

Evaluation of Implementations of Access
Mechanisms in Block Structured Languages.

by

Mats Löfgren and Boris Magnusson

CODEN: LUTFD2/(TFCS-3001)/1-20/(1982) &
LUNFD6/(NFCS-3001)/1-20/(1982)



LUND UNIVERSITY

Department of Computer Sciences

Evaluation of Implementations of Access
Mechanisms in Block Structured Languages.

by

Mats Löfgren and Boris Magnusson

CODEN: LUTFD2/(TFCS-3001)/1-20/(1982) &
LUNFD6/(NFCS-3001)/1-20/(1982)

Abstract

Statistics from dynamic analysis of programs are used to compare the efficiency of different methods to access global variables. The method where static links are used for access is shown to be at least as efficient as the display method. The use of the linked list method also leads to a simpler runtime system and allows separately compiled modules to execute at any level in the resulting program. The results of the dynamic analysis are presented in a form which allows comparison of the access methods in language implementations on specific computers. Experience with using two different access methods in an implementation of Simula67 concludes the paper.

Key words

programming languages, implementation, recursion,
block-structure, display, access mechanism, Simula, Ada,
separate compilation, dynamic analysis, compiler.

Dokumentutgivare

LUNDS UNIVERSITET, Inst för Datateknik,
Handläggare Avd för Datalogi och
Numerisk Analys

Dokumentnamn

RAPPORT LUNFD2/(TFCS-3001)/1-20/(1982)

Utgivningsdatum

AUG 1982

Dokumentbeteckning

Ärendebeteckning

Författare

Mats Löfgren
Boris Magnusson

Dokumenttitel och undertitel

Evaluation of Implementations of Access Mechanisms in
Block Structured Languages.

Referat (sammandrag)

Statistics from dynamic analysis of programs are used to compare the efficiency of different methods to access global variables. The method where static links are used for access is shown to be at least as efficient as the display method. The use of the linked list method also leads to a simpler runtime system and allows separately compiled modules to execute at any level in the resulting program. The results of the dynamic analysis are presented in a form which allows comparison of the access methods in language implementations on specific computers. Experience with using two different access methods in an implementation of Simula67 concludes the paper.

Referat skrivet av

Förslag till ytterligare nyckelord
programming languages, implementation, recursion,
block-structure, display, access mechanism, Simula, Ada,
separate compilation, dynamic analysis, compiler.

Klassifikationssystem och -klass(er)
D 3.4, D 3.3 (Computing Reviews, JAN 1982)

Indextermer (ange källa)

Programming Languages,

Processors: compilers optimization code-generation run-time environment

Language Constructs: procedures, functions and subroutines, coroutines

(Computing Reviews, JAN 1982)

Omfång Övriga bibliografiska uppgifter

20 sidor samt bilagor

Språk

Engelska
Sekretessuppgifter

ISSN

ISBN

Dokumentet kan erhållas från
Dept. of Computer Science
Box 725
S-220 07 LUND, Sweden

Mottagarens uppgifter

Pris

Blankett LU 11:25 1976-07

Contents

1. Introduction	1
2. Access methods	3
2.1 Evaluation	4
2.2 Improvements of the methods	5
2.3 Influence of many index registers	6
2.4 Estimation of machine-dependent costs	7
3. Program Statistics	9
3.1 Results from three programs	10
3.2 Results from 20 programs	13
3.3 A cost expression for the methods	15
4. Practical results in an implementation	17
5. Conclusion	19
6. Acknowledgements	19
7. References	20

Appendices

A: Calculation of the average program.

B: Calculation of the coefficients in the cost expression.

1. Introduction.

When processing programs written in block-structured languages allowing recursive procedures or other means to generate more than one dynamic instance of a given entity, the compiler cannot predict the location in memory of variables during runtime. Examples of languages with such facilities are Algol 60 [13], Simula67 [7], Pascal [16] and Ada [1]. In these languages identifiers declared in the currently active block or in one of its statically enclosing blocks are directly accessible by their names. The problem discussed in this paper is how to access entities denoted by identifiers in the enclosing blocks in an efficient way.

During runtime there may simultaneously exist several activation records representing different activations of the block (sub-block, procedure etc.) with distinct data. The static binding of variables to names defines the environment of an activation record. The environment of an activation record does not vary during its lifespan.

A common method of providing access to variables in the environment is to maintain a table of references to the accessible activation records. This table is usually called the display [11], and the method has been used in many language implementations [8,17,15]. Improved methods for maintaining and representing the display have been described in [3,18]. Another access method is to use a link stored in each activation record referencing the activation record of its closest surrounding block, the static link. The environment of an activation record is resolved by descending through the linked list of activation records.

An earlier evaluation of a Pascal system [18] indicates that the linked list method is preferable for that language. In this paper we will show that this is also true for languages such as Simula where programs normally contain more nested block structures than in Pascal. In Ada programs there will probably be at least as complex structures as in Simula programs, which indicates that the results are valid also for Ada.

The reason for the advantage of the linked list method is that accesses to variables in activation records other than the current one are usually to the outermost block, which can be accessed in a more efficient way. This means that the overhead to maintain the display does not pay off in faster access often enough. Other advantages of the linked list method are the considerably simpler runtime system and the simpler implementation of separate compilation.

In section 2 of this paper we discuss the access methods in more detail and calculate costs for a few basic operations like updating the display or accessing a non-local variable. The calculations are done in machine independent units.

In section 3 we study the runtime behaviour of 20 programs in order to compare the access methods. We do the comparison for ten different situations or versions of the basic access methods. We also present a cost expression which can be used to compare cases other than the ten presented. We conclude the section with some program statistics that help to explain the result but also are of more general interest.

In section 4 we report on the results obtained in an implementation of Simula using two different access methods.

The reader should have a working knowledge of at least one of the programming languages discussed and some knowledge about language implementation techniques.



