COO
~~CHO~~-3288-29

HOPKINS COMPUTER RESEARCH REPORTS
REPORT # 29
DECEMBER 1973

95,729

# "IDEAL" DIRECTLY EXECUTED LANGUAGES: AN ANALYTICAL ARGUMENT FOR EMULATION

BY

LEE W. HOEVEL

RESEARCH PROGRAM IN COMPUTER SYSTEMS ARCHITECTURE
COMPUTER SCIENCE PROGRAM
THE JOHNS HOPKINS UNIVERSITY
BALTIMORE, MARYLAND

Author: Lee W. Hoevel
310 Ridgemede Road, #204
Baltimore, Maryland 21210

Title: "Ideal" Directly Executed Languages:
An Analytical Argument for Emulation*

Key Words and Phrases: Emulation, Microprocessor, Machine-Oriented-Languages, Language-Oriented-Machines, Directly Executed Languages (DELs), interpretation.

Abstract:

Several methods of evaluating user programs are analyzed with respect to space and time requirements. The concept of an "ideal" directly executed language is introduced, and it is argued that the "ideal" directly executed language for a contemporary computing system will not be either its source language or the language accepted by its base machine.

* This work was supported by the U. S. Atomic Energy Commission under contract AT (11-1 3288)

# "Ideal" Directly Executed Languages:
# An Analytical Argument for Emulation

## Introduction:

In a broad sense, any computer system evaluates user-written (or source language) programs in two distinct phases. First, a user program is translated into an "equivalent" program in some intermediate language during an initial translation phase. The resulting program then becomes a "surrogate" for the original user program, being executed in its place as often as desired, over any number of subsequent interpretation phases. In real systems, there are usually several "levels" of intermediate languages involved in the translation phase (e.g., "parse-tree code" emitted by a syntactic scanner, "object code" emitted by a compiler, "relocatable code" emitted by a linkage-editor, and finally the "machine code" emitted by a loader). Similarly, there may also be multiple "levels" involved in an interpretation phase (e.g., the microprogrammed CPU of a model 65 emulating a 360-machine which is executing a 360-code program that is interpreting a LISP source program).

In this paper, such complex computing systems are abstracted into "Two-Phase Processing Systems," in which all translational processes are composed into a single compiler component, and all interpretational processes are incorporated into a single emulator component. Any combination of a source language and base machine defines a Family of Two-Phase Processing Systems where:

I: Both the compiler component and the emulator component of any member of a given Family are, in a general sense, programs which will run on the common base machine of that Family.

II: The input medium for the compiler component of any member of a given Family must be the common source language for that Family, while its output medium must be an intermediate language which also serves as the input medium for the emulator component of that member.

The intermediate languages mentioned in property II are called "Directly Executed" languages, or DELs for short, since they are the lowest "level" into which user programs are translated in their entirety prior to interpretation. Constructing a good Two-Phase Processing System, given a particular source language and base machine, can be considered as equivalent to selecting a superior member of the Family defined by that source language and base machine. If the DEL of such a superior member could be ascertained, then the design of a compiler and emulator for a good system would be constrained both by the given source language and base machine, and by the chosen DEL. From this viewpoint, it is clear that a non-trivial emulator will be required iff some language other than that accepted by the given base machine is selected as the DEL for the final system.

There are many arguments for using a non-trivial emulator to evaluate the DEL surrogates of user programs. For example, it is often stated that monitoring facilities, automatic error recovery, and dynamic resource reconfiguration are easier to implement if the interpretation phase mechanism is not completely "hardwired." Such claims are relatively easy to establish on a theoretical basis, but it is the author's opinion that the implementation of these facilities is often not followed through, or at least that they are seldom actually used by an average source language programmer.

In practice, however, those systems whose DEL has been "tailored" to a specific combination of source language and base machine show significant reductions in the space and time required to evaluate user programs (see [Weber 67], [Abrams 70], [Tucker and Flynn 71], [Zaks 71], and [Wortman 73]). The important observation here is that the choice of DEL is a critical factor in determining the potential efficiency of any system.

## Objective:

Our intent is to analyze the process of interpretation and determine those technological conditions under which the use of a non-trivial emulator will enhance the performance of a Two-Phase Processing System. The analysis is couched in terms of a search for an "ideal" DEL within a given Family. This "ideal" DEL would allow the use of minimal overall space and time in evaluating a typical user program. There are two obvious choices for such an "ideal" DEL: the given source language (SL), and the language accepted directly by the given base machine (ML). We will show that there exist currently plausible technological conditions under which neither SL nor ML is "ideal" (and hence that the further study of DELs is justifiable on the basis of space-time considerations alone, without resorting to less "tangible" arguments).

## Initial Assumptions:

It may be observed pragmatically that user programs are easier to develop in a "high level" language, while emulator programs are more efficient when "microprogrammed." Therefore, we will restrict our attention to Families of Two-Phased Processing Systems which are constrained by "high level" source languages and "microprogrammable" base machines.

For our purposes, a "high level" source language is a user-oriented language whose programs are finite strings of characters, organized into a physical sequence of syntactic units called "statements," where:

i: A virtually unlimited number of symbolic "names" (e.g., strings like the ubiquitous 'TEMP1' and 'TEMP2') may be used to identify program variables.

ii: Arithmetic expressions may be written in the traditional parenthesized hierarchial infix-operator notation (e.g., as in high school algebra).

iii: The logical sequence of the statements in a source program may be defined in a "structured" way, using a few basic constructs (e.g., IF-clauses, DO-loops, and PROCEDURE-blocks) to build up arbitrarily complex control structures.

