Tuning an Algebraic Manipulation System Through Measurements

Knut Bahr
Gesellschaft für Mathematik und Datenverarbeitung
Darmstadt, Germany

Jaap Smit
Twente University of Technology
Enschede, The Netherlands

Summary

This paper describes a measurement tool that has been used to analyze, tune, and redesign in part the PL/I-FORMAC Symbolic Mathematics Interpreter. In a number of examples, details are given both on the FORMAC system and on the application of the measurement tool. The basic tool, called invocation count measurement, is simple but quite effective. Its application enabled us to improve FORMAC considerably.

1. Measurement and Objectives

Measurement is essential to design, test, and improve a system. For algebraic manipulation systems, it was pointed out by Tobey [10] that studying the behavior of existing systems is a valuable resource and that a number of verification methods should and have been used to determine the behavior. Fitch and Garnett [4] describe a measurement tool that has successfully been used to investigate the inner workings of CAMAL. In this paper we shall present a different tool that has successfully been used to screen and tune FORMAC.

Our objective was twofold: First, the existing PL/I-FORMAC system as released by IBM [9] was to be extended by new features [2] and the effectiveness of the implementation was to be monitored. Secondly, FORMAC was to be improved in its overall speed and measurement was to provide criteria. Different measurement tools have been used, e.g.

(i) run-time measurement of an entire program, program parts, or system modules;
(ii) measurement of usage percentages;
(iii) measurement of the invocation count of system modules;
(iv) a trace of list storage space in use;
(v) a check on unintentional garbage accumulation.

We want to focus attention on the invocation count measurement. It is a simple, but quite effective tool, that serves best to observe the performance of algorithms, for either of our objectives.

In applying it, we proceed from the premise that the system under study is well structured into modules. A module is a self-contained part of the whole system that performs a well-defined task in the overall logic, such that the task of a governing module (i.e. one at a higher level in the structure) can be equally well described in terms of the tasks performed by its submodules. Invoking a module means requesting that a certain task be accomplished. During a computation, the invocation count will furnish information on the activity of a task, i.e. it will tell if and how the task is actually executed. The efficacy of the invocation count is strong if the modules reflect significant logical steps, and it becomes weak if tasks are ill-defined and modules are arbitrary.

2. The Module Invocation Count

The invocation count in its simplest form emerges from the well-known technique of counting how often certain code sections are being executed; but it is applied to the logic rather than the code level. It is obtained by introducing conditional instructions at the entry to each module, for the purpose of incrementing a count in a given measurement table. The implementation is by a macro. The count reflects the number of times any given module has been invoked. The table may be set up for some or all modules and may be reset at any time during execution. Counting can be turned on or off, so that we are able to count invocations globally, selectively, or incrementally. The fact that FORMAC is embedded in a general-purpose host language, PL/I or FORTRAN, is of great convenience: All measurement control is naturally put in the same source text that makes up the algebraic computations program; there is no need to resort to language or other artifacts. Thus, output and management of the measurement table and the measurements are controlled through PL/I and if desired, depending on algebraic results, loop parameters, and other conditions. To examine for example the system performance during an algebraic computation consisting of initialization, iteration, and final

evaluation it would be possible to select measurement of the expansion activity in the innermost loop, the substitution activity during the entire iteration, or the re-simplification activity during the final evaluation.

3. The Augmented Invocation Count

The invocation count in its simplest form gives only a quantitative statement on the activity of a module. Therefore, it is augmented by tools that give a more qualitative information indicating purpose and effect of an activity. We have concentrated on the following questions:

a) Why is a module invoked or from what other module is it invoked?
b) What is the input and output to the module for any invocation?
c) Did the module actually perform a transformation?
d) What conditions did the module set for subsequent processes?
e) How much time did the module consume?

They also enable the investigator to specify conditions under which an activity is to be monitored. For example, he might be interested in the number of calls to the substitution module, but only if no substitutions are made, and in the time taken in relation to the total time.

Question (a) is aimed at the reasoning of the governing module(s), that is, what caused the system to activate a certain subtask. Some answer is obtained from a traceback. But the standard traceback gives it in terms of statement numbers or addresses, which is unsatisfactory because it relates to the relatively low level of coding rather than the higher structural and conceptual level [10]. What we would like to see is a trace of the logic that constitutes an algorithmic interpretation. We do not have a satisfactory logical traceback yet because the necessary provisions have not been built into the system when it was designed. Verification beyond the bit level can only take place if one can get a readout that is beyond the bit level.

