COMPUTER SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY · STANFORD, CA 94305

SU-SEL 79-011

SU-326-P.39-31

A THEORY OF INTERPRETIVE ARCHITECTURES: IDEAL LANGUAGE MACHINES

by
Michael J. Flynn
and
Lee W. Hoevel

February 1979

TECHNICAL REPORT NO. 170

The work described herein was supported in part by the Army Research Office—Durham under contract no. DAAG29-78-G-0205, using laboratory facilities developed with the support of the Department of Energy under contract no. EY-76-S-03-0326-PA 39.

SU-SEL 79-011 SU-326-P.39-31

A THEORY OF INTERPRETIVE ARCHITECTURES: IDEAL LANGUAGE MACHINES

by

Michael J. Flynn

and

Lee W. Hoevel

February 1979

TECHNICAL REPORT NO. 170

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

The work described herein was supported in part by the Army Research Office - Durham under contract no. DAAG29-78-G-0205, using laboratory facilities developed with the support of the Department of Energy under contract no. EY-76-S-03-0326-PA 39.

ABSTRACT

This paper is a study in ideal computer architectures or program representations. An ideal architecture can be defined with respect to the representation that was used to originally describe a program, i.e. the higher level language.

Traditional machine architectures name operations and objects which are presumed to be present in the host machine: a memory space of certain size, ALU operations, etc. An ideal machine framed about a specific higher level language assumes operations present in that language and uses these operations to describe relationships between objects described in the source representation.

The notion of ideal is carefully constrained. The object program representation must be easily decompilable, (i.e. the source is readily reconstructable). It is simply assumed that the source itself is a good representation for the original problem, thus any nonassignment operation present in the source program statement will appear as a single instruction (operation) in the ideal representation. All named objects are defined with respect to the natural scope of definition of the source program. For simplicity of discussion, statistical behavior of the program or the language is assumed to be unknown; that is, Huffman codes are not used.

From the above, a canonic interpretive form (CIF) or measure of a higher level language program is developed. CIF measures both static space to represent the program and dynamic time measurements of the number of' instructions to be interpreted and the number of memory references these instructions will require. The CIF or ideal program representation is then compared using the Whetstone benchmark in its characteristics to several contemporary architectural approaches; IBM 370, Honeywell Level 66, Burroughs S-Language Fortran and DELtran, a quasi-ideal Fortran architecture based on CIF principles.

KEY WORDS

architecture comparisons

canonic interpretive form

directly executed

high level language machines

program representation

INTRODUCTION

Obvious inadequacies of present machine architectures, both in program size and execution time pose the problem of representing programs for direct interpretation [1,3,11]. Secondary effects lead to complicated system structures and implementations, e.g. compilers, linkage editors, as well as difficulties in recognition and exploitation of parallelism [13]. The traditional premise is that any executable machine architecture must be a fixed and, hence, universal language. The premise of this paper is that this forces interpretation to occur at too low a level and places too great a burden on translation, limiting the efficiency of a system.

Assume that programs are initially expressed in a higher level _source language_ (HLL), catering to both the user and his problem. However, programs must ultimately be evaluated by a lower level processor -- the system's _host machine_. It is in the host machine that state transitions over HLL named resources actually occur. Once the source language has been selected the issue becomes one of determining the most suitable intermediate language (or instruction set) for the system -- called its _directly executed language_ (DEL) and a suitable host machine for interpreting this DEL. It is important that this intermediate language preserve as much information concerning the user environment and original source program structure as is useful in realizing concise representation and expeditious interpretation (Figure 1).

Abstract Algorithm

[User]

Source Program (in HLL)

[Compiler]

Intermediate Surrogate (in DEL)

[Interpreter]

Individual DEL Instruction

[Execution Semantics]

Host State Transitions

Figure 1: EVALUATION PROCESS

TWO PHASED EVALUATION [7]

The basic model used in this investigation is illustrated in Figure 2. Its most obvious characteristic is that program evaluation is assumed to take place in two distinct phases. First, a source program is converted into an equivalent executable program during an initial compilation phase. Users retain this executable program, which becomes a surrogate for the original source version. In the second phase this surrogate undergoes any number of subsequent interpretations. The user must visualize the effects of interpreting a DEL program in terms of the semantics associated with its original source level representation.

Two phased-evaluation may be used as the basis for a design model. The principal components of a system in this model include: a source language -- selected for its representational capabilities; a host machine -- selected for its execution capabilities; a translator that takes a source program as input and produces a logically equivalent executable program; and finally an interpreter that enables the host machine to implement the state transitions specified by the output of the translator.