A "microprogrammable" base machine is a sequential, instruction-driven device where:

i': There are two levels of addressable memory: (a) Control Store, a "fast memory" whose cycle time $t$ is less than or (more often) equal to the primitive cycle time of the base machine, and whose access width $wc$ is at least as large as the width $wu$ of a typical base machine instruction; and (b) Main Store, a "slow memory" whose cycle time $R \cdot t$ is usually much longer than $t$, and whose access width $wm$ is usually no greater than $wc$. Control Store is usually more costly per bit than Main Store (since it is typically both wider and faster), and hence we will also assume that (c) the size of Control Store is small in comparison with the size of Main Store (due to economic considerations).

<u>ii':</u> Each ML (or micro) instruction (a) contains the same number of bits: some of which (b) specify a set of primitive operations by directly controlling key "gates" within the execution resources of the base machine (these bits form the "op-code" field of a ML instruction); and some of which (c) are the address(es) of logical successor instruction(s) (these bits form the "sequencing" field(s) of a ML instruction).

<u>iii':</u> Several ML instructions are required in order to duplicate the effect of individual, frequently used, source language operators.

Finally, we are primarily interested in <u>comparing</u> the space and time required to generate and interpret various <u>forms</u> of DEL code. In order to focus in on the pertinent factors, we will also assume that:

<u>*:</u> All DEL surrogates for a given source program q use the same general "algorithm" as q; and in particular, the storage of program variables (with respect to the size and placement in memory of their values) will be the same for any two surrogates for q.

<u>$SL \overset{?}{=} DEL$:</u>

It is easy to show that SL is not an "ideal" DEL from the standpoint of program <u>size</u> (a measure of the space required during an interpretation phase). Merely replacing the long symbolic "names" allowed by <u>i</u> with well-chosen binary encodings (see below), and replacing the arithmetic expressions allowed by <u>ii</u> with functionally equivalent "parenthesis free" forms (e.g., translating into Polish Suffix), will substantially reduce the size of a typical SL program.

It is also easy to show that SL is not an "ideal" DEL with respect to interpretation phase time. We need only consider the problem of associating <u>values</u> with <u>program variables</u>. There are, in general, far too many potential symbolic names allowed by <u>i</u> to be able to "pre-assign" a unique storage cell to each possible name (BASIC may be an exception here); hence finding or changing the value of a symbolically identified variable is an inherently indirect operation. Replacing symbolic names with binary encodings which are, in fact, the appropriate cell "addresses" (or are easily converted into the correct addresses by simple arithmetic operations) will eliminate one table look-up per access of a program variable during execution.

Assumption iii strengthens this point, both from a space and time standpoint, since each invocation of a PROCEDURE block can alter the correspondence between symbolic names and storage cells (as explained by the Contour Model, [Johnston 71]). Careful choice of "address formats" can eliminate an entire level of indirectness in accessing variables ("indirectness" measured in terms of accesses to memory), and also can obviate the need to maintain dynamic, run-time symbol tables (e.g., by using an implied "base register" to duplicate functions of the Contour Model "environment pointer").

Note that it is also faster to evaluate a "linearized" expression, where operators are all treated uniformly (e.g., as in Polish Suffix) than it is to evaluate the equivalent, parenthesized, hierarchial, infix-operator expression (as in ii). This is discussed briefly in [Lawson 68]: having "to scan to locate each instruction and prepare it for execution is hopelessly inefficient" (p. 477).

If we also assume that

> iv: the DEL surrogate for a typical source program will be evaluated several times (or, equivalently, that it contains a "loop" which will be traversed several times in a single evaluation)

then SL will not be an "ideal" DEL with respect to the overall space and time needed to evaluate a typical source program; even though it would reduce the compilation phase space and time to an absolute minimum.

ML $\overset{?}{=}$ DEL:

Using similar arguments, it is easy to show that ML is not an "ideal" DEL with respect to either program size or compilation phase time from assumptions i' through iii'. For instance, since several ML instructions are required to implement some individual SL operators (iii'), we could obviously find a more length-minimal encoding for ML surrogate programs. Further, the space and time required during the translation phase in order to make efficient use of the parallelism in ML instructions (ii'b), or to allocate specific base machine resources to symbolic SL variables, is clearly much greater than that required to translate SL programs into the encoded Polish Suffix mentioned above.

It might seem reasonable, however, to speculate that ML will always be an "ideal" DEL from a standpoint of interpretation phase time. If we value reductions in execution time highly enough (as implied by assumption iv), then ML would be a cannonical "ideal" DEL for all the systems under consideration, and our search for an "ideal" DEL would be over.

Let us consider a hypothetical system which uses ML as its DEL and where source language and base machine satisfy i through iii', as illustrated in figure 2. While its emulator may indeed be trivial, its compiler, uC, is likely to be eminently non-trivial. Since there are few such compilers currently in existence, we will describe only the properties of the output of uC, rather than delving into its inner workings. In the following, let q be a "typical" source program, and u = uC(q) be its ML surrogate.

## Attributes of u:

One basic attribute of u is that it is "dynamically short" (i.e., more instructions would have to be executed in order to evaluate any other "equivalent" ML program). It is well known, however, that in general, the shortest possible program cannot be produced by any deterministic mechanical method. What we have actually assumed here is that u is the "best that can be done" within the current state-of-the-art. In a practical sense, this means that each operation in q is simulated by an in-line "open subroutine" in u. and that the interface between any two logically adjacent "open subroutines" has been optimized, at least with respect to redundant data transfers.

On the other hand, another important attribute of u is that, in general, it is statically long, in the sense that it will "fit" within the Main Store but not within the Control Store. Essentially, this situation results from the limited size of the Control Store (assumption i'(c)) and the fact that "programs tend to expand to fill the memory available." As we will see, it is also the primary reason that u may not be the "fastest possible" surrogate for q.

