University of Glasgow
Department of Computing Science
Lilybank Gardens
Glasgow G12 8QQ

University of St. Andrews
Department of Computational Science
North Haugh
St Andrews KY16 9SS

PS-algol Machine Monitoring

Persistent Programming
Research Report 37
June 1987

Graham Kirby

PS-algol Machine Monitoring

Z. Loboz

Australian National University*

* This work was done while its author was at the Department of Computational Science of St. Andrews University, Scotland on leave from Regional Computer Center CYFRONET, Cracow, Poland on a scholarship from the Australian National University, Canberra, Australia.

Contents

Section One - on the Macintosh

Section Two - on the Vax.

Section One

Ps-algol Machine Monitoring On the Apple Macintosh

1. Problem description.

The PS-algol compiler produces an intermediate code which is interpreted by an interpreter written in C. The details of the virtual machine architecture and instruction set are described in [PPR11].

The monitored PS-algol system runs on a 512kb Macintosh computer. The Mac is equipped with 60Hz clock which allows the measurement of only those events lasting more than 1667 μs.

The interpreter works by fetching the next instruction stored at the global code pointer and then calling an appropriate C-procedure to execute it.

The objective is to design and implement a monitor for a PS-algol machine. The monitor should be cheap in terms of resources used and unobtrusive enough to be used on a regular basis. It should provide information on where the machine spends its time. The information provided should assist the tuning of the PS-algol machine performance.

2. Basic monitoring

This part describes the first version of a monitor installed in the interpreter and from which results were received. The objective of the first version was to gather enough data to have a general idea about the PS-algol machine behaviour and the various trade-offs involved, in order to build a more sophisticated system.

2.1 The monitor implementation

To reduce to an absolute minimum the changes required in the interpreter, only the main interpreter loop was altered. This was augmented with a counter that counted the number of instructions executed. Before entering the main interpreter loop, the current time is written to a disk file. Upon leaving the interpreter the current time and the table containing the instruction counters is written to a disk file.

In this way, after a program is executed by the interpreter, it is known how much time the whole execution has taken and how many times each PS-algol machine instruction was executed. Times used by the PS-algol machine instructions are **not** monitored during program execution.

To estimate the measuring capabilities of the monitor (i.e., how much information can be extracted from such data), the PS-algol compiler was compiled on a monitored PS-algol machine. The compilation of the PS-algol compiler was chosen as a benchmark of the PS-algol machine and its monitor capabilities because it is the biggest program written so far in this language, and therefore provides some non-trivial behaviour of the system.

2.2 Test's results

The compilation of the PS-algol compiler required 1234.93 seconds. The number of executed instructions was 5,505,220. Of 237 PS-algol machine instructions, 124 were used. For the complete list of instructions used and their number of calls, see the table *instruction list.*

From the above data we can infer that the average time required by an instruction execution cycle (interpreter fetch and decode time + instruction execution time) is 224 µs. Usually, in determining the timing of a program unit, a call to a system clock is made before and after the execution of the unit, which allows us to measure the time used by the program unit. Unfortunately the average PS Abstract Machine instruction execution time is well below clock resolution (224 vs. 1667 µs), and this makes this profiler method infeasible.

2.3 How to measure the time used by instruction execution.

Given the above difficulties, an alternative method must be found. In principle it is possible to measure the time used by any procedure by executing it many times in a loop. This will work only for procedures with execution times independent of the data with which they are supplied. Fortunately, many of the procedures for executing PS-algol instructions have this feature. An example of such a procedure is the procedure *gbl_1* which looks like this:

```
gbl_1()
{
        register psint  tmp1;
        tmp1 = (psint) (*gcp++);
        *lmsp++ = ( (psint * ) (lpsb[-4] ) ) [tmp1];
}
```


For such a procedure the time used is identical on each call (providing that errors such as *address out of range* do not occur). So it is possible to write a short program simultating the behavior of the PS-machine registers involved and execute this procedure (say) 1 million times. The time used will be of the order of seconds, and well above clock resolution. This method will be referred further as ***off-line timing***, because the time used by an instruction execution is not measured during a PS-algol program run.

In the case of this procedure it took 40.83 seconds to execute it 1 million times. The clock accuracy being 0.0017 sec, the relative error of estimating the instruction execution time is neglible. It could be made as small as desired by executing the procedure more times. The figure 40.83 seconds refers only to the time needed to execute the code of the procedure and it ***does not*** include the time used to call and return from the procedure. Such a time is referred to subsequently as an ***instruction execution time.***

The PS-machine instructions for which the instruction exececution times are the same on each call are further called ***elementary*** instructions. Of the total of 237 PS-machine instructions, 125 are elementary and were timed by the off line method described above. Their execution time is usually in the range from 20 to 70 µs, although some of them are over 1000 µs (in particular, the floating point operations).

How does this improve our understanding of the PS-machine behavior? During the test program run 68 of 125 elementary instructions were used. They were called 2,523,896 times, which represents 45.9% of all calls. The total time used by their execution (not including the time to fetch and decode the instruction, or to call and return from the instruction execution procedure) was 101.0 seconds, which represents 8.09% of the total test time. The three most frequently executed instructions of this type were

instruction	calls	seconds
GBL_1	422,675	17.26
PLC_1	376,319	14.29
LC_1	357,578	13.06

These results mean that for approximately half of the instructions executed on the PS-algol machine, the only kind of monitoring needed is to count them, which is indeed a simple method. But they also mean than some other method must be used to identify the instructions responsible for the remaining time used by the test.

2.4 Timing the main interpreter loop.

The main part of the interpreter, performing the fetch and decode cycle, is the most freguently used fragment of code. Its code looks as follows:

```
1)  next_instr:
2)      if ( sepend != (psint) 0 ){
3)              tmp1 = sepend ;
4)              sepend = (psint) o;
5)              sys_event ( (int) tmp1);
6)          }
7)      fns [ (int) (*gcp ++) ] ();
8)      goto next_instr;
```

This sequence of statements contains an *if* construction, therefore its execution time is not the same on each execution, depending on the *if* condition being true or false. However, in the case of the PS-algol interpreter this condition is false only at the end of program execution. So we may consider the main monitor loop to be an elementary instruction and time it in a manner similar to the other elementary instructions.

Lines 3 to 6 (inside the if) are executed only at the end of a program, so we may assume that it is enough to measure the time used by lines 1, 2, 7, 8. A program was written to simulate the behaviour of the PS-algol machine with variable *sepend* always equal 0. Line 7, in which the appropriate procedure to execute a PS instruction is called, performs several actions: update global code pointer, fetch the next instruction code and call an appropriate subroutine. All these actions were timed separately. Instead of subroutines executing real instructions, the empty subroutine (text: *emtysub () {}*) was substituted. Hence the time of the interpreter loop found in this way describes the time needed to fetch the next instruction, select an appropriate procedure, call the procedure and return from it. The time of the *goto* instruction was measured separately. Times (individual and cumulative) of components of the main interpreter loop in µs are as follows:

component	time	cumulative
if (sepend != (psint) 0)	5.12	5.12
*(*gcp++)*	7.95	3.07
*(int) (*gcp++)*	3.12	16.19
*fns [(int) (*gcp++)]*	4.37	20.56
*fns [(int) (*gcp++)] ()*	15.42	35.98
goto next_instr	1.91	37.89

To monitor the frequency of instruction execution, the following (not very optimal) line was added after line 1 of the main interpreter loop:

```
moncnta[ (int) (*gcp)] ++;
```

moncnta is an array of 32bit integers holding instruction counts. The time needed to execute main interpreter loop with this line added (it increments the instruction counter of the next instruction to be executed) is 51.10 µs.

The main interpreter loop is executed each time a PS-instruction is executed. During the test 5,505,220 instructions were executed, so the total time used by the main interpreter loop was 208.6 seconds and by the monitor 72.2 seconds. This represents 16.9 and 5.8 percent respectively of the total test time.

2.5 Preliminary conclusions

At this point it is possible to conclude that a simple monitoring augmented with an off-line (i.e. outside normal program execution) timing of interpreter behavior enables us to know what the PS-algol machine is doing for 46% of all instruction executions and for 31% of the total test time. The cost of monitoring is not exhorbitant (6%) and may be reduced (if needed) to about 3% by a more optimal coding.

The measured version of the interpreter uses procedure calls to execute PS instructions. This creates an overhead because of the time needed to call and return from a subroutine. The other version of the interpreter in use on other machines uses a *case* switch method, without execution subroutines. It was not practicable to use this method on a Mac. If it had been, it would have cost about 22.00 µs per interpreter cycle (vs. 38.6 for procedure call) and the resulting gain in this test would have been 91.3 seconds (7.4 %).

An interesting (and paradoxical) conclusion concerns the system clock. We are much better off without a microsecond clock than with. Let us assume for the moment that the Macinitosh clock is a microsecond clock with resolution fine enough to measure the time needed by any PS instruction execution. To measure the time used by an instruction execution, 3 operations are needed as a minimum: read the clock time before executing an instruction, read it afterwards and add the difference of both to the total time already used by the instruction. This costs on the Mac 142 µs, and doing it for all instructions executed would have added 781 seconds to our test time. In other words, it would have resulted in a monitor using 43% of the machine time. This is an obviously unappealling scenario for a "cheap and simple" monitor. In addition, with the Mac having a simple single-user operating system, the system clock call is relatively cheap. On a bigger computer it should be expected to cost much more.

3. Timing other PS-algol machine instructions.

The technique used in chapter 2 allows for full monitoring (i.e. number of executions and time spent) of 125 out of 237 PS-algol machine instructions. They comprise 45.9% of all instructions executed during the program run, and are responsible for 8.09% of the test's time. Because instructions are only counted during program execution, the method is cheap. It will be therefore worth trying to extend it to other instructions, if feasible.

3.1 i1 type instructions.

Let us take for example instruction JUMPF_2, for which the executing procedure looks as follows:

```
jumpf_2()
{
        register psint tmp1;
        tmp1 = (psint) (*gcp++) << (psint) 8 ;
        tmp1 |= (psint) (*gcp++)
        f( !(*--lmsp) ) gcp += tmp1 ;
}
```

This procedure contains an *if* instruction, and its execution time depends on a condition being true or false, so its execution time is not constant. It is impossible to find out the time used by the instruction during the PS-algol program execution using the previous method only. However, this procedure has only two possible execution times - one for *if(false)* and one for *if(true)*. It is therefore possible to consider the instruction as two elementary instructions and measure (off-line) the time required in each case (*if true* or *if false*).

To know the total time used by both branches of the instruction during a PS-algol program run, information on how many times one of the branches was used is needed (one only, because the total number of the procedure calls has already been counted by the basic monitor; this holds also for *if-then-else* sequences). This requires an installation of one additional counter in the instruction code.

To time such an instruction, a short program was written simulating the behavior of the PS-machine registers involved. Then the instruction was executed in a loop 1 million times for *if(false)* and 1 milion times for *if (true)*. The execution time for *if (false)* was 46.86 µs and for *if(true)* 54.36 µs.

