Commission of the European Communities

****************

ESPRIT III

PROJECT NB 6756

****************

# CAMAS

## COMPUTER AIDED MIGRATION OF APPLICATIONS SYSTEM

****************

CAMAS-TR-2.2.4.3
SAD technical report

****************

Date: March 1994 — Review 3.0

ACE - Univ. of Amsterdam - ESI SA - ESI GmbH - FEGS -
PARSYTEC - Univ. of Southampton

Authors: Berry A.W. van Halderen
Jan de Ronde
Marcel Beemster
P.M.A Sloot

March, 1994
University of Amsterdam,
Faculty of Mathematics and Computer Science
Parallel Sciencific Computing and Simulation group
Netherlands

# 1 Introduction

The performance modelling of complex programs is a tiresome and error prune process when done for a large program. Counting the number of instructions in the program, analyzing the program structure and combining them in a detailed time complexity function takes up a lot of human time and because it is a precise but often boring task is likely to be subject to errors.

More important, numerical programs can be so large and complex that the one simply looses the overview over the program. Tens of thousands lines of code are not uncommon for numerical programs; tracking the variable usage and subroutine nestings has then become a difficult task and if we want to model the performance characteristics of a program we need to analyze the program structure. With such a large structure, which can be very unclear in dusty deck programs, the modelling task has become problematic.

If this work is done only once, it is acceptable to put some effort into it to do it by hand, but in a research or development environment we want to experiment with the different factors that influence the performance. Changing these factors change the time complexity formula, and requires to redo some of the work. When this is the case, it is necessary to create an environment in which most of the mind boring tasks are automated and which assist the developer in the performance evaluation. Such a tool is described in this report.

We describe the performance of a program using a SAD formula, which expresses the time complexity. The SAD formula (as described in [1]) consist of three layers. The first level of the SAD formula describes the execution cost of a basic block, which is simply a summation of the execution cost of the various instructions. The second level adds the control flow to the SAD formula. The execution time of a program is the execution time of each basic block times the number of times each block is executed. The second level SAD formula is therefor a summation of the product of the execution cost of a basic block and a certain factor. The factors are determined by the expressions in `if` and loop program constructs.

The factors in this formula are however not independent, since in the real program the `if` constructs and loops are also nested and often depend on each other. Therefor we can split the factors in a product of a number of subfactors, where each factor may occur as a multiplication factor of more than one particular basic block. This results in the following generic SAD description:

$$SAD_2 = \sum_{i=1}^{N} \prod_{m_i}^{M_i} \prod_{k_i}^{K_i} P_{k_i} X_{m_i} Block_i \tag{1}$$

The sum of the execution costs is denoted by $Block_i$. For instance, for the statements:

```
A = 1 + B * 5
```

```
B = A ** 9
```

the $Block_i$ is equal to:

$$M(addition) + M(multiplication) + M(Store) + M(Power) + M(store)$$

.

The $P_{k_i}$ and $X_{m_i}$ are the multiplication factors derived from `if` and loop constructs respectively. In the actual execution their values are fixed for a certain input, but for simulation purposes these can be mapped to probabilistic functions.

The third level of the SAD formula is to express the data locality. In this way we will be able to describe the performance of SPMD parallel programs. This level is part of further development.

The main task of the tool which we are developing is to generate a SAD formula, suitable for execution time estimation purposes. A SAD formula when directly generated from the entire source code, however may prove too large to handle. Also, the user of these simulation tools would often like to get some more information about the program. This could vary from describing which part of the program results in a certain multiplication factor or to being able to isolate code fragments for smaller simulations or profiling. This makes the tool a more interactive program which forms a base for making a SAD formula which can then be used for simulation.

An interactive tool also allows you to annotate the source program with information you can determine, but which cannot be determined in an automatic manner. This may result in a better, smaller SAD formula.

In the next two sections the generation of a SAD formula is described, followed by a section about increasing the quality of the SAD formula.

