A Software Science Model of Compile Time

WADE H. SHAW, JR., SENIOR MEMBER, IEEE, JAMES W. HOWATT, ROBERT S. MANESS, AND DENNIS M. MILLER

Abstract—Halstead's theory of software science is used to describe the compilation process and generate a compiler performance index. A nonlinear model of compile time is estimated for four Ada compilers. A fundamental relation between compile time and program complexity measures used, and the investigation of significant relationships between program characteristics and compile time. The results suggest that the model has a high predictive power and provides interesting insights into compiler performance phenomena. The research suggests that the discrimination rate of a compiler is a valuable performance index and is preferred to average compile time statistics.

Index Terms—Ada compilers, compile time, performance indexes, software science.

I. INTRODUCTION

TECHNOLOGICAL advances in computer software are changing the way we understand the underlying processes governing software design. Computer systems are becoming more numerous, more complex, and deeply embedded in our society. Inherent in this explosion of technology exist questions concerning fundamental relationships between the processes of problem definition, algorithm selection and coding, and translation into an executable image. We can no longer write programs, but must "engineer" software for our systems to offset the rising cost of software development [1].

The Department of Defense (DoD) recognized this challenge in the 1970's and realized that a new standard language could be created to encourage the use of modern software engineering principles [2]. With the introduction of the Ada programming language for DoD, software engineering tools are needed to evaluate the performance and reliability of this language. Ada was developed under sponsorship of the DoD to support development of software for embedded computer systems. 'By definition, an embedded computer system is one that forms a part of a larger system whose purpose is not primarily computational, such as a weapons system or a process controller. Physically, an embedded system may range from a single microcomputer to a network of large computers' [2]. For example, one area of use will be in the field of avionics. In the development of avionics software, efficient compilers are needed. As more Ada compilers become available, tools are needed to validate and evaluate these compilers to determine which, if any, could best meet DoD requirements. One measure of interest in compiler comparisons is the computer time required to translate source code into executable machine code.

Currently, benchmark test suites are used; however, they have a poor reputation because the performance figures are sometimes cited out of context and overgeneralized into overall ratings [3]. What is needed is an approach that provides insight into the effect of intrinsic software characteristics on compile time. Researchers, such as Maurice Halstead [4], [5], have raised questions about the existence of fundamental principles that govern the design and execution of software. Halstead's goal was to develop objective measures of programming time and effort to make sound judgments about software quality and complexity.

The motivation for this research is threefold. First, the application of software science to the compiling problem is a straightforward extension of the theory. The degree to which concepts proposed in software science can be used to explain compile time represents reinforcement of the basic tenets offered by Halstead. Second, software science may, in fact, produce insight into the physical process of compilation. Clearly, compile time is a relatively minor aspect of compiler performance. Nevertheless, variation in source code characteristics such as operator/operand frequency are manifested in varying compile times which represent a phenomenon that is not fully understood. Any relationships uncovered by mapping characteristics of source code to a model of compile time offer some evidence of a natural process. Finally, the use of a compile time model allows direct comparison of alternative compiler implementations as well as comparison of target architectures. Use of simple averages does not yield as sensitive a metric as models specifically designed to reduce the error component always present in performance measurement. A model yields structure to the problem of compiler/machine comparisons so that statistical tests can be used with a known precision.

Manuscript received February 11, 1987; revised August 3, 1987.

W. H. Shaw, Jr. and J. W. Howatt are with the Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH 45433.

R. S. Maness was with the Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH 45433. He is now with the Air Force Satellite Control Facility, Peterson AFB, CO.

D. M. Miller was with the Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH 45433. He is now with the Air Force Operational and Test Center, Kirtland AFB, NM.

IEEE Log Number 8926735.

U.S. Government work not protected by U.S. Copyright
We propose to apply the fundamental concepts of Halstead’s software science theory to determine if an extension of his theory could be used to explain compile time and evaluate Ada compilers [6], [7]. Specifically, the research hypotheses are as follows.

1) There is no variability in compile time for Ada programs which can be explained by the relationships proposed in software science.

2) There is no variability in the performance of the software science model of compile time attributable to characteristics of the program. (We develop these characteristics later.)

3) There is no variability in the performance of the software science model explained by alternative compiler/computer systems.

