low level techniques. The main advantage is that very accurate results can be achieved with these techniques. The more information that is used in forming the model, the better is the prediction. However, errors are still introduced into the models from several sources. Cache hits and misses are one source of error. The processor is another source of error. Manufacturers are not always willing to provide all the details needed to determine the exact time for an operation. Furthermore, with pipeline flushes and dynamic branch prediction in the processor, it is not possible to have an exact time for a given operation. The main disadvantage of these models is the amount of information that is required. Generally, the entire system is modeled right down to the microinstructions used by the CPU. In addition, this can only be accomplished after the code is compiled and all the hardware is finalized. This is very late in the product cycle relative to code development.

1.2.2 Complexity Analysis

Work has been done in software complexity analysis using metrics. It is self-evident that programs that are more complex will contain more faults and will be more difficult to maintain. For example, the number of lines in a program is commonly used as one indication of program complexity. One would view a program with a thousand lines differently from one containing a million lines of code.

Munson and Khoshgoftaar proposed the creation of a complexity measure for use in reliability modeling [Munson 92]. They used the statistical technique of factor analysis, which is an elaborate form of principal component analysis, on various software metrics. Factor analysis is used to determine the major components of the metrics that are correlated with reliability. In other works, the authors show that complexity measure is strongly predictive of the number of faults in a software system.

While software metrics have been successfully used for complexity analysis, we believe that these metrics also contain information about the execution time of the structured code. For example, code that contains many branches, as indicated by the cyclomatic complexity metrics, would execute faster than code of the same size that does not contain branching instructions.

1.2.3 Tempus Project

Software Productivity Solutions, Inc. has developed a software measure, τ, that is claimed to predict the execution time of real-time systems [Tempus 97]. The dimensional unit for τ is number of run-time operations, which makes it a counting metric. The source code is translated into a machine independent op-code format. Every op-code is assigned a value and the final metric value equals the sum of the op-code along the execution path. The final metric is again weighted by the executing architecture. They have computed a Coefficient of Determination, R^2, between execution time and the software measure τ to

be 0.9511. The Coefficient of Determination measures the closeness of the relationship between the two values with a maximum value of unity. Therefore, their result of 0.9511 shows a strong relationship between execution time and the software measure.

However, there is a lack of information to evaluate this approach, as is this a commercial product. For instance, little information is given about the test data except in one of the graphics. In [Tempus 97], six series are labeled in a diagram: odd, factorial, term, sin, cos, and tan. Although it is claimed that these procedures are for real-time systems, it appears to us that these are mathematical functions with regular predictable execution patterns thus making these particular functions easier to predict. For example, reducing the acceptable error when calculating sin x would increase the execution time. Given the algorithm used to find sin x, the increased execution time can be easily predicted.

1.3 Thesis Objective

The objective of this thesis is to develop a method of predicting the execution time of structured functions based on static software metrics. This is the earliest stage in which a quantitative prediction can be done. Before the source code is produced, there exist only a timing requirement that has to be meet. Only after the source code is produced can quantitative measures be taken.

In many situations, software code is not ready to be executed. This can be due to many factors. For example, the hardware and the compilers might not be determined or

available, or other software modules are not ready for integration. Yet, the timing information is needed as soon as possible. The combined timing information of several modules can be used to insure that the whole system meets all of its timing requirements. If a module goes over its timing limit, it must be optimized or the system must be rescheduled.

To accomplish this, it is necessary to have some quantitative measures of the source code, and software metrics can be used for this purpose. The objectives of this work are, therefore, to determine the correlation between the software metrics and the execution time, to develop the methodology of predicting the execution time of source code, and to validate the proposed technique.

1.4 Thesis Organization

This thesis will be organized in the following way. Chapter 2 examines the basic concept of software metrics in detail. Chapter 3 explains our approach. In Chapter 4 the results are presented and discussed, and the conclusions are summarized in Chapter 5.

Chapter 2: Software Metrics

Software metrics grew out of the need in software engineering for quantitative measurements of software code. These quantitative measurements or metrics are used in answering several important questions such as the differences in complexity, error rate, maintainability, production time, and testability between two software programs. At the beginning, simple metrics like lines of code dominated. In 1977, Halstead published [Halstead 77] and introduced the software science metrics. A year before that, McCabe wrote [McCabe 76], which defined the cyclomatic complexity metrics. Now various hybrid metrics exist. This chapter introduces the basics of the commonly used metrics, and those that are the used in our work to predict the execution time of functions.

2.1 Lines of Code

The first set of simple metrics involved counting the number of lines in a function. The simplest metric is lines of code (LOC). It is claimed in literature that, based on LOC, it is possible to predict the reliability and maintainability of code. Due largely to its simplicity, this metric is widely used with several variances. The variances differ in what is considered a line of code. Some measure only source statements and others include comment and blank lines. One problem with LOC is its dependency on individual programmers. Programmers differ in how they format their code. Each has an individual style to writing source code. LOC is also dependent on the language used. Some languages are more

expressive than others are. The same thing can be accomplished in one line in one language that would take several tens of lines in another.

In UX-Metrics, a software used in our work to collect metrics from C and C++ programs, there are four metrics that relate to the number of lines of code. They are lines of code, physical source statements, logical source statements, and executable semi colons. In this case, a line of code is defined as a count of every physical line of code from the start of the function to the end of the function, including comments and blanks. Physical source statements are similar but exclude comments and blanks. Logical source statements are counted in only two cases. It is increased when executable statements end with a semi colon and for all control statements, like "else." Lastly, executable semi colon is a count of the number of executable semi colons in the code. Therefore, the "for" statement would have two executable semi colons. As discussed, all four variant metrics are very similar.

2.2 Software Science Metrics

In 1977, Maurice Halstead of Purdue University published his work on software metrics [Halstead 77]. He recognized that LOC is not accurate and developed a different theory to model program size. Based on his theory, programs consist of the manipulation of tokens: operators and operands. Operands are variables and constants. On the other hand, operators are the actions on the operands. From these basic tokens, he derived a large number of metrics.

2.2.1 Basic Tokens

Halstead formulated several metrics based on the number of operators and operands in a program based on four parameters:

η_1: number of unique operators

η_2: number of unique operands

N_1: total number of operators

N_2: total number of operands

The program vocabulary, η, is defined as:

$$\eta = \eta_1 + \eta_2 \tag{2.1}$$

The program length, N, is defined as:

$$N = N_1 + N_2 \tag{2.2}$$

Originally, Halstead did not include precise definition of operators and operands and relied on intuitive understanding. In many cases, what is counted as operators and operands are dependent on the user. For example, "goto x" can be taken as one operator and one operand, or it can be viewed as only a single operator. However, most work in the area suggests that minor differences are immaterial as long as the counting procedures are

consistent [Currans 86]. Also, Halstead felt only the executed part of the code is important and therefore did not include function declarations in his metrics.

2.2.2 Length Equation

The predicted length of the program ($\tilde{N}$) is defined as:

$$\tilde{N} = \eta_1 \log_2(\eta_1) + \eta_2 \log_2(\eta_2) \tag{2.3}$$

This interesting relationship is available before the code is completed since it depends only on η_1 and η_2. Usually, the number of unique operators in a language is constant and the number of unique operands can be obtained from the design specifications. High correlations have been shown between the actual measured length, N, and the predicted length, $\tilde{N}$. Correlations above 0.90 are common. Halstead suggested that the differences between N and $\tilde{N}$ are due to impurities in the coding. Impurities are due to poorly written algorithms and is measured with the purity ratio defined as:

$$P_R = \tilde{N} / N \tag{2.4}$$

2.2.3 Program Volume

Another interesting metric is the program volume, V that is defined as:

$$V = N \log_2(\eta) \tag{2.5}$$

Halstead gave two reasons for this metric. First, given that a program has η unique tokens, it would take $\log_2(\eta)$ bits to represent them all. With N as the total number of operators and operands, it would take V bits to represent the whole program. Alternatively, to understand a program with a program length of N and a vocabulary of η, it would take one mental look-up for each N. Assuming the look-up is done in a binary manner ($\log_2 \eta$), the total time to understand the program would be given by the volume metric, V.

2.2.4 Effort

This metric is based on the Volume metric, V. However, it is adjusted based on the abstraction level of the program, λ. Unfortunately, the abstraction level of a program is difficult to determine. It is usually estimated with:

$$\lambda = 2/\eta_1 \cdot \eta_2/N_2 \tag{2.6}$$

The highest possible value for λ is unity. The abstraction level depends on two terms. The first term, $2/\eta_1$, decreases as the number of unique operators increases. Fewer operators imply a more abstract program. The two in the expression $2/\eta_1$ comes from the fact that, a minimum of two operators are required to implement a function, the function call and its

argument. The second term, η_2/N_2, is the average of the number of operands, inverted. Again, each time the operand is used, the abstraction level decreases. This leads to the definition of effect as:

$$E = V / \lambda \tag{2.7}$$

2.3 Cyclomatic Complexity

The idea of cyclomatic complexity was first presented by Tom McCabe in 1976 [McCabe 76]. Whereas the Software Science metrics measure the size of the code, cyclomatic complexity measures control flows. The metric itself is extracted from the control flow graph. This graph is built with basic blocks of code being represented with nodes, n, and branch statements with directed edges, e. A basic block is a section of sequentially executed code. There is no branching within the block. Decision points or branch statements are represented with directed edges. These would include conditional statements and loop structures. The directed edges connect one basic block to another depending on which block is executed if that branch is taken in the code. Cyclomatic complexity, $V(g)$, is defined as:

$$V(g) = e - n + 2 \tag{2.8}$$

Only a year after McCabe published his paper, Glenford Myers suggested an extension on this metric. The extended cyclomatic complexity is defined similar to McCabe's. Instead of

having each branch statement represented with one edge, each edge represents only one simple predicate. A predicate is a condition without the AND/OR operators. Therefore, for every AND/OR operators, the extended cyclomatic complexity is increased by one. Thus, depending on the complexity of the conditional statement, each branch can be represented with more than one directed edge. This is claimed to be a more accurate metric than the standard cyclomatic complexity.