**arithmetic and logical** A basic computational function, like an addition, multiplication or exponential.

**procedure call** The actions needed to call a function or procedure. This consist of a subroutine call and the management of the arguments.

**array references** The action for referring to a 1, 2 or $n$ dimensional array.

**jumping and iteration** The actions needed for `DO` loop (in other languages called a `FOR` loop) like initialization and re-iterating or the action to perform a `GO TO`.

**builtin functions** intrinsic functions in Fortran or in more general library calls, like log , exp , etc.

Table 1: A classification of the abstract machine instructions

## 2 The execution cost of a basic block

In the introduction we stated that we wanted to derive a SAD formula, describing the time complexity, from the source code. We must now first ask ourselves what kind of source code. In the field of numeric computing, many applications are still written in Fortran, so Fortran seems to be a logical choice for this project. We could also choose to take the generated machine instructions by a Fortran compiler. This has the potential to be more precise because we have more detailed knowledge of the underlying machine. This however, servery restricts us in experimenting with different machines, because we need to rewrite the entire tool for this. If we base ourself on a high level language, like Fortran, we do not have this limitation.

In the same manner it is not sensible to restrict oneself to a particular high level language. Not only because we can then only process this particular language, but also because we run the risk of being so language dependent that we lose sight of the underlying programming constructs. Aiming at the basic programming language constructs helps us writing simpler algorithms for rewriting the original source code into the resulting SAD formula and obtaining further information about the program.

It is common practice to define an intermediate representation, which in our case is a very high level description. This representation is close enough to the original input source code to be a straightforward translation —and thus a close match in the performance estimation— and yet general enough to allow easy manipulation. This representation will be introduced gradually and is summarized in appendix A.

The first level of the SAD formula expresses the execution cost of basic instructions. The instructions are based upon an abstract machine which closely resembles the actions defined in the high level language, but are detailed enough to express major performance characteristics of common computers.

The table 2 gives an indication of the kind of instructions modelled. The choice of the instructions are described in more detail in [5].

It is inevitable that some of these instructions are language dependent, but including some additional instructions for a different programming language does not undermine the notion of being largely language independent. New instructions which are specific to other languages can be added easily, but the main structure of the instruction set remains the same and imperative languages share the same kind of operations. A nice characteristic of this instruction set is that it is machine independent; it is not even restricted to stack-based machines.

The execution cost of an expression can easily be determined by counting the occurrence of the operations in the expression. Since the Parasol machine database has been constructed in such a way to cover at least the main Fortran performance factors, all operations in Fortran find their counterpart in the database. Also, the main intrinsic functions available in Fortran are available in the database, and can thus be treated as the other operations.

The fact that the items in the machine database for operations in expressions have been designed for Fortran, is not a limitation of the machine database, since most imperative programming languages provide the same kind of basic operations on expressions (like addition, multiplication, power of, etc). Therefor the arithmetic and logical operations are not restricted to a certain language.

# 3 Adding control flow

A structured program is a program in which every part of the program has only one entry and one exit point. In other words, the execution of every part of the program always begins at a certain statement and it finishes after executing a specific last statement of that part.

This implies that, in a structured program, we can only use loops and if-then-else statements. It is not possible to continue execution from the body of the then part of an `if` statement to the middle of a loop statement. This violates the fact than a loop statement can only be entered through one single point.

Generating a time complexity function of a structured program is much simpler then generating it from a program containing complicated go-to structures. Also other analyzing techniques are greatly simplified or depend on the program being highly structured. One of these techniques, symbolic evaluation, we will be using in a later section of this report.

Structured programs are simpler for the process later on, but this does imply that we must transform a possible less structured program into a highly structured program. Less structured programs are in fact common usage, even in clear and well written code. Look at the following example, where an element is searched within a certain array:

```
      DO 10, I=1, 100
         IF(SEARCHVALUE.EQ.ARRAY(I)) GOTO 20
 10   CONTINUE
C     element not found, do something
      I = -1
 20   WRITE(*,*) 'index of value in array: ', I
```

Line 20 can be reached when the loop ends (after `I` becomes 100) or when the go-to is taken (the element has found), the program is thus not highly structured although this is very common programming style because this decreases the average time complexity by a factor two.