In a traditional design, the interface between the translator and interpreter (called the "machine language") is in fact a universal, host-oriented language. Its structure is largely defined by artifacts presumed to be in the host machine, i.e., a certain number of registers, memory cells, or arithmetic operations. The form of a traditional machine language corresponds closely to the form of the underlying host machine, so that interpretation of individual instructions requires a minimum number of state transitions in the host. This approach is limited since both the host and DEL are fixed and cannot adapt to specific requirements of the source language and user environment.

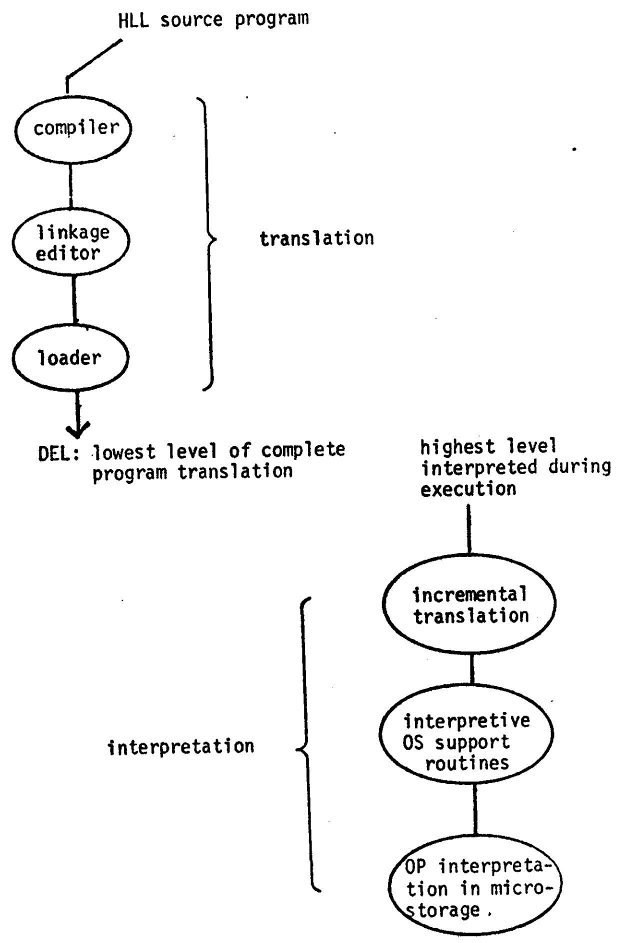
In practice, since many different user environments and corresponding source languages must use the same host/interpretation combination a good deal of complexity is forced upon a translator or compiler. The translator must produce a program representation in a fixed, universal language -- regardless of various perturbations in the source language and corresponding environment.

As an alternative, an execution architecture may be constructed to be an ideal representation with respect to the given source language and environment. The term "ideal" is used here to mean (1) transparency, i.e. translation is a simple process which preserves equivalent source information thus allowing a ready reconstruction of source contructs and (2) DEL optimality -- space to represent and time to compile and execute (interpret) are minimized. Since the DEL language will now differ as environments change, the interpreter and host machine must be more sophisticated. For example a host might be designed specifically as an efficient interpretive engine, rather than a general purpose computer. The advent of improved technology, especially fast read/write storage that can be used to contain the interpreter and its immediate execution environment, has enhanced the ability to construct adaptive, interpretively directed hosts, and forms a premise for this alternative design approach.

Translation

Translation is the process of converting a program in one language into an equivalent program in another language. Equivalence refers to similarity of transformation in the eyes of a user -- a program p in language P is equivalent to a program q in language Q if and only if users cannot distinguish between an execution of p according to the semantics of P, and an execution of q according to the semantics of Q.

Figure 2: TWO PHASED EVALUATION


HLL source program
compiler
linkage editor
loader
translation
DEL: lowest level of complete program translation
highest level interpreted during execution
incremental translation
interpretive OS support routines
OP interpretation in micro-storage.
interpretation

The virtue of compilers is that they translate a source language program into a form that can be interpreted more efficiently. This means that some sort of reduction in computational complexity takes place. In general, compilation can be viewed as a (partial) binding of operands to storage cells and operators to computational structures. The first binding is maintained in a symbol table, the second in a Program Tree -- Open Macro Definition (OMD) tree in the terminology of Elson and Rake [2].

The symbol table maintains a mapping of the source name space into the DEL name space, while the program tree maintains the syntactic structure of the source program. The macros associated with the non-terminal nodes of this tree are actually functions that generate short sequences of DEL instructions implementing the appropriate source operator. Auxiliary data structures may also have to be maintained during translation in order to produce minimal code sequences (i.e., perform optimization).

Intuitively, by requiring a direct one for one mapping of HLL operations and identifiers into DEL instructions and operands and by assuming optimum HLL source form (no further optimization) the fastest possible compilation (exclusive of none at all) is realized. The trival case of direct interpretation of HLL source is known to be inadequate for for most environments because of the time consuming and redundant process of binding HLL identifiers to values.