For the purpose of point (b), a routine has been written that prints out an expression or part of it from anywhere in the middle of the system, together with some identification. Care has to be taken that temporary structures which may not (yet) represent a valid well-formed expression do not result in abnormal termination. This measurement tool proved useful e.g. in checking the scan across a chain of terms, watching the progress of a sequence of substitution passes, and verifying the return of an expression to the free list (erasure).

Question (c) is to examine more closely those modules whose main task is to perform a certain transformation (like $\sin(-x) \rightarrow -\sin(x)$ or automatic simplification in general). One way to test if a transformation took place is to compare input and output of the module. Disadvantages are:

(i) it is time consuming and expensive;
(ii) it means being able to recognize two (possibly temporarily non-canonic) structures as identical or equivalent;
(iii) different criteria have to be applied at different times; e.g. the generation of an exact duplicate may be considered as no transformation as far as substitution goes, but as a transformation as far as duplication is concerned;
(iv) it cannot discern identity transformations; take e.g. "substitute x for v in v+w", where x is a parameter that happens to be x=v.

A better way to test if a transformation took place is to test specific activity indicators. We assume that there is such an indicator, say a bit, that is set on or off by the module in question. In many cases in FORMAC indicators exist, for example there is a bit telling whether or not an expression has been simplified, a bit telling if re-ordering is necessary, or an indicator telling if multinomial expansion took place. Others can be introduced for measurement purposes (a priori). Measurement makes use of these indicators by entering their status, together with a count in the afore-mentioned measurement table. For example, in a particular computation of SIGSAM problem number 2 [3] there occurred 363 invocations of the automatic simplification package. Measurements showed that in 231 cases no simplification took place. In these cases, the input had already been simplified before (which the package determines afterwards). So, the system could be modified to invoke the simplification package only when necessary, i.e. 132 times.

Question (d) inquires about the control output of a module. There are modules in the system that perform a transformation in combination with some tests and others that perform a test only. The outcome of such tests will in general control further processes. The expression comparison module is a typical test module; its control output affects ordering. Of more interest are modules that either do not complete a transformation and signal incompleteness upon exit, or detect during transformation that situations have arisen which may require further transformations. For instance, if during combination of like terms in a sum or product new types of expressions are generated, a flag will be set to indicate that re-simplification is needed. Being able to monitor the setting of this flag revealed that many costly re-simplifications were demanded that did not introduce any change at all. A similar situation may occur with certain substitutions. Knowing about the control indicators is what enabled us to improve algorithms, as discussed in the following sections.

In point (e), the execution time for a module has to be judged in relation to the total execution time. Two methods have been used. One is to employ some sort of software monitor to establish a time profile and to find out the critical sections in the program, that is those that take an outstanding amount of time [5].

The other method is to integrate time and invocation count measurement, by taking the times at entry to and exit from a module and accumulating the differences for each invocation in a separate timing table. The table is processed in the same way the afore-mentioned measurement table is. Thus, we can set conditions, read intermediate times as desired, or stop when a specified maximum is reached.

Timing should only be used as a tool in connection with others. It is neither fine nor informative enough to help beyond a general impression. Given a reasonable modular structure, we found that the invocation count is a quite sensitive fine comb. Execution times tend to vary considerably from run to run due to timer resolution, interrupts, and background activity. A lapse in the code of one module may not be measurable in terms of time for that particular module (although it may slow down some other part or a particular application), but if it affects the invocation count - and it typically will - it cannot be overlooked, however small the difference.

4. Localizing Weak Points

To understand our measurements and modifications, a central feature of the FORMAC system has to be appreciated. It is the automatic simplification of expressions [8, 11]. The strucure of this subsystem, as far as of interest here, is given in Figure 1.

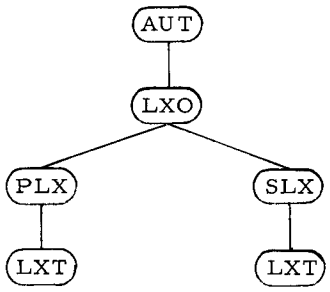

Figure 1

AUT is the general control module consisting of a set of transformation rules [9], e.g.