The above example can be rewritten into a highly structured program. A generic algorithm is described in the next subsection and two special cases are described in the next section.

Before we describe the restructuring algorithm, we define the input of the algorithm. The input is not directly Fortran source code, but an intermediate format. This format is simple and portable to use between different imperative languages. The translation from Fortran to this intermediate language is relatively easy and not described in this report.

First we define the control flow statements of this intermediate format. There are two kind of control statements for structured programs, the choice (better known as the if statement) and three loop iteration statements. The if statement has three parameters: The first argument is an expression, which outcome must be boolean and on which outcome is determined which of the other two arguments must be evaluated. The second and third arguments contain the corresponding statements which must be evaluated for a true value (then-branch) or false value (else-branch).

We define three kinds of loop iteration statements:

1. A loop statement at which the number of iterations of the body is known just before the loop is executed. In Fortran, this is the `DO`-statement; other languages sometimes call it a `FOR` statement (N.B.the C `for` statement is *not* such a statement).

   We call this kind of loop statement a `LOOP`. A `LOOP` has two parameters: An expression, which is evaluated just before the loop is entered, and which outcome determines how many times the statements in the loop are executed. The second parameter contains the statements over which the iteration process must occur.

   Note the fact that we don't include a loop counter in this statement. The initialisation of this counter, and it's evaluation during the loop must be defined separately.

2. A loop statement where just before a next (or first) iteration is determined whether or not to continue the iteration process. This is known as an while-do loop. The `WHILEDO` loop statement also has two parameters, an expression and a body of statements. Now however, the expression is evaluated every time the loop iterates.

3. A loop statement where the decision to make a next iteration is taken after an iteration has been done, is known as a do-while loop. The `DOWHILE` loop statement, like the `WHILEDO`, also has two parameters: a body of statements and an expression which evaluates to either true or false during execution.

Although it is sufficient to provide only a do-while or while-do iteration loop, it is simpler to define these basic three. The structure of most programs is maintained by defining separate loop statements and we don't need unnecessary algorithms, which may lead to an exponential increase of the input program size[7]. The other operations in a structured program can be expressed as an expression. Expressions can have side effects, as long as the execution always continues at the statement after the expression. One of the main side effects we have in mind is the assignment operator. The statement `EVAL` accepts one argument; an expression which is evaluated.

Less structured programs contain jumps from one statement to another, therefor we need an additional statement which expresses such a jump. The `GOTO` statement is just that, it accepts one argument which is a label. The jump during execution is to a corresponding label which is defined by the special statement `LABEL`, which also accepts a label as its argument. After the restructuring process, an equivalent program is generated which does not contain any `GOTO` or `LABEL` statements anymore.

## 3.1 Restructuring the source code

The restructuring algorithm we use is described in [2]. It is based on two transformations ($T_1$ and $T_2$) and a node splitting routine which operate on a control flow graph. The nodes in the graph are basic blocks and the edges are expressions, which are both already present in that form in the intermediate representation.

The $T_1 - T_2$ algorithm is not very complicated and yet efficient. The main drawback is the fact that large boolean expression are generated which can be reduced. Using a simple boolean reduction algorithm, which only handles the cases which are generated by the $T_1 - T_2$ algorithm this drawback is overcome.

## 3.2 Generating the SAD formula

Structured programs have very simple rules for generating a time complexity function. When a loop statement iterates $n$ times, the time complexity function of the entire loop statement is the time complexity function of the body of the loop statement multiplied by $n$.

Since our restructuring process led to a structured program, we can use these simple rules, formalized in table 3.2 by the translation function **tc[**[ ]**]** , to handle the generation of the complexity function SAD.

