ON THE EFFICIENT GENERATION OF DYNAMIC PROGRAM PROFILES

by

Luis Felipe Cabrera

Memorandum No. UCB/ERL M81/38

1 June 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

On The Efficient Generation of Dynamic Program Profiles

by

Luis Felipe Cabrera [†]

Department of Mathematics
and
Electronics Research Laboratory
University of California, Berkeley
Berkeley CA 94720

Abstract

In this paper we present several methods which allow us to efficiently obtain dynamic profiles of programs. By scanning the code of a program, we build a representation of its profile equations. These representations of profile equations yield the exact profile of the original program as a function of the inputs.

We want these representations of profile equations to have an evaluation cost linear in the length of the program. Since this is not possible for all syntactically correct programs, we have devised several alternative ways to handle non-optimal situations. We have also determined conditions which allow us to choose among the various alternative methods. These conditions depend on the topology of the program's flowchart, on the algebraic complexity of the statements which modify a loop's control variables and on the syntactic nature of the control variables.

[†] The work reported here has been supported in part by the Computer Systems Design Program of the National Science Foundation under grants MCS-7824618 and MCS-8012900.

1. Introduction

The performance of software systems is becoming a central issue in the implementation and utilization of novel ideas and techniques in various fields of computer science. With hardware costs constantly decreasing, the availability of systems with a relatively large amount of main memory has become more widespread. Since these systems are now normally operated in a multiprogramming mode, they are very frequently utilized by many users at a time, creating contention for the installation's resources. The behavior of any program in a multiprogramming system often differs from that of the same program in a uniprogramming one because of the effects of the actions of other users as well as of the operating system. Programs almost always exhibit performance degradation, in terms of turnaround time, in multiprogramming systems.

It is not surprising then that software which is highly utilized (including in this category the operating system) is becoming the object of an increasing number of performance evaluation studies. It is known that the performance of a system depends on all of the aspects of its hardware and software configuration and on its workload. It is thus advantageous to have software which uses appropriate algorithms and, most important, that makes suitable usage of the resources available in the installation. Unfortunately, today there are no software design tools, or methodologies, which allow us to analyze a symbiosis of this kind between a program and the installation in which it will run. Not enough work has been done in this area, although there is interest and need. In [Smi80, Boo80], for example, we see discussions, implementations of methodologies and proposals on some of the issues that need be resolved to find a solution for this problem.

In the case of an existing program, when trying to analyze and/or predict its performance in a given installation, it is necessary to be able to determine exactly what resources and in what proportions the program requires to run. It would be very convenient if one could obtain this information in an efficient way, i.e., faster than by actually running the program and measuring it. We would like to have a *performance description* of the behavior of the program as a function of the values of its input variables, which would allow us to obtain efficiently the desired performance information. If such performance descriptions for programs were available, problems like comparing distinct software packages, or distinct implementations of a given algorithm, would become easier and less resource and time consuming.

2. Our Fundamental Problem

The *behavior* of a program means different things to people with different objectives. For example, one may be interested in the I/O activity, in the cpu requirements, in the number and type of arithmetic operations performed, in the amount of paging activity generated (in the context of a paged virtual memory system) or in the total running time. Each of these performance aspects of the execution of a program is normally a function of the value of the inputs to the program. It then becomes clear that the objective of the performance study must be established beforehand. Nevertheless, there exists a performance index which enables us to unify most of these studies. This index is a count of what gets executed in a run of a program.

A *basic block* is a linear sequence of program statements having one entry point (the first statement executed) and one exit point (the last statement executed). The *dynamic program profile* is a vector whose elements express the number of times each basic block is executed in a given run [Knu71b]. We shall often use the term *profile* to mean dynamic program profile.

Given a profile, it is rather simple to obtain several of the above mentioned performance aspects. The only one that may not be obtainable, depending on how intricate the flow of control structure is, is the dynamics of the memory demands produced by the program. As for all the other performance aspects, all that is necessary is some information which normally needs to be gathered only once per program-installation combination. Some of this information may occasionally be installation dependent.

