CGTM 85
Harry Saal
February 1970

Microprogram Organization for the Execution of a Wide-Spectrum Higher Level Language

One may observe in recent language development such as that of PL/I a trend toward commonality. This commonality refers to the availability of large numbers of features within one language sufficiently broad to cover the demands of scientific, commercial and perhaps even systems programming. Many techniques have been developed to efficiently deal with these features. However, each relies upon special mechanisms (stacks, hashing, data formats, array addressing and control, segment protection, etc.. The cost of providing all these features in hardware has seemed too high to warrant their inclusion in any one machine. One might conclude, therefore, that bringing together commercial, scientific, and task management features, etc. cannot succeed in an efficient manner and consequently we should produce specialized languages and specialized machines. In fact, I believe this is not the case when we consider (writeable) microprogrammed computers, where the complete environment is not frozen into the machine. Then, however, the problem is to conserve the amount of micro-storage needed to implement these language features, since the cost of micro-storage is a substantial fraction of the entire processor cost. I will therefore assume that microprogram storage sufficient to implement all these features is not available.

We observe that whereas we have synthesized these classes of applications at the language level, in programming practice there are still highly correlated uses or clusters of certain features which are related to the application work. That is, most programs will emphasize one particular area of the wide spectrum language such as scientific calculations and only infrequently use certain features present for systems or commercial work. As well, we observe a certain common nucleus of features such as procedure entry and exit, expression evaluation, data types and structures, etc.. Consequently we propose that the language specification and translation process be uniform and unprejudiced as to usage. However, we expect that in fact the running environments (implemented through microprogramming) will emphasize

certain characteristics of typical use or might be specialized for custom applications.

Let the language L support features N and F_i.

(1) $$L = N + \sum_{i=1}^{m} F_i$$

where N is a common nucleus, F_i are dominant feature groups.

We expect user profile to look for one user to be as in (2) where ϵ_i are to be small numbers.

(2) $$U_j = N + F_j + \sum_{i \neq j} F_i \epsilon_i$$

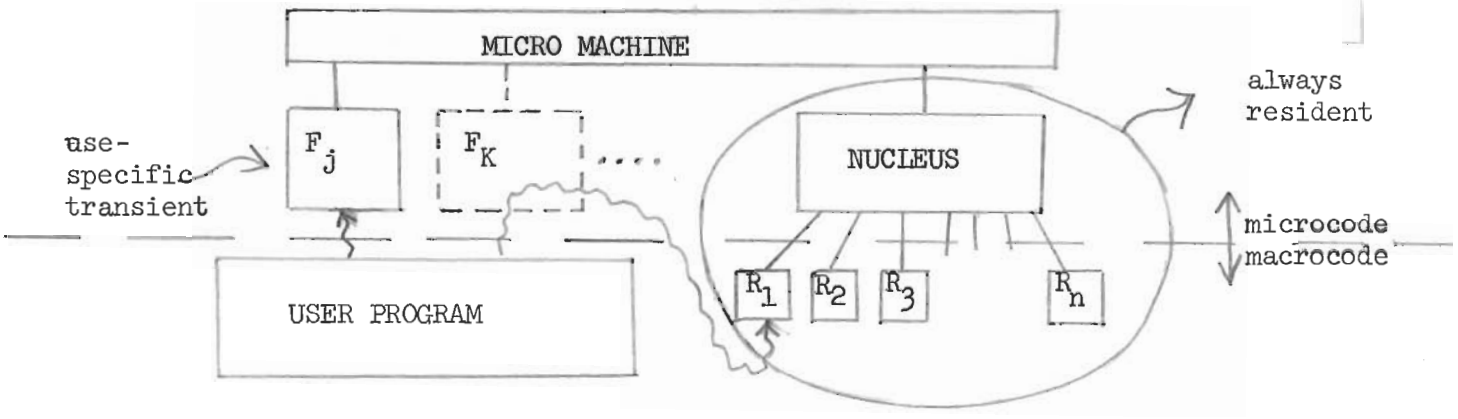
We assume there are resources sufficient to perform such activity U satisfactorily but not the totality of L. In brief, our argument runs as follows: in order to perform the activity U_j we assume that the language processor (by our uniformity argument) should produce code for the hypothetical machine L. This is presumably the best restructuring of a higher level language into a very broad class of machine primitives that we can accomplish. However, we then construct the machine environment in such a way that at run time (in a given environment) the predominantly used features are directly executed by microprogramming, whereas the others act as UUOs or SVCs, that is, they cause execution of macro routines coded in the nucleus language N.