These hypotheses allow for the development and testing of the applicability of software science in three ways. First, how well does the model predict compile time and what fundamental relationships exist between compile time and the software metrics proposed by Halstead? Second, is there a difference in the model’s performance across various categories of Ada code (such as high versus low percentage control flow code)? Third, can performance differences between compilers and machines be detected using software science measures and, if so, can a performance measure be developed?

The next section presents the theory applicable to this investigation to provide a background for the compile time model. The research methodology and experimental design used to evaluate the compiler model are presented as well as results of the experiment for four compilers. Finally, we summarize the results and conclude the paper with comments on the applicability of the compile time model.

II. HALSTEAD'S SOFTWARE SCIENCE THEORY

In his classic work on software science [5], Halstead attempted to define and measure the complexity of software by analyzing program source code. Halstead defined four basic metrics computable from the code:

\[ n_1 = \text{the number of unique operators}, \]
\[ n_2 = \text{the number of unique operands}, \]
\[ N_1 = \text{the total number of occurrences of operators}, \]
\[ N_2 = \text{the total number of occurrences of operands}. \]

Using these basic metrics, Halstead defined the vocabulary \( n \) of a program to be the total unique tokens:

\[ n = n_1 + n_2 \]  

and the length of a program to be the total operator and operand occurrences:

\[ N = N_1 + N_2. \]  

The size or volume of a program may vary when translating from one language to another. Halstead surmised that the volume of a program is a function of its vocabulary and is given by

\[ V = N \log_2 (n) \]  

where \( V \) has a unit of measurement in bits. That is, \( \log_2 (n) \) bits are needed to distinguish each of the \( n \) tokens in a program.

An algorithm may be implemented by many different, but functionally equivalent programs. When an algorithm is implemented in its most succinct form, then its potential volume \( V^* \) is

\[ V^* = (2 + n_2)(\log_2(2 + n_2)) \]  

where \( n_2 \) is the number of input/output parameters and \( "2" \) is the number of required operators: the procedure/function name and the parameter list grouping operator. This represents the size of the program if it existed as a built-in function or procedure call. Halstead then argued that the amount of time required to implement an algorithm is directly proportional to the square of the program volume \( (V) \) divided by the potential volume \( (V^*) \) and a constant \( (S) \):

\[ T = V^*/(SV^*). \]  

The constant "\( S \)" represents the speed of the programmer or the number of mental discriminations per unit of time. Halstead used a value of 18 because in his experiments, 18 gave him the best results when comparing actual versus predicted programming time. We use the parameter "\( S \)" (denoted \( K \)) as a measure of compilation rate; it therefore represents a performance index.

III. RESEARCH METHOD

Halstead’s programming time equation serves as the basic theoretical model. The equation is specified as a set of independent variables related by a set of parameters to be estimated. The dependent variable is the actual CPU time required for the compilation process. The volume \( (V) \) and the potential volume \( (V^*) \) are the independent variables. Placing (5) in parameter form yields

\[ T = KV^*(V^*)^b. \]  

This equation has the exact form as Halstead’s time equation where "\( K \)" is the discrimination rate, "\( a \)" is 2, and "\( b \)" is -1. "\( K \)" is assumed to have the same meaning in the compilation process as the constant "\( S \)" in Halstead’s equation for predicting programmer time. "\( K \)" represents how fast the compiler does its job (the processing rate) and will depend on the compiler architecture and the efficiency of the compiler itself. Clearly, "\( K \)" can be interpreted as a performance index given that "\( a \)" and "\( b \)" are known. Alternatively, "\( K \)" can be estimated across compilers and used as a performance index to statistically distinguish compilers.

Data collection represented a major portion of this research effort. Observations of compile times were required where the metrics of the source code could be accurately measured. To estimate an equation, numerous observations were required with sufficient variability in the software metrics. The observations of software metrics necessary for estimation of the proposed mathemati-
SHAW et al.: SOFTWARE SCIENCE MODEL OF COMPILATION TIME

A software science model was extracted from algorithms taken from a DoD-sponsored test suite. This required a set of rules for the identification and enumeration of each operator, operand, and I/O variables in each program. For our investigation, we expanded the counting strategy defined by Halstead. He considered tokens in only executable code. But a compiler must process all the tokens in a program, and can expect substantial resources translating data types, packages, tasks, and the like. Thus, to obtain accurate estimates, we include all program source code (except comments) in computing the measures for our analysis.