2.4 Miscellaneous Metrics

In addition to the above defined metrics, UX-Metrics uses three other metrics: span of reference, nesting depth, and average nesting depth. Span of reference is defined as the average of the maximum number of lines between variable references. A reference can be either the definition or the use of a variable. The maximum number of lines between references for each variable is calculated. These are then averaged for a function and reported as the metric. Nesting depth is the number of nesting control structures. Every nesting control structure increases this metric by one. Finally, the average nesting depth is the depth of each logical statement divided by the number of logical statements.

2.5 Hybrid Metrics

Hybrid metrics are those that combine one or more basic metrics. There are well over 100 distinct metrics of software complexity in the research literature [Zuse 90]. Each is slightly different and has its own proponents proclaiming its predictive abilities. In essence, they

combine some aspect of Halstead's or McCabe's metrics. One of the most notable variants of hybrid metrics is presented by Munson and Koshgoftaar [Munson 89, 90a, 90b, 92, 93]. Using a form of principal component analysis, they combine various commonly used metrics into a relative complexity metric. They consider these combined metrics to be more reflective of software complexity.

2.6 Software Metrics and Execution Times

It is evident that some metric can contribute to estimating the execution time of functions. The LOC metrics show how many lines are in each function. Functions with many lines will tend to take longer to execute. A similar statement can be made about the metrics that Halstead defined, even though, the exact relationship between his metrics and the execution time of each function is difficult to determine. Similarly, cyclomatic complexities are branch points in the code. The more branch points there are in the code, the faster the code will execute since branching causes the code to execute fewer statements.

The most interesting idea is presented by the hybrid metrics. As stated before, the exact relationship between the software metrics and the execution time of a function is unknown. However, by using Munson's idea, it is possible to generate principal components that are more predictive of execution time than the individual metrics. In addition, it would be simpler to define and reason with the few domains that are represented by the principal components than with the raw metrics. This approach is adopted in this thesis.

2.7 Summary

In this chapter, many of the basic software metrics have been examined. It starts with lines of code then more advanced metrics, such as the software science metrics and cyclomatic complexity, are discussed. Munson and Koshgoftaar presented a method of combining these metrics into a relative complexity metric using a form of principal components. Though they primarily used the principal component method in complexity analysis, we adopt the statistic method in our proposed approach to predicting execution time at the function level.

Chapter 3: Proposed Approach to Time Estimation

The proposed approach to time estimation is presented in this chapter. It uses software metrics as discussed in the previous chapter, to formulate a linear regression model. This model can be used for predicting the execution time of other functions. The first section of this chapter outlines the basic approach. This is followed by more information of how the model is formed and verified. Section 3.4 explains the limitations of this method.

3.1 Proposed Approach

The proposed approach is to predict execution time of structured functions based on the static software metrics of those functions. This is carried out in two steps. First, a model is formulated, by linear regression, from a standard program with both the metrics and execution time information already available. This will determine the regression coefficients of the metrics versus the execution time. Once the coefficients are determined, they are used to predict the execution time of other functions based on their metric values.

When the model is formulated from the standard program, there are two main problems if the raw metrics are used in the regression. First, there is a large difference in scale of the different metrics. Secondly, collinearity exists in many of the metrics. Collinearity happens when metrics measure similar properties. The scale problem is solved by standardization of the raw metrics. To standardize individual metric values, each is subtracted from the mean

and then divided by the standard deviation for that particular metric type. The second problem of collinearity is solved using principal components analysis. After the raw metrics are transformed, first by standardization and then by principal component analysis, they can be used in linear regression.

Once the measured execution times are provided and the principal components are calculated for the standard program, the linear regression can be carried out. One condition for selecting the standard program is that the functions in the standard program should be representative of the other functions that the resulting model will be used to predict. This is, different types of programs have different metric characteristics and this fact must be taken into account. For example, computational based programs have more intensively executed loops as compared to real-time programs. The other requirement for the standard program is that the execution times of each function must be known. The requirement can be met by selecting the standard program from previously completed projects, or from benchmark cases if available.

After the model is formed, other functions can be easily predicted. First, raw metrics are extracted from the functions. Then, the exact same procedures of standardization and principal component transformation are applied. This means that both the standard functions and the functions to be predicted are the same statistically, since the same operations are applied to both in order. Once the transformations are applied, predictions are generated by multiplying the result of the transformations by the regression coefficient obtained from the model.

To summarize, the proposed procedure to calculate the regression model is:

- Choose an existing set of functions as a standard. The standard functions should be representative of the type of programs to be tested. Extract and standardized the metrics for each function and stored the values in matrix defined as **Z**. For the convenience of discussion, assume that the program has 75 functions and 16 metrics are extracted, **Z** would be a 75x16 matrix.
- Calculate the correlation matrix ρ of the matrix **Z**. The matrix ρ will be 16x16.
- Calculate the 16x16 normalized eigenvector matrix **E** from the matrix ρ. Each column of the matrix **E** is a normalized eigenvector of the matrix ρ.
- The principal component matrix $\mathbf{P_C}$ is calculated based on:

$$\mathbf{P_C} = \mathbf{Z} \cdot \mathbf{E} \tag{3.1}$$

  $\mathbf{P_C}$ has a dimension of 75x16.
- Given the actual execution time of each function in the program in a column vector, **x**, calculate the regression coefficients β using linear regression techniques.

Once the regression coefficients β are found, we can use them to estimate the time of other programs following these procedures:

- Extract the metrics of the functions to be tested, and standardize these metrics. This would be the same transformation applied to matrix **Z** above. Then record

the standardized values in matrix $\mathbf{Z}_1$. The number of rows of $\mathbf{Z}_1$ is the same as the number of functions being tested.

- $\mathbf{Z}_1$ is transformed into principal components by

$$\mathbf{P}_{C1} = \mathbf{Z}_1 \cdot \mathbf{E} \tag{3.2}$$

- The estimated time $\mathbf{Y}_1$ is calculated as in (3.3). A column vector of ones is added for the intercept term.

$$\mathbf{Y}_1 = [\mathbf{1} \mid \mathbf{P}_{C1}] \cdot \beta \tag{3.3}$$

3.2 Model Formulation

This section will present the statistical techniques used in this thesis to formulate the model: principal component analysis and multiple linear regression. These can be found in most statistical text such as [Johnson 92]. The primary goal in principal component analysis is to find the appropriate variables, that is, to represent the entire set of variables by only a few variable domains. On the other hand, multiple linear regression is concerned with finding the relationship between a response variable, the execution time, and multiple predictor variables, the transformed metrics. Both will be used to formulate the model for the prediction of execution time.

3.2.1 Principal Component Analysis

Principal component analysis is a data transformation technique that provides the transformed variables with useful statistical properties. In this case, it is use to reduce collinearity and reduce the number of predictor variables for multiple linear regression. Collinearity happens when two variables are closely related in the linear sense. After the transformation, the variables have the property of being uncorrelated with each other. They will be used in multiple linear regression to predict the execution time of functions.

Principal component analysis can be done with either the covariance or correlation matrices. The results are two completely different solutions, one from using the covariance and another from using the correlation matrix. This is due to the scale sensitivity of the covariance matrix. When the data set is in different units, as in software metrics, the correlation matrix is generally employed. In addition, with standardized data, the covariance matrix is identical to the correlation matrix. In the treatment below, correlation matrices will be use. However, identical procedures also work for the covariance matrices. The correlation matrix is defined as:

$$\rho = \begin{bmatrix} 1 & \rho_{12} & \cdots & \rho_{1p} \\ \rho_{21} & 1 & \cdots & \rho_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{p1} & \rho_{p2} & \cdots & 1 \end{bmatrix} \qquad (3.4)$$

Where ρ_{ij} is the simple correlation between metrics i and j. The eigenvector and

eigenvalue is then extracted from ρ. Assuming the standardized data is in **Z** with data points are represented in rows with columns representing different metrics then the transformed data, $\mathbf{P_C}$, is defined as:

$$\mathbf{P_C} = \mathbf{Z} [\mathbf{e_1} \mathbf{e_2} \ldots \mathbf{e_k}] \quad (3.5)$$

Where $\mathbf{e_i}$ is the i^{th} eigenvector of ρ. If the rank of ρ is p then the maximum value for k is p. However, a few eigenvectors of ρ is usually dropped because they are insignificant. Therefore, k is usually less than p. One rule of thumb is to drop all eigenvectors when its corresponding eigenvalue is below unity. There is no hard justification for this except that an eigenvalue of less than unity implies the transformed variable explains less variance than one of the original variables.