There is however a problem with the procedure we used. First, we restructured the program and then we generated a time complexity formula. Unfortunately the restructuring process is a transformation of the original program to a new, but semantically equivalent program. Semantic equivalence does however not mean that the time complexity formula of the original and transformed program are equal. There are two main reasons for the arisen problems:

1. The transformation of an irreducible control flow graph leads to the duplication of program code. Although for this has no direct influence on the time complexity function as defined in table 3.2, there is a concern for misinterpretation of the formula when we want to extend it to model cache influences. In the time complexity formula we generate using the rules so far, we have no indication that two blocks of statements originate from the same original source. This would lead to a possible increase of the execution cost we may have a more expensive instruction-fetch when the statements are cached.

   Currently, we have no real support for multi level memories, so caches are not modelled. We do however want to keep this option open.

2. Not only statements are duplicated, but also expressions and new conditional statements may be introduced. Let us look at an earlier example program:

```
      DO 10, I=1, 100
         IF(SEARCHVALUE.EQ.ARRAY(I)) GOTO 20
 10   CONTINUE
      ...
```

$$\begin{array}{ll}
\mathbf{tc}[[\texttt{WHILEDO}\ expression\ statements]] & \rightarrow \\
 & \quad \mathbf{tc}[[expression]] + M(jump) \\
 & \quad +P(expression = true)* \\
 & \qquad\qquad (\mathbf{tc}[[statements]] + \mathbf{tc}[[expression]] + M(jump)) \\
 & \\
\mathbf{tc}[[\texttt{DOWHILE}\ expression\ statements]] & \rightarrow \\
 & \quad \mathbf{tc}[[statements]] + \mathbf{tc}[[expression]] \\
 & \quad +P(\mathit{expression} = true)* \\
 & \qquad\qquad (\mathbf{tc}[[statements]] + \mathbf{tc}[[expression]] + M(jump)) \\
 & \\
\mathbf{tc}[[\texttt{LOOP}\ expression\ statements]] & \rightarrow \\
 & \quad \mathbf{tc}[[expression]] + M(overhead) \\
 & \quad +expression * (\mathbf{tc}[[statements]] + M(iteration)) \\
 & \\
\mathbf{tc}[[\texttt{IF}\ expression\ stats_{then}\ stats_{else}]] & \rightarrow \\
 & \quad P(expression = true) * \mathbf{tc}[[stats_{then}]] \\
 & \quad +P(expression = false) * \mathbf{tc}[[stats_{else}]] \\
 & \quad +M(jump) \\
 & \\
\mathbf{tc}[[\texttt{EVAL}\ expression]] & \rightarrow \mathbf{tc}[[\mathit{expression}]]
\end{array}$$

$M(jump)$, $M(overhead)$ and $M(iteration)$ are machine parameters from the machine database.

Table 2: Rules for translating into a time complexity formula. **tc[[** expression **]]** is defined by the summation of the execution cost of all its operations (as mentioned in the previous section)

After the transformation into a structured program this program looks about this:

```
i = 1
DO
   b = (searchvalue = array[i])
   IF not b THEN
      i = i + 1
   ENDIF
WHILE (i <= 100) and not b
...
```

Although the semantics are the same, the time complexity formula will be what we expected, since the do-loop has changed into a DO-WHILE and the two (sub)expressions `not b` were not present in the original code.

The solution is simple, separate the generation of the SAD time complexity formula into two processes. One process is done before the transformation of the original source code into structured code, the other is done after the transformation. The determination of the execution cost parameters not be done on structured code. We can annotate the program with the execution cost parameters from the machine database using a special `COST` statement. The cost statement has one argument, an expression containing a summation of the cost parameters of a previous statement. If the program is annotated with these special statements, these statements follow the same structure as the original program. The rules for the insertion of these annotation statements are described in table 3.2.

The transformation to structured code can now be performed without the problems described earlier, since the execution cost has already been extracted and because the `COST` statements act just like `EVAL` statements, the transformation results in a semantic equivalent program.

