January 2, 1973
Kent Curtis

Some Experiments with SETLB Programs

This newsletter summarizes some experiments performed with a simple SETLB program and a comparison which was made between this program and an equivalent FORTRAN program.

I. Nature of Program

The program studied calculated the sequence of numbers generated by the following rules:

Given a starting integer N;

1. If N even then $N^1 = N/2$
2. If N odd then $N^1 = (3\ N + 1\ )\ /2$
3. Iterate on the generated integers until the sequence goes to 1.

The program generated the sequences of odd integers derived from initial odd integers in the range FIRST $\leq$ N $\leq$ LAST with the provision that redundant calculations would not be performed but linkages to previously computed sequences would be indicated. In these experiments, the 17 bit integer arithmetic of SETLB constrained the range to $1 \leq N \leq 701$.

To eliminate redundant calculations, a set was used to accumulate the computed integers and each sequence was followed until it generated a member of that set. In FORTRAN, the set was represented by a linear array.

A variety of modifications of this program provided the comparisons reported herewith.

II. SETLB-FORTRAN Comparison.

No particular attempt was made to design efficient data representations in either the SETLB or FORTRAN versions (integers and integer arithmatic were used throughout) but some care was exercised (especially in the FORTRAN versions) to eliminate superfluous tests and comparisons from the logic. As a result, this comparison is believed to be fair for rather casually created experimental programs.

It should be noted that this problem is really on FORTRAN's home ground of numerical calculations where one might expect SETLB to compare especially unfavorably to FORTRAN.* Since, based on comparison of elementary operations, execution time ratios in excess of 30:1 have been predicted between the present SETLB system and FORTRAN, the results are a bit surprising. They must be taken as indicative of the potential of the SETL dictions to be competitive in program execution as well as in providing advantage for program construction on problems which can use those dictions to advantage.

The results are shown in the following tables:

SETLB-FORTRAN Comparison

(Program Version 1, requiring 2 set membership tests in the inner loop)

Item	SETLB	FORTRAN	Ratio(SETLB FORTRAN)
No. Program Statements	25	60	0.4
Compile Field Length	66 K (octal)	40 K(octal)	1.6
Execution Field Length	160 K(octal)	17 K(octal)	9.4
Compile Time (CPU)	3.8 sec.	0.3 sec.	12.6
Execution Time (CPU)	24.6 sec.	3.1 sec.	8.0
Total PPU Time	28.6 sec.	5.5 sec.	5.2

* The SETLB program reflected this problem characteristic since it looked much like a FORTRAN program, consisting primarily of many short statements.

SETLB-FORTRAN Comparison
(Changes from above table)

(Program Version 2, requiring 1 set membership test in inner loop)

Item	SETLB	FORTRAN	Ratio
No. Program Statements	26	62	0.4
Execution Time (CPU)	24.2 sec.	2.0 sec.	12.1
Total PPU Time *	21.3 sec.	8.5 sec	2.5

It is interesting to note that the execution times of the SETLB programs were nearly independent of the number of set membership tests performed in the inner loop whereas the FORTRAN program was highly sensitive to this, as expected. This reflects the high efficiency of the set membership test in SETLB (a source of substantial power in SETL since it is a basic constituent of many SETL expressions and operators and contributes to both programming and execution time efficiency) but its true significance is obscured by other side effects of the program changes involved (e.g. in the definition and handling of tuples). Some further study of these effects follows.

Comparison of SETLB Programs

(In all of the following comparisons, the compile field length was taken to be 66K (octal). This may or may not be a minimum for the present SETLB system.)

---

* The total PPU time recorded by the CDC SCOPE system seems to have large uncertianties which depend upon the momentary multi-programming job environment, not on the individual user job.

As a starting point, the next table shows data on compiling and executing the SETLB program consisting of a single NOOP as the only executable statement.

SETLB NOOP Program

Item	Data	Comments
No. Program Statements	4	
Execution Field Length	150 K (octal)	
Compile Time (CPU)	1.7 sec.	
Execution Time (CPU)	1.6 sec.	
Total PPU Time	25.2 sec.	
No. Symbol Table Entries	1156	
No. Garbage Collections	0	
Max. Height of Heap	103773 (octal)	a Function of the Execution Field Length
Actual Height of Heap	77377 (octal)	

This table reflects the minimum time and space overhead of the SETLB system as of this date (Dec.21, 1972). As noted above, the PPU time is not very significant but is reported for its indication of order of magnitude. Memory space data in this table reflects the requirements of the SETLB run time library that provides the SETLB operations.