$A * 1 \rightarrow A$

$\sin(-A) \rightarrow -\sin(A)$

$A + A \rightarrow 2 * A$.

For sum and product chains, AUT calls upon the lexicographic ordering module, whose task it is to generate a canonically ordered chain. Its control module LXO decides in what way, say a new term is to be merged into an already partially sorted chain. The actual work is done by PLX and SLX for sum chains and product chains, respectively. Both PLX and SLX employ a module LXT for expression comparison to find out if a new term belongs before, after, or at another term.

The test example discussed here is the first version of a FORMAC solution to SIGSAM problem number 2 [3] as given in [1]. All one needs to know is that the problem generates fairly long sums; the complexity of the generation process as well as the length of the sums increase with increasing parameter M.

As a first step, a software monitor was used to establish a running profile for three values of M = 6, 7, 8. Since we wanted to break the profile down in terms of FORMAC modules, we had to write our own utility program that re-arranged the output which was given on the basis of program addresses. This was done in 1973 by T. Kobin at the Southern Illinois University. The profile is useful (i) to get a first idea on where the critical sections are and (ii) to check a presumably more efficient implementation. It is not so convenient to single out modules or groups of modules, and to time them only under certain boundary conditions.

	M = 6		M = 7		M = 8	
	old	mod	old	mod	old	mod
1. LXO	8.17	6.24	6.43	13.54	6.58	14.82
2. LXT	32.17	13.15	41.21	10.74	43.61	13.92
3. PLX	16.50	14.37	21.65	21.24	28.10	23.79
4. APB	2.67	4.59	2.36	5.25	1.72	6.01
5. AUT	8.50	12.84	5.16	11.44	3.99	12.62
6. ENT	2.00	5.81	1.76	4.43	1.30	3.71
7. INS	4.83	10.09	2.47	4.90	0.90	2.45
8. Rest	25.16	32.73	18.96	28.46	13.80	22.68
Σ23	48.67	27.52	62.86	31.98	71.71	37.71
Σ123	56.84	33.94	69.29	45.52	78.29	52.53
run time in sec.	13	6.4	43.4	14.2	131	32.2

Figure 2
Usage percentages and run time, Sigsam 2.1

Figure 2 lists the usage percentages for a few modules in columns under the heading "old". Σ123 gives the sum of 1., 2., and 3., i.e. the percentage for the whole lexicographic module. The peak is conspicuous: For M=8, 78% of the time is spent in this module, 44% is spent in LXT alone.

Our approach to finding the reasons is discussed in the next section, a fix is mentioned in [1]. We modified essentially the modules LXO and PLX to the effect that less activity is requested from PLX and LXT. The new percentage figures are found in columns under the heading "mod". We notice a considerable decrease in the share of LXT, as well as in the percentage of the whole lexicographic ordering module (Σ123). The usage is more evenly distributed than before.

To analyze the speed-up, we cite the run times given in the bottom row of Figure 2. The total runtime was decreased by a factor of 2 to 4, going from M=6 to 8. The activity of the two particular modules

we had aimed at, namely PLX and LXT (see Σ23), was actually reduced by a factor of 3.5 to 7.7. As a consequence, the rest of the system was speeded up about 50% to 80%. The total factor is obtained as a weighted sum.

5. Looking for Reasons

The observed peak in the LXT activity led to the first assumption that this module might be poorly coded. In addition, the opinion was frequently voiced that FORMAC was inherently slow due to its "unwieldy" internal data structure. LXT namely, compares two expressions by going down and across list structures which are essentially polish prefix representations with nodes containing explicit operators and operand pointers [9]. However, these hypotheses cannot be upheld, as we shall see.

Analysis of LXT alone does not take us much further. More informative data is obtained from the invocation count taken at this point (Table 1).

module	count
AUT	363
LXO	10663
PLX	20723
SLX	3957
LXT	43193

Table 1

module	count
AUT	363
LXO	10663
PLX	20720
SLX	3957
LXT	24677

Table 2

This and the following tables are given for the second version of a program to solve SIGSAM problem number 2, for M=7 [1], unless otherwise indicated.