Interpretation

Interpretation is the process of executing the actual computations defined by a program without further changes in its overall representation. Each DEL instruction defines a specific transformation to be applied to the current state image within the host machine. The interpreter implements this transformation and passes control to the next DEL instruction to be interpreted (Reigel [12]).

The interpreter (Fig. 3a) may be visualized as consisting of a primary control loop, a set of interface routines, and a set of semantic routines. The primary control loop maintains the DEL instruction stream, and parses at least one subfield of each instruction. The encoded symbol in this subfield defines the layout of the rest of the instruction (the format for the immediately following bit stream), as well as what to do with these fields (the semantic structure of the instruction). The primary control loop then transfers control to the appropriate interface routine -- which actually parses any operand references within the instruction.

Interface routines are responsible for creating a standard interface that defines the location (and possibly the value) of each operand in an instruction. The interface routine then returns to a known point in the primary control loop, which transfers control to the semantic routine that actually implements the action rule defined by the instruction (Fig. 3b). It does this by transforming the values previously loaded into standard interface. Upon completion of all semantic processing, control returns to the top of the primary control loop, and another cycle of interpretation begins. Results may be stored by the semantic routine itself, or within the primary control loop.

Within control loop, interface, or semantic processing, the state of the DEL data store and program state vector is, of course, temporarily undefined. Definition occurs at the boundaries; the overall process will be correct if the DEL data store and program status vector agree with defined constructs whenever the last DEL instruction in the expansion for a source statement has been executed. Note that as in pipelined machines the interpreter need not process the DEL serially, even if the original source program is itself serial.

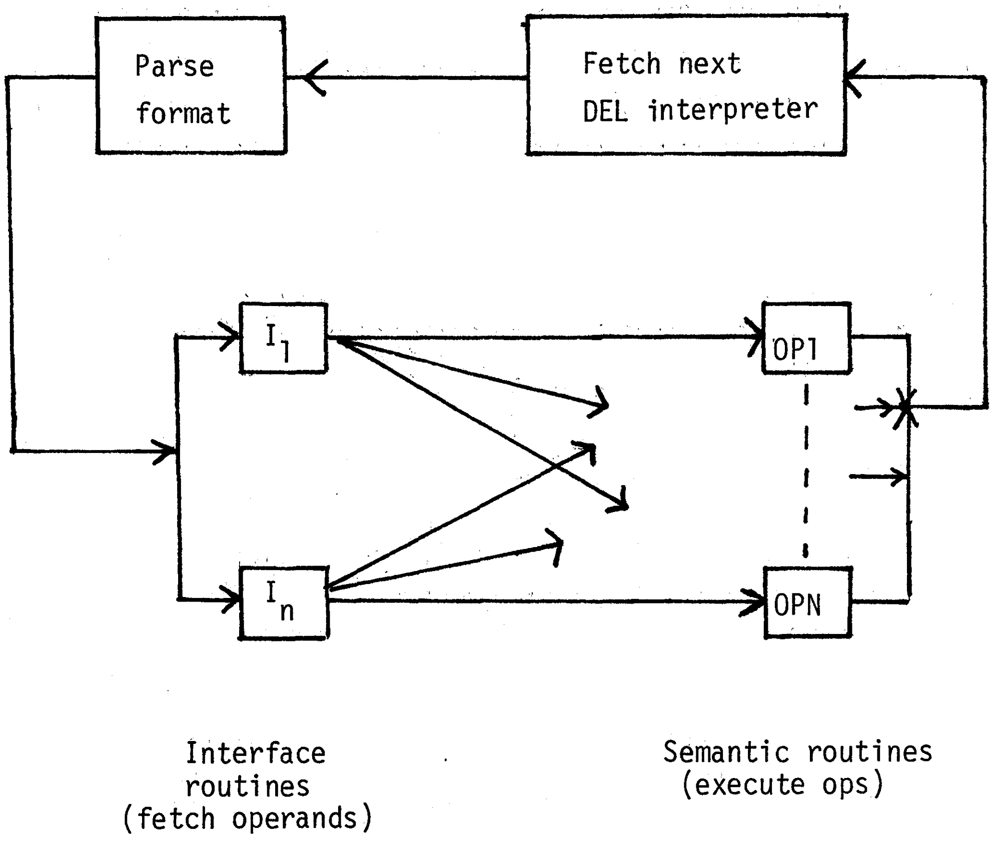

Parse format
Fetch next DEL interpreter
I_1
I_n
OP1
OPN
Interface routines (fetch operands)
Semantic routines (execute ops)