2. Access methods.

The term "block" is used here to denote sub-blocks, procedures and similar constructs. The storage space needed for execution of a block is allocated during runtime, when such a construct is entered. The variable-size allocation units are called activation records. The environment which consists of the directly accessible activation records can be represented in basically three different ways (Fig. 1).

1. Global display:

A vector stored in a fixed place in memory or in registers [11, 2,8]. The display contains references to the environment of the currently active block instance and has to be updated every time the environment changes, due to procedure calls or returns, sub-block execution or process switching.

2. Local display:

A vector stored in the activation record itself [2,3]. The display contains references to the environment of the block instance and is constructed when the activation record is created. The currently used display is that of the currently active block, so no updating of the display is needed when the environment changes. Compared with other access methods this method may need larger activation records to hold the display which can cause reduced efficiency. This aspect, however, is not considered here.

3. Linked list:

A linked list of activation records, where every record references the activation record of its closest surrounding block [2]. The chain has to be followed for every access to a non-local identifier.

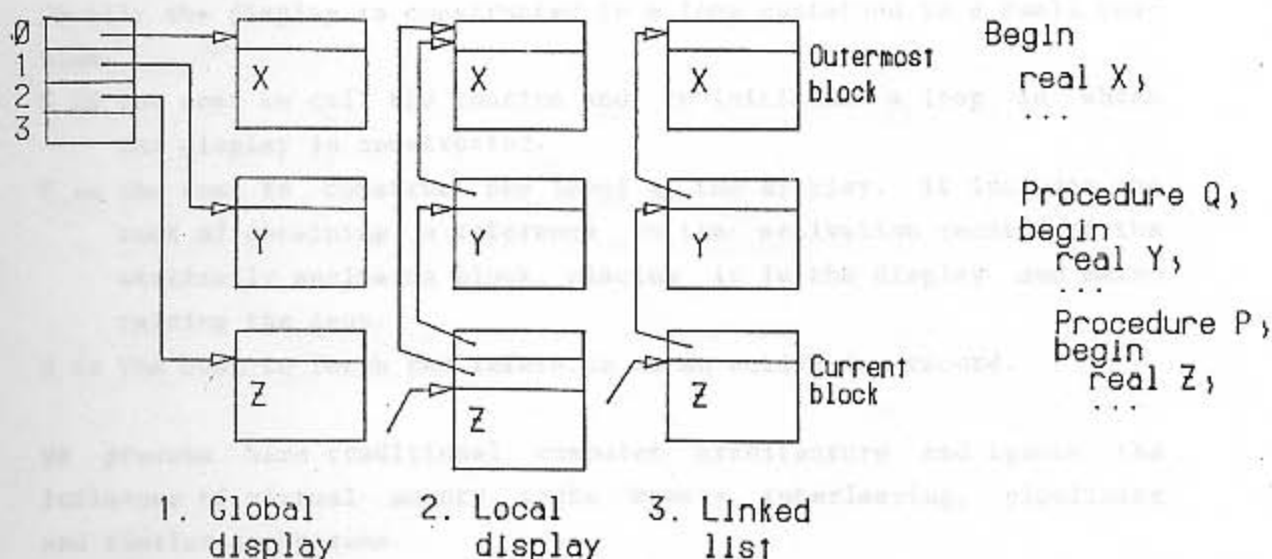


Fig. 1. The three methods to represent the environment.

2.1 Evaluation.

The choice of access method has a great influence on system performance. The execution time spent on the following tasks is of major interest:

- a. Entry to or exit from a procedure, sub-block or other block-like construction.
- b. Process switching (in programming languages with coroutine facilities or facilities for concurrent execution e.g. Simula [7], Concurrent Pascal [6] and Ada [1]).
- c. Referencing activation records in order to access variables or call procedures declared in the corresponding block.

In order to compare the different methods we will calculate the costs for tasks a, b and c above, adopting the following notation:

S is the static block level (= zero for the outermost block).

S_p is S for a called procedure, block etc.

S_b is S for a calling block, procedure etc.

S_v is S for a referenced variable.

The impact of the hardware used is isolated in three constants: K, C and A, which can take different values in the three methods.

Usually the display is constructed in a loop contained in a small routine.

K is the cost to call the routine and to initialize a loop in which the display is constructed.

C is the cost to construct one level of the display. It includes the cost of obtaining a reference to the activation record of the statically enclosing block, placing it in the display and maintaining the loop.

A is the cost to fetch one reference to an activation record.

We presume here traditional computer architecture and ignore the influence of virtual memory, cache memory, interleaving, pipelining and similar techniques.

Access to the current activation record is usually optimized by keeping a reference to it in a fixed memory location or in a register, since this record is expected to be very frequently referenced. Furthermore, the activation record of the outermost block can be allocated in a fixed position, which makes it possible to access its contents directly. These improvements can be applied to any of the three access methods, which leads to the same costs for accessing variables in these two activation records independent of the access method used. Since we are interested in the difference between the access methods, we will not include accesses to these two blocks in our calculations.

In some of the access methods the number of index registers available will have a significant effect on performance. This will be discussed later. To start with we presume only one index register available for access to activation records at intermediate levels.

	1. Global display	2. Local display	3. Linked list
a. Entry	$K+C*(Sp-1)$	$K+C*(Sp-1)$	0
Exit	$K+C*\max((Sb-1),0)$	0	0
b. Switching	$K+C*\max((S-1),0)$	0	0
c. Access ($Sv \neq 0$ and $Sv \neq Sp$)	A	A	$A*(Sp-Sv)$

Table 1. Cost for interesting events depending on access method. Observe that the activation records of the current and outermost blocks are not included in the display.

2.2 Improvements of the methods.

In most cases for method 1 it will be possible to determine at compile time how the display has to be changed during runtime [18]. This is typically the case when a procedure is called using its name in the source program. Usually a large part of the displays of the procedure and the calling block overlap and do not have to be changed.

Calling a sub-block is a special case where the display only has to be extended with one reference.