In the test case the instruction was called 441,781 times and the *if(true)* branch was executed 134,892 times. So the total time used by both branches of an instruction during the test was 23.00 seconds.

Such instructions will be further called **i1** instructions (i for if, 1 for one additional counter). There are 15 such instructions in the PS-algol machine.

3.2 I1x type instructions

The next class of instructions which can be timed off-line are instructions similar to the **i1** class, but containing in one of the *if* branches a call to another procedure. Such instructions are further called **i1x** instructions (x for external).

An example of such instruction is PGBL_1:

```
pgbl_1()
{
        register psint tmp1,tmp2,*ptr1;
        tmp1 = (psint) (*gcp++) ;
        ptr1 = (psint) (*lspb[-4]) ;
        tmp2 = ptr1 [1] / (psint) 4 - tmp1 ;
        tmp1 = ptr1 [tmp2];
        if ( tmp < (psint) 0 ){
                comp1 = ptr1 ;
                tmp1 = (psint) (ill_adr (tmp1)) ;
                comp1 [tmp2] = tmp1 ;
        }
        *--lpsp = tmp1 ;
}
```

This instruction, inside an *if* , contains a call to the *ill_adr* procedure. This makes full timing of the instruction impossible, as the procedure called is in itself complicated enough to discourage any attempts to time it. It is possible, however, to time the part of an instruction executed when the if condition is false. Similarly, as for an **i1** instruction type, an additional counter must be added to count during a PS-program execution, the number of times one of the instruction branches was used.

Using the same technique of simulating the PS-machine registers involved in a short program, the time of executing this instruction was found out to be 131.8 µs for *if(cond)* false (i.e. external procedure not called).

In all, there are 14 such instructions in the PS-algol machine. The method does not allow one to find out the whole time used by such instructions during the PS-algol program run, because it is impossible to find out off-line the time required by an execution of any called procedures. However, in most cases, procedures are called to handle unusual situations (like errors), and so the number of **i1x** type instruction calls which could not be timed should be small.

3.3 i2,i2x, ... type instructions

There is a small group of instructions which can be considered as a combinations of instruction types **i1** and **i1x**. Their code contains 2 or 3 *if*'s and sometimes a call to a procedure in one of the if branches. They may be timed using the technique described in the previous sections. Installation of additonal counters to monitor how many times the particular branch of an if instruction was executed is required. Execution time for such instructions may be timed off-line, either fully, in the case of **i2,i3** instructions, or partially, in the case of instructions calling procedures in one of the if branches.

There are 7 such instructions in the PS-algol machine.

3.4 Special cases

The results of the basic monitoring had pointed out a singnificant percentage of executions of instructions which do not belong to any of the instruction types described so far. To increase the percentage of fully monitored (i.e. number of calls **and** time used) instructions two special cases were timed off-line.

3.4.1 String comparison.

There are three instructions (CJUMP_S_2, EQ_S, NEQ_S) which comprise 6.05% of all instruction executions during the test run. The procedure for the CJUMP_S_2 contains an if instruction, while the procedures for EQ_S and NEQ_S contain a simple sequence of commands. They cannot, however, be timed directly, because they call an *eq_string* procedure. In the case of CJUMP_S_2, the procedure is called while evaluating the if condition, and therefore even partial off-line timing is impossible. The *eq_string* procedure compares two PS-algol strings and returns the value *true* if they are equal and *false* otherwise.

To measure their instruction execution time, instead of *eq_string* a dummy procedure was substituted and a time used by these instructions without comparing the strings was measured. In the case of CJUMP_S_2 the method used was similar to **i1** type instructions, in two other cases as for the elementary instructions. The execution time found reflects only the time needed to execute the instructions. It does not include the time for a: call and return from the instruction procedure (for consistency with instructions measured so far); string comparison; nor call and return from string comparison instructions.

The eq_string procedure looks as follows:

```
psint eq_string( s1,s2 )
register psint *s1,*s2 ;
{
        register char *c1,*c2 ;
        register psint len ;

        if ( s1 == s2 ) return( PSTRUE ) ;
        if ( !s1 || !s2 ) return( PSFALSE ) ;

        len = *s1++ & LOWER16 ;
        if ( len != ( *s2++ & LOWER16 ) ) return( PSFALSE ) ;

        c1 = ( char * ) s1 ; c2 = ( char * ) s2

        while (len-- > ( psint )0) if ( *c1++ != *c2++ )return(PSFALSE);
        return( PSTRUE ) ;
}
```

It is basically a sequence of statements. After each statement the procedure may finish its execution. This holds also for the *while* loop. To find all information about all possible execution paths several counters must be added, practically one after each line. One counter was added also in the *while* loop to count how many characters were compared.

The execution time was measured for each situation (each possible *return*). In the case of the *while* instruction, an average time of comparing two characters was measured.

Using this data it is possible to find out how much time was spent by the PS-algol program executing these 3 instructions (without the time used to compare strings). It is also possible to compute separately the time used by the *eq_string* procedure. This does not give all information about the time used to execute each of the 3 instructions, because a) procedure *eq_string* may be called also from other places, and b) the number of characters compared by *eq_string* may not be the same when called from EQ_S procedure as when called from NEQ_S or CJUMP_S_2 procedures.

3.4.2 Instruction APPLY_OP

This instruction comprised 3.66% of all instruction executions during the test run, so therefore an attempt was made to time it off-line as fully as possible.

The procedure execution is complicated. It contains two nested *if* constructions and 3 *while* constructions. It also calls 4 other procedures: *go_stand* (which, in turn, calls standard functions), *claim*, to obtain heap space, *garb_coll*, to perform garbage collection, and *sys_event*, in error situations.

Additional counters were installed to count the number of times each branch of an *if* instruction was executed (and a procedure was called). The counters were also installed to monitor execution of *while* loops.

A call to the *claim* procedure is used to evaluate one of the *if*'s conditions. Fortunately the *claim* procedure is similar to an **i1** type instruction and can be timed separately. Additionally, those paths of the APPLY_OP instruction not calling other procedures were timed off-line. This effectively allows us to compute the time used by the procedure in situations when neither standard function nor garbage collection nor sys_event procedures are called.

3.4.3 Garbage collection timing

The procedure performing garbage collection was called relatively rarely (41 times) during the test. Neverthless, the algorithm is complicated, and its execution time exceeds clock resolution. Therefore two counters were installed in that procedure: one counting the number of calls; another computing the time used by the procedure.

It is worth pointing out that this is the only situation in the monitor when a time used by the PS-algol machine is measured directly during the PS-algol program execution.

4. Extended monitor - implementation and results.

This part describes the implementation and results received from the second version of the monitor. The second version is built on the same principle as the first one. It does not *time* any instructions (save garbage collection). But additional counters were added to count instruction execution in more detail. The added counters reflect the need to compute the frequency of any given execution path for instructions described in the previous chapter. An off-line measurement of instruction execution time for each path was done. These two types of data combined allow us to compute the time used by instructions of the types described in chapter 2, while executing the PS-algol compiler program. In addition, further counters were added to count the number of times each standard PS-algol function was called.

At the beginning of execution of a PS-algol program the monitor file is opened and the current clock time written to it. Upon finishing the program execution the current clock time and all the counters are written to this file. Hence the time used by the program is known. By analysing the counters we can also deduce how many times each instruction was called, and in the case of certain instructions, how many times each path inside an instruction in question were executed. Combining this information with the data received from an off-line instruction timing (as described in chapter 2) it is possible to ascertain the details of the behaviour of the PS-algol Abstract Machine.

This chapter describes the results received as seen from the point of view of evaluating various aspects of **monitor performance**.

4.1 The test

The test performed was identical to that used during the basic monitor test. The PS-algol interpreter was augmented with additional counters. The PS-algol compiler was compiled on the PS-algol machine using the monitored version of the interpreter. The monitor file produced by the interpreter was analysed with other programs.

4.2 The monitor performance criteria

The two basic monitor performance criteria are: the amount of system resources (such as disk space, procesor time, memory) used by the monitor, and the amount of information obtained from it. In the case of this monitor, because of the off-line timing technique used, it is also interesting to know how much of the PS-algol program running time can be accounted for.

4.3 The results

The total time used by the compilation of the PS-algol compiler was 1249.22 seconds. It is 14.29 seconds more than during the test of the basic monitor. Because the same program was compiled in both cases, the additional time represents the time used by counters added in the second version of the monitor.

This monitor extension may be considered a very cheap one, in terms of processor time used, using only 1.15% of total test time. The basic monitoring (see 2.4) costs 72.2 seconds (5.8%), so the total cost of monitoring is 7% of the PS-algol machine time. This is not exhorbitant, and the monitor time could be halved by some simple changes in coding. Apart from the processor, the monitor uses one table for counters. The table contains 512 entries occupying 2Kbytes of memory. It is difficult to determine precisely the memory required by the code inserted to count the instructions, but it is certainly less than 1Kbytes. Apart from this, some 2056 bytes disk space is required for a monitor file containing times and counters.

Considering the above, the monitor may be described as "a cheap and dirty one". It remains to be seen how much information we can extract from it. It is important to know what percentage of instruction executions can be fully monitored and what percentage of the time used by a program can be explained.

4.3.1 Elementary instructions.

The results received are, of course, identical during the basic monitoring (chapter 2). Of all 125 elementary instructions, 68 were used during the test. Of 5,505,220 instructions executed during the test 2,523,896 were elementary, or 45.85%. Their combined execution time was 101.0 seconds, or 8.09 % of total test's time. The average time to execute an elementary instruction is 40 μs.

The three most frequently executed instructions of this class and the times used by them were:

instruction	calls	seconds
GBL_1	422,675	17.26
PLC_1	376,319	14.29
LC_1	357,578	13.06

4.3.2 i1 instructions.

Of 15 instructions of this type, 10 were used during the test. They were called 991,461 times, or 18.01% of all instruction executions. Their combined time was 65.2 seconds, or 5.22% of total test time. The average execution time in this class was 66 μs.

The three most frequently executed instructions of this class and times used by them were: JUMPF_2 (441,781 calls, 23.00 seconds), LL_INT_1 (327,468 c, 11.82 s), JUMPFF_2 (122,076 c, 6.33 s).

4.3.3 i1x instructions

Of 14 instructions of this type 7 were used. They were called 827,699 times. Only in 757 cases (0.09%) a call to them resulted in calling an external (to the interpreter) procedure. This means, that it is possible to compute the time used by this instructions in 826,942 cases - 15.02% of all instructions executed. Their total execution time (for measurable 826,942 executions) is 108.0 seconds. The average execution time is 131 μs.

The three most frequently executed instructions in this class and times used by them were:

instruction	executions	seconds
PGBL_1	460,894	60.73
PGBL_2	252,300	36.24
SVA_S	67,782	4.51

In the case of PGBL_1, 6 executions resulted in a call to an external procedure. The time used in these cases is not included.