Again there is a peak for LXT, but this time it signifies frequent invocation as a reason for high usage. That takes us one level higher to the question what other modules call upon LXT so heavily and why do they do so. With additional details, we can in this way carry the investigation higher and higher in the modular structure, up to the real cause(s). We shall see that in this particular example causes can be found at all levels PLX, LXO, AUT.

The measurement data exhibit a striking misbalance between the number of simplifications (AUT) and the amount of lexicographic activity. It becomes even worse if we recall that only 132 of the 363 AUT invocations are "real". Looking at the counts of other modules that are not given in Table 1, does not change the picture. The corollary that nearly all simplification activity is lexicographic, would support the thesis of an inherently slow FORMAC system. To refute the thesis, we have to show that the slowness is not inherent and thus can be reduced.

From the system structure is known that either PLX or SLX call upon LXT. The sum of their invocation count is 24680 versus 43193 for LXT. The difference of 18513 means that at least one of the two requests more than one comparison, in fact about two on the average. To investigate this we have to appreciate the logic of PLX/SLX:

```
PLX/SLX :=
 if skeleton(a) = skeleton(b) then combine(a, b);
 else if skeleton(a) > skeleton(b) then GT;
 else LT;
```

where

- skeleton is a function extracting the part of the argument expression that is relevant for comparison,
- combine is a function combining like expressions and returning EQ,
- LT, EQ, GT are constants indicating the lexicographic order (<, =, >) of a relative to b.

Evidently, an efficient implementation would be contended with one comparison, and we conclude that PLX/SLX may be less efficient. Measurements with the augmented invocation count reveal that indeed at certain points in PLX the module LXT is called for a re-comparison. The reasons lie solely in the particular implementation and are of no interest here. After elimination of the re-comparisons, we obtain the counts given in Table 2. The sum of the counts of PLX and SLX is now equal to the count of LXT.

To further reduce the lexicographic activity, we turn to LXO. It attemps to insert a given term x into a partially sorted chain of terms and distinguishes three cases:

x is a numeric constant (LXOconst)
a variable (LXOvar)
other (LXOother) .

Constants are handled by LXO directly and do not require PLX and LXT. Insertion of variables is peculiar: a first attempt is for a new variable to be appended at the end of the chain, a second is for variables already existing in the chain, a third (= var retry) is a full insertion scan.

Measurement yields count #1 for the system so far, and count #2 for the system with LXO modified, as discussed below, in Table 3.

module/condition	count #1	count #2
AUT	363	363
LXO	10663	10663
LXOconst	5658	5658
LXOvar	496	496
LXOother	4509	4509
PLX	20720	6651
PLXvar	0	0
PLXvar retry	0	0
PLXother	20720	6651
SLX	3957	3384
SLXvar	496	496
SLXvar retry	119	0
SLXother	3342	2888
LXT	24677	10035

Table 3

We observe (count #1):

1. Over 50% of the LXO invocations are "trivial", i.e. for constants and requiring little work. That leaves a relation LXO-nontrivial to LXT of about 1:5.
2. All LXO invocations for variable insertion are product cases. How many of these were a hit on the first attempt has not been determined; but we know that 119 retries were necessary - a fairly high rate.
3. For one insertion, LXO calls PLX/SLX several times. How often, can be measured. But in this example one may as well estimate: The product chains are of an estimated length of 3. So, if only 1500 of the 4509 LXO invocations were for SLX (assuming an average of a little more than 2 attempts on product insertion) then about seven PLX attempts (or more) come on one sum insertion. This is good enough to raise suspicion. For, if LXO employed some sort of logarithmic search the number of attempts would have to be lower for sums of the actually given length.

Subsequent analysis of LXO showed that it did indeed not utilize a logarithmic search, but a straight linear insertion. Thus, LXO was modified to employ a binary insertion scheme [6] that looks at successive midpoints. At the same time, the peculiar variable retries were eliminated: variables are no longer treated special, since the few instances where it would be advantageous do not justify the overhead. The measurement results after modification of LXO are given in Table 3 under count#2.

The system at this stage is the one discussed in section 4, Figure 2. The activity of LXO has been reduced from a count of 43193 to a count of 10035, the PLX activity from 20723 to 6651, while the invocations for AUT and LXO remain unchanged.