The code to update the display can be generated inline, in which case there is no need for a loop. This will reduce the C-factor by the cost of the loop, and reduce K to zero.

	Simple Procedure calls	Sub- blocks	other cases
a. Entry	$C * \max((Sp - Sb), 0)$	$C * 1$	$C * (Sp - 1)$
Exit	$C * \max((Sb - Sp), 0)$	0	$C * \max((Sb - 1), 0)$
b. Switching			$C * \max((S - 1), 0)$
c. Access ($Sv \neq 0$ and $Sv \neq Sp$)			A

Table 2. Overhead in the improved version of method 1 (Global display).

In method 2 the code to create a local display may also be generated inline, in which case the K and C costs will be affected in the same way as for method 1 above.

A further improvement may be achieved by collapsing sub-blocks into their textually enclosing blocks [3,2]. This will reduce the length of the display in a program, which affects all the access methods in a similar way and is therefore not discussed in this investigation.

2.3 Influence of many index registers.

If the number of index registers is large enough to hold the complete display the performance of the various methods can be improved.

The overhead in method 1 is affected in two ways. The access cost, A, is reduced to zero when accessing variables and decreased from a memory access to a register transfer when allocating a procedure. The cost of the transfer, however, is ignored here. The construction-cost, C, will decrease, since the references have not to be stored in memory.

Method 2 is distinguished from method 1 with respect to costs by the fact that no action is needed on block exits and process switches, as shown in table 1. However, if a copy of the display is kept in registers these must be updated also on block exits and process switches, and the advantage of method 2 disappears. Thus method 1 and method 2 behave identically when the display is held in registers although the K and C costs may differ.

In method 3 the extra registers can be used to save references to already accessed blocks. This technique may be combined with logic during code generation to move register loadings out of program loops when possible. This improved version of method 3 will come very close to a version of method 1 where only the used portions of the display are actually constructed. After special events, such as calls of other procedures, the display has to be reloaded (but only when used again), which corresponds to the exit-procedure cost in method 1.

Thus in the highly optimized case, and with enough index registers available, the difference between methods 1 and 3 is very small. In method 3 registers will only be loaded when actually used. If method 1 is used, some time may be wasted on keeping unused registers updated. On the other hand, in the improved version of method 1 no update of the display has to be performed when the displays overlap. These improved versions of methods 1 and 3 appear to be comparable with respect to effort of implementation and overall complexity.

2.4 Estimation of machine dependent costs.

In this section we estimate the machine dependent costs with lower limit values for typical cases. The costs are expressed in instructions, except for the cost for repetitious instructions which are estimated in number of memory references. On a specific computer it will be more interesting to do the calculations in time-units because the execution time for different instructions may vary considerably. The values presented in table 9 may be used for more accurate calculations in specific situations.

When a general routine is used to update the global display, or to create a local display, there will be a fixed overhead to call and exit the routine and to initialize the loop to perform the construction. In method 1 the display is usually constructed from a linked structure of activation records. In method 2 the display of the block statically enclosing the entered block can be copied and extended.

As an example we show an implementation of the update display routine for method 1 in a fictitious assembly language.

```

LOAD INDEX1,LEVEL      (K: Index1:= Position in new Display
SKIP INDEX1,#0          (K: -- If there are zero entries
JUMP DONE              (K: -- we are done
LOAD INDEX2,SOURCE      (K: Index2:= Top element in Display.
ONEMORE:
STORE INDEX2,DISPLAY,INDEX1 (C: Display[Index1]:=Index2
LOAD INDEX2,STACHAIN,INDEX2 (C: Index2:=Index2.StatChain
DECR INDEX1,#1          (C: Index1:=Index1-1
SKIP INDEX1,#0          (C: -- check if more entries to fill
JUMP ONEMORE           (C: if Count1#0 then goto ONEMORE
DONE:                  (

```

The code for method 2 is slightly more complicated.

The display in memory.

The C cost in a particular implementation depends heavily on the target computer. On a machine with a limited instruction set up to 5 instructions may be needed as shown above. On computers with more sophisticated hardware, such as automatic register-incrementation and powerful loop handling instructions, 3 instructions are commonly needed. If a machine has advanced repetitious instructions or is microcodable the C-operation may be implemented with one instruction. However, the time to execute one loop in that instruction will come close to that of executing one load and one store operation because two memory references are needed. We therefore assume a lower limit of 2 for the C-operation.

The loop set-up operation, K, includes initialization of three entities; how many levels should be loaded into the display, where to find the first entry and where the first entry should go. The first loop test must be included in the K cost adding up to 3 instructions. When zero entries are needed in the display a JUMP instruction has to be executed instead of one of the LOADs giving the same sum.

The cost to call and exit the routine is neglected here since the routine is usually small and can be generated inline in the runtime system where it is needed.

The display in registers.

When the display is kept in registers the STORE operation is not needed. The C cost will decrease by one to 4, 2 or 1 instructions depending on instruction set. K is still 3 instructions.

The update display code generated inline.

When the update display code is generated inline no loop is needed. The K cost is zero and the C cost is equal to 2 when the display is kept in memory (LOAD and STORE) and 1 when the display is kept in registers (LOAD).

The A cost is determined from the code to obtain a reference to one activation record. In method 1 this is done by indexing the display vector. In method 2 the reference is found in the activation record of the current block. In both cases one instruction is needed. In method 3 the reference is found by following a chain of static links from the current activation record to the desired record, each step of which is implemented with one instruction, giving an A value of 1.

3. Program Statistics.

In the previous section three access methods were studied and expressions for their overhead were developed. In order to predict which method will actually have the best performance we need runtime statistics on significant events (block entry/exit, process switching and access of non-local entities). The relevant data cannot be obtained by examining the program text, since these events are dependent on program execution. In fact, statistics obtained from one program can differ from one execution sample to another, but normally different sets of data will exhibit similar trends.