4.3.4 Combinations of i1x and i1 instructions.

There are 10 instructions whose type is a combination of **i1** and **i1x** types. They were executed 403905 times, in 3453 of these executions (0.09%) an external procedure was called. So there were 400,499 measurable executions, or 7.27% of all instruction executions. The total execution time for measurable executions was 106.0 seconds, or 8.49 % of test time. The average execution time was 256 μs.

The three most frequently executed instructions of this class and times used by them were

instruction	executions	seconds
DPGBL_2	134,397	22.80
SRTN_V	93,274	40.84
SUBV_S	78,142	11.33

4.3.5 Instructions using the string comparison.

The three instructions calling *eq_string* procedure to compare strings were called 333,028 times, or 6.05% of all instruction executions. Their total time, without the time used to compare the strings, was 16.1 seconds, or 1.29 % of total test time.

The *eq_string* procedure comparing the strings was called 346,818 times (so 13,790 calls to this procedure, or 0.04%, were made from other places than these three instructions). The total time used by this procedure was 42.2 seconds, or 3.38% of total test time. It may be estimated, however, that due to the significant number of counters installed in it, about 6 seconds were spend on monitoring.

4.3.6 APPLY_OP instruction.

This instruction was called 201,141 times. 46,264 of these calls were calls for a standard function. 41 calls resulted in a call for a garbage collection. In all, 154,386 calls representing 2.81% of all instruction executions may be timed. Their total execution time is 90.2 seconds, which represents 7.22% of total test time. This time does not include either the time spent calling and executing standard functions, nor the time spent performing garbage collection.

4.3.7 Garbage collection.

The garbage collector was called 46 times during the test. The time used by the garbage collector was 42.2 seconds, which gives an average time to perform garbage collection of 0.92 seconds. The relative error of the measurement is about 1%.

4.3.8 The basic interpreter loop.

As described in the chapter 2, the time used by the basic interpreter loop is 212.8 seconds, or 17.04% of the total test time. The time spent monitoring the loop is 72.2 seconds, or 5.8 %.

4.3.9 Summarizing percentages.

In all, the instructions mentioned in this section comprise 95.01% of all instructions executed, giving thus nearly complete coverage in this aspect. This means that it is possible to compute the time used by almost all instructions executed on a PS-algol machine using the technique of line monitoring and on-line instruction counting.

The total time used by the instructions for which the execution time may be measured off-line, together with the time used by the monitor, the main interpreter loop, the garbage collection and the string comparison, is 878.2 seconds. This represents 70.3% of the total test time.

4.4 What is not covered?

The results received so far point out that although 95% of all instruction executions on the PS-algol machine can be accounted for, only 70% of the time is accounted for. In other words the remaining 5% of instruction executions are responsible for 30% of test time. What kind of instructions are executed during this 5% of executions? Is it possible to explain (if not measure directly or off-line) this missing time?

4.4.1 Estimating i/o time.

The most frequently executed instruction which was not timed was the READ_OP. It was executed 128,749 times, or 2.34 % of all instructions executed. Another one was the WRITE_OP, with 57,665 executions and 1.05% of all instruction executed. The third one (a part of the APPLY_OP instruction which calls a standard function 46,305 times) represents 0.84% of all calls. In all, these three items total 4.2 % of all calls. The remaining 0.8% belong to IS_OP - 0.3% (with logic too complicated to time it off-line easily) and several other instructions with the number of executions less than 10,000.

Of these instructions only the input/output operations can be suspected of using much of the remaining 30% of test time. The other ones are relatively short procedures with estimated execution time too small to account for the missing 371 seconds. The input /output operations are, however, too complicated to be timed off-line. They cannot also be monitored with a direct timing, because even superficial analysis of the problem suggests that their average execution time is about the same as the clock cycle time, and a short test confirmed this.

However, analysing what our test is doing, it is possible to devise a method of estimating, although roughly, the time used by the input/output. There are 3 basic i/o operations executed while compiling the PS-algol compiler: reading the compiler text by one character at a time and testing for the end of file, writing on a display a compiler listing with line numbers appended, and writing the pseudocode file.

To estimate how much time was used reading the compiler text, a short program in PS-algol was written with a basic loop *while ~eof(infile) do ch := read(infile)*. Then the program was executed using the PS-algol compiler as input. The resulting monitor file was analysed to find out how much of the program time was unexplained (the assumption being that the unexplained time was spent performing i/o, as no other operations were performed by the program). Of 88 seconds execution time, 58 were accounted for - interpreter time, monitor, some simple instructions. That left 30 seconds for the file read.

To estimate how much time was spend producing the program listing, with line numbers appended, another short program was written. As the compiler text is 2477 lines long with an average line length 29 characters, a short program was written. Using a*for* loop and a string 29 characters long, 2477 lines were printed on a screen together with their line numbers. The resulting monitor file was then analysed, as in the previous case. Of the total time of this test 300 seconds were unexplained. So, it may be assumed that this is the minimal time to get that amount of text on the screen. (Minimal, because the listing may be produced by a compiler in many ways and each one of them will require more time than the one tested).

The time to write out the PS-algol code file cannot be estimated that way. It is difficult to create a model program to perform the writing in a similar way to the PS-algol compiler. However, let us assume for a moment that the time to write this file is of the same order as the time to read it character by character. This will require about 20 seconds.

So the time used by the 3 i/o operations is about (very roughly) 350 seconds, leaving about 20 seconds of the main test time unexplained. (And 1.6% of all instruction executions not timed).

4.5 Summary of monitor performance.

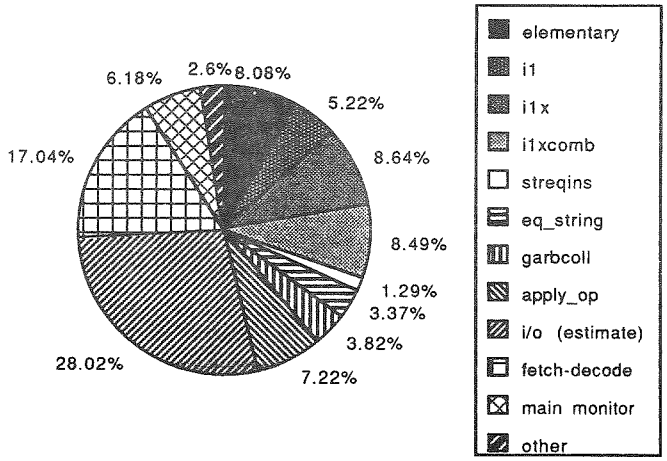
Of 237 different PS-algol machine instructions 142 may be timed fully using the off-line timing. 125 of them are elementary, 15 of **i1** type, 2 of **i2** type. A further 24 instructions may be timed partially during executions not calling another procedure.

During the test run it was possible to account for the time of 95% of all instructions executions. Together with the time used by the fetch-decode cycle, monitoring, garbage collection - 70% of the test time can be precisely accounted for.

Input/output operations were responsible for 3.34 % of all instruction executions and a rough estimation suggests that the time used by them is almost 30% of total test time. That leaves 1.76% of all instruction executions too difficult to time off-line, and about 20 seconds (2%) of the test time unaccounted.

The monitor used about 7% of the test time, about 3 Kbytes of memory and about 2Kbytes of disk space.

It is cheap enough to be incorporated into the interpreter as a standard option.



SUMMARY OF THE TIME USED BY THE VIRTUAL MACHINE

5. Analysing the PS-algol machine.

This chapter describes the results received during the test from a point of view of the PS-algol machine.

The compilation of PS-algol compiler was chosen as a benchmark, and it is assumed that the reader is aware of the advantages and disadvantages of this.

The time needed to compile the compiler was 1249.22 seconds. 5,505,220 PS-algol machine instructions were executed during that time. Of 237 PS-machine instructions 124 were used.

5.1 Description of tables.

The table *INSTRUCTION LIST* lists the names of all PS-machine instructions, their codes, number of executions during the test, and the time used. The following legend is used.

Name instructions names as defined in the PS-algol interpreter.
Code instruction code.
Type type of instruction as seen by the monitor. **e** for elementary instructions, **i1**, **i1x**, as described in chapter 2.
Utime time (in μs) needed to execute the instruction once. For instructions of **i1** class the time of only one branch is given here.
Calls number of times the procedure executing the instruction was called (= number of instruction executions).
Pccall number of instruction executions as a percentage of all instruction executions.
Tottime total time used by the instruction during program execution. For **i1** instruction type this is the time used by both branches of the instruction. For **i1x** instruction type this is the time used by the instruction branches which do not call an external procedure. For instructions using the *eq_string* procedure the time given does not include the time spent comparing the strings. For the APPLY_OP instruction this is the time spent executing only measurable branches, without garbage collection, standard function calls.

The second table, *FREQUENCY*, contains similar information as the first one, but only for the most frequently used instructions sorted in frequency of use. There is one additional column.

Cpccalls cumulative percentage of number of instruction executions.

Additionally, data for certain instructions should be regarded with caution (see the next section for explanations).

5.2 Comments.

This section contains additional explanations to the instructions listed in table FREQUENCY

CJUMP_S_2 calls *eq_string* procedure, the total time given for the instruction does not include the time to call and execute this procedure.

APPLY_OP_2 calls several procedures depending on the situation in the PS-algol machine. The total time given does not include the time used by this instruction and procedures called by it, while a call to perform garbage collection or execute standard function was made. It does however include the time used by the claim procedure.

READ_OP no total time given, as this instruction was not timed off-line.

EQ_S see CJUMP_S_2

WRITE_OP see READ_OP

NEQ_S see CJUMP_S_2

IS_OP see READ_OP. Also: this instruction can be timed off-line partially, but it was used too rarely to undertake this rather complicated job. The same holds for all other non-timed instructions in this table.

5.3 Additional results.

The tables do not cover some additional results obtained, because they were not directly connected with specific instructions.

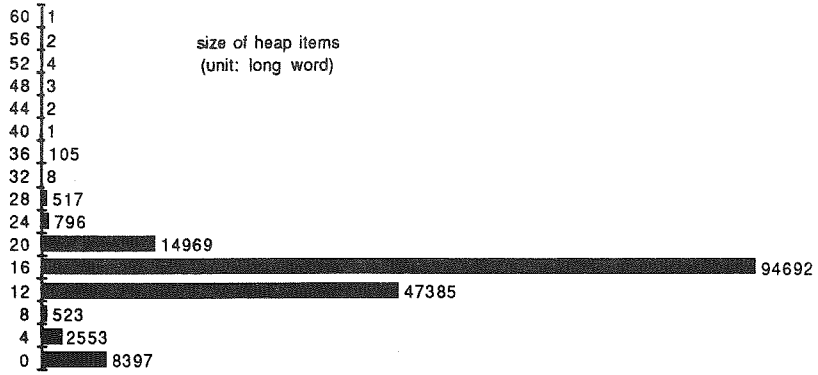
The timing of the main interpreter loop is described in 2.3 and the results in this version of the monitor are identical, because that part of the PS-algol machine was not changed. The same holds for the time used by the part of the monitor installed in the main loop.

