論 文

論文 77-14-6-2

Microprogram Organization for the Execution of A General Purpose Language

趙 廷 完*

(Cho, Jung Wan)

요 약

범용의 컴퓨터언어를 정의하고 이 언어의 수행에 적합한 컴퓨터 architecture를 제시하였다. 이 architecture는 마이크로프로그램기법을 이용하며 이의 특징은 마이크로모듈 swapping의 필요성을 줄이므로서 이에 소용되는 시간을 절약할 수 있는 점과, 원하는 마이코로듈이 제어 기억장치에 부재시에도 프로그램 수행이 가능한 점이다.

Abstract

A general purpose language is defined and a computer architecture suitable for this language is proposed. The proposed architecture utilizes the microprogramming technique. The significant features of the proposed architecture are the reduction of the overhead in micro-module swapping and graceful degradation of the execution upon detection of the micro-module fault in the time-sharing environment.

1. Introduction

One may observe in the recent language development such as PL/1 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 applications. Many techniques have been developed to efficiently deal with these features. However each relies upon special mechanisms such as stacks, data formats, array addressing and control, segment protection, etc. The cost of providing all these mechanisms in hardware has seemed too high to warrent their inclusion in any one machine. We may conclude, therefore, that bringing together the features required by all of the scientific, commercial and systems applications cannot succed in an efficient manner and consequently we should produce specialized languages and specialized machines. We believe, however, in fact, this is not the case when we consider the user-microprogrammable computers, where complete environment is not fixed into the machine. Then, however, the problem is to conserve the amount of control storage needed to implement these language features, since the cost of control storage is a substantial fraction of the entire processor cost. We will therefore assume that micoprogram storage sufficient to implement all these features is not available in the following section.

* 正會員, 韓國科學院 컴퓨터 싸이언스科
본 연구는 부분적으로 한국과학원 자체연구비의 지원을 받았음.
(Korea Advanced Institute of Science, Computer Science Department, This research was supported in part by the KAIS-IRF 0403-4214.)
接受日字 : 1977年 11月 14日

2. General Purpose Language

In this section we give a descriprion of the

general purpose language. The description that will be given here is conceptual rather than formal grammar such that it provides the environments that the general purpose language supports. We observed the classes of applications at the language level in the previous section.

In programming practice there are still highly correlated uses or clusters of certain features which are related to the specific application. That is, most programs will emphasize one particular feature of the wide spectrum of features supported by the general purpose language. For instance, some program will emphasize scientific calculations and only infrequently use the certain features present for systems or commercial applications. Furthermore, there exist certain common nucleus of features such as procedure entry and exit, expression evaluation, data type and structure, etc. From these phenomena, we may conclude that the language specification and translation process are uniform and independent of the usage, however, the running environments will emphasize certain characteristics of typical use or might be specialized for custom applications.

Based on the developments in the above phrase, we can derive a description of the general purpose language L in terms of the environment, E, that L supports.

$$E = N + \sum_{j=i}^{n} F_j \quad \cdots\cdots (1)$$

In(1), N is the common nucleus, F_j is one of the n dominant feature groups that L supports.

For(1), it may be difficult to develop an algorithm for partitioning the features into dominant feature groups. However it is not too difficult to list some of the inherent partions. Hardware/firmware implemented instructions for multiply, divide and evaluation of the mathematical functions may belong to one group which is dominantly used by the programs for the scientific applications. Hardware/firmware implemented instructions for decimal computations, file handling, and high performance input/output may belong to the other group which is dominantly used by the programs for the business applications. On the other hand, hardware/firmware implemented instructions for interrupt handling and privileged instructions for task managements may belong to still other group which is dominantly used by the programs for the systems applications.

3. Machine Architecture

It is obvious that the construction of the machine architecture for general purpose language is straight forward when the construction cost is not one of the major design criteria. In this case one may implement the common nucleus and all of the dominant feature groups, i.e. F_j's in(1) in the host machine with the combination of hardware and firmware. However this is unrealistic as described in the introduction. What we must consider is the cost and performance tradeoffs.

When one wants to simplify the construction of the machine architecture, a specific user's environment, E_j of the user U_j can be described as shown in(2).

$$E_j = N + F_j \quad \cdots\cdots (2)$$

Although the interpretation of (2) for N and F_j is somewhat different from what we proposed in(1), Wilkes[1], Thomas[2], and Cho and Woo[3,4] proposed user-microprogrammable architectures based on(2). In their approaches N and F_j are considered as the micro-modules for the base machine instruction set and for the user-defined special instruction set, which is microprogrammed by user U_j, respectively. In these architectures N is stored permanently in ROM. The differences between these architectures are the storage mediums for the micro-codes for F_j and the mechanisms for execution of F_j. These architectures are suitable when one process remains in the execution state until it finishes. However this is not likely to happen in the multiprogramming environments. In this case, whenever the process switches, loading and storage allocation for the new micro-codes pose as overhead. Furthermore, since each user may have a different micro-module and, for $i \neq j$, F_i and F_j may contain some identical micro-codes, microprogramming efforts may be wasted by the redundant micro-codes among users.