## Methods of Evaluating u:

Even though u cannot be stored entirely within Control Store, there are still several possible methods that might be partitioned into two parts: one part in Main Store and one part in Control Store. Individual instructions would be executed directly out of the memory in which they were stored. In general, the placement of u itself might even be reorganized dynamically to achieve "optimal" execution times. However, we will consider only two representative methods of evaluating u:

Method A: The direct evaluation out of Main Store: u is stored entirely within Main Store and is evaluated by fetching individual instructions directly out of Main Store (one at a time) and executing them within the internal resources of the base machine. There is no dynamic reorganization of the placement of u in this method.

Method B: The indirect evaluation out of Main Store: u is initially stored entirely within Main Store, but is evaluated by fetching "segments of u" from Main Store into Control Store upon demand, executing as many individual instructions as possible directly out of Control Store in between demands for new segments (called faults).

Now, since Main Store and Control Store have different physical characteristics, the time in between the execution of individual instructions could vary from Method A to Method B. In order to compare these methods, we must assume that

iv': the effect of executing a sequence of instructions from u is not affected by arbitrary time delays between the execution of individual instructions in that sequence; i.e., no "time dependent" code is generated by uC.

Actually, it is much easier in practice to generate "time independent" code than it is to generate "time dependent" code which will be correct for any possible input. Assumption iv' also appears to be reasonable in light of the fact that such "time dependencies" would eith have to be fielded dynamically at a fault, or anticipated by uC in order for Method B to be a viable alternative.

Under these assumptions, let us form an estimate of the execution time required by each of these methods. For comparative purposes, we will focus on the time required to fetch and execute instructions and ignore the time needed to access data. This is an acceptable practice under assumption *, which assures us that the general algorithm, data storage techniques, and hence the time needed to access data values will be constant over all of the methods under consideration.

Time Estimate for Method A:

In this method, u is stored entirely within Main Store, and individual ML instructions are fetched and executed, one at a time and in the proper sequence, directly out of Main Store, as illustrated in figure 2. The time needed to fetch an ML instruction may be estimated as $[wu/wm]\cdot R\cdot t$, where:

> wu is the width of an ML instruction (a constant by ii'(a));
>
> wm and R·t are the width and cycle time of Main Store respectively (as in i'(b));

and $\lfloor wu/wm \rfloor$ is just wu/wm if the base machine uses "instruction buffering" and "branch anticipation" (see [Tucker and Flynn 71]), or is the least integer not less than wu/wm if it does not use these techniques (denoted by $\lceil wu/wm \rceil$).

Let #u be the total number of ML instructions which must be fetched and executed during the evaluation of u, and denote the jth ML instruction so fetched and executed as $u_j$ (for j = 1, ...,#u). Then the total time needed to evaluate u using Method A is given by:

$$TA\cdot t = \sum_{j=1}^{\#u} ([wu/wm]\cdot R + e_j)\cdot t = \#u\cdot([wu/wm]\cdot R + E)\cdot t, \text{ where:}$$

> $e_j\cdot t$ denotes the time needed to execute $u_j$ that is not "overlapped" with a memory access;

and $E = (e_1 + \ldots + e_{\#u})/\#u$ is the "dynamic average" execution time for a ML instruction.

If the base machine employs an "overlap" logic (i.e., fetches $u_{j+1}$ in parallel with the execution of $u_j$), then E is likely to be close to zero; otherwise E is likely to be close to one (although it may assume other values depending on the architechture of the base machine).

Time Estimate for Method B:

As illustrated in figure 3, in this case there are two nested 'loops' which are iterated several times during a single evaluation: an 'inner loop' (steps [1], [2a], [3]; [1], ...), during which ML instructions are fetched directly out of Control Store for execution; and an 'outer loop' (steps [1], [2b], 'inner loop'; [1], ...), in which fresh segments of u are brought from Main Store into Control Store so that 'inner loop' processing may continue. The dynamic ML instruction stream executed during 'inner loop' processing should be the same as in Method A, since u itself is unchanged. This time, however, we denote an individual instruction $u_{j,k}$ in this stream by using a "double subscript" that indicates the 'outer loop' and 'inner loop' iteration in which it is executed. The 'outer loop' index, j, runs from 1 to #f (the total number of segment transfers, or faults). The index for the 'inner loop' contained in the jth iteration of the 'outer loop,' k, runs from 1 to $\#u_j$ (the total number of ML instructions fetched from Control Store between the jth and j+1st faults).

Note that $\#u = \#u_1 + \ldots + \#u_{\#f}$ by iv' (and the definition of u); and that the "faults" referred to in this section deal only with instruction fetches and not data fetches. The time needed to access data will be the same for any of the methods under consideration and therefore may be safely excluded from these comparative time estimates.

The time used in the jth 'outer loop' iteration, excluding the time used within its nested 'inner loop' (calculated below), will be:

$(\$_j + \lceil ws_j/wm \rceil \cdot R + \lceil ws'_j/wm \rceil \cdot R) \cdot t$ where:

$\$_j \cdot t$ denotes the time needed to determine where the segment causing the jth fault is located in Main Store and where it should be placed in Control Store (i.e., the decisional "cost" of the jth fault);

$ws_j$ denotes the width of the segment to be moved from Main Store to Control Store;

and $ws'_j$ denotes the combined width of any segments which must be moved from Control Store to Main Store in order to make room for the new segment.

Since this discussion centers on instruction streams rather than data streams, it seems reasonable to assume that no changes are made to segments while they are in Control Store, and hence that new segments may overlay old segments with impugnity. Under this interpretation $ws'_j$ will be zero.

If we further assume that all segments are the same size (i.e., "pages" in Denning's terminology), then a simple cyclic replacement algorithm may be used at each fault. In this case $ws_j$=ws, a constant for all j, and $\$_j$ need only account for the time needed to calculate the starting address of the required segment. This may be accomplished using a simple mask-and-shift operation followed by a look-up in a "memory-map" table and an addition; hence we estimate $\$_j=\$J=3$. This estimate is based on the assumption that there is sufficient parallelism in the ML instruction set so as to allow table updates to be effected during the transfer of the new segment.