5.3.1 Garbage collection.

Garbage collection was performed during the test 46 times. The total time used was 42.2 seconds, the only case when the time used by the PS-algol machine is measured directly during a program execution. The average time to perform the garbage collection was thus 0.92 seconds. The garbage collection time constitutes only 3.7% of the total test time and several of the instructions used more time. Neverthless, with the time per execution being so high, this overhead in the PS-machine may show up in applications requiring more intensive garbage collection.

In all 170,002 items were put onto heap. The total size of all items was 2,730,372 4-byte units, which gives average item size 16 (64 bytes).

These values do not include 2 memory units needed for each item in the heap for administration. The size of heap that would have been required to execute the program without any garbage collection is 12,281,504 bytes.



5.3.2 String comparison.

Procedure *eq_string* was called 346,818 times. It is called from various PS-machine instructions to return the value *true* if the two strings to be compared are equal and the value *false* otherwise.

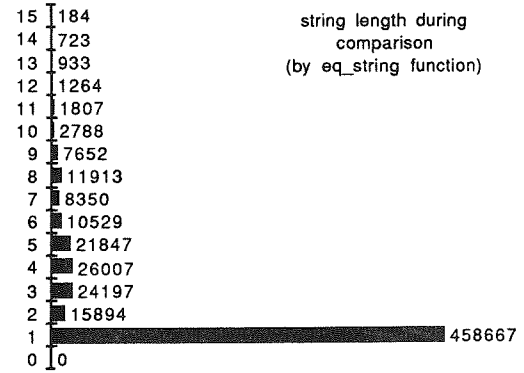
Counters installed to monitor an execution of this procedure allow us to obtain some additional information.

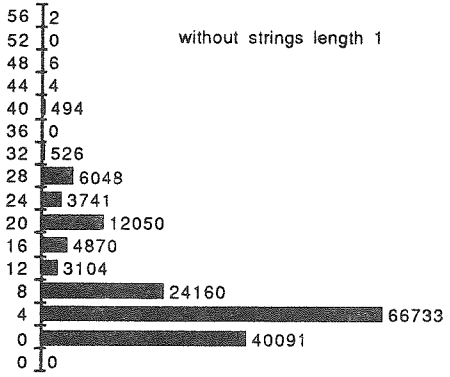
The total time used by this procedure was 42.2 seconds (it may be estimated that about 6 seconds of this are spend on monitoring).

At the beginning of its execution the procedure tests two trivial items: if the passed string pointers are identical, or one of them is zero. This enables it to return the result faster (without going into a detailed string check). This happened 36,515 times (10% of all calls).

In the next step the subroutine computes lengths of strings to be compared: if they are different it returns value *false*. This happened 77,166 times (22% of all calls). The average length of the string to compare was 2.99 characters.

All the preliminary tests being positive, the procedure performs character by character comparison of strings using the while loop. This loop was executed in 68% of all calls to this procedure. A total of 279549 characters were compared, which gives on the average 1.2 characters comparisons per string to compare.





5.3.3 Standard functions.

The standard functions are called through APPLY_OP instruction (which also performs other duties). There were 46,264 calls to the standard functions. Nine of them were called: LENGTH (9,717), IFORMAT (10,663), LETTER (2), LINE_NUMBER (8,973), SHIFT_R (3), SHIFT_L (60), B_AND (20), B_OR (36), B_NOT (4).

The time to execute standard functions was not measured off-line or by any other method. It is, however, possible to do so for certain functions.

The time used by APPLY_OP instruction to issue a call to a standard function is 180 μs. This does not include the time used to call and execute the standard function procedure. So the total overhead (not included in the APPLY_OP time in the tables) was about 8.3 seconds.

5.3.4 Input/output operations.

The READ_OP instruction was executed 128,749 times. The following input operations were used: READ (50,088), EOF (69,622), READ_NAME (9,036).

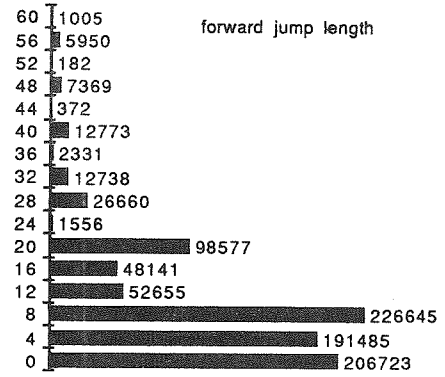
The WRITE_OP instruction was used 57,665 times. The following output operations were used: WRITEI (2,536), WRITES (55,095), OUTBYTE (34).

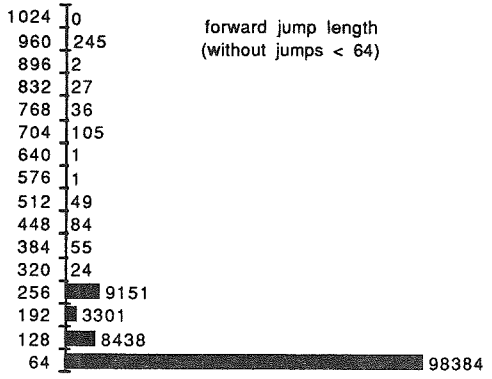
The length of the file containing the compiler text is 74,000 bytes (2477 lines). The number of READ and EOF operations performed allow us to deduce that this file is read predominantly in a character-by-character way with an end-of-file test after each character. On this assumption the time used by the two operations was estimated (see 4.5.1) to be about 30 seconds.

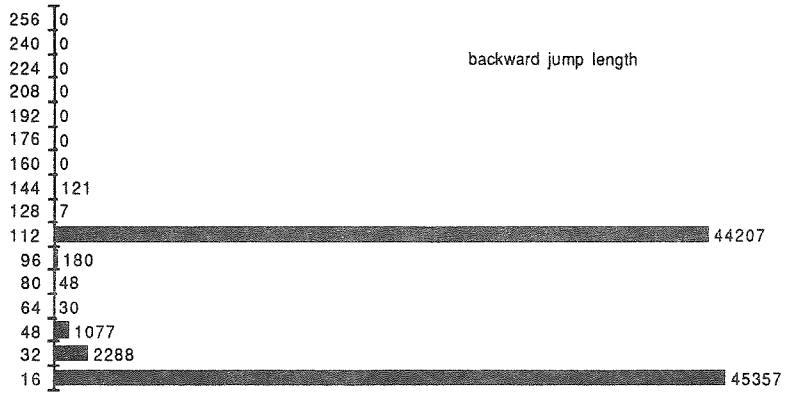
It is difficult to estimate the time spend writing the code file, because both output files (the program listing and the code file) were written as a sequence of strings of unknown length. Therefore the time used to produce a listing was estimated in a way described in 4.4.5, and yielded a value of about 300 seconds. The estimate of the time used to produce the code file is 20 seconds (see 4.5.1).

5.3.5 Jumps.

Jumps comprised about 20% of all processed instructions. Most of them (1 million) were forward jumps, 800,000 of which were conditional jumps. 200,000 conditional jumps were executed, i.e. after the condition was evaluated, the program counter was updated. In all a jump (including loop evaluations) was executed on the average after each ten instructions.

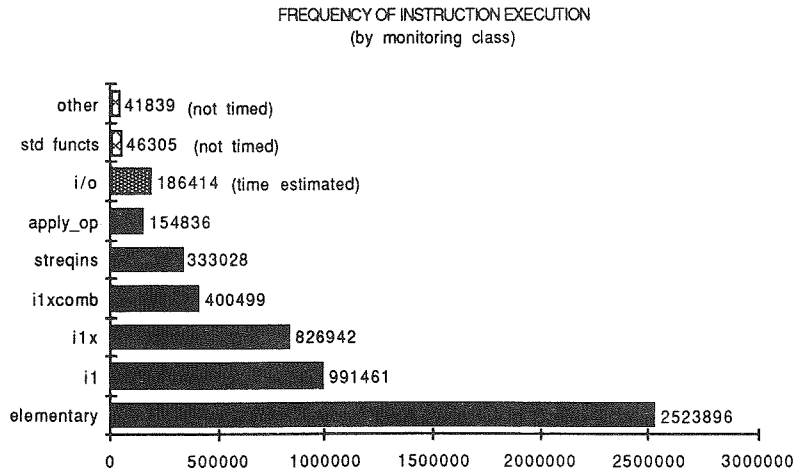




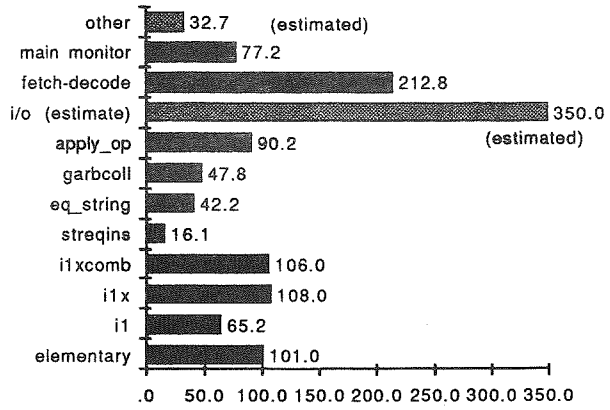


5.4 Results' summary.

The first chart depict the distribution of calls to the instruction classes as defined for the monitor. Except for the i/o instructions, calls to standard functions and 0.8% of other instruction executions, all instruction executions are fully monitored.



The next chart depict the distribution of time used while executing the PS-algol interpreter. The biggest single item is i/o, about 350 seconds.



6.Tuning Possibilities.

This chapter describes possible improvements to the PS-algol interpreter, which, when implemented, will result in its increased speed.

Results received during the testing phase of the monitor allowed the identification of some PS-machine instructions which are using enough machine time to be worth more detailed analysis. The interpreter is written in C. The code is extremely tight and there are practically no possibilities left to improve interpreter speed by trying to write the instructions in a "more clever way".

There are however some possibilities for speeding up the interpreter if we consider the interface between the microprocesor used and the C compiler. There are also some tuning possibilities in the area of string comparison and the coding of the main interpreter loop (the fetch-decode cycle).

6.1 Division - shift.

The most frequently used instruction is PGBL_1 (see its code in 3.2). Its execution time is 131.8 µs and the total time used by it during the test is 60.73 seconds. The execution time is surprisingly high for such a simple sequence of statements. One statement in this procedure looks as follows :

```
tmp2 = ptr[1] / (psint) 4 - tmp1;
```

Analysis of the code produced by the C compiler for this instruction shows that the division is really translated into a 32-integer division, which is a time consuming algorithm for the 16-bit arithmetic unit of M68000. In such a situation, division by 4 may be replaced by a right shift of 2 bits - the result will be identical.

Such a change will improve the instruction execution time by about 51 microseconds (38 %), bringing it down to 81 µs. The time used by this instruction during the test will go down from 60 to 37 seconds.