6. Interplay of Design and Measurement

We shall now illustrate how the effectiveness of an implementation can be monitored by invocation count measurement. Since the tool is so inexpensive to apply one can readily run tests to check if the desired effect turns up, if the effect is strong enough to justify an implementation, and if there are side-effects. It can, better than time measurement, provide the designer with hard data on the performance of an algorithm, to complete an initial theory or vague feeling.

For the purpose of clarity, we concentrate mostly on our standard example. However, it is understood that a variety of tests is needed to give enough evidence. Also, the invocation count should be complemented by other tools such as those mentioned in the first section.

From our attempt to further improve the FORMAC system we pick a few design points related to AUT.

Design point 1: Is it worthwhile to improve merging of two ordered chains?

We recall that LXO is used to build a canonically ordered chain from given elements. If two already ordered chains are to be merged, the elements of the second one are inserted into the first one without utilizing the fact that they come in sorted. Therefore, a module LXO1 was introduced that merges two already ordered chains into one. The method is obvious [6]. The search for insertion of the next element needs not scan the portion before the element inserted last.

One can construct cases in which the method does not bring any noticeable advantage, for example (A+D+F+I) + (C+E+G), assuming alphabetic ordering. Therefore, we are interested in the degree of improvement achieved. In 80% of our test applications, there was an improvement and our standard example appears to exhibit a typical situation:

module	count before	count after	
AUT	363	363	
LXO	10663	6629	} 8178
LXO1	-	1549	}
PLX	6651	5887	
SLX	3384	2552	
LXT	10035	8439	

Table 4

There are now 1549 LXO1 invocations in place of 4034 LXO invocations for chain merging, i.e. about 2.6 former LXO calls are handled by one LXO1 call. The LXT activity has been reduced by about 15%.

Design point 2: How does the Reaut scheme at hand compare with a scheme that simplifies terms while they are generated?

The Reaut scheme is the following: The lexicographic module, though it combines like terms, does not put the result in a simplified form:
PLX: $c_1 * x + c_2 * x \rightarrow (c_1 + c_2) * x$
SLX: $x ** e_1 * x ** e_2 \rightarrow x ** (e_1 + e_2)$
c_1 and c_2 are numeric constants. Instead, such terms are simplified afterwards and LXO is invoked again to establish the correct order. We say, LXO is in a Reaut situation. In our example, we find that 1327 times (out of 6629) LXO is in a Reaut situation.

The above question can be answered in favor of the Reaut scheme at hand. Assume alphabetic order and consider the following example: Add to an already simplified sum W = A+D+E+F five terms W = W+C+C+C+C+C. Table 5 illustrates what goes on. Method (a) is the Reaut scheme at hand, method (b) works by simplifying and re-inserting a term right after it has been generated. Column s gives the number of simplifications, column c gives the number

of comparisons. Method (b) - which can be viewed as simulation of a recursive scheme - needs more comparisons and simplifications because it "hastily" puts every intermediate result into its canonic form, while method (a) "considerately" waits out temporary variations.

method (a)				method (b)			
sorted chain	term	s	c	sorted chain	term	s	c
A+D+E+F	+C	1	3	A+D+E+F	+C	1	3
A+C+D+E+F	+C	1	2	A+C+D+E+F	+C	1	2
A+2C+D+E+F	+C	1	2	A+D+E+F	+2C	1	3
A+3C+D+E+F	+C	1	2	A+2C+D+E+F	+C	1	2
A+4C+D+E+F	+C	1	2	A+D+E+F	+3C	1	3
A+5C+D+E+F				A+3C+D+E+F	+C	1	2
re-simplification:				A+D+E+F	+4C	1	3
A	+5C	1	1	A+4C+D+E+F	+C	1	2
A+5C	+D		1	A+D+E+F	+5C	1	3
A+5C+D	+E		2	A+5C+D+E+F			
A+5C+D+E	+F		2				
A+5C+D+E+F							
sum		6	17			9	23

Table 5

For our Sigsam 2.2 example, the invocation count reflects the difference in methods:

module	method(a)	method(b)
LXO	6629	8021
LXO1	1549	1549
PLX	5887	7540
SLX	2552	3152
LXT	8439	10692

Table 6

Design point 3: How can the number of Reaut situations be reduced?

The Reaut scan is expensive because it includes re-ordering of the entire chain. So, we look for ways
(i) to improve re-ordering;
(ii) to avoid a Reaut scan.