The time used within the jth 'inner loop' may be expressed as:

$$\sum_{k=1}^{\#u_j} (\$_{j,k} + [wu/wc] + e'_{j,k})\cdot t, \text{ where:}$$

<u>$\$_{j,k}\cdot t$</u> denotes the time needed to determine whether the ML instruction "next to be fetched and executed" (i.e., the logical successor of $u_{j,k}$) is on a segment currently in Control Store, or whether a fault has occurred (in which case the successor of $u_{j,k}$ would be denoted $u_{j+1,1}$;

<u>$[wu/wc]\cdot t$</u> is the time needed to fetch $u_{j,k}$ from Control Store, as above;

and <u>$e'_{j,k}\cdot t$</u> is the "unoverlapped" time required to execute $u_{j,k}$ (again, note that $e'_{j,k}$ is usually one for simple base machines and zero for complex base machines.

It can be argued that $\$_{j,k}$ (the "time-cost" of calculating the location of the logical successor for $u_{j,k}$) should be a constant for all j and k. Typically, the "virtual address" of the next ML instruction is masked-and-shifted (one or two t), the result used as an index in a table look-up (using one t, if the table is kept in Control Store), and then the table entry is examined for a fault indication and perhaps combined with previously masked-out bits to form the "true address" of the next ML instruction. We estimate $\$_{j,k}=\$K=3$ as a minimal value, but note that this figure may be low (for instance, some of the schemes presented in [Denning 70], pp. 162-163, would clearly require several primitive machine cycles in order to generate a "true address" from a "virtual address").

The total time needed to evaluate u using Method B can be expressed as a sum of the 'outer loop' and 'inner loop' time estimates derived above:

$$TB\cdot t = \sum_{j=1}^{\#f} (\$J + \lceil ws/wm \rceil \cdot R + \sum_{k=1}^{\#u_j} (\$K + [wu/wc] + e'_{j,k}))\cdot t$$

(or more concisely as:

$$TB\cdot t = \#f\cdot(\$J + \lceil ws/wm \rceil \cdot R)\cdot t + \#u\cdot(\$K + [wu/wc] + E')\cdot t$$

where E' is the average value of the $e'_{j,k}$).

<u>SL ≠ DEL ≠ ML</u>:

Now consider a Two-Phase Processing System which uses the same base machine to evaluate programs in the same source language, but which employs a different DEL, V, where SL ≠ V ≠ ML. Such a system <u>must</u> contain both a non-trivial compiler, vC, <u>and</u> a non-trivial emulator, vE.

Let q be a typical SL program, as before, and v = vC(q) be the V surrogate for q. Whereas u is a sequence of <u>in-line open ML subroutines</u>, v is merely a sequence of <u>V instructions</u>. During an interpretation phase, these V instructions are processed, one at a time in logical order, by the emulator vE as follows:

> vE determines the address of the next logical instruction in v to be processed;
>
> vE fetches this V instruction (say, $v_j$, the jth instruction in the instruction stream generated by this execution of v) from the Main Store;
>
> vE decodes $v_j$ into an "op-code" and a list of "operands";
>
> vE then invokes the <u>semantic routine</u> for the operation indicated in the "op-code" of $v_j$, passing the "operands" of $v_j$ as parameters to the appropriate <u>closed ML subroutine</u> in Control Store (see figure 4).

The main body of a semantic routine will be similar to the main body of corresponding open subroutines in u. They should differ only in the "interface instructions," which were eliminated from the open subroutine expansions in u during the translation phase by uC, or which were included in the closed subroutine implementations for the requisite semantic routines in order to provide an efficiently uniform interface for vE during interpretation phases. For consistency, let us call this method of evaluating q "Method C."

Time Estimate for Method C:

As illustrated in figure 4, there are also two nested 'loops' in this evaluation process. During the $j^{\underline{th}}$ iteration of the 'outer loop' (steps [1], [2], [3], 'inner loop'; [1], ...), vE determines the address of $v_j$, fetches it from Main Store, decodes $v_j$ into the address of and parameters for the proper semantic routine, and then invokes this routine. The $j^{\underline{th}}$ nested 'inner loop' (steps [4], [5]; [4], ...) accounts for the execution of this ML routine on the base machine. In the $k^{\underline{th}}$ iteration of this 'inner loop', the ML instruction denoted $u''_{j,k}$ is fetched and executed directly out of Control Store; by definition $u''_{j,\#r_j}$ causes control to be returned to vE, beginning a new cycle of the 'outer loop'. Hence we can compute the time needed to evaluate v using Method C in much the same manner as we computed the time needed to evaluate u using Method B.

The only real differences are that in 'outer loop' iterations, some time, denoted <u>A·t</u>, must be expended to determine the address of the next V instruction (step [1]); some time, denoted <u>D·t</u>, must be expended in order to decode this V instruction (step [3]); and <u>no</u> time need be expended in order to determine the location of succeeding <u>ML</u> instructions (i.e., no \$K term). The resulting estimate for the time required to evaluate v using Method C is:

$$TC\cdot t = \sum_{j=1}^{\#v}(A + [wv_j/wm]\cdot R + D + \sum_{k=1}^{\#r_j}([wu/wc] + e''_{j,k}))\cdot t \text{ , where:}$$

<u>#v</u> is the total number of V instructions processed by vE;

<u>$wv_j$</u> is the width of the $j^{\underline{th}}$ V instruction in the instruction stream for this execution of v (i.e., the width of $v_j$);

<u>$\#r_j$</u> is the number of ML instructions executed during the invocation of the closed subroutine associated with $v_j$;

and $\underline{e''_{j,k} \cdot t}$ is the "unoverlapped" time required to execute $u''_{j,k}$.
There is also a more concise formulation for this time estimate:

$$TC\cdot t = \#v\cdot(\underset{[1]}{A} + \underset{[2]}{[wv/wm]\cdot R} + \underset{[3]}{D} + \#r\cdot(\underset{[4]}{[wu/wc]} + \underset{[5]}{E''}))\cdot t$$

where $\underline{wv}$ is the average width of a V instruction, $\underline{\#r}$ is the average number of ML instructions executed per invocation of a semantic routine, and $\underline{E''}$ is the average unoverlapped time needed to execute a ML instruction. The bracketed numbers refer to the steps shown in figure 4: each term in the equation accounts for the time used during the step whose number appears beneath it.