There are several such instructions in the PS-algol machine. SRTN_IB, for example, uses 3 such divisions and its execution time is 337.8 µs. Substituting shift for division brings down its execution time to 175.8 µs.

Shift was substituted for division in such instructions and a new version of the interpreter was used to perform the same test. The new execution time was 1143.80 seconds (vs. 1249.22 for the standard version). So the improvement was 105.4 seconds.

6.2 Getting an integer from code stack.

Instruction parameters frequently reside on the code stack. Many instructions require as a parameter a 16-bit integer, which is taken from the code stream byte by byte. The following sequence is used:

```
tmp1 = ( psint )( *gcp++ ) << ( psint ) 8 ;
    tmp1 |= ( psint )( *gcp++ ) ;
```

Such a sequence is optimal at the C-language level.

It results in the following assembly code:


```
tmp1 = ( psint )( *gcp++ ) << ( psint ) 8 ;
move.l    gcp_,a2          ; addres of the code pointer into a2
add.l     #1,gcp_          ; increment the cp in memory
move.b    (a2),d3          ; contents of the cp into d3
and.l     #$ff,d3          ; convert to psint
move.l    d3,d4            ; result to d4
asl.l     #8,d4            ; and shift it to the second byte
tmp1 |= ( psint )( *gcp++ ) ;
move.l    gcp_,a2          ; once more cp address into a2
add.l     #1,gcp_          ; increment cp address in memory
move.     (a2),d3          ; next byte from cp stack to d3
and.l     #$ff,d3          ; convert to psint
or.l      d3,d4            ; construct 16-bit integer
```

This is far from optimal for this (and practically any other) microprocessor. The 16-bit integer cannnot be taken from the code stack in one operation, because such operations have to be aligned on a word boundary, and it is difficult to assure that it will always happen. Moreover, the bytes taken from stack are in reverse order, due to the addressing scheme of the M68000. Neverthless, some obvious redundancies may be eliminated and following code used instead:

```
move      gcp_,a2          ;address of cp to a2
add.l     #2,gcp_          ;update cp
clr.l     d4               ;prepare result
move.b    (a2)+,d4         ;get first byte from stack
asl.l     #8,d4            ;shift it to the second position
move.b    (a2),d4          ;get second byte
```

reside

This sequence is faster than the previous one by 14.79 µs. As an example let us take an instruction like PGBL_2, which uses both division (as described in the previous section), and a 16-bit integer from code stack. Its execution time is 143.6 µs. After a shift is substituted for the division, the time is 92.9 µs. With additional optimization of stack access, this becomes 78.1 µs.

Although this optimization requires changes on the assembler level, it is simple to implement. The C compiler produces, optionally, an assembly language file and using the editor, one set of lines is replaced by another. Such a replacement was made and a new version of an interpreter was tested using the same test as previously. The execution time was 1118.25 seconds, some 25.5 seconds faster than previously.

6.3 Comparing strings.

Strings were compared rather frequently during the test program (see 5.3.2) and the total time used by the *eq_string* function was 42.2 seconds. This function is called mostly (96% of all calls) from 3 instructions: CJUMP_S_2, NEQ_S, EQ_S. The results show that the average length of compared strings is 2.99 characters. On average 1.2 characters are compared before finding an answer.

These results suggest a need for another approach to the algorithm used. The procedure tests firsts (5.3.2) some trivialities and afterwards, in a while loop, checks characters one by one. The overhead is caused by the procedure call and the while loop.

The string in the PS-algol starts with a long word containing its length on the lower 16 bits. The long words after the first one contain characters padded to a long word boundary. The new algorithm should be similar to the previous one - first test trivialities, after that test equality of first 4 characters in the string (there are always first 4 characters, due to padding). This should be done not in a subroutine, but directly in a PS-instruction. If the result is true - only then should the *eq_string* procedure be used. The *eq_string* procedure should be rewritten to compare 4 characters at a time. Such a solution will significanly reduce the number of calls to the subroutine.

The following macro may be used instead of a call to the *eq_string* subroutine in string-comparison instructions:

```
#define COMPSTR \
        (!ptr1 || !ptr2 ||\
                ((*ptr1 & LOWER16) != (*ptr2 & LOWER16)) ||\
                (ptr1[1] != ptr2[1])\
        ) ? PSFALSE : eq_string (ptr1,ptr2)
```

Its second line checks if one of the pointers is zero, the third line tests if the lengths of strings are not equal, the fourth line compares the first four characters of both strings. The only mising (in comparison with original procedure) part of an algorithm is a test for identical pointers - rather difficult to implement in this way and rarely producing the *true* answer.

The times used (in µs) to return a result to compare strings with the following properties:

	eq_string	macro
one of string pointers zero	60.10	7.88
different length	87.17	24.38
different 1st char	115.67	32.40
different 2nd char	139.55	32.40
different 3rd char	163.34	32.40
different 4th char	187.32	32.40

As it may be seen from this an elimination of the subroutine call overhead brings a significant speed-up. Also impressive is the reduction due to the elimination of the *while* loop to compare the first four characters.

The estimate of the time which would have been used by this algorithm during the test is 9.72 seconds (instead of 42.2). (Only an estimate is possible, as only the average number of characters compared is known, and its distribution is not.)

The disadvantage of the new solution is lack of opportunity to monitor about 99% of string comparison operations. Judging by the test results, the new *eq_string* procedure will be called rarely and all the monitoring counters can be installed only there.

6.4 Main interpreter loop.

The main interpreter loop is the most frequently executed part of the PS-algol interpreter. It performs the fetch-decode cycle. With the monitor installed, it counts also the number of times each instruction was executed. This is performed by the following 2 lines:

```
moncnta [(int) (*gcp)] ++; /*  increment instruction counter */
fns [ (int) (gcp++) ] ();  /* call a procedure to execute instruction */
```


The fetch-decode cycle costs 37.89 µs per execution, for the interpreter with basic monitor - 51.10 µs. Transformation of a current byte (instruction code) into an array index is done twice. Going to the assembly language level we can remove the second transformation and optimize the first - the new execution time is 41.43 µs. This gives the total execution time for main interpreter loop 228.1 seconds instead of 281.3 previously. The difference between 41.43 µs needed to execute the main interpreter loop now and 38.89 µs to execute main interpreter loop previously is (approximately) equal to the time used by the basic monitor. It may be estimated that the total time used by basic monitor is now about 20 seconds instead of 73 seconds.

6.5 The code pointer handling.

The global code pointer is the most frequently used register of the PS-machine. It is used in the fetch-decode cycle of the main interpreter loop. Many instructions are also using it to obtaing a parameter. The standard C construction used to obtain the next byte from the code stack and update its pointer is

```
tmp1 = (psint) (*gcp++);
```

which is translated into following machine code.

```
move.l   gcp_,a2
add.l    #1,gcp_
move.b   (a2),d3
and.l    #$ff,d3
move.l   d3,d4
```

There are two possibilities to optimize this code. In the first step we can substitute "clear register" instruction, instead of using the masking, and remove one interregister transfer by moving stack contents directly to the result register. This will result in the following code:

```
move.l   gcp_,a2
add.l    #1,gcp_
clr.l    d4
move.b   (a2),d4
```

The time to execute the original sequence is 17.79 µs, the modified one - 15.39.

So the improvement is a very small one - 2.4 microseconds. It was implemented in the main interpreter loop. But its implementation in all instructions of the interpreter may bring improvement of only (about) 10 seconds - which does not seem worthwhile. (The estimate is, that while 5.5 mln instructions are executed and 80% of them used this sequence the resulting gain in speed will be too small to justify such an troublesome change).

The second method of speeding up this operation is to have pointer to the global code stack in the processor register all the time. This would have saved the need to move it from memory to the register each time. Also, instead of updating in memory (by ADDQ instruction of M68k) postincrementation could be used while getting the next byte from the stack. The code would be as follows:

```
clr.l    d4
move.b   (a2)+,d4
```

The time needed to execute this code is 7.36 µs. This code would have been used during the test about 10 mln times (5.5 in the main loop, 4.5 executiong instructions) - and the resulting gain would have been abount 80 - 100 seconds.

Such an optimization, however, would have required extensive changes in the interpreter.

6.6. The "case" solution

The interpreter wersion monitored uses a subroutine call to execute a PS-machine instruction. The overhead, while using a case solution, is smaller (see 2.4). And the resulting speed-up would have been 91 seconds for the test. The case solution is commonly used for interpreters in other PS-algol implementations, so this should not be considered as an optimization comparable with the ones described earlier.

6.7 Processing jumps

Forward jump instructions are 3-bytes long, one byte for an instruction code, 2 bytes for the jump length. During the test 99% of forward jumps processed was for lengths shorter than 256 bytes - i.e. they could have been coded in one byte. This would have reduced the dynamic code size for processing forward jump instructions from 3 to 2 mb. Total dynamic code size of the test program was 12mb (static: 30kb), so the reduction in the dynamic code size would have been small but significant. Such optimization would have also resulted in time savings, because only one byte would have been required as a parameter - see 6.2 for timings.

Further possible optimization along these lines is to introduce 'a very short jump instructions' - 25% of all used jump lengths may be coded in 2 bits, 56% in 3 bits. This would have reduce the dynamic code size by further 0.5 mb - although at a cost of significantly changed instruction set.

One kind of optimization which does not require any changes in the virtual machine is different coding of conditional jump instructions. With the present coding an instruction takes jump length (two bytes) from code area, evaluates condition and updates (or - more frequently - not) code pointer. The coding in which condition is evaluated first and *if true* the jump length is obtained and program counter updated (otherwise program counter updated by the length of a parameter) would have required in worst case the same time. And less memory acceses for jumps not executed.

6.8 Optimization possibilities - summary

The optimization possibilities described in 6.1..6.4 total (in test case) to about 216 seconds and they are easy to implement.

In the test case, of 1249 seconds total time, about 350 were spend on i/o. The remaining 900 seconds may be considered to be spend executing processor-bound instructions. If the "case version" of the interpreter would have been used, the time to execute processor-bound instructions would have been abount 810 seconds. Therefore it may be estimated that the optimization described in sections 6.1 to 6.4 improve the overall processor speed of the PS-algol machine by 27%. (These optimizations are not version-dependent.)

The optimization described in 6.5, if implemented, would have improved the speed by further 10%.

Section Two

Monitoring the PS-algol virtual machine

VAX implementation

1. Introduction

This text describes a continuation of work on monitoring the PS-algol virtual machine, started by monitoring the Mac implemetation and described in a separate report. The monitored interpreter of the PS-algol machine used for the Mac version of the PS-algol was transferred to the VAX.

Because of the machine architecture and the peculiarities of the UNIX clock, time measurements done extensively on the Macintosh version were not made on the VAX. The VAX has a cache memory - and this means that an off-line timing method cannot be used, as the execution time of any PS instruction is not the same on each call and depends on cache administration algoritm. This obstacle could be partially overcome during an off-line timing by filling in the cache memory with (say) an array of numbers before and after an execution of each timed PS-instruction. In this way the speed-up effects of the cache memory would have been nullified. However, they are present during real execution of PS programs and their effect is difficult to estimate.