For example, if we are interested in counting the different kinds of atomic operations that the program performs, then we need the information that associates with each basic block an itemized description of all the atomic operations performed by the statements in the basic block. Then, once we obtain the profile for basic blocks, we only have to multiply the value associated with a specific basic block by the number of times each atomic operation is performed in it to obtain the counts of the operations executed. This procedure is certainly installation independent, thus, once this information has been found for a program, it never needs to be recomputed.

However, if we are interested in estimating the running time of a program when executed with a given set of inputs, we need installation dependent information. In particular, if we assume a uniprogramming environment, what we need to find out is the (average) time each atomic operation takes as well as the execution time of each kind of branching statement appearing in the program. For machine and assembly language programs this information may be obtained from tables supplied by the hardware manufacturer. If we are dealing with a program written in a high level language, then we need information which is both compiler and system dependent, because what we are really interested in is the (average) time a compiled atomic operation takes. It should be clear that each time a new compiler is installed or a new system considered, a new table has to be obtained for the atomic operations we are interested in.

We shall call *profile equations* of a program those expressions which express the frequency counts of basic blocks as functions of the input data. Thus, if we had an appropriate representation for the profile equations, we would be able to obtain the profile of the program in an efficient way. The best achievable is to have an evaluation cost linear in the length of the representation of the profile equations.

This paper explores different alternatives which will enable us to obtain profiles for programs in an efficient manner. In fact, we will describe automatic ways of representing the profile equations for a program and conditions under which they will yield the profiles with linear time evaluation cost. These methods will allow us to obtain profiles much faster than by actually running (a properly instrumented version of) the programs.

3. Some Related Work

Donald Knuth has pioneered the area of the mathematical analysis of algorithms [Knu71a, Knu71b, Knu78]. In this analysis, for the execution time of a given algorithm or program, one attempts to determine the four quantities

<maximum, minimum, average, standard deviation>.

The fourth quantity refers to the standard deviation of the distribution of execution times around the average. In [Knu78] we can see that the complete analysis of a rather simple algorithm may require complex mathematical knowledge and expertise. It then becomes quite clear that analyzing large real-life programs may be an enormous task. The required amount of sophistication and level of reasoning about the program seems to go beyond the current level of what can be automated.

With a different approach, since 1974 Jacques Cohen and his collaborators have been *microanalyzing* structurally simple programs, i.e., determining the above mentioned four quantities as functions of each elementary operation involved in the program. In [Coh74] Cohen presented a system which would accept programs in a restricted Pascal-like programming language and would return an expression of its execution time as a function of the processing time of elementary operations. However, the evaluation of this expression requires the user to specify the number of times the body of a loop would be traversed and the branching probabilities of conditional statements. These two conditions make this approach very difficult to use when one is trying to *gain* knowledge about the behavior of a program.

The simple structure of many algorithms has proved that the method can yield interesting results. In [Coh76a] we see an analysis of Strassens's matrix multiplication algorithm. A non recursive version of the algorithm has all loops traversed a fixed number of times and no conditional statements within loops. This allows the authors to find a closed form expression for the processing time of the algorithm whose evaluation does not lead to inconsistencies. In their expression, specifying the number of times a loop is to be traversed is given by the dimension of the matrices. Then, as all the bodies of the loops are basic blocks, the evaluation yields the exact profile of the run.

We shall call Cohen's approach the *deterministic microanalysis* of programs because of the requirement that the user provide the number of times a loop will be executed and the (fixed) probability that a conditional branch will be taken. A big drawback of this method is that, in any relatively complex program, the interrelationships between statements may become very obscure and intricate. It is unreasonable to expect that a user will master them and provide consistent data for the evaluation of the expressions. The fact that these expressions do not depend on the input variables of the analyzed algorithm or program appears to be responsible for most of the method's deficiencies.