In order to collect the necessary data we chose to modify the compiler and runtime system of an existing language implementation. The compiler was modified to generate code for counting accesses wherever an activation record was accessed. The runtime system was modified to count all block entries/exits and process-switches. The compiler used was a Simula system on a mini-computer [12].

All programs examined in this paper were written in Simula [7].

3.1 Results from three programs.

We will here present statistics obtained from three programs in different application areas, and show how the values are normalized for comparison. The programs have been chosen to cover quite different ways of using the facilities of Simula.

Program 1 solves partial differential equations, written in traditional Algol style without deeply nested levels. Due to the emphasis on calculation it contains a large number of variable references. Program 2 is a lexical scanner for Simula. It was written with heavy use of the coroutine facilities of Simula, which results in a large number of process switches. Program 3 is a model of a Concurrent Pascal [6] system with an unusually complex block structure.

Program	Block creations	Process switches	Block references
1	Level=0 1		Outermost 341193 12 %
	1 33693		-1 158121 5 %
	2 10801		Current 2474836 83 %
2	0 1	0 1	Outermost 205397 46 %
	1 6363	1 61372	-1 17460 4 %
	2 7519	2 0	Current 224026 50 %
3	0 1	0 1	Outermost 140502 8 %
	1 10	1 17547	-3 0
	2 70230	2 3	-2 140435 8 %
	3 122909	3 17544	-1 721264 42 %
	4 26347	4 0	Current 721986 42 %

Table 3. Distribution of relevant events in three programs.

An interesting observation in Table 3 is that for these three programs global variables at long static distance are rarely accessed, except for those declared in the outermost block.

In order to make a more compact comparison, we will use the total number of process switches and global references (excluding the outermost block) to characterize a program. These figures are normalized by the number of block-creations as presented in Table 4.

Program	Switches/block	Global accesses/block
1	0	3.55
2	4.42	1.26
3	0.16	3.93

Table 4. Characteristic figures for three programs.

The index register available for access of blocks at intermediate levels is not reloaded if it already contains a reference to the requested record. The costs for accessing the current block and the outermost blocks are the same in all three methods and are therefore not included in the calculations. With no other optimizations included the costs for each access method are presented in Table 5.

Pro-gram	Method 1	2	3	1-Improved	3-Improved (many registers)
1	88,998*K + 24,128*C +158,121*A	44,499*K + 10,801*C +158,121*A	158,121*A	86,001*K + 24,128*C +158,121*A	21,134*A
2	89,146*K + 15,868*C + 17,460*A	13,887*K + 7,519*C + 17,460*A	17,460*A	89,144*K +14,483*C +17,460*A	7,489*A
3	474,096*K +860,358*C +861,699*A	219,501*K +395,089*C +861,699*A	1,002,134*A	474,092*K +579,386*C +861,699*A	385,075*A

Table 5. Overhead costs in machine dependent units. The constants K, C, and A are defined in section 2.1. The improvements of the methods are described in section 2.2 and 2.3.

For direct comparison the following ten versions of the methods were chosen.

- Method 1 with the display update as a routine and with the display in memory. We use $A=1$, $K=3$ and $C=3$ according to section 2.4.
- Method 1 with the display update code generated inline and the display in memory, giving $A=1$ and $C=2$. (The result of this version is later used for normalizing the figures.)
- Method 2 with the display update code as a routine ($K=3, C=3, A=1$).
- Method 2 with the display update code generated inline ($C=2, A=1$).
- Method 3 with $A=1$.
- Method 1, improved so that unnecessary restorations of the display are avoided and update code is produced inline ($C=2, A=1$).
- Method 1 assuming that the hardware registers can hold the display ($K=3, C=2, A=0$).
- Method 1 with the display in registers and update code inline ($C=1, A=0$).
- Method 1, improved so that unnecessary restorations of the display are avoided, update code is produced inline and the display is kept in registers ($C=1, A=0$).
- Method 3 with the improvement that many available registers are used to save already loaded block references, combined with logic to move such loads from program loops as described in section 2.3.

P r o g	F e w registers						M a n y registers			
	1	1	2	2	3	1 Im-	1	1	1 Im-	3 Im-
						proved			proved	proved
	K=3		K=3				K=3			
	C=3	C=2	C=3	C=2		C=2	C=2	C=1	C=1	
	A=1	A=1	A=1	A=1	A=1	A=1	A=0	A=0	A=0	A=1
1	497	206	324	180	158	206	315	24	24	21
2	333	49	82	32	17	46	299	16	15	7
3	4865	2582	2705	1652	1002	2020	3143	860	579	385

Table 6. Overhead cost for the three programs in thousands of instructions.

To ease comparison the numbers are normalized with the cost for method 1 with update code generated inline. The normalized values are presented together with the number of process switches/block creation and global accesses/block creation.

			F e w registers						M a n y registers			
P r o g	sw	acc	1	1	2	2	3	1Imp	1	1	1Imp	3Imp
			K=3		K=3				K=3			
			C=3	C=2	C=3	C=2		C=2	C=2	C=1	C=1	
			A=1	A=1	A=1	A=1	A=1	A=1	A=0	A=0	A=0	A=1
1	0	3.55	2.41	1	1.57	0.87	0.77	1.00	1.53	0.12	0.12	0.10
2	4.42	1.26	6.76	1	1.66	0.66	0.35	0.94	6.08	0.32	0.29	0.15
3	0.16	3.93	1.88	1	1.05	0.64	0.39	0.78	1.22	0.33	0.22	0.15

Table 7. Result example. Abbreviations: Sw/bl = no. of process switches per created block, acc/bl = no. of intermediate accesses per block, Imp = Improved version of the method.

To illustrate the use of this table we will first focus on program 1. This program uses global variables to some extent, but does not use coroutines at all. Comparing the methods we find that method 2 and 3 have a slightly better performance than method 1, and that the improvement of method 1 does not give any effect at all. When many index registers are available we find a significant increase in performance for all the methods.