Elaboration of the scheme: we construct environment W_j for a U_j type job as follows: all facilities N and F_j are directly supported (in microprogram). Any features $F_k \neq F_j$ required by U_j is supported by initiation of a "pure U_j process"; i.e., a routine supported solely by N and F_j. This imposes certain sufficiency conditions on all sets $[N, F_j]$ in their ability to support the other features F_k. For example, N must be powerful enough to effect the same set of state changes that any F_i can. In general we would expect that these features could be most readily satisfied within the nucleus N. Consequently we require

a series of routines $R_k^{N,j}$ where K is the service F_k provided and N,j is the environment R functions within. Hopefully the ratio

(3) $$\frac{\sum_k \epsilon_k R_k^{N,j}}{N + F_j}$$

is sufficiently small compared to one that we are not overwhelmed by the type of non-U_j work that needs to be done. It is clear from counting that if there are m feature sets available and if each R indeed depends on both N and j, we require a total of $1 + m^2$ routines to be written (including the nucleus and the feature F_j). On the other hand, if all routines R can be supported by the nucleus along, we then need a total of $1 + 2m$ routines. For m=4, this is a 2:1 saving in the amount of programming effort required, although in principle one may consider taking advantage of the presence of microroutine F_j in coding the other features F_k. As well, it also simplifies the amount of actual macro-program swapping that might be necessary to implement such a feature. If we remove the j dependence from routines R, then in fact all we need to have resident in main core are the series of routines $R_1 \cdots R_n$, all of which are supported by the nucleus (which is always present in the microprogram memory). The only swapping is of microroutines F_j. The F_j are considered our fast primary resources. If a feature not supported by F_j is used, the system uses one of the backup routines R_k which share a common interface with the nucleus language. See Fig. 4.



Thus in a time sharing environment from user to user only one F_j need be swapped.* All routines R are at fixed external locations and any microprogram "fault", i.e., a usage of an $F_K \neq F_j$ by U_j causes backup routine R_K to be initiated.

Certain Considerations

There are several advantages to this scheme. At the language level one achieves uniformity of design. That is, the translation process is based entirely upon the actual features or primitives necessary to support those features and not fixed to some existing environment. Thus in fact, such a translator program could be used on a wide spectrum of micro-program machines as long as they can be microprogrammed to support the environment in the manner presented above.

In a multi-user system, a great danger arises due to the lack of protection mechanisms at the microprogram level. Generally a micro-program segment has access to all facilities of the inner machine and can easily cause severe inadvertent problems until well debugged. In this proposed organization one can first provide the R routines which are not micro-coded and easier to install. Language processors can now utilize these features, and once debugging on a simulator has provided sufficient checks on the micro-routines they can be loaded in a transparent fashion.

* Interestingly, should the system decide not to swap F_j, the next user will still execute correctly but more slowly.

In actual implementation we may go to either extreme in the number of environments to be provided. We may, due to lack of time and effort, simply provide one environment. This approach is equivalent to the present choice of buying a machine with a fixed instruction set from a manufacturer. At the other end of the spectrum we can provide specialized environments, either by writing them as they appear necessary to optimize a certain task, or preferably we can produce such environments in some automatic fashion (for example loading a library of subroutines as now done from a subroutine library).