Nominal values for A and D might be one and two respectively, corresponding to the addition of the width of the current V instruction to a "V-program" counter and a look-up in an "op-code" table in Control Store. These values could be even lower, however, given an efficient base machine and a clever encoding for the "op-codes" of V instructions.

Comparison of TA, TB, and TC:

The point of all these time estimates is that Method C is faster than Method A iff $\underline{TA\cdot t - TC\cdot t > 0}$, and faster than Method B iff $\underline{TB\cdot t - TC\cdot t > 0}$. Note that the result of these comparisons is not dependent on either the time needed to access data values (by $\underline{*}$), or on the magnitude of the micro cycle itself (t cancels out immediately). As formulated, however, these equations _do_ depend on the particular source program selected as "typical."

Although it seems fair to expect the "averaged" parameters ws, wv, E, E', E", and #r to be largely independent of the choice for q (i.e., q is "typical" in a meaningful way), there is no reason to expect _absolute_ quantities like #u, #v, and #f to be invariant with respect to q. In order to factor out these q-dependent variables, we define:

$\%f = 100\cdot(\#f/\#u)$ to be the _fault rate_;

and $\phi = \#r - (\#u/\#v)$ to be the _V-overhead_.

Certainly it is not too objectionable to speak of an "average fault rate;" and by considering #u/#v to be the "average worth" of a V instruction (in ML instructions) and #r as the "average cost" of a V instruction (also in

ML instructions), we may also view $\phi$ as being independent of q. In this context, the assertion that q is a "typical" SL program merely means that the estimated values for these "averaged" parameters are valid.

In practice, a reliable estimate for $\phi$ can probably be formed simply by counting the number of ML instructions required per semantic routine in order to establish a "standard interface" for vE. Typically, one such ML instruction will be needed for each operand/parameter that must be passed through this interface, although fewer will be required if individual ML instructions allow a high degree of parallelism (especially with respect to data transfers. Note that in any case ws, wv, E, E', E", #r, %f, and $\phi$ are all "observables" in the sense that their values are easily determined either by direct observation or through simple experiments.

In fact, by substituting $\phi$ and %f for #u and #f in the comparison formulas above, we find that all of the remaining parameters are both observable and independent of the choice of q. Solving for R and %f (as representative "technological" parameters), we obtain the following results:

[R1] Method C will be faster than Method A iff:

$$((\#r - \phi)\cdot[wu/wm] - [wv/wm]) > 0 \quad \text{and}$$

$$R > \frac{A + D + \#r\cdot([wu/wc] + E'') - (\#r - \phi)\cdot E}{(\#r - \phi)\cdot[wu/wm] - [wv/wm]}$$

[R2] Method C will be faster than Method B iff:

$$((\#r - \phi)\cdot\%f\cdot[ws/wm]/100 - [wv/wm]) > 0 \quad \text{and}$$

$$R > \frac{A + D - (\#r - \phi)\cdot(E' - E'' + \$K + \%f\cdot\$J/100) + \phi\cdot(E'' + [wu/wc])}{(\#r - \phi)\cdot\%f\cdot[ws/wm]/100 - [wv/wm]} ;$$

or

$$((\#r - \phi)\cdot\%f\cdot[ws/wm]/100 - [wv/wm]) < 0 \quad \text{and}$$

$$R < \frac{(\#r - \phi)\cdot(E' - E'' + \$K + \%f\cdot\$J/100) - A - D + \phi\cdot(E'' + [wu/wc])}{[wv/wm] - (\#r - \phi)\cdot\%f\cdot[ws/wm]/100} ;$$

[R3] and/or

$$\%f > \frac{A + D + [wv/wm]\cdot R - (\#r - \phi)\cdot(E' - E'' + \$K) + \phi\cdot(E'' + [wu/wc])}{(\#r - \phi)\cdot(\$J + R\cdot[ws/wm])}$$

Conclusions:

[R1] specifies the minimal value of R such that Method C will be preferable to Method A. For existing machines, this limit appears to be near two (i.e., for a "Tucker and Flynn" machine or a series 360 CPU; for a QM-1, the limit is actually *less* than *one* due to the extremely horizontal nature of its micro instructions). This result is essentially based on a code density argument, and since the required value of R is so low, we conclude that emulation will remain faster than the direct execution of a micro surrogate out of Main Store within the forseeable future. Note that this equation provides us with an absolute measure of the quality of a DEL in the sense that if the denominator goes negative, then emulating that DEL will always be slower than the direct execution of a micro surrogate, regardless of the value of R.

[R2] provides us with two equations that define minimal and maximal values for R, respectively. The first equation is applicable when the "fault rate-weighted" time required to transfer a segment from Main Store to Control Store exceeds the time needed to fetch one DEL instruction. If the numerator is positive, this equation defines the minimal value of R such that emulation is faster than a paged or "cached" execution of a micro surrogate; under the current technology, the numerator is *negative*, indicating that the $K factor makes Method C faster than Method B for *any* value of R.

The second equation defines the *maximal* value of R for which Method C is preferable to Method B, assuming that the time it takes to fetch one DEL instruction from Main Store is longer than the "fault rate-weighted" time needed to move a segment from Main Store to Control Store. For the "Tucker and Flynn" machine, using a segment size of twenty ML instructions and assuming a fault rate of 1%, we find that R must be greater than about 20 before we should abandon emulation, while for a 360 series model 65 CPU we must assume a fault rate of 0.25% in order to bring the critical value of R down below 100. These are extremely low fault rates, and certainly unrealistic values for R.

