SETL Newsletter 177 August 6, 1976

<u>Measurement Utilities for the Optimized SETL System</u> J. Schwartz

We will want to provide the new SETL system with both HELP (debugging) and measurement utilities. This newsletter will outline measurement utilities that can be provided easily and that should be quite useful. Note first of all that such utilities need to serve two somewhat different classes of user. Users of the first kind are those who are developing a SETL program which they intend to bring to acceptable efficiency by the use of basing declarations and without having to rewrite it in any lower level language. Users of the second kind are developing an algorithm in SETL and experimenting with it preparatory to redeveloping a production version of the same algorithm in a lower level language. The second kind of user will be interested principally in the distribution of execution over the instructions of a SETL program; users of the first kind will want more detailed information, and in particular will want to know:

(a) the actual distribution of execution time over the program, allowing for the time actually required to execute each particular instruction (this time is highly variable).

(b) the number of calls to offline library routines. This information can be important to a user trying to assess the adequacy of his basing declarations.

(c) the places at which space is being allocated, possibly needlessly. Time required for garbage collection deserves to be charged against instructions in proportion to original allocations of space. Moreover, points of excessive space allocation can pinpoint failures in our copy-elimination mechanisms.

All the sorts of information alluded to above can be collected in a fairly uniform way by attaching small groups of counters to each of the basic blocks of a compiled SETL program, and by incrementing these counters appropriately.

These counters, and the rules for incrementing them, define our system of measurements completely. At the end of execution, the counters should be examined and printed out (perhaps in a bar chart representation) in appropriate relationship to the source text of the program which generates them. Note that multiple counts may be supplied for simple statements with internal iterations, e.g. set formers or quantifiers.

The following counts and incrementation rules correspond to the measurements described above.

1. Block Entrance Count. Incremented each time a block is entered.

2. Block-calls-library Count. Each time a block is entered, we set a global *present block indicator* variable PBI to an appropriate value. Each time the library is called, we increment BCL(PBI) by 1.

3. Space Allocation Profile Count. Each time the space allocator is called to allocate N words of heap space, we execute SAPC(PBI) = SAPC(PBI) + N.

4. Execution Time-consumption Profile. When block PBI is entered, execute ETP(PBI) = ETP(PBI) + K, where K is the number of instructions comprising the block. When the run-time library is entered, begin a count of the number of instructions executed (e.g. by zeroing a global count accessible to LITTLE, which is and then incremented at the start of each basic block of LITTLE, by an amount equal to the number of instructions in the block). On each return from LITTLE, increment ETP(PBI) by the total number of instructions executed since LITTLE was called.

An alternative method for gathering the information (4), which requires less change to the LITTLE code-generator, but does require additional systems support, is to use an auxiliary 'SPY'-like systems utility program which periodically samples PBI and builds up a histogram of its distribution.