Program 2 has a different behaviour, with a very large number of process switches and a small number of accesses to global blocks. The penalty for using a routine to update the display in method 1 is thus very high. Methods 2 and 3, which handle process switching better than method 1, show much better performance. The importance of many registers is less for this program than for program 1 because of fewer global accesses, but the improvement is still high. Program 3 yields similar relationships for the performance of various methods.

So far the results indicate that method 3 has a better performance than the other methods, but to draw a definite conclusion we need statistics from more programs.

3.2 Results from 20 programs.

20 Simula programs were selected for additional statistical evaluation. Due to the small size of the memory on the computer used (64 Kbytes), the size of a program was restricted to 3000 source lines. Further, the following restrictions were adhered to:

1. The programs were to be chosen from different application areas.
2. The programs were not to be written mainly to check the system itself.
3. The size of a program should not be less than 200 source lines.

				F e w registers						M a n y registers					
P r o g	S i z e	sw	acc	1	1	2	2	3	1Imp	1	1	1Imp	3Imp		
				K=3		K=3						K=3			
				C=3	C=2	C=3	C=2		C=2		C=2	C=1	C=1		
				A=1	A=1	A=1	A=1	A=1	A=1	A=0	A=0	A=0	A=1		
1	300	0	0	74.36	1	37.18	0.50	0	0.99	73.86	0.50	0.49	0		
2	500	0.16	3.93	1.88	1	1.05	0.64	0.39	0.78	1.22	0.33	0.22	0.15		
3	700	0.24	2.18	2.62	1	1.38	0.67	0.48	0.88	1.90	0.28	0.22	0.13		
4	900	0	2.79	1.88	1	1.04	0.60	0.39	0.53	1.26	0.39	0.15	0.14		
5	2400	0.08	0.73	3.45	1	1.77	0.60	0.24	0.97	2.83	0.38	0.37	0.17		
6	1300	0.49	1.64	3.10	1	1.66	0.76	0.39	0.53	2.40	0.30	0.07	0.18		
7	600	0	3.55	2.41	1	1.57	0.87	0.77	1.00	1.53	0.12	0.12	0.10		
8	2100	0	1.59	2.74	1	1.61	0.72	0.39	0.70	2.05	0.31	0.16	0.14		
9	200	0	0.60	9.41	1	5.25	1.00	0.83	1.00	8.49	0.09	0.09	0		
10	2000	0	2.46	2.02	1	1.19	0.67	0.37	0.43	1.39	0.37	0.08	0.16		
11	1000	4.42	1.26	6.76	1	1.66	0.66	0.35	0.94	6.08	0.32	0.29	0.15		
12	200	0.09	20.79	1.33	1	1.07	0.91	0.83	0.84	0.42	0.08	0.00	0.04		
13	200	0	5.14	1.91	1	1.32	0.85	0.68	0.69	1.10	0.19	0.04	0.11		
14	400	0.97	0	100.67	1	34.22	0.54	0	0.50	100.17	0.50	0.25	0		
15	200	0	2.30	2.01	1	1.11	0.61	0.28	0.66	1.39	0.38	0.21	0.13		
16	700	0	0.86	3.88	1	1.90	0.53	0.39	0.89	3.19	0.32	0.26	0.11		
17	400	1.69	20.67	1.52	1	0.98	0.81	0.79	0.87	0.68	0.16	0.09	0.12		
18	700	0.59	0.89	3.93	1	1.90	0.70	0.30	1.00	3.29	0.35	0.35	0.23		
19	200	1.25	1.15	3.32	1	1.15	0.45	0.23	0.76	2.71	0.39	0.26	0.14		
20	700	0	4.38	1.96	1	1.27	0.78	0.55	0.59	1.20	0.23	0.03	0.12		
Average program				0.50	3.85	2.22	1	1.22	0.72	0.57	0.77	1.47	0.25	0.14	0.12

Table 8. Overhead for access method in 20 programs. Values for 10 different cases of the 3 basic methods are presented. The calculations of the values representing the average program are explained in section 3.3.

Criterion 1 is required to guarantee a fair use of the different facilities of the language. Criteria 2 and 3 are intended to avoid distorting the statistics by too much use of special facilities.

The resulting table also includes the number of source lines in each program, since we initially felt that this should affect the behaviour of the program. However, this does not seem to be the case.

The use of a routine to update the display gives a very high overhead in both methods 1 and 2. It gives extreme results in the cases where nothing has to be updated in the majority of the calls of the routine. Clearly the use of a routine to update the display should be avoided.

When few registers are used method 2 shows good behaviour compared to method 1. As a whole, method 2 has 72% of the cost of method 1. In programs without process switching the number of display constructions are cut in half by this method, but overall gain is less since the costs to access intermediate levels are present in both methods.

Method 3 shows even better performance than method 2 in all cases and yields 57% of the overhead of method 1 for the average program. In cases with many global accesses (Programs 12,13,17), it still compares favourably with method 1. It performs extremely well in cases where there are many process switches (11,14,17,19). It is interesting to find two programs with no global accesses at all and thus no overhead.

The improved version of method 1 shows an overall performance close to that of method 2. The effects of the improvement vary a great deal, from cutting the overhead to less than half to having no effect.

These results indicate that on a machine with few hardware registers the linked list method is more efficient. The performance obtained by this method is better than the improved version of method 1 without the added complexity of implementing these improvements.

When comparing the figures for the cases where the display can be held in registers we find that the performance of basic method 1 improves considerably. The improved version of this method cuts the remaining overhead to about half and the improved version of the linked list method shows similar figures. A choice between these latter two methods should be made on other grounds, such as simplicity and ease of implementation.

3.3 A cost expression for the methods.

The statistics obtained have also been used to construct a cost expression for each of the methods. The expression can be used to calculate the overhead in other combinations of the parameters K, C and A than that used in the evaluation above. The expression has the form:

$$k*K + c*C + a*A$$

where K, C and A are the machine dependent factors as described above. The coefficients k, c and a are constructed from the statistical material to reflect the method dependency of the different factors. If K, C and A are calculated in time units for a particular implementation, the expression gives the access method dependent overhead to execute an "average block" as it appears in our statistical material.