Unix being a multiprogramming system executes several processes at the same time. The process time logging is done in an approximate way, namely at the end of each 1/60th of a second it is assumed that the process active at this moment was active for the last whole 1/60th of a second and such a value is added to the process time. However, a processor scheduling algorithm allows the processor assignment to the process at any moment during such interval. The total effect of these two factors can be seen during off-line timing of some PS instructions: their measured execution times were accurate only to 2 significant digits regardless of the number of iterations (while on the Mac accuracy was a simple function of the number of iterations).

But the VAX implementation has some bonuses. Monitoring anything on the Mac severely restricted the range of tests, because of memory limitations. In fact, on a 512k Mac full compilation of the PS-algol compiler by the PS-algol compiler was impossible and some errors had to be introduced into the text, reducing the memory requirements. Although the test obtained was more similar to an average user program (they more often than not contain errors) it can hardly be maintained to be a satisfactory solution. In that aspect there were no such limitations while monitoring the VAX version - there was plenty of memory available. This enabled to perform not only the kind of monitoring done on the Mac, but also extend it to measure PS-machine behavior in detail.

2. The VAX implementation monitoring

The same version of an interpreter and monitor was used as for the Mac version - each PS-machine instruction was executed by a separate C procedure. Some additional counters were added to obtain more details about machine behavior. Heap traffic and garbage collection were monitored in greater detail.

Two tests were used. The first one consisted of running on a monitored PS-algol machine the PS-algol compiler (a PS-algol program in itself) which was compiling a PS-algol program. The second test consisted on running on a monitored PS-algol machine the S-algol compiler (an S-algol program in itself) which was compiling an S-algol program. These tests will be further referred to as the **PS-test** and **S-test**, respectively. In both cases the program to compile was a PS-algol compiler. Because standard PS-algol compiler is written in S-algol it could not be used for PS-test. The second version of PS-algol compiler, this time in PS-algol, was created. This version was as similar as possible to the original and functionally identical. In this way, a comparison of the results received during the S-test and the PS-test is, to some extend, also a comparison of these two languages.

From the point of view of the PS-algol machine, on which they are processed, the programs differ only in one area - heap load. S-algol does not have first class procedures therefore, upon procedure return, the space on the heap used by a procedure can be reclaimed immediately. In PS-algol any procedure containing a block stays on the heap till the garbage collector reclaims the space used by it.

In general, the PS-machine interpreter installed in the system tries to grab as much memory as possible to satisfy the heap requests. In the monitored version of the interpreter heap size was preset to be 1 megabyte during program execution and all the results are for this version - unless specified otherwise.

Additional counters were installed in the interpreter to obtain distribution of such characteristics as number and type of parameters on procedure call, procedure memory requirements, location, age and type of heap items. To accomodate these counters the monitor table was expanded to 8 kilobytes. These changes resulted also in a singnificant increase in processor usage.

The monitored version of the PS-algol interpreter was run as a separate program, with its own data base. THis ensured that the operation of the interpreter was independent of any system database activity.

3. General results

During the S-test 13,311,308 PS-machine instructions were interpreted. Total time used was 1550 seconds (with 'unix accuracy' -+ 50 seconds).During the PS-test the number of instructions interpreted was 14,067,987 - 5.7% more than in the S-test . Time used: 1700 seconds (again with +- 50 seconds accuracy). The above timings are for tests during which the distribution-registering facility of the PS-algol machine monitor was turned off. With that mechanism working, the times received are about 1000 seconds bigger.

It must be pointed out once more, that the timings quoted are very approximate. While running indentical tests several times user process time as returned by the UNIX system varied by a factor of 2 - which shows once more how unreliable time measurements.are on this version of UNIX.

It is worth noting, that the same operation as performed by the S-test while performed by an S-compiler installed in the system takes 150 seconds - but this compiler is running in a native mode, not on an interpreter. To perform the same task as the PS-test, the PS-algol compiler installed in the system (and running in a native mode, not on an interpreter) used 210 seconds. Further results suggest strongly, that for the system compilers this 60 second difference in speed is largely due to the garbage collection. As for any comparisons of speeds of system and interpreted compilers we shuld take into account that the monitored interpreter is not the fastest version of the PS-algol interpreter and, additionally, cost of monitoring was in this case about 10% of test's time.

Of 237 PS-machine instructions 127 were used. The frequency of PS-machine instruction usage on all 3 tests (Mac, S-test, PS-test) was similar and its distribution nearly identical (despite the fact that on the Mac test about 5 milion instructions were processed). The diagram *'frequencies of most frequently executed instructions'* shows that for each test only first 25 of most frequently executed instructions had execution frequency over 1%. The diagram *'cumulative percentages of instruction executions'* shows also very similar distribution for all tests - with first 35-37 most frequently executed instructions comprising 90% off all instructions executed.

4. Jumps

In all, jump instructions comprised 22.97 of all processed PS-instructions. Their detailed characteristics looks as follows

	count	% all jumps
unconditional jumps forward	433699	13.42
conditional jumps forward	2648901	81.96
unconditional jumps backward	147075	4.55
conditional jumps backward	2179	0.07
conditional jumps forward executed	1540058	58.14 % of cdf
conditional jumps backward executed	724	33.23 % of cdb

distribution of forward jumps length:
28.83 % - encoding in 2 bits
43.50 % - encoding in 3 bits
96.18 % - encoding in 7 bits
98.92 % - encoding in 8 bits

distibution of backward jumps length
0.00 % - encoding in 3 bits
64.11 % - encoding in 7 bits
96.05 % - encoding in 8 bits

Practically all forward jump instructions processed are for jumps shorter than 256 - which means that they may be coded in one byte, instead of 2. Such encoding would have required significant changes in the compiler itself or a postprocessor to convert intermediate code appropriately.

The size of the code processed by the interpreter while processing jump forward instructions was about 9 megabytes, and one-byte encoding of shorter jumps would have saved processing of 3mb. It may be estimated that, similarly like for the Mac version, the total dynamic code size of the executed program would have been reduced by this about 10%.

For forward jumps there is also a significant percentage of jump instructions which lengths may be coded in 2 or 3 bits. This points to the possibility of using 'very short jump' instructions with opcode directly specyfying jump length. The reduction in the dynamic code size would not be as high as in the previous case, but still significant.

If any such changes are implemented, the percentage of short jumps should increase, because their presence in a static code will lead to the overall shortening of it and some jumps presently over the threshold will became "short".

Of all conditional jump forward instructions processed, 58% had really performed a jump. For backward jump instructions this percentage was 33%. Interpretation of a conditional jump instruction is performed in three steps: first the jump length is obtained fromthe code area, then a condition is evaluated and (if true) the program counter is updated. The percentages quoted point out that, to obtain slightly faster execution of such instructions, they should be coded to evaluate the condition first, and update the program counter later. In this way, if a condition if false, no time will be wasted to obtain the jump length from the code area With this, the amount of processing will be indentical for jumps executed, but for the jumps with a condition *false* no time will be wasted to obtain jump length - which will save some memory accesed.

5. Procedures.

The PS-instruction 'procedure call' (APPLY_OP) comprised 4% of all instructions executed during the test. 12.2% of all procedure calls were for standard functions. The remaining 87.75 % were calls for user-defined procedures.

Due to the PS-algol machine architecture a procedure may be characterized by 4 characteristics. The first two deal with the number of scalar and pointer parameters a procedure has been called with. The other two describe how much space on the main stack and on the pointer stack a procedure requires.

frequency of procedure parameters on call

pars	total	%	main	%	pointer	%
0	10154	21.74	259631	55.57	284196	60.83
1	221094	47.33	171184	36.6	81968	17.55
2	126663	27.11	27581	5.90	101015	21.62
3	9134	1.96	8690	1.86	0	0.00
4	8741	1.87	93	0.02	0	0.00
5	0	0.00	0	0.00	0	0.00

The following table lists the distribution of the space required by procedures on both stacks. The link words, 2 for main stack and 5 for pointer stack, are not included. The results are as received for the S-test.

procedure stack requirements - the S-test

space	total	%	main	%	pointer	%
0	0	0.00	716	0.15	12648	2.71
1	3	0.00	192517	41.21	31898	6.83
2	1890	0.40	34175	7.32	88880	19.02
3	12273	2.63	119139	25.50	102994	22.05
4	92745	19.85	85136	18.22	22489	4.81
5	50267	10.76	16961	3.63	54491	11.66
6	104766	22.43	6344	1.36	58690	12.56
7	48113	10.30	11655	2.49	66755	14.29
8	77488	16.59	16	0.00	18379	3.93
9	40538	8.68	7	0.00	2240	0.48
10	2432	0.52	197	0.04	280	0.06

For the PS-test the results received were slightly different - they are listed below.

procedure stack requirements - the PS-test

space	total	%	main	%	pointer	%
0	0	0.00	727	0.15	103943	21.41
1	6048	1.25	231117	47.60	37986	7.82
2	31874	6.57	55766	11.49	41442	8.54
3	27145	5.59	92096	18.97	77450	15.95
4	151168	31.14	94199	19.40	24478	5.04
5	48970	10.09	2199	0.45	64873	13.36
6	29732	6.12	4483	0.92	55916	11.52
7	69613	14.34	4329	0.89	71246	14.67
8	73687	15.18	0	0.00	324	0.07
9	25782	5.31	7	0.00	1	0.00
10	283	0.06	245	0.05	196	0.04

In general, it can be concluded, the that total stack space requirement is usually between 4 and 8 words. The procedure call mechanism employed by PS-algol machine adds to this 11 words for administration. This means, that there is a significant memory overhead in procedure call.

6. Strings

An operation frequently performed by the PS-machine during the test was string comparison. PS-instructions which execution involves string comparison are calling various procedures passing them strings' pointers and receiving a boolean result. Practically the only such procedure really used is procedere eq_string, which was called 790390 times (5.9% of all executed instructions tested strings for equality). The other function used was lt_string, called 58971 times (0.04 % of all instructions).

The eq_string procedure is called with two paramaters - strings' pointers - and returns a boolean result of string comparison. The algorithm it uses is:

1. if string pointers identical return true
2. if any of them 0 return false
3. compute string lengths
4. if differ return false
5. organize while loop, check strings char-by-char

Some counters were installed in this procedure to obtain detailed characteristics of string comparison.

7.45% of comparisons returned a value after trivial checks, before reaching step 3 of the algorithm. 56.88% comparisons returned after step 4, because of different lengts of compared strings. At this stage average string length was 10.16 characters.

This left 281,958 string comparisons (35.67% of eq_string function calls) to be performed by the while loop. At this stage the average lenght of strings to compare was 1.41 characters and, on average, 1.25 characters were compared before obtaining the result.

Of 281,959 string comparisons using the *while* loop 91.73% were comparisons of strings length one (i.e single characters). In all, 96.45% of strings compared by the *while* loop were of length 4 or less, 2.90% of length 5..8.