3.2.2 Multiple Linear Regression

Multiple linear regression is concerned with finding a relationship between a response (independent) variable and multiple predictor (dependent) variables. It will be used here to find the relationship between the execution time of functions and software metrics. In this case, only one response variable is needed to represent time for each function. Therefore, the linear regression model for the i^{th} function is:

$$y_i = \beta_0 x_{i,0} + \beta_1 x_{i,1} + \ldots + \beta_k x_{i,k} + \varepsilon_i \quad (3.6)$$

Where y_i is the actual execution time, β_0 to β_k are the regression coefficients, $x_{i,0}$ is the constant 1, $x_{i,1}$ to $x_{i,k}$ are the principal component values, and ε_i is the error. In other words, the response equals to the sum of some factor of each predictor variable and error. The term linear is used because the parameters $\beta_0, \beta_1, \ldots, \beta_k$ are linear. In matrix form (3.6) is:

$$\mathbf{y} = \mathbf{x}\beta + \varepsilon \tag{3.7}$$

Where $\beta = [\, \beta_0, \beta_1, \ldots, \beta_k \,]$.

The method of least square error is usually employed to find the value of the parameters β. The least square function is:

$$S(\beta_0, \beta_1, \ldots, \beta_k) = \Sigma\, \varepsilon_i^2 \tag{3.8}$$

The summation is done over all the functions used in the model formation. The idea is find $\beta_0, \beta_1, \ldots, \beta_k$ which minimizes the total square error of all the data points. Taking the partial derivative of (3.8) and setting it to zero will achieve this. The result is the least square estimation for the parameters $\beta_0, \beta_1, \ldots, \beta_k$. In matrix form:

$$S(\beta) = \varepsilon^T\varepsilon = (\mathbf{y} - \mathbf{x}\beta)^T(\mathbf{y} - \mathbf{x}\beta) \tag{3.9}$$

Rearranging (3.9) gives:

$$S(\beta) = \mathbf{y}^T \mathbf{y} - \beta^T \mathbf{x}^T \mathbf{y} - \mathbf{y}^T \mathbf{x}\beta + \beta^T \mathbf{x}^T \mathbf{x}\,\beta$$

$$= \mathbf{y}^T \mathbf{y} - 2\beta^T \mathbf{x}^T \mathbf{y} + \beta^T \mathbf{x}^T \mathbf{x}\beta \qquad (3.10)$$

Taking the derivative of (3.10) with respect to β and setting to zero gives:

$$\mathbf{-2x^T y + 2x^T x\beta = 0} \qquad (3.11)$$

Solving (3.11) for β gives:

$$\beta = (\mathbf{x}^T \mathbf{x})^{-1}\mathbf{x}^T \mathbf{y} \qquad (3.12)$$

In this case, (3.12) gives the parameters of the regression model based on the metrics and execution times of the standard functions.

3.3 Model Verification

After a model is obtained, it is necessary to check if the model is valid. The best way is to examine the results graphically. Any patterns in the residual plots would indicate if a systematic error exists. Residual is the difference between the predicted and actual value. Besides the residual plots, there are various numerical tests that indicate the validity of the model.

3.3.1 Significance of the Whole Regression Model

This test determines if (3.7) is a valid model for the data being studied. The correct hypotheses are:

$$H_0: \quad \beta = \mathbf{0}$$
$$H_1: \quad \beta \neq \mathbf{0} \tag{3.13}$$

If the null hypothesis, H_0, is rejected, then at least one of $\beta_0, \beta_1, \ldots, \beta_k$ is not zero. In other words, at least one of the parameters is useful. On the other hand, if H_1 is rejected in favor of the null hypothesis then none of the parameters are useful and the model is meaningless. The null hypothesis is rejected if the value of F_0, defined below, is greater than the F-value. The F-value can be obtained from a chart at a significance level of α and degrees of freedom k+1 and n-k-2, where the number of predictor variables is k and the total number of data points is n, and α is the confidence or tolerance level. The value F_0 is defined as below:

$$F_0 = \text{MSR/MSE} \tag{3.14}$$

F_0 is a ratio of MSR (mean square of regression) and MSE (mean square error). MSR is the sum of square due to regression and MSE is the sum of square of the residual each divided by its corresponding degrees of freedom.

$$MSR = SSR / (k+1) \tag{3.15}$$

$$MSE = SSE / (n-k-2) \tag{3.16}$$

The sums of squares for the analysis of variance are:

$$SSR = \beta^T \mathbf{x}^T \mathbf{y} - nY^2 \tag{3.17}$$

$$SSE = \mathbf{e}^T \mathbf{e} = \mathbf{y}^T \mathbf{y} - \beta^T \mathbf{x}^T \mathbf{y} \tag{3.18}$$

$$SST = SSR + SSE = \mathbf{y}^T \mathbf{y} - nY^2 \tag{3.19}$$

Where Y^2 is the square of the mean of the response variable, SST is the sum of square of total variance and is composed of two parts, SSR and SSE. It is the sum of all the observed values of the response variable squared and corrected for the mean value.

3.3.2 Test for Individual Regression Coefficients

It is generally necessary to test individual predictor variables to determine whether they contribute to the regression model. A model could be more or less efficient with the inclusion or exclusion of one of the predictor variables. The appropriate hypotheses for testing an individual β_j is:

$$H_0: \quad \beta_j = 0$$

$$H_1: \quad \beta_j \neq 0 \tag{3.20}$$

If the null hypothesis is not rejected then it suggests that the j^{th} predictor variable can be removed from the model. The test for this is:

$$t_0 = \frac{\beta_j}{\sqrt{\delta^2 \cdot C_{jj}}} \tag{3.21}$$

Where δ^2 is $(\varepsilon^T\varepsilon)/(n\text{-}k\text{-}1)$ and C_{jj} is the diagonal element of $(\mathbf{x}^T\mathbf{x})^{-1}$ corresponding to β_j. Basically, the t-value is found by dividing each β_j by its corresponding standard error. If $|t_0| > t_{(\alpha/2, n-k-1)}$ then H_0 can be rejected in favor of H_1. It is more accurate to use the simultaneous confidence interval and the F-test. Namely, if $|t_0| > (k+1) \cdot F_{(\alpha, k+1, n-k-1)}$, then H_0 can be rejected in favor of H_1. However, most practitioners use the t-test for checking individual regression coefficients.

3.3.3 Coefficient of Multiple Determination

The coefficient of multiple determination, denoted R^2, measures the reduction in the variability of the prediction obtained by using the predictor variables. It is defined as:

$$R^2 = \text{SSR/SST} = 1 - (\text{SSE/SST}) \tag{3.22}$$

SSR, SSE, and SST are defined in (3.17) to (3.19). Clearly, R^2 will always fall between zero and one inclusively. This is the same R^2 that was obtained from the correlation between the observed and predicted values of the response variable. Unfortunately, R^2

near unity does not necessarily imply that the model is good. Adding more predictor variables always increases R^2 and never reduces it. This is because SST is constant for given set of responses and SSR always increases with additional predictor variables. Therefore, an adjusted coefficient of multiple determination is also defined as:

$$\text{adjusted } R^2 = 1 - (n\text{-}1)/(n\text{-}k\text{-}1)(\text{SSE/SST}) \qquad [3.21]$$

3.4 Limitations of the Approach

The main limitation with predicting the execution time of a structured program from source code alone is loop structures. It is impossible to determine the number of times a loop is executed without executing the whole program. At present, the metrics used do not take into account loop structures. To test our hypothesis, we remove each loop structure and replace it with equivalent code. With the equivalent code used for testing, the prediction errors are dramatically reduced. This is referred to as expanding the code.

Some programming languages require a loop bound on each loop structure. This can be an explicit part of each loop structure or a type bound on the index variable. Therefore, to apply our technique, either the upper bounds or the average number of times a loop is executed must be provided.

Unlike loop structures, branching instructions present little problems. It appears that the various metrics like cyclomatic complexity account nicely for the branching effect. Of

course, the programmer could supply some information concerning each branch, for example, the percentage that the true branch of an if statement is taken. Fortunately, this does not appear necessary, since the majority of the error in the prediction is from loop structures.

3.5 Summary

This chapter describes the technique and procedure of predicting the execution time of a structured program using software metrics. Software metrics are problematic from a statistical viewpoint since many exhibit large differences in scale and have collinearity. Therefore, it is necessary to transform the data with standardization and then principal component analysis. After the transformation, the principal components are used in linear regression. The resulting model can be used to predict the execution time of other structured functions. Equation (3.7) givens the general regression model and (3.12) shows how the parameters are chosen. As discussed, the model can be validated in a number of ways. Prediction on other functions can be made by applying the same transformations and then applying the regression model, as shown in (3.3).