Now we use rules simular to the old rules to extract the SAD formula, but now we must not evaluate the **tc[**[ ]**]** transformation of expressions and statements within loops and if statements, but *collect* the `COST` statements from the program (See table 3.2).

| | |
|---|---|
| **cst[[** GOTO **]]** label | →COST M(jump)<br>GOTO label |
| **cst[[** LABEL **]]** label | →LABEL label |
| **cst[[** EVAL **]]** expression | →COST **tc[[** expression **]]**<br>EVAL expression |
| **cst[[** IF **]]** expression<br>statements1<br>statements2 | →COST **tc[[** expression **]]** + M(jump)<br>IF expression<br>**cst[[** statements1 **]]**<br>**cst[[** statements2 **]]** |
| **cst[[** DOWHILE **]]** statements<br>expression | →DOWHILE **cst[[** statements **]]** ; COST **tc[[** expression **]]** + M(jump)<br>expression |
| **cst[[** WHILEDO **]]** expression<br>statements | →WHILEDO expression<br>**cst[[** statements **]]** ; COST **tc[[** expression **]]** + M(jump) |
| **cst[[** LOOP **]]** expression<br>statements | →COST **tc[[** expression **]]** + M(overhead)<br>LOOP expression<br>statements ; COST M(iteration) |

Table 3: Rules for annotating the program with COST statements

| | |
|---|---|
| $\mathbf{col}[[\mathtt{WHILEDO}\ expression\ statements]]$ →<br>$P(expression = true) * \mathbf{col}[[statements]]$ | |
| $\mathbf{col}[[\mathtt{DOWHILE}\ expression\ statements]]$ →<br>$(1 + P(expression = true)) * \mathbf{col}[[statements]]$ | |
| $\mathbf{col}[[\mathtt{LOOP}\ expression\ statements]]$ →<br>$expression * \mathbf{col}[[statements]]$ | |
| $\mathbf{col}[[\mathtt{IF}\ expression\ stats_{then}\ stats_{else}]]$ → | $P(expression = true) * \mathbf{col}[[stats_{then}]]$<br>$+P(expression = false) * \mathbf{col}[[stats_{else}]]$ |
| $\mathbf{col}[[\mathtt{COST}\ expr]]$ | → $expr$ |
| $\mathbf{col}[[statements]]$ | → $\sum every\ statement$ |

Table 4: Rules for collecting the COST statements into a time complexity formula.

# 4 Symbolic execution

The generation process as defined in the previous section generates a SAD time complexity formula which consists of machine parameters multiplied by factors. These factors are derived from the source code and are either expressions from LOOPs or are probabilistic functions notating the chance that the expressions in IF, DOWHILE or WHILEDO are true or false.

There is very little we can do with solely this SAD formula, since we have lost all other information about the algorithm and thus about the values of the factors in the SAD formula. The factors are all probabilistic functions of the problem size and statistical properties of the input data, so although we have many factors in the SAD formula (every control flow statement results in such a factor) we actually have only a few parameters on which the time complexity depends. Many factors in our current SAD formula have therefore the same value or are dependent on each other. The current SAD formula is therefore not useful and we must find a way in which more information can be extracted from the formula.

Before collecting the cost statements we can perform an action known as symbolic execution [3, 6]. In symbolic execution we execute all steps in the program, as if we were simulating the execution, but without any input data. Therefor, many variables have no value and expressions cannot be evaluated entirely. Not being able to evaluate the expressions entirely means that we don't know how many times loops must be executed and which branch to take in an if-then-else expression.

The idea to track the definitions of variables and to substitute the use of of those variables with the expression that is assigned to that variable, *but* only if that definition is still valid within the context of the usage of that variable. The symbolic execution is a transformation of the intermediate structured representation of the program which result is a semantic equivalent program in which the usage of the variables has been replaced by what we know about that variable.

We use a simple algorithm for performing the symbolic execution action. The algorithm steps through the program replacing variables in expressions with their definition in a global name space. If an assignment statement is encountered the definition of a variable is added to the name space. Furthermore it "kills" the definition of a variable by removing it from the name space when that becomes necessary. This is necessary when for example the definition of a variable took place within the then branch of an if-statement and the variable is used also after the if-statement is encountered. More complicated rules are necessary for loop-statements.