These results mean that the string comparison comprises significant percentage of the PS-machine activities. They suggest also that another approach should be taken to optimize string comparisons. To reduce an overhead arising from a procedure call, instructions calling the eq_string procedure should do some preprocessing in-place. If this preprocessing would have consisted only of checking string lengths, it would have reduced the number of calls to the eq_string procedure by 64%.

Another possible optimization involves checking the equality of strings done by step 5 of the string comparison algorithm. Because most of the compared strings are of length one their first characters should be compared before entering the *while* loop - in over 90% of cases there will be no need to orgranize and enter the while loop.

The above is true also for many other PS-algol-like machines. In case of the PS-algol this may be further improved by taking into account a string representation. It starts with a word containing its length and followed by words containg string characters **padded to a word boundary**. This means that for strings we can compare words, not characters. It takes the same time to compare 2 words, four characters per word, as to compare 2 characters (on computers with 32-bit memory bus). And comparison of the first word of strings is equivalent to comparing four characters at once - which would have taken care of 96.45 of all string comparisons involving presently while loop.

If the basic preprocessing of string comparison would have included length check and comparison of the first words the eq_string function would have been called in about **1.3%** of cases. The C macro implementing this was described in the Macintosh version report. The results obtained in the Macintosh test had shown similar predominance of strings length one.

7. Heap traffic

There are significant differences between heap traffic in the S-test and in the PS-test - they are mostly caused by the first class procedures in PS-algol. The S-algol procedures, compiled to run on PS-algol machine are using special return instructions to return the space on heap dynamically. PS-algol procedures can do this only if no block was declared inside them. The net result of this is seen in an increased usage of the heap.

	thePS-test	the S-test
total space in kilobytes	46466	36716
items	648229	487628
size in words	10598987	8424075

note: the size, if specified in words, does not include 2 words per item used for the heap administration.

The differences included also the type of heap items (the types of heap items here defined as they originate in the PS-machine - by which procedure creates them):

types of heap items for the S-test

item type	items	%items	size	% size	av size
procedure	467179	95.81	8276114	98.24	17.72
block	26	0.01	545	0.01	20.96
procedure vec	0	0.00	0	0.00	0.00
ib vec	4	0.00	12019	0.14	3004.75
real vec	0	0.00	0	0.00	0.00
code vec	0	0.00	0	0.00	0.00
string vec	0	0.00	0	0.00	0.00
cstrc vex	0	0.00	0	0.00	0.00
cfile vec	7	0.00	924	0.01	132.00
fram vec	4	0.00	1015	0.01	253.75
mkstrn vec	9080	1.86	21645	0.26	2.38
f_str_2	5976	1.23	27740	0.33	4.64
f_str_4	0	0.00	0	0.00	0.00
makev_ib_2	16	0.00	80	0.00	5.00
makev_r_2	0	0.00	0	0.00	0.00
makev_p_2	1	0.00	252	0.00	252.00
makev_pr_2	0	0.00	0	0.00	0.00
fortest	4122	0.85	68368	0.81	16.59
cvec_pntrs	4	0.00	331	0.00	82.75
pushframe	0	0.00	0	0.00	0.00
readr	0	0.00	0	0.00	0.00
writer	0	0.00	0	0.00	0.00
ll_n_pr	12	0.00	60	0.00	5.00
err_recs	1	0.00	8	0.00	8.00
imdesc	0	0.00	0	0.00	0.00

types of heap items for the PS-test

item type	items	pct items	size	pct size	av size
procedure	485495	74.90	8118393	76.60	16.72
block	142195	21.94	2330648	21.99	16.39
procedure vec	0	0.00	0	0.00	0.00
ib vec	4	0.00	12019	0.11	3004.75
real vec	0	0.00	0	0.00	0.00
code vec	0	0.00	0	0.00	0.00
string vec	0	0.00	0	0.00	0.00
cstrc vex	0	0.00	0	0.00	0.00
cfile vec	7	0.00	924	0.01	132.00
cfram vec	4	0.00	995	0.01	248.75
mkstrn vec	9314	1.44	21723	0.20	2.33
f_str_2	5837	0.90	27424	0.26	4.70
f_str_4	0	0.00	0	0.00	0.00
makev_ib_2	16	0.00	80	0.00	5.00
makev_r_2	0	0.00	0	0.00	0.00
makev_p_2	1	0.00	219	0.00	219.00
makev_pr_2	0	0.00	0	0.00	0.00
fortest	4127	0.64	70768	0.67	17.15
cvec_pntrs	4	0.00	331	0.00	82.75
pushframe	0	0.00	0	0.00	0.00
readr	0	0.00	0	0.00	0.00
writer	0	0.00	0	0.00	0.00
ll_n_pr	13	0.00	65	0.00	5.00
err_recs	1	0.00	8	0.00	8.00
imdesc	0	0.00	0	0.00	0.00

The above type statistics does not include heap items created during an initialization phase. The difference between the S-test and the PS-test in a heap items type frequency is caused by a significant number of blocks on heap in the PS-test .

The basic difference between the S-test and the PS-test is lack of the first class procedures in S-algol. Therefore, S-algol programs executing on the PS-algol virtual machine can use special return instructions which reset the heap top pointer to the value it had before a subroutine was entered. In this way space on the heap used by an S-algol procedure usually does not have to be reclaimed by the garbage collector. For PS-algol programs only procedures which does not have a block inside them can return the heap space this way. The table below summarizes an extent to which dynamic returns differentiate between execution of this tests

dynamic returns

	the S-test	the PS-test
% of heap items	87.72	50.09
% of heap space returned	88.94	50.87
% of procedures	91.56	66.87
% of procedure space	90.53	66.42

Because for the S-test almost 90% of the heap spece were returned dynamically and for the PS test only 50%, the remaining space had to be reclaimed by the garbage collector - resulting in higher garbage collection frequency for the PS-test .

It is worth pointing out that big percentage of heap space is used for administration purposes. An average item size in words is (the S-test) 17.26. To this 2 words are added for the heap administration. But an average item is a procedure - and of this 17 words 11 are used for links. So, from 19 words for an item on the heap 13 (70%) are used for administration.

8. Garbage collection

Memory available for the heap was limited to 1 megabyte and kept constant during tests. The main characteristics received were as follows:

	the S-test	the PS-test
no of collections	**5**	**31**
time	14.7	87.7
time/collection	2.94	2.83
% of active space	22.29	21.93
items encountered	66,653	406,812
items per collection	13,306	13,113
mark time per collection	0.42	0.37
pass1 time/collection	1.43	1.41
pass2 time/collection	1.11	1.04

So, the difference in dynamic returns resulted in six times as many garbage collections in the PS-test as in the S-test . All other main characteristics were similarly changed. Active heap space is defined as a ratio of occupied heap space after and before garbage collection. It was identical in both cases, which is a further proof that first class procedures tend to keep heap space for no reason at all. Looking at values of the garbage collection characteristics per number of garbage collections it can be concluded, that garbage collections were practically identical for both tests - the only difference being that for the PS-test six times as many garbage collections were required to perform identical task - the compilation of the PS-algol compiler.

To obtain more details about the garbage collection some additional counters were installed. The counters are updated while garbage collection procedures are performing their job. This means that the number and distribution of items counted in this way will be different than in heap traffic analysis. There each item was counted once (during creation) while here any item may be counted many times - if it lives through subsequent garbage collections.

8.1 Type of item and its activity

To find out which item types are active in the heap, types of active and inactive items were counted during the garbage collection.

items type and activity

the PS-test

type	% active items	% total	%inactive	%total
string	**22.17**	5.06	2.99	2.30
file	0.20	0.05	0.00	0.00
structure	**72.03**	16.46	0.98	0.76
pntr vec	1.87	0.43	0.00	0.00
clos vec	0.40	0.09	0.00	0.00
ib vec	0.20	0.05	0.01	0.00
real vec	0.00	0.00	0.00	0.00
frame	1.30	0.30	96.02	74.08
code vec	1.83	0.42	0.00	0.00
image	0.00	0.00	0.00	0.00

the S-test

type	% active items	% total	%inactive	%total
string	22.37	5.26	17.70	13.54
file	0.19	0.05	0.01	0.01
structure	72.43	17.03	6.25	4.78
pntr vec	1.79	0.42	0.01	0.01
clos vec	0.38	0.09	0.00	0.00
ib vec	0.19	0.05	0.04	0.03
real vec	0.00	0.00	0.00	0.00
frame	0.89	0.21	75.97	58.11
code vec	1.76	0.41	0.02	0.02
image	0.00	0.00	0.00	0.00

From the above data it can be concluded that items active consists predominantly of structures and strings and inactive items are procedures.

A coefficient describing activity of any given type of items may be constructed by taking a ratio of frequencies an item of this type may be encountered among active items and among all items. For example for the S-test strings comprised 18.8% of all items and 22.37% of active items - so the coefficient is 1.19. Structures, while comprising 22% of items encountered during the garbage collection found to be active items in 72% of cases - 3.3 times over their share if uniform distribution would have been assumed.

activity coeff	the S-test	the PS-test
string	1.19	3.0
file	3.3	4.0
structure	3.3	4.2
pntr vec	4.3	4.4
closure vec	4.2	4.4
ib vec	2.4	4.0
frame	0.015	0.017
code vec	4.2	4.4

From the above coefficients we can see that, upon encoutering a structure while garbage collecting, its probability of being active is like 3.3:1, while for a frame (procedure) like 0.015:1 (for the S-test).

8.2 Items activity and age

For each item on heap there are 2 bits unused in its header. This two bits were used to mark the time spend by an item on the heap. During each garbage collection one is added to this bits if the item is active. If both bits are set (value = 3) no furter updating takes place. In this way it is possible to find out (with obvious limitations) how much time items are spending on heap (how many garbage collections they have lived through).

age of active and inactive objects

	the PS-test		the PS-test without first 3 garb.colls	
age	active items	inactive items	active items	inactive items
0	4932	312628	3406	279289
1	3631	1234	2505	1166
2	3617	1	2949	1
3	80763	6	80873	6

	the S-test		the S-test without first 3 garb.colls	
age	active items	inactive items	active items	inactive items
0	3793	50886	370	19706
1	3534	100	681	50
2	3336	0	1185	0
3	5004	0	5004	0

From the above data it is obvious that in the PS test items that were active had, at the same time, high probability of staying on heap for a longer time, while inactive items were created recently, between subsequent garbage collections. In case of the S-test the tendency of active data to stay on heap is not so strongly marked - only 5 garbage collections took place and there was not enough many of them for the active items to reach age 3. (After first 2 garbage collection no item can reach age 3, so for the S-test only 3 garbage collections were ageing items to age 3, while during the PS-test - 29) Even with this, it shows clearly that inactive items were practically always the ones created recently.

To find out the picture without the distortion introduced by the first three garbage collections another test was performed without counting active and inactive items during the first three garbage collections. Comparison of results shows beyond any doubt - even for the S-test , with only two collections taken into account - that for an item once active there is a high probability of living through subsequent garbage collections . At the same time items inactive during a given garbage collection were created recently - before this and the previous one.