Concerning (i), note that in a Reaut situation the chain to be reordered consists of "old" elements that are in correct order, interspersed with a few "new" ones that may be out of order. Two changes have been made:
1. In Reaut situations, LXO will first compare a new element with its current predecessor, as the new element is likely to be already in the correct position.
2. Old elements are not reconsidered and only new ones are given to LXO for re-insertion.

Measurement shows (Table 7) that the effect of the changes is a more than 10% reduction in the LXT activity.

module	before	change 1	change 1+2
LXO	6629	0	-141
LXO1	1549	0	0
PLX	5887	-764	-905
SLX	2552	0	0
LXT	8439	-764	-905

Table 7

Concerning (ii), there are two questions to answer:
1. Are there transformation rules which apply to new elements?
2. What is the lexicographic position of the new elements?

As for transformations, we distinguish "common" transformations such as e.g.

$c_1*x + c_2*x \rightarrow (c_1+c_2)*x \rightarrow c_3*x$ (*)

and "non-common" transformations which are in particular those that are under user control. In the example of Table 5, the term 5 C is such a common case that should be dealt with on the spot rather than in a Reaut scan. If so, method (a) would not require re-simplification at all, so that s=5 and c=11. In order to obtain a quantitative measure of the effect of a change before its full implementation, a test was run with PLX modified such as to apply (*) on the spot and not to request a Reaut scan. The difference in the invocation count is noticeable (Table 8). The figures give a good projection and justification for a planned effort.

module	before	change
LXO	6629	-1267
PLX	5123	-465
SLX	2552	-301
LXT	7675	-766

Table 8

The implementation is now along the following lines: The lexicographic module may be allowed to tacitly perform common transformations, in which case no re-ordering is needed. That takes care of the slowness of repeated addition of a term to a sum. If however AUT knows about additional transformations then it informs LXO to report back on certain new terms, e.g. x**c with power series truncation, or $x^{**}(\sin^2 u)*x^{**}(\cos^2 u) \rightarrow x^{**}(\sin^2 u+\cos^2 u)$. AUT keeps book if the latter transformation changed the skeleton of the new term. If so, a re-insertion is requested, otherwise it is not.

The given figures hopefully illustrate how measurement serves to verify, i.e. support or refute a design concept. What has not been illustrated but has been experienced is that it also helps to point out side-effects, adverse or beneficial ones, in system parts that were not taken into consideration. The invocation count was clearly superior to timing.

7. References

(1) K. Bahr: A Speed-Up of FORMAC.
SIGSAM Bullletin 27, Sept. 1973

(2) K. Bahr: Some Enhancements to the FORMAC System. SEAS Conference, Leuven, Belgium, Sept. 1973

(3) J.A. Campbell: Problem #2. The Y_{2n} Functions.
SIGSAM Bulletin 22, April 1972

(4) J.P. Fitch + D.J. Garnett: Measurements on the Cambridge Algebra System.
Proc. ACM Int. Comput. Symposium, April 1972, Venice, Italy, pp. 139-147

(5) D.E. Knuth: An Empirical Study of Fortran Programs. Stanford Memo AIM-137, Nov. 1970

(6) D.E. Knuth: The Art of Computer Programming Vol. 3: Sorting and Searching.
Addison-Wesley, Reading, Mass., 1973

(7) B.M. Leavenworth(ed.): Control Structures in Programming Languages.
SIGPLAN Notices 7, no.11, Nov. 1972

(8) P. Marks: Design and Data Structure: FORMAC Organization in Retrospect.
Proc. 1968 Summer Institute on Symb. Math. Computation (R.G. Tobey, ed.), IBM Federal Systems Center, June 1969

(9) R.G. Tobey et al.: PL/I FORMAC Symbolic Mathematics Interpreter.
Contributed Program Library, IBM Program Information Dept., Hawthorne, N.Y., 360D-03.3.004, Sept. 1969

(10) R.G. Tobey: Symbolic Mathematical Computation - Introduction and Overview. Proc. 2nd Symposium on Symbolic and Algebraic Manipulation, March 1971, Los Angeles, pp. 1-15

(11) J. Xenakis: The PL/I-FORMAC Interpreter.
Proc. 2nd Symposium on Symbolic and Algebraic Manipulation, March 1971, Los Angeles, pp. 105-114