A different approach can be found in Ferrari's work, [Fer78], where programs are viewed as D-charts and formulae are built in a bottom up fashion taking into account all the data dependencies. Unfortunately, the methodology used there did not clarify *when* one could obtain such expressions. Only very simple examples were found to be manageable. However, the expressions obtained were functions of the input variables and thus when supplied with values for them a correct profile was obtained. The task of finding expressions became more complicated but their evaluation required no further information from the user, and the answer obtained was always correct.

To obtain the four quantities desired using Ferrari's expressions, one has to find suitable input data that would exercise the program in such a way as to achieve its minimum and its maximum; then, making some probabilistic assumptions on the nature of the input data, one is able to determine the average and standard deviation with some predetermined degree of statistical confidence by measuring enough samples of the input data. In fact, Cohen's approach requires the same kind of hypothesis with the additional problem that, for a given assignment of values to the number of times loops are traversed and branches taken, one may obtain evaluations which do not represent the execution of the program under any given set of inputs.

In [Weg75], the system Metric is presented. With it a very limited class of Lisp programs can be correctly microanalyzed. The highlights of Metric are that it knows how to find closed form formulae for recursive programs (in its restricted Lisp environment), deals with algebraic simplifications and expresses the execution behavior as a function of the size of the input. Moreover, Metric also allows several measures of performance to coexist. This provides a degree of flexibility that Cohen's system does not have. However, when computing the maximum and minimum execution time of a program, as in Cohen's system, several "simplifying" hypotheses are made which yield bounds not necessarily tight. In other terms, there may be no set of inputs which would make the program attain these bounds.

The very fertile area of Symbolic Evaluation or Symbolic Execution of programs has undisputed relevance to our problem. In [Che79, Che76, Che78, Kin76, How78] we read about different systems which attempt to express in a symbolic way the results of the computations performed by a program. Common to all of them, and to any system which performs such a task, is the problem of dealing with loops. The effect that such a construct has on the value of a variable is central to the analysis in all approaches. All of these authors are primarily concerned with the correctness, and not the performance, of the analyzed programs.

4. Obtaining Dynamic Profiles

4.1. The Skeleton Approach

When we are interested in a count of the basic blocks as a function of the values of the input variables of a program and all we have is the program to work with, we will normally find that the program has many statements which do not play any role in this process. We shall now make this observation more precise.

Given a variable name x, we shall say that x is a *control variable* if its value affects the flow of control of the program. It is clear, then, that all statements in the program which do not modify directly or indirectly control variables may be excluded from an analysis whose sole purpose is to find the profile of a program.

We have thus found a first approach to our problem of efficiently generating program profiles. Given a program P, we construct a program P_1 and suitable tables $T_1, \dots, T_n$, which will enable us to obtain performance information about P much more rapidly than by actually running an instrumented version of P in the following way:

(1) P_1 is obtained by deleting from P all those statements which do not affect the flow of control of P, i.e., those statements which do not modify control variables. Moreover in each basic block, i, we add a statement of the form $B_i := B_i + 1$, where the variable B_i does not appear in the original program, has not been used before in P_1 and is associated with the basic block in a unique way. We also add, at the very beginning of P_1, statements which initialize each and every one of these new variables B_i to zero.

(2) The tables $T_1, \dots, T_n$, represent mappings between these names B_i and different kinds of information required for our performance studies. For example, one of these mappings will always be the one which matches the names B_i and the actual sequences of statements which constituted the corresponding basic block.

We shall call P_1 the (flow-of-control) *skeleton* of P.

Notice that, in the first clause defining P_1, we do not need to introduce so many new variables B_i to describe the profile of P, because in any program there is redundancy of flow-of-control information which can be used advantageously [Knu73]. All one needs is to have one variable counting the executions of each independent path. Basic blocks which will always execute sequentially may be accounted for by the use of only one variable.

Discovering which statements affect the flow of control of a program can be done in an automated way by the global data flow analysis methods used for code optimization [Aho79]. The methods essentially consist of a multipass procedure where, in the first pass through the code, data flow information is gathered and suitable data structures that will keep this information for further processing are built. In subsequent passes the information is appropriately used.