This paper inspects the user's environment

somewhat differently. When we inspect the user profile, a specific user's environment E_j, for the user U_j in terms of the features required by U_j, may be expressed as shown in (3). In (3) ϵ's are

$$E_j = N + F_j + \sum_{i=1, i\neq j}^{n} \epsilon_i F_i \quad \cdots\cdots (3)$$

small numbers. From (3) we may conclude that, for U_j, there must be sufficient resources to carry out the activities of the programs of U_j. This implies that the machine architecture may provide sufficient resources for E_j but not the totality of E. In order to perform the activities of the U_j's program, we assume that the language processor should produce codes for the hypothetical machine L. This is actually restructuring of the higher level language into a very broad class of machine primitives that the architecture can accomplish.

In (3), if F_j is in fact a predominantly used feature group, then the ratio of $\sum_{i=1, i\neq j}^{n} F_i$ and $N+F_j$ will be significantly smaller than 1. There are several ways of computing this ratio. One way is to compare the frequeucies of usages of each feature group. At any rate, since this ratio is significantly smaller than 1, loading of micro-codes for F_i's, where $i\neq j$, for user U_j is not economically justifiable. Therefore, a cost effective general purpose anguage architecture can be constructed from (3) ch that the common nucleus N is implemented by hardware or microprogram permanently stored in ROM, the dominant feature group F_j for user U_j is mplemented by microprogram loaded in the writable control storage, and the rest of feature groups, F_i's, where $i\neq j$, are implemented by using either N or F_j, or both. This implies that F_i's must be implemented via software routines that utilize the features N and F_j. Furthermore all the routines for F_i's must be main memory resident in order to maintain reasonable execution speed.

From the development in the previous paragraph, we can now rewrite (3) as shown in (4)

$$E_j = N + F_j + \sum_{i=1, i\neq j}^{n} \epsilon_i R_i(N,j) \quad \cdots\cdots (4)$$

where $R_i(N,j)$ represents a set of software implemented macro-routines that behaves as the feature group F_i using N and F_j. In order to construct a general purpose language architecture for (4), it requires to construct $1+n$ micro-modules for N and Fj's, and n^2-n macro-modules for $R_i(N,j)$'s. However, in (4), if we transform $R_i(N,j)$ into $R_i(N)$ or simply R_i, then it only requires to construct $1+n$ micro-modules for N and F_j's and n macro-modules for R_i's. Such a transformation requires that the macro-modules R_i's must be constructed by using the common nucleus N only. The reduction of the number of macro-modules to be constructed has advantages; first, savings in the macro-module development efforts, second, savings in the macro-module storage space, and third, reduction of the macro-module swapping time overhead. By doing this, (4) can be rewritten as (5).

$$E_j = N + F_j + \sum_{i=1, i\neq j}^{n} \epsilon_i R_i \quad \cdots\cdots (5)$$

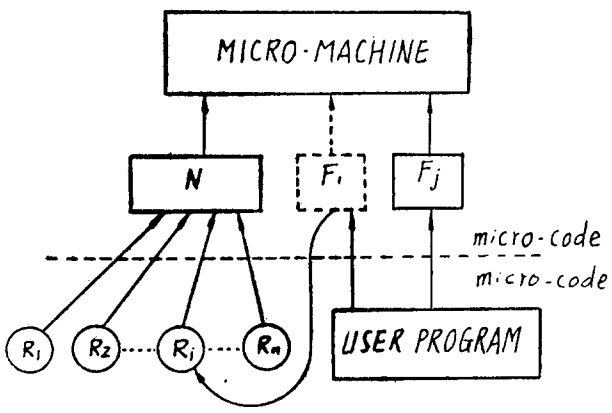

Fig. 1. General purpose language architecture.

From (5), we propose a machine architecture, that supports the general purpose language, as shown in Fig. 1.

In Fig.1, the series of routine R_1, R_2,…, R_n are resident in the main memory and all of which are supported by the nucleus, which is always present in the microprogram memory. The only swapping is of micro-modules F_j. If a feature not supported by F_j is used, say F_i, the system uses one of the back up routines R_i which share a common interface with nucleus. Interestingly, should a system decides not to swap F_j, particularly in the time-sharing environment, the next user may have a microprogram fault, i.e. user U_k expects to have $F_k \neq F_j$ in the writable control store, but U_k's program will still execute correctly but more slowly using R_k.

4. Conclusion

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. 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 probably we can produce such environments in some automatic fashion. For example loading a library of subroutines as now they are done from a subroutine library.

References

[1] M.V. Wilkes, "The Use of A Writable Control Memory In a Multiprogramming Environment," Preprints of the 5th Annual Workshop on Microprogramming, Sept. 1972, pp.62~65.

[2] R.T. Thomas, "Computer Organization for Allowing Dynamic User Microprogramming", SIGMICRO Newsletter, Vol. 4, July 1973, pp. 28~39.

[3] J.W. Cho and N.S. Woo, "Design of A User Microprogrammable Computer, "Journal of Korean Institute of Electrical Engineers, Vol. 26, Jan. 1977, pp.71~76.

[4] J.W. Cho and N.S. Woo, "A Fourth Generation Microprocessor Architecture: User Microprogramming with Low Cost," Digest of Papers in 15th IEEE Computer Society International Conference, Sept. 1977, pp.451~453.