Method:	1	2	3	1-Improved	3-Improved
k	2.50	1		2.49	
c	1.96	0.88		1.07	
a	3.85	3.85	4.45	3.85	0.89

Table 9. The method dependent factors for an average block.

As an example of the use of Table 9 we calculate the overhead in the following case:

We use method 1 with a microcoded routine to update the display (K=3 and C=1). The display is kept in registers (A=0). The yield value is 9.46 ($=3*2.50+1*1.96$), or 1.22 when normalized as above.

The constants in Table 9 have been calculated based on an "average program" constructed by taking the average of the characterizing numbers for the 20 programs. The distribution of block accesses, block creations and process switches over levels has been constructed by taking the average of the distributions in the 20 programs. A more detailed description of the method used is found in appendix A and B.

Block creations	Process switches	Block references
Level:	Level:	Level:
0 (Main program)	0 0.2 %	Outermost 28.9 %
1 36.1 %	1 62.9 %	-4 0.06 %
2 43.4 %	2 31.1 %	-3 0.38 %
3 16.6 %	3 5.8 %	-2 2.2 %
4 3.3 %		-1 17.5 %
5 0.2 %		Current 51.0 %
	0.50/block	19.1/block of which 3.85 at intermediate levels

Table 10. Distribution of events in the average program.

In Table 10 we find that nearly 80% of all created blocks are declared in the user main program or one level deeper. Thus the display is usually not very long. The distribution of process switches shows the same tendency.

The distribution of block creations gives us the possibility of calculating the average length of the display and thus the extra memory needed in method 2. In blocks created at level 1 no entry is needed as references to the main program are optimized. Blocks at level 2 will have 1 entry and so on. The average length of the display is thus:

$$0 \cdot 0.36 + 1 \cdot 0.43 + 2 \cdot 0.17 + 3 \cdot 0.03 = 0.87$$

This indicates that the needed memory is in fact a little less than in the other methods, where normally space for a static link is reserved in all activation records.

The results regarding accesses are supported by an earlier investigation made by Arnborg [4] where the static behaviour of 51 Simula programs is evaluated. He shows that of 17,166 textual identifier accesses, 75% were to an item in the activation record of the current block, 2% to the outermost block, and 24% to blocks at intermediate levels. The discrepancy with our results may be explained by the following observations.

In the results presented above the references to standard input-output files are included in the values for the outermost block, which considerably increases the figure for that block. If the values for the outermost block are discarded, and the remaining values are normalized, we obtain 72% for the current block and 28% for intermediate levels. The remaining difference may be explained by the

block-reduction technique used in [4] which increases the number of accesses to the current activation record. Furthermore, only accesses to identifiers denoting data are considered in [4], but in this paper the cost of obtaining a reference to the activation record of the closest surrounding block of an activation record to be allocated is also included.

With these observations in mind it is interesting to note that the figures match so well in spite of the difference in methodology (counting static occurrences in [4] rather than performing a dynamic analysis of running programs as in this paper).

The amount of computation per block varies considerably from over 100 variable accesses per block to virtually none. The average, 19 accesses per block, will probably decrease, as the ideas of structured and modular design of programs and information hiding principles often lead to programs with many, very small, procedures [14].

4. Practical results in an implementation.

The results from this study were used to change the access method in an existing implementation of Simula 67 for the Data General Eclipse computer [12]. Due to the very restricted memory size on this machine (64 Kbytes) the Simula system has been heavily optimized to minimize space used.

The access method originally used was a version of the global display method with the display kept in memory, since the Eclipse has only two index registers. One register was reserved to reference the activation record of the current block. However, references to the activation records of the current block and the outermost block were included in the display. The display update code was implemented as a routine with a heavy overhead to call. The optimization of method 1 discussed earlier in this paper was employed, but when the display had to be updated it was completely reconstructed to avoid parameters to the routine.

The K cost was found to be 15.6 μ sec, the C cost 3.67 μ sec and the A

cost 1.2 μ sec. Using the statistics above on one of the programs (no. 3, Table 5), remembering that more than actually needed was restored, we got a total overhead of 13.2 seconds.

Doing the same calculations with method 3 (linked list) gives a total overhead of 1.2 seconds.

The expected saving was very encouraging, so the access method was changed from method 1 to method 3. Although both the compiler and the runtime system are written in assembly language, the change could be carried out fairly easily due to modular design and adopted principles of information hiding and abstract data types.

The CPU time consumed by the same program executed under the new system decreased from 67.0 to 55.7 seconds, a saving of 11.3 seconds, which matches the expected saving of 12.0 seconds.

It was feared that the time advantages obtained by using method 3 instead of method 1 would be outweighed by an increase in storage space, due to the extra code produced for descending static links when activation records of enclosing blocks were referenced. However, it was found that most programs actually shrank in size. This surprising result is due to the smaller size of the runtime routines responsible for block entry/exit etc. and the fact that extra code is produced very rarely. The activation records of the currently active block, the block immediately surrounding the current block and the outermost program block can all be accessed with the same overhead as in method 1. Of all block accesses, 97% are to one of these three blocks.

The runtime system also became simpler as the concept of block level disappeared. This has the nice sideeffect that separately compiled procedures and classes can execute on any level in a system using the linked list access method, since the only entity they share with the main program is the system defined I/O-package, located in the outermost block. Accesses to the corresponding activation record are not made via the static links.

Using a program to calculate runtime costs of different Simula constructs [5] we measured a saving of 25% in execution time on procedure calls at the lowest user level, and 37% for coroutine switching and remotely accessed procedures. A saving of 5-10% of execution time seems to be typical for most programs. This indicates

that approximately 20 to 30% of the total CPU time was spent on administration of procedure calls, now reduced to 15 to 20%, which is still quite high.