As for the tables required, some of them can easily be generated automatically. For example, while parsing the program, the compiler can identify all the basic blocks, when they are encountered, and count the distinct types of atomic operations which appear in each block. However, it should be clear that some other tables require extra effort to obtain. For example, suppose we are interested in determining the page trace of a run of a program; then, we need to know the physical layout of a compiled version of the program to determine which pages correspond to a given basic block. This has to be done after the actual machine code has been produced.

To illustrate the form, effectiveness and usage of the skeleton of a program, as well as its limitations, we shall present two examples. They are taken from a large FORTRAN program (11000 lines of code) called SPICE. SPICE [Coh76b, Nag75] analyzes integrated circuits to determine their electrical and thermal properties. We have chosen to study (parts of) it because it is an example of a large program which is constantly used at Berkeley and whose behavior has been analyzed using only conventional techniques.

For our immediate purposes we have chosen two parts of the subroutines TMPUPD and MATLOC for which we will build the skeletons. In both cases we will see that obtaining the profile from the skeleton is much faster than obtaining it from the original code. Appendix A contains the original FORTRAN code for each of the two portions of the subroutines we use.

Example 4.1.1

In Table 4.1.1 we show the code for the skeleton of part of the subroutine TMPUPD. We see that all the statements are very simple and thus of quick execution. We have not included redundant counters. When analyzing the results, one reconstructs the full profile by taking into account the interdependencies used to eliminate statements from the original code.

```
410 IF (LOC.EQ.0) GO TO 1000
```

```
   B26=B26+1
   IF(IPRNT.NE.0) B27=B27+1
   IPRNT=0
   IF (ITEMNO.LE.2) GO TO 415
   B29=B29+1
415 CONTINUE
   IF (ITEMNO.LE.2) GO TO 420
   B31=B31+1
420 CONTINUE
430 LOC=NODPLC(LOC)
   GO TO 410
```

Table 4.1.1 Skeleton of a part of TMPUPD

■

Example 4.1.2

Table 4.1.2 depicts the skeleton of part of the subroutine MATLOC. What we should notice in this example is that, since we have nested loops, the execution time of the skeleton would improve substantially if we could "linearize" them. In fact, from analyzing the code we see that each of the inner nested loops, is traversed NDIM times each time the T-branch of their respective outer loop is taken. NDIM is a variable which is not modified within MATLOC, it is an input value for this subroutine. Linearizing these loops is then a fairly straightforward matter. After the count for the corresponding outer loop is found, one multiplies it by NDIM and obtains the count for the inner loop.

```
768 LOC=LOCATE(7)
   B17=B17+1
770 IF (LOC.EQ.0) GO TO 772
   B18=B18+1
   DO 771 I=1,NDIM
   B19=B19+1
771 CONTINUE
   LOC=NODPLC(LOC)
   GO TO 770
```

Table 4.1.2 Skeleton of a part of MATLOC

■

The only reference known to us that uses an idea similar to the skeleton is in [Pra79], where programs are decomposed into a control part (a subset of our skeleton) and a kernel part (which is only concerned about computing output values). The author uses this idea to study program equivalence, termination and code optimization. It should be clear that any two programs which share control structures will behave identically in their profile equations, and moreover, from the viewpoint of their termination, one will halt if and only if the whole class of programs with the same control structure halts.

In the following sections we shall study families of representations for programs which improve on our skeleton approach (by having a faster running time) and which will always be dependent on the input data. To achieve this we need to introduce pertinent formalisms.

4.2. The Program Performance Formulae Approach

We shall study non recursive *goto*-less programs. As done in [Fer78], we shall represent this kind of programs by single-entry single-exit directed graphs called D-charts. From now on, we shall use program as synonym for the D-chart which represents it. In [Cab81] we find the generalization of this discussion to all programs.

With each D-chart D, we associate a unique *program performance formula* ψ_D (ppf ψ_D for short) as follows:

(1) For each indecomposable elementary D-chart B (i.e., B represents a basic block of instructions), we assign to the basic block a special denotation symbol B_i (never to be used again for any other basic block) and the ppf $1B_i$ to the elementary D-chart.