[R3] is derived from the same comparative equation as [R2] (i.e., TB·t - TC·t 0), and specifies the *minimum* fault rate for which emulation is faster than a paged or "cached" execution. The current range of values for R is approximately four to sixteen; given reasonable estimates for #r and $\phi$ (i.e., $(\#r - \phi) > 1$ and $\phi < 4$), the numerator of this equation is negative -- indicating that Method C will be faster than Method B for <u>any</u> fault rate! Essentially, this is due to the fact that Control Store is implicitly the "fastest available" memory device, and hence that \$K must be of the same order as the total time required to fetch and execute a micro instruction.

Hence, we may conclude that Method C is currently superior to either Method A or Method B, and that ML is therefore unambiguously not an "ideal" DEL. Together with our previous comments, this establishes the plausibility of the contention that the "ideal" DEL for a contemporary computing system lies somewhere between its source language and the language accepted by its base machine.

## Acknowledgments:

The author is extremely grateful for the guidance and insight provided during the development of this paper by his advisor, Professor Michael J. Flynn. This research was partially supported by the United States Atomic Energy Commission under Contract AT(11-1) 3288.

## References:

1. [Abrams 70] Abrams, P. S., "An APL Machine," AD 706741, February 1970.

2. [Denning 70] Denning, Peter J., "Virtual Memory," <u>Computing Surveys</u>, vol. 2, September 1970, pp. 153-190.

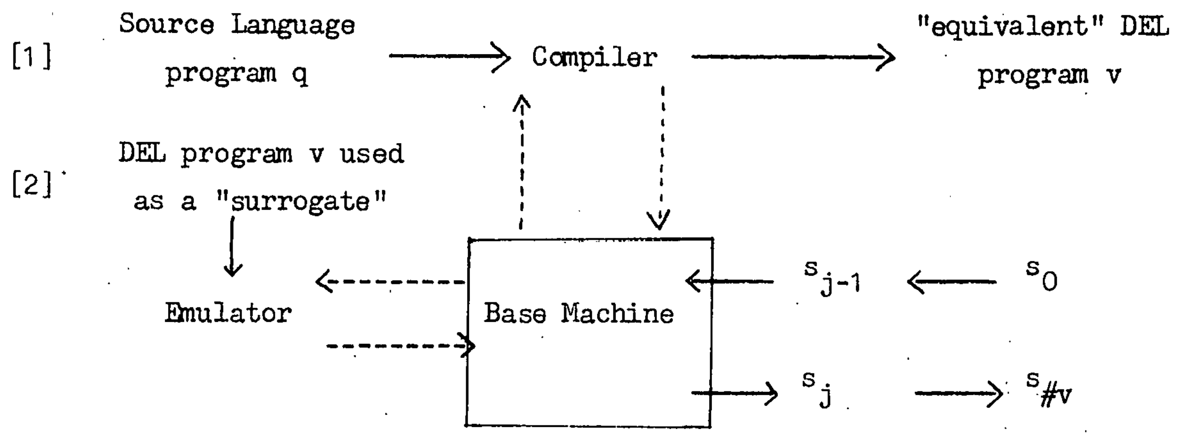
3. [Johnston 71] Johnston, John B., "The Contour Model of Block Structured Processes," <u>SIGPLAN Notices</u>, vol. 6, February 1971, pp. 55-82.

4. [Lawson 68] Lawson, Harold W., Jr., "Programming-Language Oriented Instruction Streams," <u>IEEE Transactions on Computers</u>, vol. C-17, May 1968, pp. 476-485.

5. [Tucker and Flynn 71] Tucker, Allen B., and Flynn, Michael, "Dynamic Microprogramming: Processor Organization and Programming," <u>C.A.C.M.</u>, vol. 14, no. 4, April 1971, pp. 240-250.

6. [Weber 67] Weber, Helmut, "A Microprogrammed Implementation of EULER on IBM System/360 Model 30," <u>C.A.C.M.</u>, vol. 10, no. 9, September 1967, pp. 549-558.

Figure 1: Evaluating a Source Program on a Two-Phased Processing System


[1]
Source Language program q
Compiler
"equivalent" DEL program v
[2]
DEL program v used as a "surrogate"
Emulator
Base Machine
$s_{j-1}$
$s_0$
$s_j$
$s_{\#v}$

This diagram illustrates the relation between the three basic components of a Two-Phased Processing System: a base machine; and two programs which run on the base machine, a compiler and an emulator. A user written program q is evaluated by such a system in two broad steps, or phases:

[1] First, q is translated into an "equivalent" DEL program, v, by the compiler (translation phase).

[2] Afterwards, v is used as a "surrogate" for q, as often as desired, being executed directly by the emulator (interpretation phase).

The input values for a given evaluation of v are determined by the initial store $s_0$ ("store" is used here in the conventional programming language sense to denote a mapping of "program variables" into "values"). Each successive store, $s_j$, is produced as a result of executing the $j^{th}$ DEL instruction, $v_j$, in the environment of store $s_{j-1}$. (for j from 1 to #v, the total number of DEL instructions executed during the given evaluation of v). The sequence $[v_j]$ is called the DEL instruction stream, while the corresponding sequence of stores, $[s_j]$, is called the effect of evaluating v starting in the initial environment of $s_0$. The result of the execution is normally considered to be only the store $s_{\#v}$.

The meaning of "equivalent" depends on how finely the effects of executing v are to be observed: two programs are deemed "equivalent" iff a user can not distinguish between the effects (or results, for "weak equivalence") of their evaluations.