Chapter 4: Results and Discussion

In this chapter, the technique proposed in Chapter 3 is applied to a number of test cases. First, the software tools employed are examined. Then, detail on the test data is given. Following the proposed method outlined, a model is formulated. Finally, various aspects of the modeling are discussed, and the results presented.

4.1 Software Tools

Three major software packages are used to extract the various metrics from the test programs. They are GCT, UX-Metrics, and Quantify.

Brian Marick of the Testing Foundation wrote the generic coverage tool, GCT for C programs. It is a C preprocessor. It inserts tags into each possible branch of the program. When the instrumented program is executed, the path taken by the program is recorded by the tags along the execution path. Note that the instrumented program runs significantly slower than uninstrumented versions of the program. By examining the tags after a test program runs to completion, it is possible to reconstruct how many times a loop structure is executed. This information is used to expand the source to remedy the loop effect discussed in section 3.4.

UX-Metrics is a commercially available program from Set Laboratories Inc. It was used to

extract the following metrics from the test data:

Table 4-1: UX-Metrics supported metrics

LOC	Halstead	Cyclomatic	Misc. Metrics
Lines of Code, LOC Physical source statements, PSS Logical source statements, LSS Executable semi colons (;)	η_1, η_2, N_1, N_2 Predicted length, $\tilde{N}$ Volume, V Effort, E	Cyclomatic complexity, VG1 Extended cyclomatic complexity, VG2	Average maximum span of reference of variable, SP Nesting depth, Dpth Average nesting depth, AvgDp

Lastly, Quantify too is a commercially available program from Pure Software Inc. Again, it is basically a C preprocessor. It is used to record the number of cycles various parts of a program take to execute. It is interesting to note that most system calls are not included during timing measurements.

These three programs are used to gather all the test data. The execution time of all the functions is measured using Quantify. GCT serves as a dynamic tracer. It traces the execution path of the test programs. Using the information from GCT, the source code is expanded. Finally, UX-Metrics is used to gather the metrics information from the expanded code.

4.2 Test Programs

A total of seven software programs were used in the experiments. They can be roughly divided into three categories shown in Table 4-2. The number of functions, the language, whether expanded code is available, and whether the expanded code contains timing overhead for each program are also listed.

Table 4-2: The Data Set

Category	Program	Functions	Language	Expanded Code	Timing Overhead
1	BEN	7	C	Yes	No
	FFT	5	C	Yes	No
2	Sim1	60	C	Yes	No
	Sim2	75	C	Yes	No
	Xboard	145	C	Yes	Yes
3	Program X	24	C	Yes	No
	Program Y	100	C++	No	N/A

The first category of programs consists of the two small programs. They are the burst error network simulator (BEN) and the fast Fourier transform (FFT) calculation program. These are locally written programs.

The second class of programs contains Sim1 and Sim2, which originated from the Rochester Connectionist Simulator, and Xboard. The Rochester Connectionist Simulator

is a system tool designed to simulate network programs. Data was collected from two runs of this program, Sim1 and Sim2. Sim1 is a network program that tries to color a map with four colors so that no two neighboring regions have the same color. Sim2 is a simulation program for a three-layer neural network containing eight, three, and eight cells. The task is to learn to reproduce the input pattern at the output. Essentially the cells must learn to binary encode the input, and the output layer needs to binary decode. Finally, Xboard is a graphical X windows user interface program for playing chess. The Xboard client uses TCP/IP to connect to a chess server over the Internet and the user can play chess with other opponents connected to the same server. Due to the interactive nature of Xboard, it was not possible to execute it with a predetermined set of inputs like the rest of the data. Consequently, Xboard contains the timing overhead from GCT since a separate run to collect timing information might have a different execution path. All three test programs in this category were downloaded from the Internet.

The third category of programs is made up of Program X and Program Y. Both are proprietary programs used in Nortel. Both were prototype programs to simulate switching telephone networks. Because Program Y was written in C++, the expanded code is not available for it. They were written to determine if timing requirements could be satisfied.

The programs in category 1 and 2 were tested on a Unix Sun workstation running SunOS Release 4.1.1. However, category three programs were executed on a HP UNIX machine at Nortel.

4.3 Model Formulation

In this section, the Sim2 model formulation is investigated. The ideas from the Chapter 3 will be applied. Various other models were tried but Sim2 seems to be the best. However, other models were formed in a similar way and tested. This should serve as an example.

4.3.1 Finding the Normalized Z matrix

The first step is to find the normalized **Z** matrix. The raw metrics are extracted by UN-Metrics from Sim2. Table 4-3 lists the mean and standard deviation for each metric. To form the matrix **Z**, each metric value is subtracted by its corresponding mean in the table. The result is then divided by its corresponding standard deviation value, also given in Table 4-3. The reason for standardization is clear from the different scales of the metrics. For example, E is extremely large when compared to AvgDp. In linear regression, such large-scale difference would produce models that are heavily weighted towards E as the dominant predictor. By standardization, this effect is eliminated.

4.3.2 Finding the Principal Components

The first step to finding the eigenvectors is to produce the correlation matrix ρ. Using a numerical program, the eigenvalues and eigenvectors of ρ is found. Table 4-4 listed only the first seven eigenvectors ($\mathbf{e}_1 \ldots \mathbf{e}_7$). They have the largest eigenvalues of the 16 eigenvectors. Finally, the matrix **Z** is multiplied by $\mathbf{e}_1 \ldots \mathbf{e}_7$ to give $\mathbf{P}_C$, as shown in (3.5).

Table 4-3: Standardization of Sim2 Metrics

Metric	Mean	Sd
η_1	19	10
η_2	20	16
N_1	409	1495
N_2	272	992
N^	178	142
V	4115	15682
E	7672800	43504000
VG1	25	111
VG2	32	125
LOC	121	431
;	62	220
SP	11	42
Dpth	2	4
AvgDpth	1	1
PSS	115	413
LSS	57	192

Table 4-4: Sim2 Eigenvectors

Metric	$\mathbf{e}_1$	$\mathbf{e}_2$	$\mathbf{e}_3$	$\mathbf{e}_4$	$\mathbf{e}_5$	$\mathbf{e}_6$	$\mathbf{e}_7$
η_1	0.151	-0.388	0.151	-0.570	-0.077	-0.302	0.097
η_2	0.141	-0.468	0.141	0.466	0.101	0.259	-0.091
N_1	0.289	0.091	0.289	0.064	0.168	-0.212	-0.172
N_2	0.287	0.099	0.287	0.121	0.105	-0.433	-0.020
N^	0.156	-0.461	0.156	0.147	0.052	0.120	-0.011
V	0.291	0.085	0.291	0.077	0.110	-0.110	-0.131
E	0.281	0.152	0.281	-0.174	0.012	0.391	0.126
VG1	0.285	0.109	0.285	-0.128	-0.076	0.468	0.176
VG2	0.273	0.114	0.273	-0.236	0.656	-0.025	0.279
LOC	0.290	0.086	0.290	0.002	-0.007	0.174	-0.187
;	0.288	0.092	0.288	0.175	-0.131	-0.339	-0.096
SP	0.275	0.095	0.275	0.055	-0.634	-0.101	0.490
Dpth	0.125	-0.409	0.125	0.224	0.109	-0.119	0.479
AvgDp	0.146	-0.378	0.146	-0.448	-0.176	0.043	-0.391
PSS	0.291	0.087	0.291	-0.008	-0.016	0.199	-0.054
LSS	0.289	0.054	0.289	0.158	-0.177	-0.011	-0.375

4.3.3 Finding the Useful Principal Components

Table 4-5 lists the correlation between the principal components and the execution time of the functions in Sim2. As shown, the components most strongly associated with execution time are $\mathbf{e_1}$, $\mathbf{e_4}$, $\mathbf{e_5}$, $\mathbf{e_6}$, and $\mathbf{e_7}$. These were used in the regression as the predictor variables.

The meaning of each component can be identify with Table 4-4 by examining the relatively large values, in absolute terms, in the corresponding vector. Component 1 seems be to an overall average of the various raw metrics. Component 4 is a measure of the function's complexity. Component 5 measures the number of boolean literals in the function. This is strongly correlated with the case statement. Component 6 measures the size of the function and compares it to the number of possible branches. Component 7 measures variable use, how far between references and how deep into nested structures.

Table 4-5: Correlation between P_C and Time

P_C	r	P_C	r
e_1	0.9088	e_9	-0.0137
e_2	0.0995	e_{10}	0.0179
e_3	-0.0029	e_{11}	0.0213
e_4	0.1671	e_{12}	-0.0146
e_5	0.1342	e_{13}	-0.0005
e_6	-0.2348	e_{14}	0.0007
e_7	-0.2248	e_{15}	0.0248
e_8	-0.0726	e_{16}	-0.0303

4.3.4 The Sim2 Model

The model is formulated as outlined above. In this case, the standard program used is the expanded code for Sim2. The 16 metrics, obtained using UX-Metrics, were first standardized. Each would average to zero and have a standard deviation of one. Using (3.5), the raw metrics were transformed into 16 principal components. For the 16 principal components, the five with the strongest correlation to the execution time were used in the linear regression.