(2) If B_1 and B_2 are elementary D-charts with assigned ppf's ψ_1 and ψ_2 respectively, then the concatenation $\psi_1\psi_2$ is the ppf assigned to their composition.

(3) Given an alternation construct where D_1 and D_2 are the elementary D-charts associated with the T and F branches respectively and φ is the predicate, the ppf associated with it is

$$\text{IF } (\varphi \text{ , } 1) \text{ THEN } \psi_1 \text{ ELSE } \psi_2 \text{ FI,}$$

where ψ_1 and ψ_2 are ppf's associated with D_1 and D_2 respectively, and 1 represents the real valued constant function whose value is 1, i.e.: $1(x) = 1$ for all $x \in \mathbb{R}$.

(4) Given an iteration construct D where D_1 is the elementary D-chart associated with the T-branch and φ is the predicate having n variables, the ppf associated with it is

$$\text{IF } (\varphi \text{ , } f) \text{ THEN } \psi_1 \text{ ELSE } \Lambda \text{ FI ,}$$

where ψ_1 is the ppf associated with D_1 and f is an n-place function symbol with the same variables as φ which, when evaluated with the value of the variables at the entrance of the iteration, yields the number of consecutive times that φ would evaluate to true in the corresponding run. We shall denote such a function f associated with φ by $\#\varphi$. $\#\varphi$ is called the *counting function* of φ.

The semantics for ppf's is quite simple because there are no quantifiers in the language. In fact, the lack of quantifiers enables us to evaluate any ppf in a one-pass left-to-right manner. The interpretation of a ppf will yield a (finite) sequence of symbols which is meant to represent the profile of a program, when the program is run with the inputs used to evaluate the ppf.

We define the *interpretation function* I by induction on the complexity of ppf's:

(1) for any special denotation symbol B_i, $I(B_i)[i] = 1B_i$.

(2) for any performance formula ψ, where ψ is $\psi_1\psi_2$, $I(\psi_1\psi_2)[i] = I(\psi_1)[i]I(\psi_2)[i]$.

(3) For any formula φ, n-place function symbol f , terms $t_1, \ldots, t_n$ and ppf's ψ_1, ψ_2, I(IF (φ , $f(t_1, \ldots, t_n)$)THEN ψ_1 ELSE ψ_2 FI)[i] is equal to $f^{\mathbb{R}}(i(t_1), \ldots, i(t_n)) \times$ I(ψ_1)[i] if $\varphi[i]$ is true, and equal to $f^{\mathbb{R}}(i(t_1), \ldots, i(t_n)) \times$ I(ψ_2)[i] if $\varphi[i]$ is false, where for any $x \in \mathbb{R} \cup \{\infty\}$, $\infty \times x = x \times \infty = \infty$. If ψ_j is Λ, for j $\in$ $\{1,2\}$, then we say that $f^{\mathbb{R}}(i(t_1), \ldots, i(t_n)) \times$ I(ψ_j)[i] is Λ.

In this formalism, the inner iteration in Example 4.1.2 is expressed by the ppf IF (I $\leq$ NDIM , **NDIM**) THEN B_{19} ELSE Λ FI, and its interpretation is **NDIM** B_{19}, where **NDIM** represents the function whose value is the value of the variable NDIM.

It is not difficult to prove that if a given program P halts, then the interpretation of its associated ppf is a string of the form $a_0 B_0 a_1 B_1 \cdots a_n B_n$ where each a_i is a positive integer [Cab81].

We have thus described a way to associate ppf's with any program. Moreover, the interpretation of any ppf yields a vector of numbers $<a_0, a_1, \ldots, a_n>$ which is identical in form to the profile of a program. The question we now consider is the following: when does a ppf represent the profile equations of a program? That is to say, under what conditions does the evaluation of a ppf always yield a sequence which *is* the profile?

The answer to this question is given by the following two theorems which can be proved by induction on the complexity of ppf's [Cab81].