Figure 2: Evaluating a Source Program Directly out of Main Store , ML = DEL: Method A

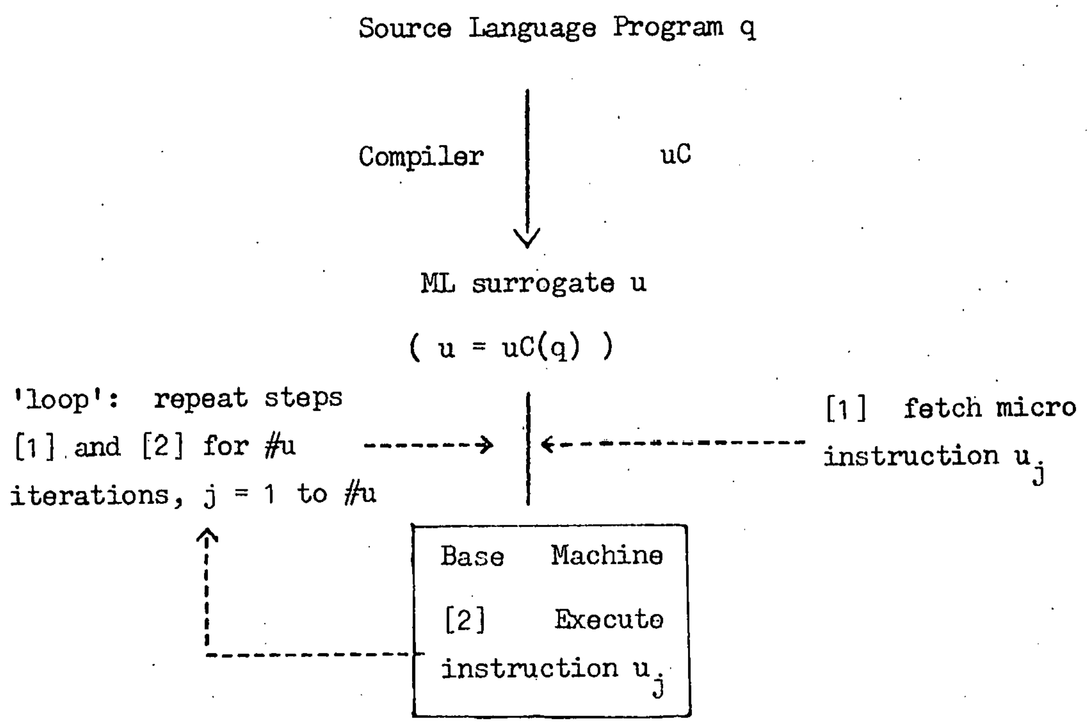

Source Language Program q
Compiler
uC
ML surrogate u
( u = uC(q) )
'loop': repeat steps [1] and [2] for #u iterations, j = 1 to #u
[1] fetch micro instruction $u_j$
Base Machine
[2] Execute instruction $u_j$

Figure 3: Evaluating a Source Program Indirectly out of Main Store , ML = DEL: Method B