This implies that tokens found in declarations as well as in executable code be counted. Consequently, in constructing these rules, consistency with the syntax diagrams in Appendix E of the Military Standard Ada Programming Language (MIL-STD-1815A) [8] was enforced. Not only did this approach offer a rigid basis upon which to build the counting strategy, it also offered the benefit of using the same syntax charts that authors of a compiler must use in designing an Ada compiler. We develop and implement a counting strategy specifically for Ada [9].

The experimental design required that the algorithms be selected with a wide range of software metrics. Therefore, the Prototype Ada Compiler Evaluation Capability (ACEC), a collection of approximately 300 Ada modules, was selected. DoD sponsored the creation of this benchmark test suite to validate and evaluate Ada compilers. The programs provide information about language features that must be present in a full implementation of a MIL-STD-1815A compiler [10].

The programs were compiled on four different computers having validated Ada compilers and the compilation times were recorded. Table I summarizes the experimental environment.

CPU compile time was measured as accurately as possible. On a multiuser computer system, compilation time cannot be measured simply by a stopwatch because of the contention with other users. Therefore, total CPU time used in the compilation process was used. This time was obtained from the list or history file generated by the compiler for the AOS/VS and VMS systems. The UNIX system did not provide this information; therefore, the system command "time" was used. The Ada library was reinitialized after each program was compiled to ensure that librarian functions did not affect timing between successive compilations.

The compiler model was then estimated using the analysis of variance method and the linear regression tool on the SAS [11] software package for data analysis. To use linear regression analysis, the compiler model had to be linearized. This was done by taking the natural logarithms of both sides of the compile time equation. As a result, the equation for the compiler model now becomes

$$\ln(T) = \ln(K) + a \ln(V) + b \ln(V^*) + \text{error}. \quad (7)$$

The analysis was divided into three parts. The first part evaluated the compiler model to determine its ability to explain compile time across alternative Ada compilers. The second part evaluated different module characteristics to determine which, if any, of them had a measurable impact on compile time on the UNIX system. Finally, we investigated the development of a performance index to compare the speed of various Ada compilers.
Thirteen program characteristics were considered in this investigation. Table II summarizes the program attributes used to divide the test suite. Each attribute average was computed for the entire test suite and the mean was used to distinguish between two levels of the attribute. Regression models were then computed for each partition.

IV. Results

Table III shows the percentage of error reduction relative to the average compilation time if the compiler mode is used to predict compile time. By error reduction, we mean the average difference between predicted and actual compile times compared to use of simple test suite average. That is, the software science model reduced prediction error 83.8 percent compared to the arithmetic average of 27.1 CPU seconds for the AOS/VS system. It is interesting to note that the slowest compiler, the AOS/VS system, provided the best model.

Based on the statistical analysis of the compiler model, the estimated exponents for \( V \) and \( V^* \), "\( a \)" and "\( b \)", respectively, are shown in Table IV. The models’ \( R^2 \) is the coefficient of determination and measures the degree to which the data fits the model. An \( R^2 \) of 1 would indicate perfect correspondence between the observed compile times and the software science model. Low \( R^2 \) indicates poor performance of the model. Since \( R^2 \) is influenced by the number of observations and the number of terms in a linear model, the adjusted \( R^2 \) is used to avoid any upward bias. The adjusted \( R^2 \) is therefore a conservative estimate of the degree of model fit.

The \( F \) statistic is a test statistic which indicates the likelihood that some parameter in the model is nonzero. That is, higher values of the \( F \) statistic indicate more confidence in the model. This confidence is expressed by the number in parentheses which indicates the probability that the data observed could produce the estimated model and still be incorrect. For example, a value of 0.0001 is 0.01 percent chance that the model has incorrectly established a linear fit between the dependent and independent variables.

Likewise, the column labeled "Prob > T" is the likelihood of error in estimating a parameter. Usually, likelihoods of less than 1–5 percent are deemed significant and indicate the parameter should be retained in the model. The evidence is overwhelming that the models are statistically valid and that the majority of variability in compile time can be explained by the software science model.