Theorem 1

For any elementary D-chart D where there are neither alternations nor iterations within an iteration the ppf ψ_D represents the profile equations of D.

■

In any D-chart we say that an alternation within an iteration is *well behaved* if the same branch of the alternation is traversed each time the T-branch of the iteration is traversed. We say that an iteration within an iteration is *well behaved* if there exists an integer n such that each time the T-branch of the outer iteration is traversed the T-branch of the inner iteration is traversed n times.

Theorem 2 (Representability of Profile Equations)

For any D-chart D, ψ_D represents the profile equations of D iff for all assignments to the free variables i, all alternations and iterations are well behaved.

■

Theorems 1 and 2 characterize the class of programs for which we can obtain their dynamic profiles in linear time. It is worth noting that even though this is a fairly restricted subclass of all programs, a survey of the published literature has showed that all the examples discussed in it satisfy the hypotheses of at least one of the theorems.

4.3. Our Final Approach

The efficient representation of dynamic profiles of programs requires a combination of the techniques presented in the two previous sections. There are many programming constructs which are not amenable to the automatic discovery of counting functions. In fact, both of our examples show that the (outer) iterations are traversed while a given list does not finish. Finding the length of these lists, which is exactly the number of times the body of the loops will be traversed, cannot be done without external information. However, in Example 4.1.2 we have that the inner loop can be treated using our ppf approach. Thus the most appropriate way to handle programs in general is to combine both methods. Whenever a ppf representing the profile equations of a (sub)program is found, it should be used. Otherwise, a combination of ppf's and a skeleton will also improve on a pure skeleton. In each of the three

cases, if the programs to be analyzed perform a fair amount of computations, our methods yield the profile faster than obtaining it from an instrumented version of the program. An extensive discussion of different alternatives is given in [Cab81].

Example 4.3.1

Table 4.3.1 depicts Example 4.1.2 when we use both techniques simultaneously. The interpretation of the ppf corresponding to the inner iteration has been written in FORTRAN.

```
768 LOC=LOCATE(7)
   B17=B17+1
770 IF (LOC.EQ.0) GO TO 772
   B18=B18+1
   LOC=NODPLC(LOC)
   GO TO 770
   B19=B17*NDIM
```

Table 4.3.1 Final Representation of a part of MATLOC

■

5. Conclusions

In this paper we have presented an approach which allows us to efficiently obtain the dynamic profile of programs. Its effectiveness depends on the nature of the programs analyzed. We have determined the class of programs whose dynamic profile can be obtained in linear time. A survey of the literature has shown that all published examples fall into this category.

For those programs whose dynamic profile cannot be obtained in linear time, we have devised alternative strategies which also allow us to build efficient representations of their profile equations. Even though the evaluation cost of these representations may not be linear in their length, in most cases they produce the profile of the program substantially faster than when obtained from the program. We have also introduced a method, the *skeleton approach*, which is applicable in all circumstances but whose running time may be close to that of the original program.

Our most efficient representations rest on the ability to determine the *counting functions* of an iteration. This is a function which gives the number of consecutive times the body of an iteration will be traversed when presented with the values of the control variables at the entrance of the iteration. In [Cab81] it is shown how to obtain automatically counting functions for a wide variety of cases.

Using known techniques of data flow analysis one can gather all the information needed to produce our efficient representations of the profile equations of programs. It is thus feasible to implement a system which would produce, after scanning the code of a program, an efficient representation of its profile equations.

6. Acknowledgements

I thank Domenico Ferrari for introducing me to this area of Computer Science and to the members of the PROGRES group for their useful comments and insights.

7. Bibliography

[Aho79] Aho, Alfred V. and Ullman, Jeffrey D., *Principles of Compiler Design*, Addison Wesley (April 1979).

[Boo80] Booth, Taylor L. and Wiecek, Cheryl A., "Performance Abstract Data Types as a Tool in Software Performance Analysis and Design," *IEEE Transactions on Software Engineering* **SE-6**(2) pp. 138-151 (March. 1980).