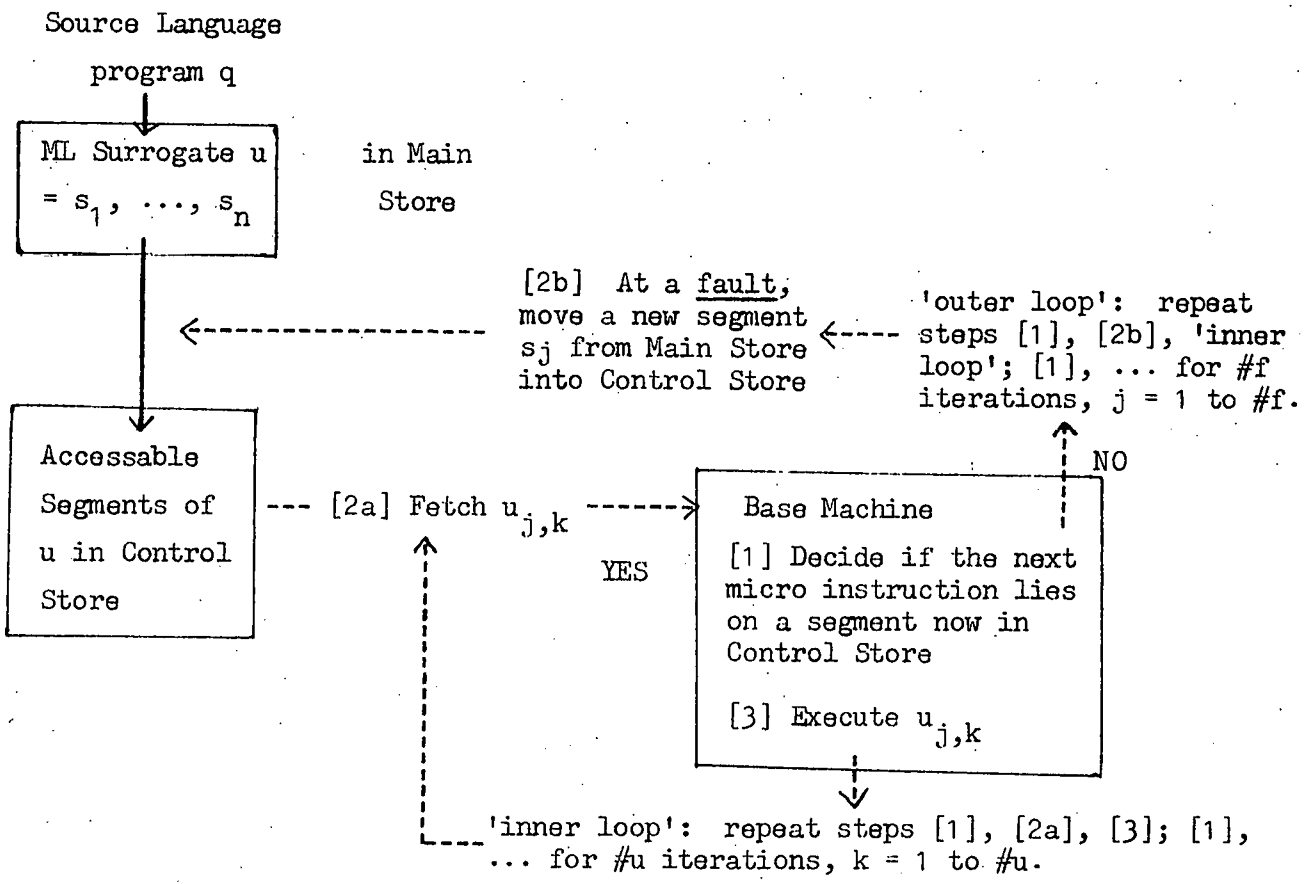
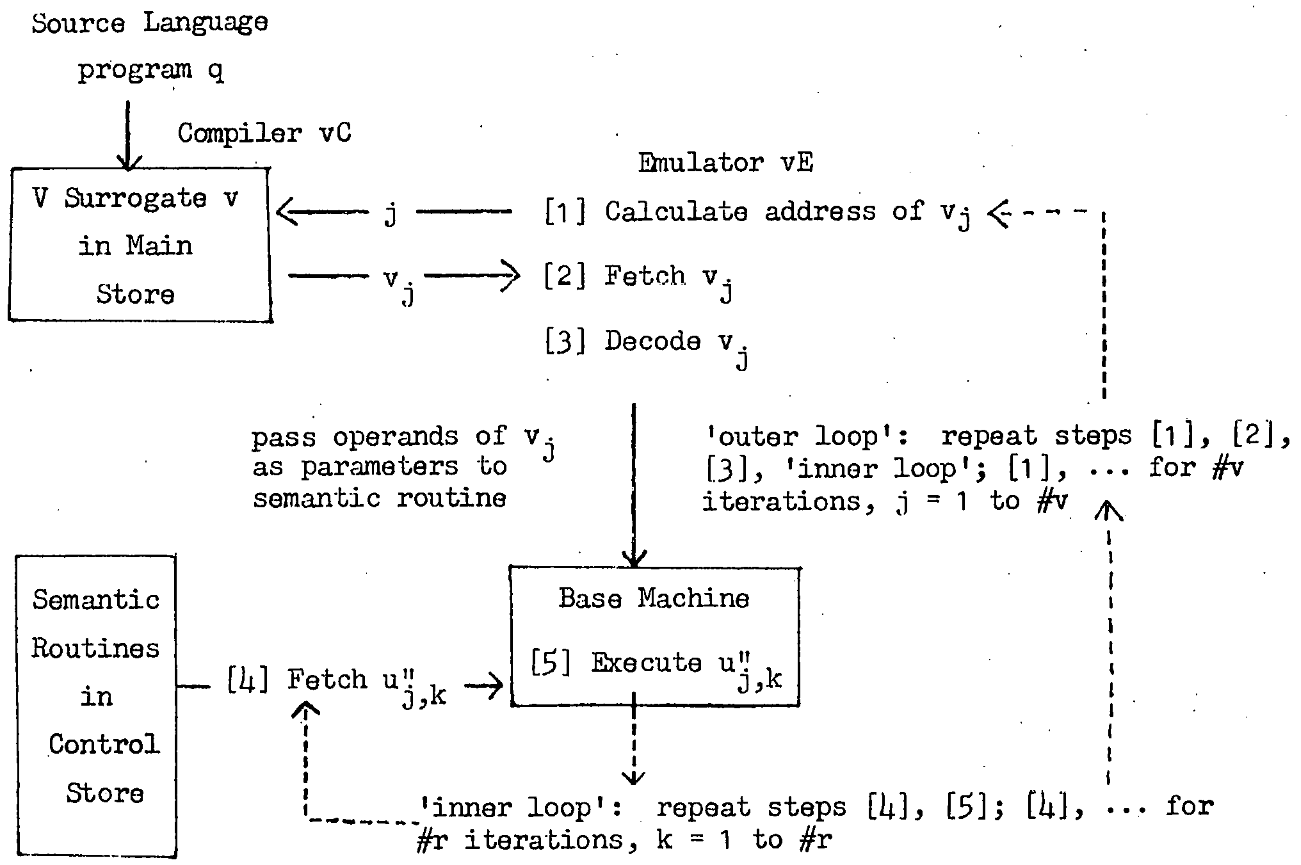

Source Language program q
ML Surrogate u = $s_1$, ..., $s_n$
in Main Store
[2b] At a fault, move a new segment $s_j$ from Main Store into Control Store
'outer loop': repeat steps [1], [2b], 'inner loop'; [1], ... for #f iterations, j = 1 to #f.
Accessable Segments of u in Control Store
[2a] Fetch $u_{j,k}$
NO
YES
Base Machine
[1] Decide if the next micro instruction lies on a segment now in Control Store
[3] Execute $u_{j,k}$
'inner loop': repeat steps [1], [2a], [3]; [1], ... for #u iterations, k = 1 to #u.

Figure 4: Evaluating a Source Program by Emulation ,
V = DEL: Method C



Source Language program q
Compiler vC
Emulator vE
V Surrogate v in Main Store
j
[1] Calculate address of $v_j$
$v_j$
[2] Fetch $v_j$
[3] Decode $v_j$
pass operands of $v_j$ as parameters to semantic routine
'outer loop': repeat steps [1], [2], [3], 'inner loop'; [1], ... for #v iterations, j = 1 to #v
Semantic Routines in Control Store
[4] Fetch $u''_{j,k}$
Base Machine
[5] Execute $u''_{j,k}$
'inner loop': repeat steps [4], [5]; [4], ... for #r iterations, k = 1 to #r