Tables 4-6, 4-7, and 4-8 all show that the regression used to formulate this model is reasonable. The regression model is highly significant as suggested by the F of over 600 as shown in Table 4-7. Since this means that the significance level of F almost zero, the model is highly accurate. Also, the individual coefficients are significant as evidence by the t-values and the corresponding almost zero p-values in Table 4-8. The p-values are the significance level for the t-test. As well, both the adjusted and normal coefficient of multiple determination is near unity. The extremely low significant level for both the F-test and the t-test as well as the near unity of the correlation coefficients are all consistent with a very accurate model.

Figure 4-1 shows the regression scatter plot of the predicted and actual execution times. Although some values were predicted as negative, this is normal behavior when predicting small values with normalized data. The plot shows that the predicted execution times are very close to the actual execution times. Again, this points to a very strong model.

Table 4-6: Regression Model Summary

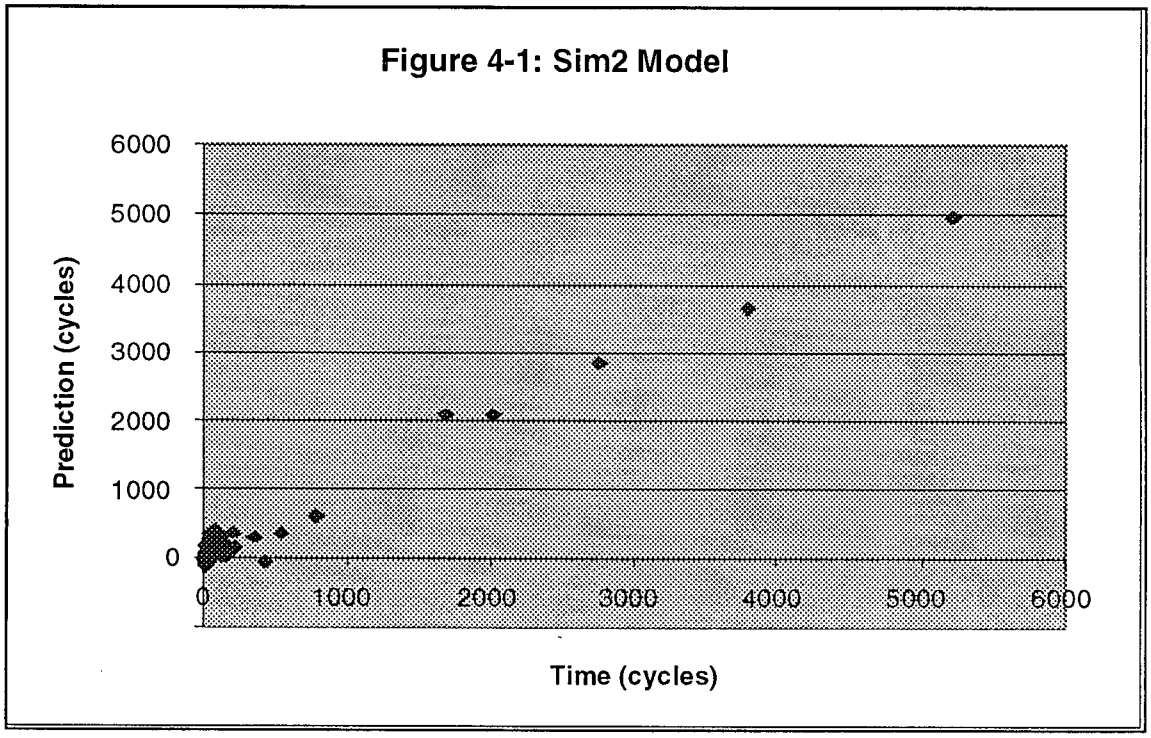
REGRESSION STATISTICS	
Multiple R	0.988710
R Square	0.977548
Adjusted R Square	0.975921
Standard Error	129.9329
Observations	75

Table 4-7: Regression Model Significance

	DF	SS	MS	F	Significance F
Regression	5	50719844	10143969	600.8551	2.04E-55
Residual	69	1164896	16882.55		
Total	74	51884740			

Table 4-8: Regression Model Coefficient Significance

	Coefficients	Standard Error	t Stat	P-value
Intercept	277.403	15.00336	18.48940	1.94E-28
PC 1	224.483	4.45556	50.38266	3.7E-56
PC 4	297.640	32.12460	9.26518	9.85E-14
PC 5	264.273	35.51869	7.44039	2.1E-10
PC 6	-604.866	46.47531	-13.01480	3E-20
PC 7	-651.674	52.29414	-12.46170	2.48E-19



4.4 The Sim2 Model Predicting the Expanded Code

The sim2 model is used to predict five test programs: BEN, FFT, Sim1, Xboard and Program X. Table 4-9 summarizes the result of the predictions. The R^2 is the square of the correlation between the prediction and the actual value of the functions in each of the expanded source code program. It ranges from 0 to 1 and is a summary of the strength of the relationship between the predicted and actual values of the execution time. Program Y was not expanded and therefore not listed. Overall, an excellent correlation is shown between the actual and the predicted execution times.

Table 4-9: Summary of Predictions

Category	Program	R^2
1	BEN	1.0000
	FFT	0.9843
2	Sim1	0.9684
	Xboard	0.6538
3	Program X	1.0000

4.4.1 Category One Programs

The prediction results of BEN and FFT are shown in Figures 4-2 to 4-4. Figure 4-3 is the same as Figure 4-2 with the slowest function in BEN removed. The behavior of both the BEN and FFT predictions is similar. Both programs are characterized by a dominant function that takes the majority of the execution time. These dominate points lead to an overstatement of the R^2 value. This accounts for the near unity values for these two tests programs. Yet, Figure 4-3 shows that the R^2 value listed for the category one programs in Table 4-9 is misleading.

Figure 4-3 demonstrates an important weakness in the theory even given these computationally based programs. Because linear regression is disproportionally affected by the larger values, regression models based on large values are poor predictors of extremely fast functions. Fortunately, in most situations, the fast functions are not of interest in scheduling real-time systems and the slow functions are ones that are important.