[Cab81] Cabrera, Luis Felipe, "Syntax Oriented Analysis of the Run Time Performance of Programs," ERL Memorandum M81/30, University of California, Berkeley (May 13, 1981). Ph.D. Dissertation

[Che76] Cheatham, Thomas E. and Townley, Judy A., "Symbolic Evaluation of Programs, A look at Loop Analysis," pp. 90-96 in *Proceedings ACM Symposium on Symbolic and Algebra Computation*, (1976).

[Che78] Cheatham, Thomas E. and Washington, D., "Program Loop Analysis by Solving First Order Recurrence Relations," TR-13-78, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts (1978).

[Che79] Cheatham, Thomas E., Holloway, Glenn H., and Townley, Judy A., "Symbolic Evaluation and the Analysis of Programs," *IEEE Transactions on Software Engineering* SE-5(4) pp. 402-417 (July 1979).

[Coh76a] Cohen, Jacques and Roth, Martin, "On the Implementation of Strassen's Fast Multiplication Algorithm," *Acta Informatica* **6** pp. 341-355 (1976).

[Coh76b] Cohen, Ellis, "Program Reference for SPICE2," ERL-M592, ERL Memorandum, University of California, Berkeley (June 1976).

[Fer78] Ferrari, Domenico, *Computer Systems Performance Evaluation*, Prentice-Hall (1978).

[How78] Howden, William E. , "DISSECT- A Symbolic Evaluation and Program Testing System," *IEEE Transactions on Software Engineering* SE-4(1) pp. 70-73 (Jan. 1978).

[Kin76] King, James C., "Symbolic Execution and Program Testing," *Communications of the ACM* **19**(7) pp. 385-394 (July 1976).

[Knu71a] Knuth, Donald E., *Mathematical Analysis of Algorithms*, IFIP Congress, Ljublijana (Aug. 1971).

[Knu71b] Knuth, Donald E., "An Empirical Study of FORTRAN Programs," *Software - Practice and Experience* 1(1) pp. 105-133 (1971).

[Knu73] Knuth, Donald E. and Stevenson, F. R., "Optimal Measurement Points for Program Frequency Counts," *BIT* **13** pp. 313-322 (1973).

[Knu78] Knuth, Donald E. and Jonassen, Arne T., "A Trivial Algorithm whose Analysis isn't," *Journal of Computer and Systems Sciences* **16**(3) pp. 301-322 (1978).

[Nag75] Nagel, Lawrence W., "SPICE2: A Computer Program to Simulate Semiconductor Circuits," ERL-M520, ERL Memorandum, University of California, Berkeley (May 1975).

[Pra79] Pratt, Terrence, "Program Analysis and Decomposition Through Kernel-Control Decomposition," *Acta Informatica*, (9) pp. 195-216 (1979).

[Smi80] Smith, Connie U., "The Prediction and Evaluation of the Performance of Software From Extended Design Specifications," TR-154, University of Texas, Austin, Texas (Aug. 1980). Ph.D. Dissertation

[Weg75] Wegbreit, Ben, "Mechanical Program Analysis," *Communications of the ACM* **18**(9) pp. 528-539 (Sep. 1975).

8. Appendix A

In this appendix we include the fragments of the FORTRAN source code of the subroutines TMPUPD and MATLOC from SPICE used in Examples 4.1 and 4.2.