These results show that an efficient implementation of the access mechanism pays well, since procedures and other sub-program features are heavily used [14].

5. Conclusion.

In this paper we have developed a method to compare different mechanisms to access global variables in block structured languages. From a number of running programs we have gathered statistics on the frequency and distribution of significant events. From this material we have developed an expression which gives the overhead of the different methods, expressed in hardware dependent parameters, for the execution of an average block.

With the use of this technique we have compared the different methods to access global data in some realistic situations. This evaluation indicates that the linked list method is not only easier to implement, but also has a better performance on hardware with few index registers. On hardware where the display can be kept in fast access registers, the difference between the linked list method and the global-display method in improved versions is very small.

The linked list method is also attractive since it allows separately compiled sub-programs to execute at any level in the resulting program.

The results obtained from implementing two different access methods in a Simula system support the conclusion above. The experiences have been used to change the access method to the linked list method in that implementation. The results here also indicate that further work on improving the performance of sub-program constructs is well spent.

6. Acknowledgements.

We are very grateful for the comments, suggestions and help received from Göran Eriksson, Per Holm and Ron Danielson, both during the investigation and during the preparation of this paper.

7. References.

- [1] Reference Manual for the Ada Programming Language.
US Department of Defence, July 1981.
- [2] Aho A.V. and Ullman J.D.: Principles of Compiler Design.
Addison - Wesley 1977.
- [3] Arnborg S.: Storage Administration in a Virtual Memory Simula
System. BIT 12(1972) 125-141.
- [4] Arnborg S.: Simula User Program Semantics and Code Generation
Effectiveness. FOA C 8498-M3(E5) June, 1974.
- [5] Babcock K.: Simula Programming Efficiency Considerations.
(Paper presented at ASU Workshop Dec 1978) NCC, Oslo, Norway.
- [6] Brinch Hansen P.: The Programming Language Concurrent Pascal
IEEE Trans. Software Eng. 1, 2(June 1975), 199-207.
- [7] Dahl O-J., Myhrhaug B. & Nygaard K.: Simula 67 Common Base Language,
Publication S-22, Norwegian Computing Centre, Oslo, Norway
- [8] Dahl O-J. and Myhrhaug B.: Simula 67 Implementation Guide.
Publication S-9, Norwegian Computing Centre, Oslo, Norway
- [9] Dahl O-J. and Hoare C.A.R.: Hierarchical Program Structures
in Structured Programming, Academic Press, London 1972.
- [10] Dijkstra E. W.: Recursive programming.
Num. Math. 2(1960), 312-318.
- [11] Dijkstra E. W.: ALGOL-60 Translation.
Algol Bulletin Supplement nr 10. Nov 1961.
- [12] Holm et al.: Simula for Data General Eclipse, Users Manual.
Dept of Computer Science, Lund Institute of Technology,
Sweden Feb 1981.
- [13] Naur P. (Ed): Revised report on the algorithmic language Algol 60.
Comp Journal. 5, 349-367.
- [14] Shimasaki M. et al.:
An Analysis of Pascal Programming in Compiler Writing.
Software - Practice and Experience vol 10 (1980), pp 149-157.
- [15] Wichmann B. A.: Algol 60 - Compilation and Assessment.
Academic Press, London 1973.
- [16] Wirth N.: The Programming Language Pascal.
Acta Informatica, 1, 1971, 35-63.
- [17] Wirth N.: The Design of a Pascal Compiler.
Software - Practice and Experience vol 1 (1971), pp 309-333.
- [18] Wirth N.: On "Pascal", Code Generation and the CDC 6000 computer.
STAN CS-72-257, Feb, 1972

Calculation of the average program.

When forming averages of data obtained from executing programs it is not appropriate to simply add the data together since this would mean that a program with very large values would affect the averages to a large extent. It would also mean that running one of the programs in a larger execution sample would change the averages. This seems not reasonable. One way to avoid this is to normalize the figures from each program with something that measures execution volume.

A simple way to do the normalizing would be to divide all the figures by the consumed CPU-time. However, since we are only interested in the significant events, the amount of "ordinary" computation should be left out. For this reason, but also because execution time depends very much on the hardware, we have not used this method. Instead, we have used the number of block creations in the program run as a normalization factor. This makes the figures independent of the hardware and the amount of execution.

A drawback of normalizing by the number of block creations is that a program with extremely many process switches or accesses can affect the average more than desired. This can be prevented by normalizing the number of block creations, process switches and accesses within each program in some way, but we felt that the benefit achieved by

block creations	process switches	block accesses
0.34	0.11	0.55

Table A1. Average of the distribution of significant events.

Method:	1	2	3	1-Imp	3-Imp
k	2.32	1		2.31	
c	1.88	0.88		0.99	
a	1.62	1.62	1.88	1.62	0.57

Table A2. Method dependent factors as in table 9, but from an average program constructed with the distribution of events from table A1.

this does not motivate the loss of simplicity and surveyability of the tables. Nevertheless, we have for comparison made an average program where we, within each program, have normalized with the sum of block creations, process switches and accesses to blocks at intermediate levels. The k-, c- and a-constants of such an average program are presented in table A2 in the same way as in table 9.

These figures give the alternative figures in table A3. We also present the figures obtained by taking straight averages in table 8.

		F e w registers						M a n y registers			
sw	acc	1	1	2	2	3	1Imp	1	1	1Imp	3Imp
		K=3		K=3				K=3			
		C=3	C=2	C=3	C=2		C=2	C=2	C=1	C=1	
		A=1	A=1	A=1	A=1	A=1	A=1	A=0	A=0	A=0	A=1
bl	bl										
used average											
0.50	3.85	2.22	1	1.22	0.72	0.57	0.77	1.47	0.25	0.14	0.12
alternative as above											
0.32	1.62	2.64	1	1.35	0.63	0.35	0.67	1.99	0.35	0.18	0.11
straight average											
0.50	3.85	-	1	-	0.69	0.43	0.78	-	0.30	0.19	0.12

Table A3. Normalized overhead costs for the different access methods calculated for three different "average" programs.