Figure 4-2: Sim2 Predicting BEN

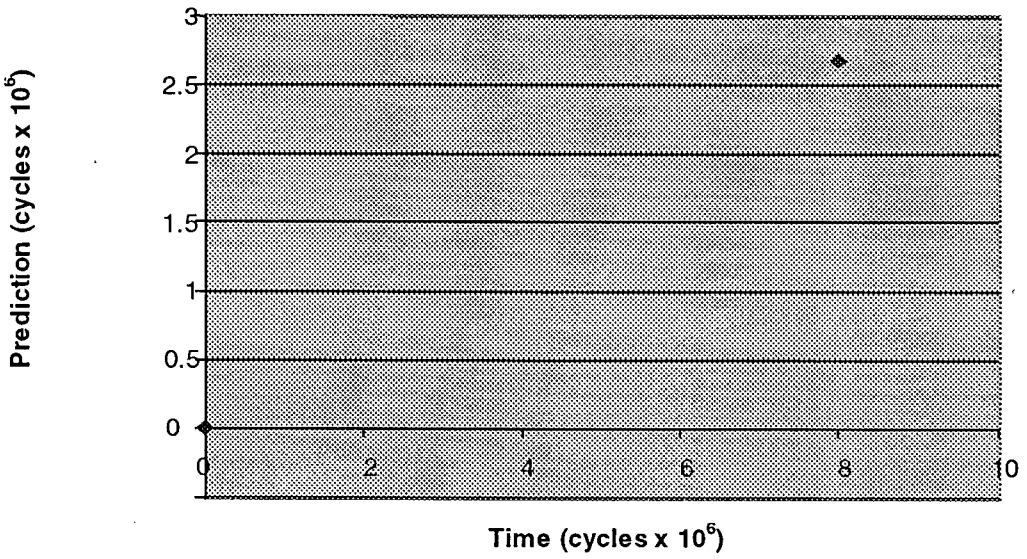
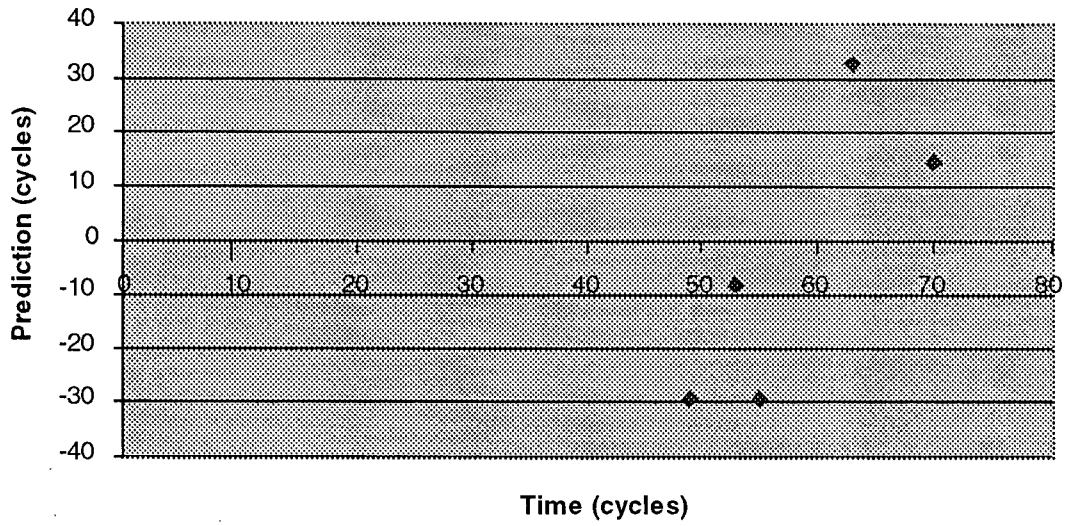
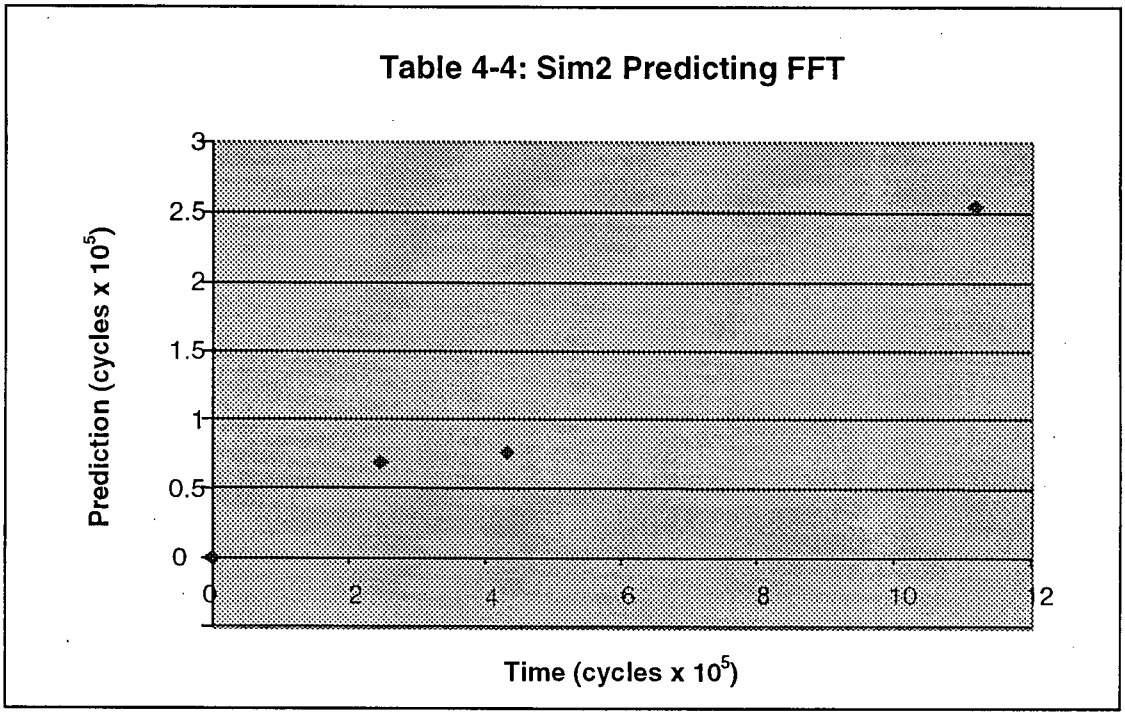


Figure 4-3: Sim2 Predicting Fast Functions



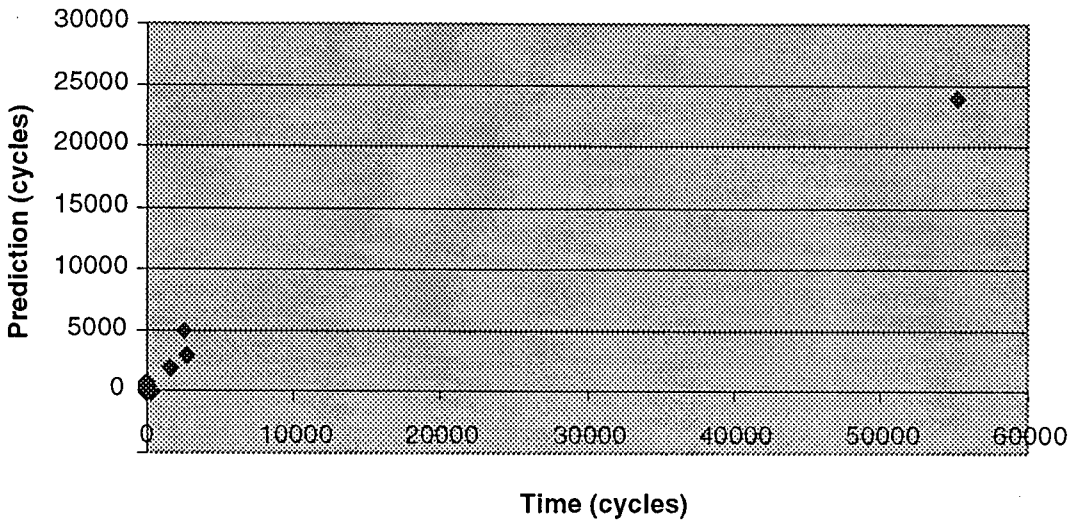


4.4.2 Category Two Programs

Figures 4-5 and 4-7 show the prediction of Sim1 and Xboard using the Sim2 model. Figure 4-6 is the same as Figure 4-5 but has the slowest function removed. Three functions in Xboard (*main*, *FindFont*, and *yylex*) were treated as outliners and removed from the calculations. Outlines is a term that means "bad values" for data points that fall outside the line implied by the rest of the data. The *main* function is removed because it contains the bulk of the GCT path tracing overhead, making the actual time slower than the time without GCT overhead. The function *FindFont* could have been executed without Quantify configured properly to catch its system call. The function *yylex* is removed because it contains "go to" statements. Our approach to time estimation cannot account for the irregular control flows generated using "go to" statements. The assumption is that only structured programs will be involved.

As shown in Table 4-9, the square of the correlation between the predicted and actual execution times for Sim1 is near unity. It implies that the prediction model explained about 97% of the variance. The actual execution times of the Xboard functions are slower than the predictions. However, all the functions seem to be at a constant factor of four times slower and Figure 4-7 does shows a linear relationship. This four-times slower prediction is due to the GCT overhead resulting in a constant speed reduction. Xboard was the only function that contains the GCT probes while data was collected by Quantify. Nevertheless, the R^2 in Table 4-9 shows that over 65% of the variance in the actual execution time is explained with this prediction. Taking this into account, both of the programs in category two are well modeled.

Figure 4-5: Sim2 Predicting Sim1



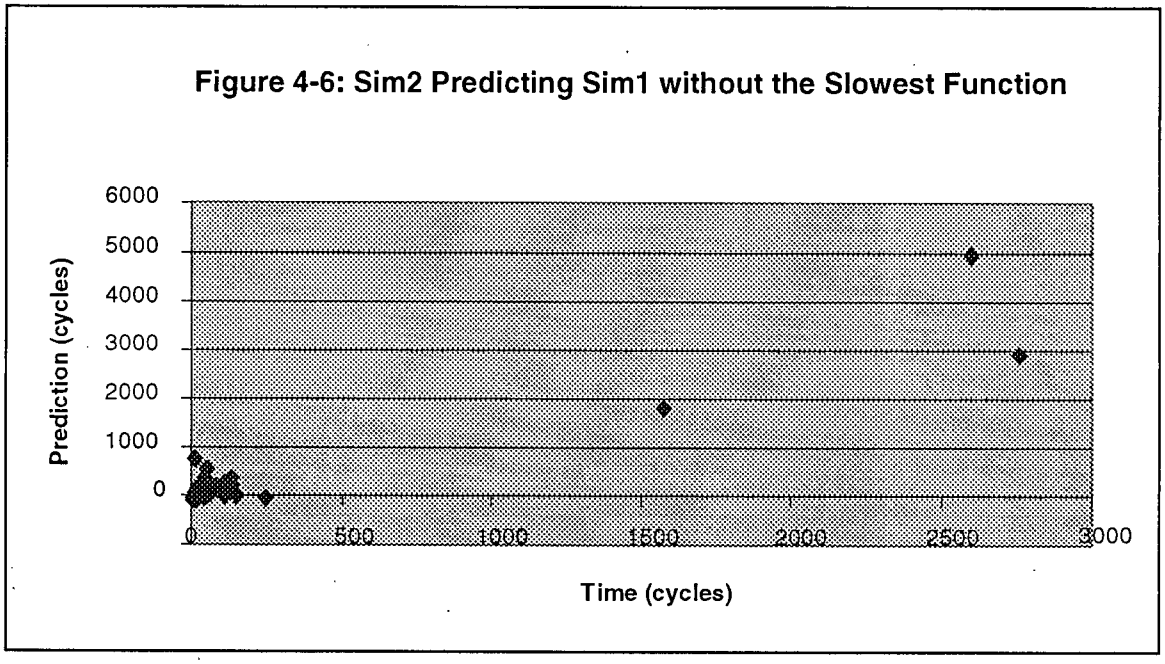


Figure 4-6: Sim2 Predicting Sim1 without the Slowest Function
Prediction (cycles)
6000
5000
4000
3000
2000
1000
0
0
500
1000
1500
2000
2500
3000
Time (cycles)