The first comparison simply replaced a statement in the previously described program of the form

$$\text{Tuple} = \text{Tuple} + \langle a,b \rangle \quad \text{(Program A)}$$

with two statements of the form

$$\left.\begin{array}{l} \text{Tuple} \ (\downarrow \text{Tuple} + 1) = a \\ \text{Tuple} \ (\downarrow \text{Tuple} + 1) = b \end{array}\right\} \quad \text{(Program B)}$$

The data is as follows:

Item	Program A	Program B
No. Program Statements	26	27
Execution Field Length	160k(octal)	160k (octal)
Compile Time (CPU)	3.8 sec.	3.8 sec.
Execution Time	24.2 sec.	23.8 sec.
Total PPU time	21.3 sec.	36.4 sec.
No. Symbol Table entries	1157	1157
No. Garbage Collections	4	3
Max. Height of Heap	113412(octal)	113412(octal)
Final Height of Heap	105047(octal)	110271(octal)

These results suggest some sensitivity to the particular choice of SETLB dictions used. Since Program A results in the formation of Tuples which are not required in Program B, one might expect it to be more demanding in time and space. This shows up in the need for an additional garbage collection. The PPU time figures do not correlate with any known differences in the programs and seem spurious.

Further comparisons were made with an expanded program which provided two additional features:

a) a binary representation of each generated integer

b) a set of ordered pairs which could be used to derive a tree representation of the generated sequences.

This program used 40 statements and required 5.1 seconds to compile. It generated 1161 symbol table entries and used 20 seconds $\pm$ 30 % of PPU time in several runs.

This program was run at several execution field lengths to compare time and memory utilization. It is interesting to note that this program executed at each of the field lengths but terminated before completing the construction of its sets in some cases. For the range $1 \leq N \leq 701$, the two principal sets formed had approx. 550 members each. The percentage completion reported below is the percentage of these set members computed before the program terminated due to lack of memory space.

Field length (octal)	Number garbage collections	Maximum Heap size	Execution Time	Comments
150000	5	103773	9.66sec.	3% complete
155000	18	110544	41.22	43% complete
160000	20	113412	52.89	58% complete
165000	24	120163	74.39	92% complete
170000	22	123031	79.61	100%
200000	12	132450	67.93	100%
240000	4	170544	59.14	100%
300000	3	226640	58.08	100%

It appears from this data that each garbage collection cost approximately 1.1 seconds of execution time on this program with a memory space-execution time trade-off of the expected form. Most efficient operation would appear to be at a field length approximately 10% above the minimum for completing the problem.

As an additional comparison, I recoded the problem (without changing the final results or output) to eliminate one auxiliary tuple of the form ⟨integer, string⟩ used in the inner loop. The statisties on three runs were:

Field length	Number garbage collections	Maximum Heap size	Execution time	Comments
155000	16	110544	37.74	46% complete
165000	23	120163	70.79	94% complete
170000	20	123031	73.75	100%

Comparing these results with the previous results shows a significant time saving (7%) by eliminating one auxiliary variable. Some saving of memory space is also apparent since the programs ran further than previously at 155K and 165K field lengths with fewer garbage collections.

Finally, the 550 member set of ordered pairs was removed from the program without otherwise altering results or output. Statistics on that were:

Field length	Number garbage collections	Maximum heap size	Execution time	Comments
150 000	36	103773	68.77	97% complete
160 000	14	113412	54.32	100%
170 000	8	123031	49.56	100%

These results are suggestive because that set of ordered pairs was not used in the inner loop of the program in any way. It was constructed in the inner loop and used once each time around the outer loop. If facilities were available in SETL for the purpose, I might have elected to form and keep that set in secondary storage, thereby freeing memory space, reducing the number of garbage collections needed, and improving turn-around time without serious degredation in program performance The cited data suggests approximately a 10% reduction in field length required to complete this problem and a 50% reduction in garbage collections needed by removing one of the two large sets from central memory.

Conclusions

These crude experiments permit one strong conclusion and suggest two others.

1. SETL programs compete well in execution speed for problems which use the SETL dictions in essential ways. These experiments demonstrated that phenomenon for a maximally simple program which involved only one statement that is unique to SETL. More complete programs which lead to more complicated relationships and dictions can be expected to show even more compelling advantage for SETL if similar comparisons were to be programmed and carried out. Writing such programs in a language other than SETL is difficult, however, and such comparisons are unlikely to be performed.

2. SETL programs tend to rise in memory space utilization like leavened dough. The garbage collector kneads the program down at some cost in time but it rises again, each time with a gradually growing irreducible volume that is inaccessible to the garbage collector.

Providing the programmer or the system with options for allocating portions of the program generated data to secondary storage and for performing SETL operations on data in secondary storage is one possible mechanism for reducing the rate of growth and thus increasing the range of useful application of SETL using existing machines.

Two possible implementation strategies are:

a) Add to SRTL a routine to translate SETL objects from internal CM representation to a linear representation appropriate to secondary sequential access storage and add a set of routines to apply SETL primitives to objects flowing through a buffer. This approach would seem to favor the movement of entire SETL objects and variables to secondary either at the programmer's option or as determined by an optimizer.

b) Introduce a virtual memory concept into the SETL system. This would seem to favor storage of fragments of SETL objects in CM and fragments in secondary.

Some review of SETL algorithms developed to date may suggest which is preferable.

3) SETL programming brings to the fore another type of program optimization which is worthy of consideration both as an automatic optimization facility and as a matter of programming style. The yeasty growth of SETL programs during execution favors elimination of auxiliary variables whereever possible and the reduction of program span during which variables are live. Some thoughts on this will be summarized in Newsletter 93.