The figures vary depending on which method of taking the average was used. Especially method 3 shows varying figures as the a-cost here ranges from 4.45 to 1.88. However, the relative order of the different access methods are the same. This could be expected as their relative order is the same in almost all the 20 programs. We conclude that the way averages are formed is not critical when comparing access mechanisms.

The effect of extream programs.

A reasonable presumption when taking an average is that one diverging value should not affect the result too much. In table 8 we can see that the programs 12 and 17 have extremely many accesses. Of 3.85 accesses per block of the average program, these two programs represent about half.

To determine the influence these two programs have on the behaviour of the average program we have made two other averages by deleting these two programs respectively inserting them twice. The overhead is presented both in number of instructions and normalized as before. In table A5 we present the result of the same stability analysis but with the use of the alternative method to take the averages.

	F e w registers						M a n y registers			
	1 K=3 C=3 A=1	1 C=2 A=1	2 K=3 C=3 A=1	2 C=2 A=1	3 A=1	1Imp C=2 A=1	1 K=3 C=2 A=0	1 C=1 A=0	1Imp C=1 A=0	3Imp A=1
18 prog. cost/bl. norm.	15.17 2.59	5.85 1	7.61 1.30	3.73 0.64	2.29 0.39	4.07 0.70	11.26 1.92	1.94 0.33	1.05 0.18	0.74 0.13
20 prog. cost/bl. norm.	17.23 2.22	7.77 1	9.49 1.22	5.61 0.72	4.45 0.57	5.99 0.77	11.42 1.47	1.96 0.25	1.07 0.14	0.89 0.12
22 prog. cost/bl. norm.	18.88 2.03	9.32 1	11.02 1.18	7.14 0.77	6.24 0.67	7.54 0.81	11.53 1.24	1.97 0.21	1.08 0.12	1.02 0.11

Table A4. The influence of two diverging programs on the average program when averages are taken after normalizing by the number of block creations.

	F e w registers						M a n y registers			
	1 K=3 C=3 A=1	1 C=2 A=1	2 K=3 C=3 A=1	2 C=2 A=1	3 A=1	1Imp C=2 A=1	1 K=3 C=2 A=0	1 C=1 A=0	1Imp C=1 A=0	3Imp A=1
18 prog. cost/bl. norm.	13.92 2.72	5.12 1	7.02 1.37	3.14 0.61	1.60 0.31	3.36 0.66	10.67 2.08	1.87 0.37	0.99 0.19	0.60 0.12
20 prog. cost/bl. norm.	14.22 2.64	5.36 1	7.26 1.35	3.38 0.63	1.88 0.35	3.60 0.67	10.72 1.99	1.88 0.35	0.99 0.18	0.57 0.11
22 prog. cost/bl. norm.	14.50 2.56	5.66 1	7.54 1.33	3.66 0.65	2.20 0.39	3.88 0.69	10.72 1.89	1.88 0.33	0.99 0.17	0.54 0.10

Table A5. The influence of two diverging programs on the average program when averages are taken after normalizing by the sum of the significant events.

The tables indicate that the first method (normalizing per block) is more sensitive for diverging programs, but there is no dramatic effect. Normalizing per block makes the tables easier to understand, and still gives reasonable results.

Calculations of the distributions of significant events over levels.

The average distributions over levels of the significant events are calculated as simple averages of the percental distributions in the twenty programs. Also in this case other possibilities exist but this one is straightforward and seems reasonable.

We have also calculated the distribution obtained when taking the average of the number of accesses per block at each relative level. The distribution differs not very much from the one obtained by the method explained above, but a single program with many accesses affects the distribution much more than in the first method.

relative level	percental average	per block average
-4	0.3 %	0.1 %
-3	1.9 %	0.8 %
-2	10.9 %	8.6 %
-1	86.9 %	90.5 %
average number of levels to descend	1.16	1.11

Table A6. Distribution of accesses to blocks at intermediate levels for two different methods for calculating the average.

To be able to calculate the cost also in the improved version of method 1 the distribution of block entries and exits are divided into sub-distributions with associated figures representing their parts of one average block creation.

The average number of levels constructed in the display on each occasion given by the distributions are:

block creation	0.88
block exit	0.86
process switch	0.43

optimized case of method 1

block entry	0.43
block exit	0.43

Calculation of the coefficients in the cost expression.

The figures in table 9 are calculated from the average program constructed as explained in appendix A.

In section 2 we have found that for method 1 (global vector), the display has to be updated at block creation, block exit and for every process switch. Since the average number of process switches per block is 0.50, we get the total number of display updating per block:

$$k = 1 + 1 + 0.50 = 2.50$$

The average number of levels of the display that have to be updated is for block creations 0.88, for block exits 0.86 and for process switches 0.43. This gives the c-factor:

$$c = 1 * 0.88 + 1 * 0.86 + 0.50 * 0.43 = 1.96$$

The number of accesses to blocks at intermediate levels, a, is given as a simple average of the values from the twenty programs and is found in table 8.

For method 2 (local vector) the display is constructed once for each block, giving $k=1$. The average number of levels to construct is 0.88, giving $c=0.88$.

For method 3 (linked list) the a-factor is given by the number of block accesses of intermediate levels, 3.85, and the average number of levels to descend, 1.16:

$$a = 3.85 * 1.16 = 4.45$$

In the improved version of method 1, sub-block exits do not change the display vector. This gives a slightly decreased k-factor. Due to other optimizations the number of levels of the display that have to be updated of the display at each occasion is decreased to 0.43 for both block entries and exits. The corresponding value for process switches is still 0.43.

$$c = 1 * 0.43 + 0.99 * 0.43 + 0.50 * 0.43 = 1.07$$

The value of the improved version of method 3 is a simple average of the number of needed register loadings of block references within a block in the twenty programs, given by direct measurements in the running programs.