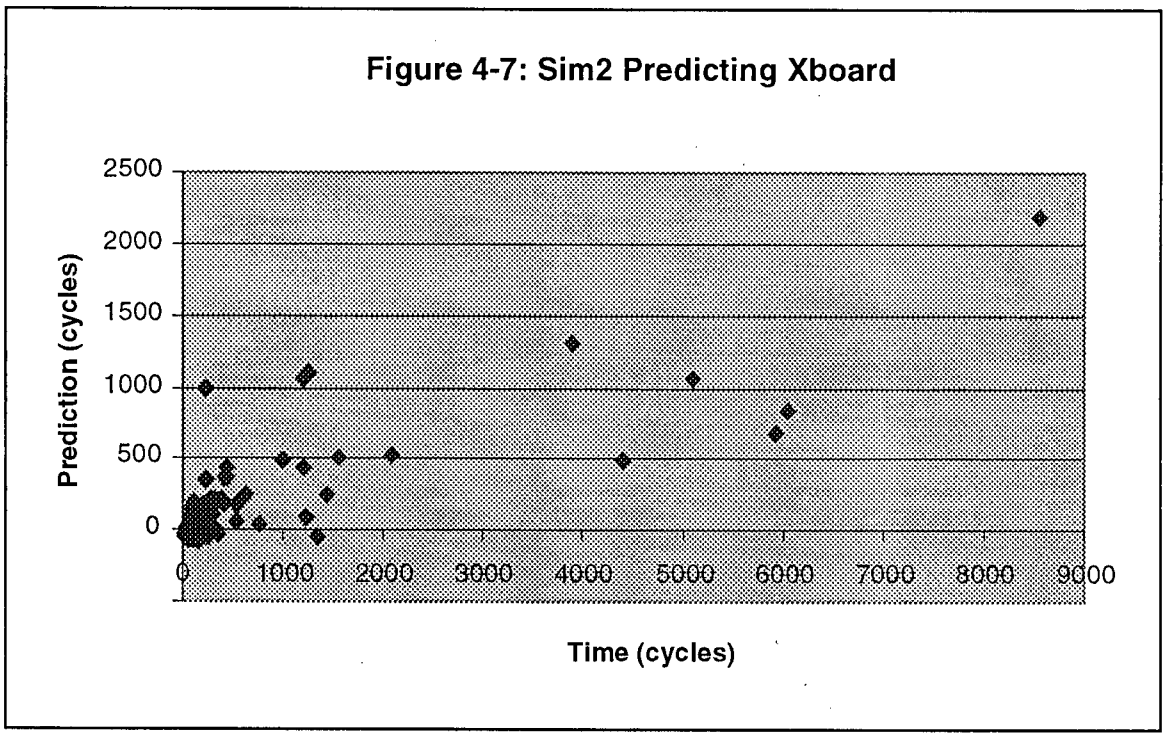
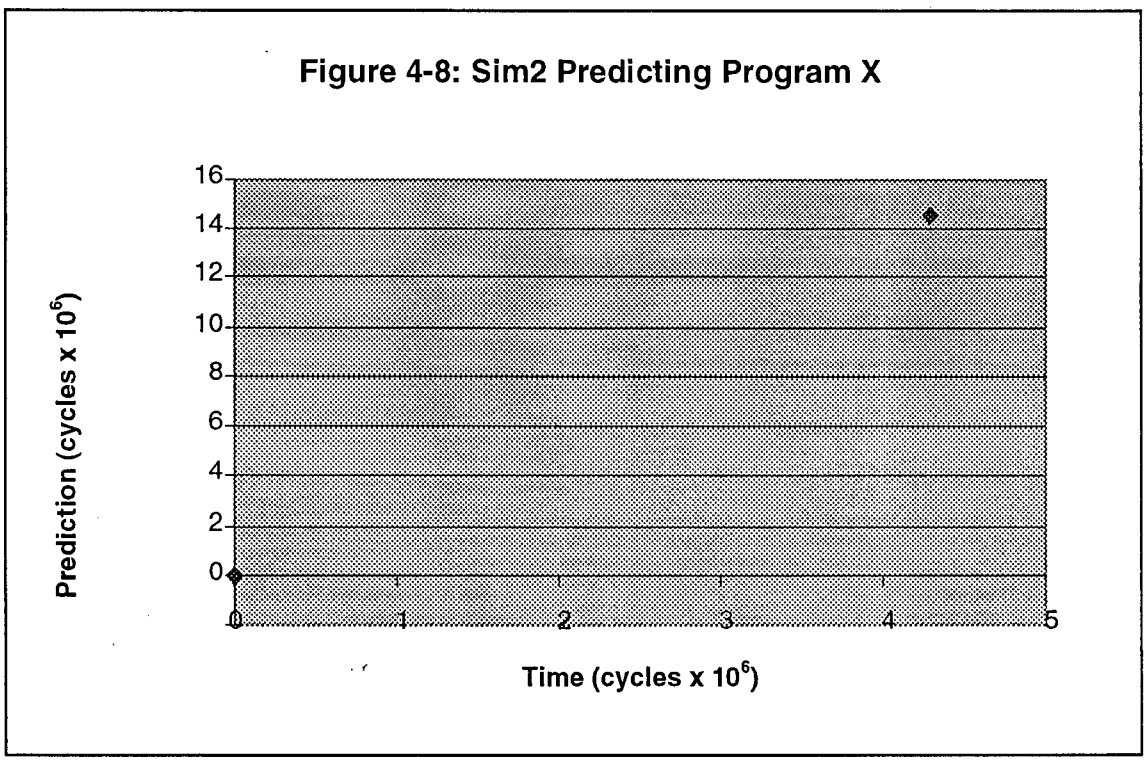


Figure 4-7: Sim2 Predicting Xboard
Prediction (cycles)
2500
2000
1500
1000
500
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Time (cycles)

4.4.3 Category Three Program

The only program in category three that had expanded code available is Program X. Figure 4-8 shows the predictions from the regression model on Program X's functions. Similar to category one programs, Program X is dominated by one slow function. This is the main reason for the R^2 value of unity. Figure 4-9 is the same graph with a reduced scale. The only point not show in 4-9 is the slowing function in Program X. Excluding the slowest function, the R^2 value is 0.7417. Even with the R^2 value at 0.7417, this is still a strong prediction.



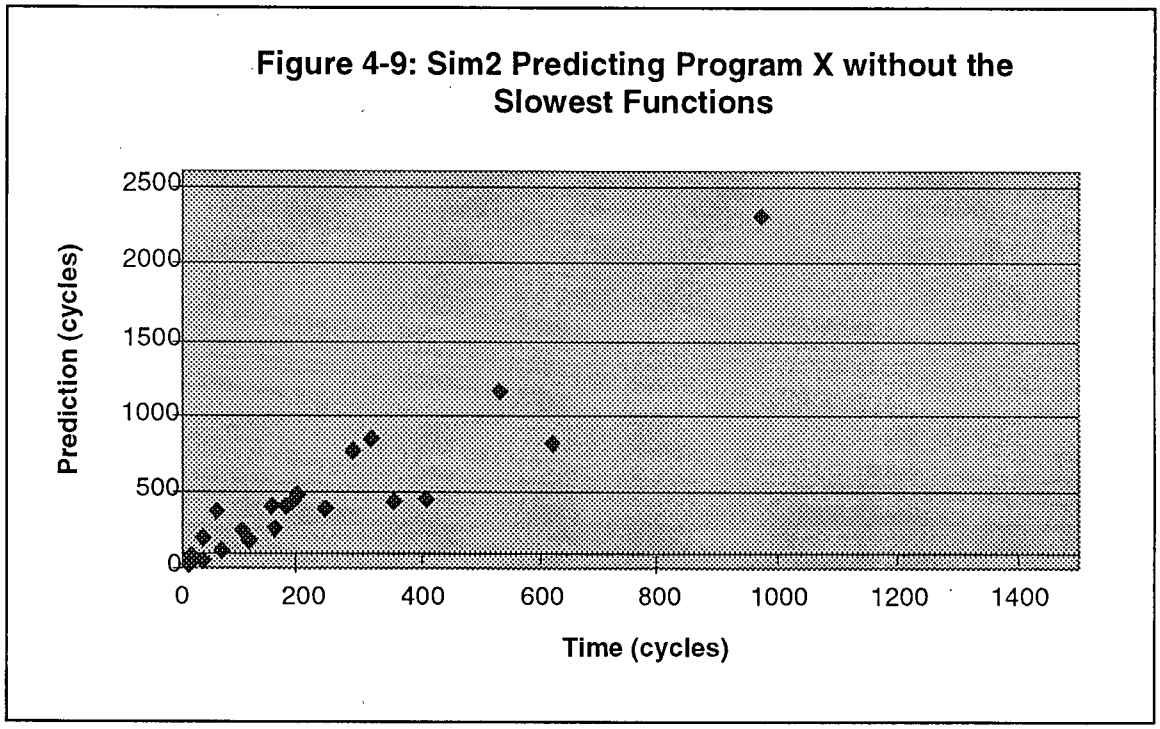


Figure 4-9 shows that the predicted value is about twice as slow as the actual values. This is caused by the model being formed on older Sun machine while Program X was tested on a more advanced HP machine. To reduce the error in this prediction, it would be necessary to formulate the model from a standard program tested on the HP machine. Nevertheless, there is clearly a linear relationship between the two. This strongly supports the idea that source code can be used to predict execution times. The error in the prediction is only due to cross platform differences.