The algorithm is a bit more intelligent by annotating every expression which is used as a definition of a variable with the location of that expression within the source code. This annotation is then also used in the instantiation of the variables where they are used. Variables which could not be instantiated (because their value was removed by "Kill") also receive an annotation of the possible locations where it may have received its value. This in effect brings the program in a static single assignment form ([4]), which may be used later by other analyzing techniques.

## 4.1 Simulation

In the previous section we have shown how we can perform a symbolic execution on the input program. This symbolic execution action takes place before “collecting” the cost statements, by which we generate the time complexity function. The time complexity formula generated now consists of far less parameters than before. Because of many factors the value is now known, or we have found that some factors really depend on only a few parameters (e.g. the problem size).

We may also know by performing the symbolic execution, that a factor can be only within a certain range. This can be used by a simulator together with the definition of the value of certain variables as being stochastic functions to generate random values for those variables. With the instantiation of random values for the variables in the expressions of the factors of the SAD formula, we can calculate one single sample in a simulation process.

An expert user must define the stochastic functions of the variables which were left unevaluated by the symbolic execution process. The tool that is being developed helps such a user in tracking down these variables and defining their stochastic function on the proper position within the algorithm.

# 5 Status

It is important to realize that total automatic performance prediction is provable impossible, because it is similar to the solving the halting problem[7]. However we believe that the tool we are developing can assist an expert user in understanding large programs by creating an environment in which he can track down the usage of variables know which possible values it can have.

The interactive tool which incorporates the ideas within this report has been built, together with a necessary graphical interface to it. The tool is still in a alpha phase and needs extensive testing with some medium and large size programs to test the robustness and to find out if the current model is sufficient to allow for meaningful simulation purposes.

The further development of the tool will include the incorporation of SPMD message passing constructs, which form level three of SAD.

# A Intermediate format

Although the intermediate format used in this paper is actually stored in some datastructures, rather than generated in ASCII form, a formal description can give a good overview over the expressive power and completeness of the format. The table below describes the BNF grammar of the intermediate representation.

| | | |
|---|---|---|
| *expression* | ::= | `Const` *constant* |
| | \| | `Var` *variable* |
| | \| | `Elt` *array-variable index* |
| | \| | `Op1` *unary-operator expression* |
| | \| | `Op2` *binary-operator expression-1 expression-2* |
| *index* | ::= | *expression* |
| *statements* | ::= | $statement^+$ |
| *statement* | ::= | `GOTO` *label* |
| | \| | `LABEL` *label* |
| | \| | `EVAL` *expression* |
| | \| | `ASSIGN` $expression_{lval}$ $expression_{rval}$ |
| | \| | `COST` *expression* |
| | \| | `IF` *expression* $stats_{then}$ $stats_{else}$ |
| | \| | `LOOP` *expression statements* |
| | \| | `WHILEDO` *expression statements* |
| | \| | `DOWHILE` *statements expression* |

Any (sub)expression is furthermore tagged with the type (integer, real, double precision, etc) of the expression as well as with the location where the subexpression originates from (as described in 4).

# References

[1] _Camas technical report 2.2.4.2. Technical report, University of Amsterdam, 1993.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] G. Colman, P. Andreae, and L. Groves. Program analysis by symbolic execution and generalization. In *The unified computation Laboratory, Modelling, specifications and tools*, volume 35 of *The institute of Mathematics and its applications conference series*, pages 367–379. ISBN 0-19-853684-4.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficient computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, october 1991.

[5] Arjan de Mes. Parasol position paper 2.1.1.4. Technical report, University of Amsterdam, 1994.

[6] M.R. Girgis. An experimental evaluation of a symbolic execution system. *Journal Software Engineering*, 7(4):285–290, july 1992.

[7] D. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison-Wesley, 1968.

[8] Clive G. Page. *The professional programmers guide to Fortran 77*. Pitman, 1988.