In Halstead’s original work, he set the exponent of \( V \) and \( V^* \) in the programmer time equation to 2 and -1, respectively. In Table IV, the estimated exponents are shown to be approximately 0.5 and 0.1. \( V^* \) is not significant in two of the estimated models. Instead of dividing \( V \) and \( V^* \), the compiler mode multiplied these two parameters where \( V^* \) was very small and perhaps insignificant. On the other hand, taking the square root of \( V \) is interesting because of the effect on modularization. That is, the exponent for \( V \) being 0.5 rather than 2, as suggested by Halstead, is a contrast about the effects of program modularization between programming time and compile time.

Assume a program can be broken into \( n \) modules such that the volume of the program is equal to the sum of the volume of the \( n \) modules. This can be expressed as

\[
V = \sum_{i=1}^{n} v_i.
\]

If Halstead’s time equation is a function of the power of \( V \) and that power is greater than 1, then

\[
V^2 \gg \sum_{i=1}^{n} (v_i)^2.
\]

As a result, modularization reduces programming time. That is, the program will take longer to write than it would take to write an equivalent modularized program. However, the compiler model seems to indicate the opposite since the exponent was fractional. If compile time is a function of the power of the volume, then the total
modularizing software causes the compiler to suffer in performance. Intuitively, this makes sense because the library and symbol tables.

The time reduced by a programmer when the actual and predicted compilation times are shown in Table V. If the parameter was not significant ($t > 0.05$), then it was not included in the model. That is why $V^*$ is not in the Unix or AOS/VS model. As shown in Table V, the correlation between predicted and observed compilation times for each compiler are all quite high. Consequently, the model fits well. Note also that the correlations between the actual and predicted times on the VAX computers are identical. This makes sense because the same compiler is used in each computer. However, the discrimination rate is different, and therefore represents a relative measure of computer speed.

The predicted times compare relatively well to the actual times as shown in Fig. 1 for the UNIX compiler. In this figure, all observations were sorted in ascending order based on the Unix compile times. The other compiler models generated similar curves with close correspondence between actual and predicted compile times. Note that as the compile time increases, the difference between actual and predicted times increases. This seems to suggest that the error term in the estimation process may enter the model as a product term and not a sum. However, below 30 s of CPU time, the model does very well. The figure suggests the existence of a structural break which may influence the model parameters as compile time increases. In general, the actual and predicted compile times correspond quite well.

Hypothesis 2 was introduced to determine if the predictive power of the compile time model remains consistent between levels of program characteristics. The first test divided the test suite into one group of modules that did not have a tasking function and the other group having a tasking function. Another test divided the suite based on module length with the dividing point being the average module length for the test suite. Each of the other 11 tests divided the test suite into two groups based on low versus high percentage of the characteristic being tested. Modules that fall below the average percentage of a characteristic were put in the low category and all other modules were put in the high category.

Table VI presents the results of the investigation for each of the 13 characteristics presented earlier. Clearly, differences in model performance exist which provide useful insight into the compilation process. For example,
programs with high a percentage of I/O statements yield similar average compile times, but the higher the I/O concentration, the better software science was able to explain compile time.

The next major area of investigation was the development of a performance index. For this analysis, dummy variables were used to observe the effect of each compiler and to analyze each compiler separately while maintaining the same exponents for volume and potential volume. The revised model was then estimated using aggregate data from the entire test suite and the dummy variable’s values was used to determine the compiler’s relative rate of compilation. The revised model is given as

\[
\ln T = \ln K + a \ln V + b \ln V^* + e + f + \text{error.}
\]

Here, \( e \) and \( f \) represent coded variables (0 or 1) depending on the compiler being estimated. For example, the Unix compiler was designated \( e = 0 \) and \( f = 0 \) so that the discrimination rate for the Unix machine simply \( \ln K \). The AOS/VS compiler was designated \( e = 0 \) and \( f = 1 \) so that the least squares estimate of \( f \) was added to \( \ln k \) to estimate the discrimination rate of the AOS/VS compiler by itself. Likewise, the VMS-780 was coded \( e = 1 \), \( f = 0 \), and the VMS-785 was coded \( e = 1 \), \( f = 1 \). In this way, the \( V \) and \( V^* \) coefficients were constant across the data and the impact of the alternative compilers was directly available in \( e \) and \( f \).

Using binary dummy variables results in more efficient estimation because the use of a variable for each compiler would introduce four variables instead of the two shown. The use of fewer variables raises the degrees of freedom associated with the error term, and thus more realistically estimates the impact of each compiler. This model ensures that each compiler effect is estimated independently of the other compilers, while the software science parameters (\( a \) and \( b \)) are estimated across all the data.