Note that two functions in Program X *xfer* and *sendmesg* were treated as outliners and were removed. They contain eight and nine lines of logical statements respectively. Since only the metric values and not the source code were available, it is assumed that these

functions only contained irregular loop structures, for example "goto" statements that were not expanded.

4.5 Model Robustness

Three points suggest that the theory is sound. First, similar models formulated with the other test programs resulted in similar predictions. Second, the programming language does not effect the overall theory. Finally, the computing environment has little effect on the results. Though, the last two claims require further investigation.

The theory does not depend on either the data or the model presented in this chapter. A different data set would produced similar predictive results provided that the standard program used to formulate the model is representative. This is to say that some care must be taken to produce the model. Sim2 was used as the standard because it produced the best regression results with the rest of the data. However, a similar model based on Sim1 had comparable results. A third model based on Xboard was also produced. Its predictive ability was the worst of the three. The other programs were considered of an insignificant size to generate a reasonable model due to the limited number of functions in each program.

A comparison was done between Program X and Program Y. Program X was written in C and Program Y in C++ (See Table 4-2). Principal component analysis was done to examine the raw metrics form the C and C++ code. Interestingly, they mapped to the same

domain. This supports the intuitive notion that C and C++ are similar at least with respect to their metrics. Both have a similar distribution of control and data statements. To predict C++ code, a model should be generated from a C++ standard program. That being said, the same modeling seems applicable to C++ as well as C source code.

As stated in before, all but the category three programs were executed on Sun machines. The category three programs were tested with HP machines. Since machine cycles ignore the clock speed of a particular machine type, all the data is taken in terms of machine cycles and not actual time units. Although Program X was tested on a HP and the model is formulated on a Sun machine, there is not an effect on Program X's predictions. It still appears as a linear relationship even though they are executed over two completely different architectures. This suggests that the model is machine type independent.

4.6 Summary

This chapter discusses the results of the proposed approach to time estimation. GCT, UX-Metrics, and Quantify are the software used to gather metrics and timing information from the test programs. The sim2 model is formulated as outlined in Chapter 3. It is validated and shown to be extremely significant by the various statistical tests. Then, the model is used to predict the execution times of the other test programs. The results for the slow functions are very good. One can see a clear linear relationship between the predicted and actual execution times. However, there is a large error when it comes to the fast functions. We believe the main reason for this error is due to the metrics chosen. They were unable to quantify the specialized nature of the fast functions.

Chapter 5: Conclusion

This work is focused on the development of a method to predict the execution time of structured programs based on their static source code metrics. Since source code is available early in the development cycle, these predictions can be made to spot potential future problems in meeting timing requirements. Various software metrics are analyzed. The most interesting are the relative complex metrics generated by the principal component method. Using this idea of principal component metrics, a linear regression model is formulated to predict the execution time of functions. The resulting model is shown capable of predicting the execution time of other functions.

There are a few things to note about our approach to timing estimation. To solve the indeterminate nature of loop structures in the source code, a bound is placed on each individual loop and the code inside is expanded. Then, principal component analysis was used both to reduce the number of metrics used and to produce predictor variables for linear regression. This prediction is reasonable for the slow functions. However, extremely fast functions are problematic for the model to predict. Fortunately, it is the slow functions that are of interest in most cases. This method is meant to be a first approximation of the execution times, to be used as soon as the source code is available. It is also relatively robust, being insensitive to changes in computing language and environment.

The main contributions of this thesis are:

- A systematic method of predicting the software execution time is proposed. It predicts the execution times of structured functions based on the source code. Modeling is carried out using principal component analysis and linear regression. Since source code prediction is one of the earliest quantitative information available, it can be used to detect possible areas where timing requirements are not met. This can result in reduced overall system production cost.
- Though meant only for first approximation, this method seems relatively robust, being insensitive to changes in computing language and architecture. Though this claim required further examination since it is based only on a few test cases.
- Results show that the functions selected from the standard program have to be representative of the other functions to be tested. That is, the program types must be similar. This leads to the conclusion that the prediction models are different for different types of programs.
- We have found that although branch structures do not present a problem, loop structures are problematic. The proposed technique can not handle programs with loops. One solution to the loop structure problem is to expand the code inside the loop according to the number of times that loop is executed. It is also found from the test cases that only about 30% of functions in real-time programs have loops. Therefore, this technique can be applied without a large amount of additional effort.

- We have found from the test cases that this technique can predict slow functions better than fast functions. We believe a partition method, that is, to group the functions into fast and slow categories will improve the prediction since a different prediction model can be made for each of the two categories.
- The 16 metrics that were used can be mapped to only five domains represented by principal components: an overall average component, a complexity component, a case-statement component, a size versus branching component, and lastly, a variable/nesting depth component.

Future research can be extended from the current work into such areas as:

- Developing new metrics. The current procedure uses raw metrics that were not originally designed for timing estimation. New metrics that are design to measure some aspect of execution timing can replace the raw metrics and should give a better prediction.
- Using higher order models. Currently, only linear transformations and regression is used in the modeling. Moving away from a linear model could improve the prediction.
- Checking the assumption that the model is insensitive to changes in languages and computing environments. Only one test case was examined for each of these and they may be abnormal cases. More testing is needed to solidify the conclusion.
- Distinguishing between fast and slow function. This would allow a partition method to be employed. The current method predicts slow functions well, however fast functions are not well modeled. Partition would allow the fast functions to be model differently from the slow functions.

- Extending this research to unstructured programs. The assumption throughout is that only structured programs are involved. It may be possible to apply this method to unstructured programs as well.
- Integrating this approach to timing estimation into existing software. It should be possible to gather all the needed metrics automatically. Then, after the estimation is accomplished, the data is transferred into a scheduling software to determine if a system meets timing requirements. Such an automated system would save both time and money in the development of real-time systems.

References

[Arnold 94] R. Arnold, F. Mueller, D. Whalley, M. Harmon, "Bounding Worst-Case Instruction Cache Performance," *Proceedings of the IEEE Real-Time Systems Symposium*, Los Alamitos, California: IEEE Computer Society Press, December 1994, pp. 172-181.

[Basili 84] V. Basili, D. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, SE-10 No. 6, New York, New York: Institute of Electrical and Electronics Engineers, November 1984, pp. 728-738.

[Boehm 81] B. Boehm, *Software Engineering Economics*, Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[Currans 86] N. Currans, *Fourth Annual Pacific Northwest Software Quality Conference*, Portland, Oregon: Lawrence & Craig, 1986.

[Gupta 94] R. Gupta, P. Gopinath, "Correlation Analysis Techniques for Refining Execution Time Estimates of Real-Time Applications," *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, Los Alamitos, California: IEEE Computer Society Press, 1994, pp.54-58.

[Halstead 77] M. Halstead, *Elements of Software Science*, New York, New York: Elsevier, 1977.

[Johnson 77] R. Johnson, D. Wichern, *Applied Multivariate Statistical Analysis* (3rd ed.), Englewood Cliffs, New Jersey: Prentice Hall, 1992.

[Puschner 89] P. Puschner, C. Koza, "Calculating the Maximum Execution Time of Real-Time Programs," *Journal of Real-Time Systems*, 1(2), 1989, pp. 159-176.

[McCabe 76] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, SE-2 No. 4, New York, New York: Institute of Electrical and Electronics Engineers, December 1976, pp. 308-320.

[Munson 89] J. Munson, T. Khoshgoftaar, "The Dimensionality of Program Complexity," *Proceedings of the 11th International Conference on Software Engineering*, Washington, D.C.: IEEE Computer Society Press, 1989, pp. 245-253.

[Munson 90a] J. Munson, T. Khoshgoftaar, "Applications of a Relative Complexity

Metric for Software Project Management," *The Journal of Systems and Software*, 12 No. 3, New York, New York, Elsevier North Holland, July 1990, pp. 283-291.

[Munson 90b] J. Munson, T. Khoshgoftaar, "Regression Modelling of Software Quality: Empirical Investigation," *Information and Software Technology*, Vol. 32 No. 2, London, England: Butterworths, March 1990, pp. 106-114.

[Munson 92] J. Munson, T. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering*, SE-18 No. 5, New York, New York: Institute of Electrical and Electronics Engineers, May 1992, pp. 423-433.

[Munson 93] J. Munson, T. Khoshgoftaar, "Measurement of Data Structure Complexity," *The Journal of Systems and Software*, 20, New York, New York: Elsevier North Holland, 1993, pp. 217-225.

[Tempus 97] Tempus Project, "Real-Time Performance Measurement: A White Paper," *Software Productivity Solutions Inc.,* March 11, 1997.

[Zuse 90] H. Zuse, *Software Complexity: Measures and Methods*, Berlin: Walter de Gruyter, 1990.