8.3 Active items, their type and age

Using mechanisms installed so far in the PS-algol interpreter we can find out dependecies between age and type of active items. Similarly like in the previous cases age is limited to 4 values, all items living through more than three succesive garbage collections reaching age 3 - and 3 is ageing limit. To remove boundary affects additional test was also performed, in which during first three garbage collections no counting of items age took place. The tables contain results of 4 tests: 2 for the S-test s (with and without boundary effects) and two for the PS-test .

age of active items by types

the PS-test

age	0	1	2	3
string	1092	791	788	17930
file	7	6	6	168
structure	2978	2689	2678	58606
vec of pntrs	56	56	56	1568
vec of procs	12	12	12	336
vec of int or bool	7	6	6	168
vec of reals	0	0	0	0
frame	725	16	16	447
code vector	55	55	55	1540
image	0	0	0	0

the PS-test without first three garbage colletions

string	779	593	682	17930
file	0	0	0	168
structure	1963	1912	2266	58606
vec of ptrs	0	0	0	1568
vec of procs	0	0	0	336
vec of int or bool	1	0	0	168
vec of reals	0	0	0	0
frame	663	0	1	447
code vector	0	0	0	1540
image	0	0	0	0

the S-test

age	0	1	2	3
string	839	771	739	1156
file	6	6	6	12
structure	2744	2612	2446	3546
vec of pntrs	56	56	56	112
vec of procs	12	12	12	24
vec of int or bool	6	6	6	12
vec of reals	0	0	0	0
frame	75	16	16	32
code vector	55	55	55	110
image	0	0	0	0

the S-test without first three garbage colletions

age	0	1	2	3
string	70	133	221	1156
file	0	0	0	12
structure	279	548	964	3546
vec of pntrs	0	0	0	112
vec of procs	0	0	0	24
vec of int or bool	0	0	0	12
vec of reals	0	0	0	0
frame	21	0	0	32
code vector	0	0	0	110
image	0	0	0	0

8.4 Position of active items on heap

While monitoring information about location on heap of active items was gathered. Heap space was divided into 16 segments and a number of active items in each segment was counted. Segment number 0 was at the bottom of the heap space, segment number 15 at the top

location and no of items

	the PS-test	the S-test
0	837	135
1	5116	825
2	48375	6842
3	34214	4597
4	223	253
5	277	227
6	265	300
7	330	286
8	294	318
9	329	158
10	335	108
11	308	288
12	204	329
13	347	341
14	498	332
15	991	328

Each of 16 heap segments represents 6.25% of heap space. Remembering that for both tests the active heap space was 22% it is easy to explain why so many active items are grouping in lower 4 segments. The garbage collection slided active items to lower regions and, because if the item is active it is at the same time long-living, they stayed there for a long time. This tendency is less visible in the S-test , due to the smaller number of garbage collections performed and bigger masking effects of inactive item distribution during the first garbage collection, when - lacking any history - item distribution was probably more homogenous.

8.5 Heap size and garbage collection.

To find out how garbage collection characteristics depend on heap size the S-test and the PS-test were executed with different (but constant during the execution of a test) heap sizes. Table *'heap size and garbage collection'* summarizes the results. The times of test's execution and garbage collection shows extend of vagaries of Unix user time clock - running the test for heap size 10x128kb times user process time received varied from 1800 to 5600 seconds with all other characteristics unchanged.

9. Summary and disclaimers

This project proves that it is possible to design and implement a monitor under rather adverse conditions and have it running without paying much (in terms of computer resources) for its use. Information received is of such quality that enables to spot some tuning possibilities in an extremely well tuned machine - after all this virtual machine is in use for some years and close contact with it enabled me to appreciate the degree of tuning done so far.

However, after seeing the results obtained so far some misconceptions may arise and, to avoid this, some warnings are in order.

The results received reflect as much virtual machine performance as the specific test being used, C-compiler quality, underlying micropocessor and system architecture - therefore at each step we should be aware which is which. But this does not mean that the results should be discarded as beeing too specific - or taken wholly in good faith because all these factors should balance themselves in a long run. Simply running a several additional programs to have 'another test' would not have helped also - benchmarking and benchmark programs are - at best - tricky things. The only good test is to run some real applications - and they are (almost by definition) very application dependent and rather scarce at this moment.

The monitor performance is satisfactory - but one should keep in mind that the numbers describing its time requirements do not include measuring **distributions** (of jump lengths, heap item sizes and so on). Analysis of distributions is costly (up to 50% of test's time) and cannot be done in a routine way. Incremeting counters (especially by 1) is cheap and such counters will give us averages - in many cases enough, but not always satisfying.

Costs of interpretating the code. Although the main interpreter loop (performing the fetch-decode cycle for instructions) used only 17% of test's time, in fact main costs of interpeting are spread (hidden) among instructions executions. Because the interpreter is written in C no explicit control of register assignments is possible - which is usually a key factor to any speed up. The *'register'* declarations' in C code do not change the fact that on each instruction entry the virtual machine registers are loaded into the processor registers. And, because most of the instructions are short ones, this overhead is at least as big as fetch-decode loop - to say nothing of the fact that next instruction interpreted does not know what the last one had left in registers.

On the other hand, this virtual machine works on many different architectures and close binding to any one of them would have been counterproductive - and lack of portability can cost heavily in software engineering.

10. Acknowledgments.

My sincere thanks go to Dr. John Hurst from the Australian National University and Prof. Ron Morrison from St. Andrews University for organizing my stay in Scotland. I would like to thank also all members of the PISA group, especially Alfred Brown and Alan Dearle for patient and numerous explanations about the intricacies of the PS-algol system and the PS-algol machine. Thanks to Dr. John Hurst this report's final appearance is much better than the submitted one. Blame for any remaining errors rests on me - only.

References

PPR-11-85 PS-algol Abstract Machine Manual

PPR-12-85 PS-algol Reference Manual

Kane, G., Hawkins, D., and Leventhal, L.

68000 Assembly Language Programming

OSBORNE/McGraw-Hill, 1981

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

or

The Secretary,
Persistent Programming Research Group,
Department of Computational Science,
University of St. Andrews,
North Haugh,
St. Andrews KY16 9SS
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", Pergammon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

Cole, A.J. & Morrison, R.
"An introduction to programming with S-algol", Cambridge University Press, Cambridge, England, 1982.

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages". Software, Practice and Experience 7, 5 (July 1977), 535-537.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", Proc. of the DECUS conference, Bath (April 1978), 5-10.

Atkinson, M.P.
"A note on the application of differential files to computer aided design", ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

Atkinson, M.P.
"Progress in documentation: Database management systems in library automation and information retrieval", Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", Information Processing Letters 9, 1 (July 1979), 1-4.

Atkinson, M.P.
"Data management for interactive graphics", Proceedings of the Infotech State of the Art Conference, October 1979. Available as EUCS departmental report CSR-51-80.

Atkinson, M.P. (ed.)
"Data design", Infotech State of the Art Report, Series 7, No.4, May 1980.

Morrison, R.
"Low cost computer graphics for micro computers", Software Practice and Experience, 12, 1981, 767-776.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in Database, Pergammon Infotech State of the Art Report, Series 9, No.8, 299-318 (January 1982) - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", Computer Bulletin II/31 (March 1982).

Morrison, R.
"The string as a simple data type", Sigplan Notices, Vol.17,3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure", see PPRR-8-84.

Morrison, R.
"Towards simpler programming languages: S-algol", IUCC Bulletin 4, 3 (October 1982), 130-133.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University of Pennsylvania. October 1982. Circulated (revised) in the Workshop proceedings 1983, see PPRR-2-83.

Atkinson, M.P.
"Data management", in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3 (March 1983), 273-285. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983, Vol.26, No.4, 360-365 - see PPRR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983, 70-79 - see PPRR-2-83.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D. Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", proceedings of the third British National Conference on Databases, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, (July 1984).

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985) - see PPRR-9-84.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", 8th International Conference on Software Engineering, Imperial College, London (August 1985), 166-172 - see PPRR-15-85.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24 - see PPRR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 278-283 - see PPRR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117 - see PPRR-16-85.

Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303 - see PPRR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383 - see PPRR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 399-410 - see PPRR-16-85.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438 - see PPRR-16-85.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, Vol.14, No.3, (1986) - see PPRR-10-84.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", proceedings of the 19th Annual Hawaii International Conference on System Sciences, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA - see PPRR-19-85.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", proceedings of the 9th Australian Computing Science Conference, January, 1986 - see PPRR-21-85.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages"; proceedings ACM SIGMOD Conference 1986, Washington, USA May 1986 - see PPRR-22-86.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 147-157 - see PPRR-14-86.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", proceedings of Information Processing 1986, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics", University Computing, Vol. 8, N0. 2, (Summer 1986) - see PPRR-23-86.

Internal Reports

Morrison, R.
"S-Algol language reference manual", University of St Andrews CS-79-1, 1979.

Bailey, P.J., Maritz, P. & Morrison, R.
"The S-algol abstract machine", University of St Andrews CS-80-2, 1980.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
"EDQUSE reference manual", Department of Computer Science, University of Edinburgh, September 1981.

Hepp, P.E. and Norrie, M.C.
"RAQUEL : User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

Theses

The following Ph.D. theses have been produced by members of the group and are available from the address already given,

W.P. Cockshott
Orthogonal Persistence, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those produced and those planned plus their status at 15th December 1986. Copies of documents in this list may be obtained by writing to the addresses already given.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., Chisholm, K.J. and Cockshott, W.P.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-4-83	The PS-algol reference manual - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R	Presently no longer available
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P.,Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00
PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-86	PS-algol Reference Manual - third edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - second edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for an Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00
PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-86	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00
PPRR-26-86	Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P.	£1.00
PPRR-27-87	A Context Sensitive Addressing Model - Hurst, A.J.	£1.00
PPRR-28-86b	A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A.	£1.00
PPRR-29-86	A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P	£1.00
PPRR-30-86	Data Types for Data Base Programming - Buneman, O.P.	£1.00
PPRR-31-86	An Introduction to PS-algol Programming - Carrick, R., Cole, A.J. & Morrison, R.	£1.00
PPRR-32-87	Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment - Morrison, R, Brown, A, Connor, R and Dearle, A	£1.00
PPRR-33-87	Safe Browsing in a Strongly Typed Persistent Environment - Dearle, A and Brown, A.L.	£1.00
PPRR-34-87	Binding Issues in Database Programming - Atkinson, M.P., Dearle, A., Cooper, R.L. and Abderrahmane, D.	£1.00
PPRR-35-87	A Persistent Architecture Intermediate Language - Dearle, A.	£1.00

PPRR-36-87	Persistent Information Architectures - Atkinson, M.P., Morrison R. & Pratten, G.D.	£1.00
PPRR-37-87	PS-algol Machine Monitoring - Loboz, Z.	£1.00