Table VII shows the results of this effort. The results of using dummy variables does not change the compiler model from Table V except for the discrimination rate \( K \). Note that the exponents for \( V \) and \( V^* \) are approximately the same, i.e., 0.5 and 0.1, respectively.

However, “\( K \),” the translation rate, changes slightly. The discrimination rate for each compiler is significantly different from the base, UNIX, due to nonzero values of \( e \) and \( f \). The equations for each compiler now become

- \( \text{Unix} = T = 0.37(V^{0.48})(V^*^{0.07}) \)
- \( \text{AOS/VS} = T = 0.34(V^{0.48})(V^*^{0.07}) \)
- \( \text{VMS-780} = T = 0.21(V^{0.48})(V^*^{0.07}) \)
- \( \text{VMS-785} = T = 0.19(V^{0.48})(V^*^{0.07}) \)

As the equations suggest, the compilers would be ranked as shown in Table VIII.

Note that the compilers on the VMS machines were the same. Therefore, it can be concluded from above that the VAX 11/785 computer is faster than the VAX 11/780 computer by 10.1 percent (\( K \) being 0.21 versus 0.19). Clearly, the compiler/machine implementations are significantly different in performance. This difference is statistically valid and represents an index which we feel is more useful than simple averages. In fact, simple averages generated a rank order different from the software science approach!

The use of a software science model of compile time takes into account much more information about the algorithm being compiled, and therefore is able to predict the compile time much better than use of a simple average. The use of the discrimination rate is a sensitive estimate of the compiler and machine speed and generates a rank order which more closely represents the true processing power of the alternatives being compared.

V. CONCLUSIONS

A number of conclusions can be drawn from this analysis. First, the attempt to develop a measure which would provide a suitable approximation of the amount of time expended during the compilation process has been validated. The results suggest that the software science compiler model is a good tool for predicting compilation times and provides some fundamental insights concerning the compilation process. Hypothesis 1 is therefore rejected.

The signs and the magnitudes of the estimated parameters were not within the proximity of the theorized values. In particular, the value of the exponent for the volume (0.5 instead of 2.0) was unexpected. However, this value seems reasonable in that a compiler must expend more resources compiling several modules separately than compiling a single program containing all the modules. In addition, the exponent for the potential volume \( V^* \) was not negative and was not as significant in this application as comparing processing time.
The 13 tests that analyzed the predictive power of the model based on module characteristics showed that the compiler model was statistically significant at the 0.05 level for both categories of all 13 tests. Therefore, the compile time model is deemed useful regardless of program characteristics. Hypothesis 2 is therefore rejected and the complex tradeoffs between program characteristics are noted. It is useful to note the overall high explanatory power of the model regardless of the modules’ characteristics.

The explanatory power of the compile time model was consistently encouraging across four compiler/machine combinations. The predictive ability of software science in explaining differences in compilation rate across alternative Ada compilers was found significant and useful. The use of the discrimination rate proved to be a sensitive and appealing way to evaluate a compiler’s performance and is not as biased by aberrations in measurement or outliers as the simple arithmetic averages. Therefore, Hypothesis 3 is rejected.

The results of this research are twofold. First, a model of compile time has been developed and tested. The model suggests that modularization of code is costly in terms of compile time and appears valid across levels of program characteristics. Second, the development of a compiler performance index has been proposed. Clearly, the value of "K" represents the processing rate of a compiler. The results indicate that ranking a compiler’s performance solely on the average compile time is suspect. Based on the average compile time, the AOS/VS system was slower than the Unix system. In contrast, the software science model ranked the Unix system slower than the AOS/VS system. It was also shown that "K" can also compare computers when the operating environment is the same.

In this case, as expected, a VAX 11/785 was faster than a VAX 11/780.

In summary, a model of Ada compile time has been developed and validated. This study represents a preliminary exploration of the applicability of software science metrics to compilers. The results have indicated that there is enough evidence to continue investigating this area. Future research testing this model on other compilers and languages on a broader spectrum of data may further illuminate the compilation phenomena of compilers. A compiler index has been proposed, developed, and tested. With further research, the software science compiler model may become a valuable tool in evaluating compiler performance.

ACKNOWLEDGMENT

The authors wish to thank the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING Editor as well as three anonymous reviewers for their helpful suggestions and comments.

REFERENCES