Fragment of SUBROUTINE TMPUPD

```
410 IF (LOC.EQ.0) GO TO 1000
    LOCV=NODPLC(LOC+1)
    TYPE=NODPLC(LOC+2)
    IF(IPRNT.NE.0) WRITE (IOFILE,401)
401 FORMAT(//'0**** MOSFET MODEL PARAMETERS',/,'0NAME',8X,'VTO',8X,
  1  'PHI',9X,'PB',7X,'IS(JS)',7X,'KP',9X,'UO'//)
```

```
      IPRNT=0
      RATIO4=RATIO*DSQRT(RATIO)
      VALUE(LOCV+3)=VALUE(LOCV+3)/RATIO4
      VALUE(LOCV+29)=VALUE(LOCV+29)/RATIO4
      OLDPHI=VALUE(LOCV+5)
      PHIO=VALUE(LOCV+5)
      IF (ITEMNO.LE.2) GO TO 415
      PHIO=(VALUE(LOCV+5)-PBFAT1)/FACT1
  415 VALUE(LOCV+5)=FACT2*PHIO+PBFACT
      PHI=VALUE(LOCV+5)
      VFB=VALUE(LOCV+44)-TYPE*0.5D0*OLDPHI
      VFB=VFB+0.5D0*(OLDEG-EGFET)
      VALUE(LOCV+44)=VFB+TYPE*0.5D0*PHI
      VALUE(LOCV+2)=VALUE(LOCV+44)+TYPE*VALUE(LOCV+4)*DSQRT(PHI)
      VALUE(LOCV+11)=VALUE(LOCV+11)*DEXP(-EGFET/VT+OLDEG/OLDVT)
      VALUE(LOCV+21)=VALUE(LOCV+21)*DEXP(-EGFET/VT+OLDEG/OLDVT)
      OLDPB=VALUE(LOCV+12)
      PBO=VALUE(LOCV+12)
      IF (ITEMNO.LE.2) GO TO 420
      PBO=(VALUE(LOCV+12)-PBFAT1)/FACT1
      GMAOLD=(OLDPB-PBO)/PBO
      COEOLD=1.0D0+VALUE(LOCV+18)*(400.0D-6*DTEMP-GMAOLD)
      VALUE(LOCV+9)=VALUE(LOCV+9)/COEOLD
      VALUE(LOCV+10)=VALUE(LOCV+10)/COEOLD
      VALUE(LOCV+17)=VALUE(LOCV+17)/COEOLD
      VALUE(LOCV+19)=VALUE(LOCV+19)/(1.0D0+VALUE(LOCV+20)
     1    *(400.0D-6*DTEMP-GMAOLD))
  420 VALUE(LOCV+12)=FACT2*PBO+PBFACT
      GMANEW=(VALUE(LOCV+12)-PBO)/PBO
      COENEW=1.0D0+VALUE(LOCV+18)*(400.0D-6*DTEMP-GMANEW)
      VALUE(LOCV+9)=VALUE(LOCV+9)*COENEW
      VALUE(LOCV+10)=VALUE(LOCV+10)*COENEW
      VALUE(LOCV+17)=VALUE(LOCV+17)*COENEW
      VALUE(LOCV+19)=VALUE(LOCV+19)*
     1  (1.0D0+VALUE(LOCV+20)*(400.0D-6*DTEMP-GMANEW))
      PBRAT=VALUE(LOCV+12)/OLDPB
      VALUE(LOCV+37)=VALUE(LOCV+37)*PBRAT
      VALUE(LOCV+38)=VALUE(LOCV+38)*PBRAT
      CSAT=DMAX1(VALUE(LOCV+11),VALUE(LOCV+21))
      WRITE (IOFILE,31) VALUE(LOCV),VALUE(LOCV+2),VALUE(LOCV+5),
     1  VALUE(LOCV+12),CSAT,VALUE(LOCV+3),VALUE(LOCV+29)
  430 LOC=NODPLC(LOC)
      GO TO 410
```

Fragment of SUBROUTINE MATLOC

```
  768 LOC=LOCATE(7)
  770 IF (LOC.EQ.0) GO TO 772
      NODE1=NODPLC(LOC+2)
      NODE2=NODPLC(LOC+3)
      NDIM=NODPLC(LOC+4)
      LOCVS=NODPLC(LOC+6)
      LMAT=NODPLC(LOC+7)
      DO 771 I=1,NDIM
      LOCVST=NODPLC(LOCVS+I)
      IBR=NODPLC(LOCVST+6)
      NODPLC(LMAT+1)=INDXX(NODE1,IBR)
      NODPLC(LMAT+2)=INDXX(NODE2,IBR)
      LMAT=LMAT+2
  771 CONTINUE
      LOC=NODPLC(LOC)
      GO TO 770
```