Figure: 3a

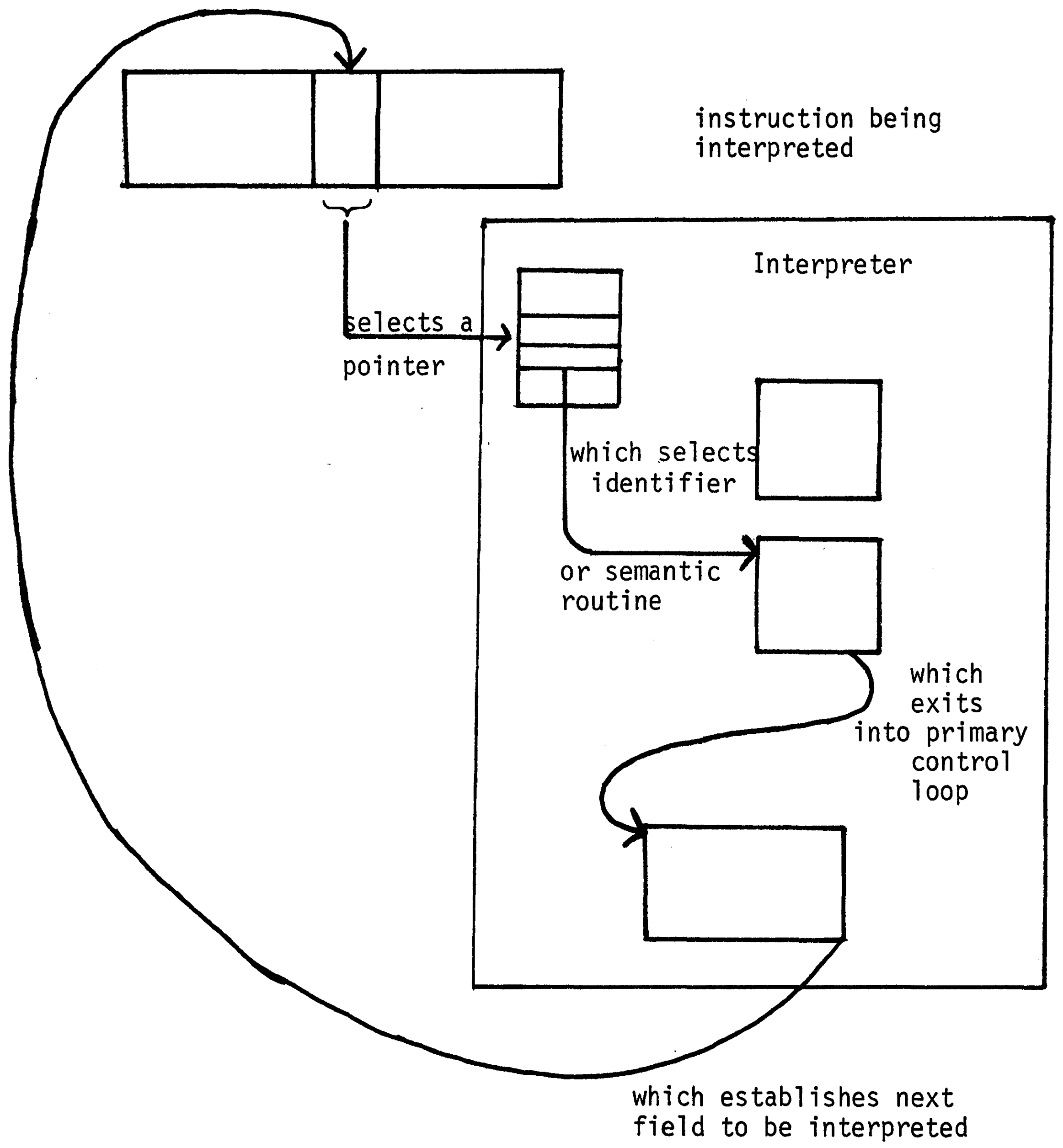

instruction being interpreted
Interpreter
selects a pointer
which selects identifier
or semantic routine
which exits into primary control loop
which establishes next field to be interpreted

Figure 3b

FUNDAMENTAL CONCEPTS

At this point, it is useful to examine a number of abstract concepts related to the mechanistic model developed above.

Identifiers, Objects and Name Spaces

Programs at any given level in an evaluation hierarchy use identifiers as surrogates for objects at that level. Objects -- arguments or results -- are only associated with values during execution, as defined by the states of a computation. In this sense, an identifier is only a specific instance of the abstract name for an object; the name exists independently of a language, while the identifier is closely tied to a particular syntax and semantics (Fig. 4).

Typically, source level identifiers are mnemonically selected alphanumeric strings. DEL identifiers are usually one, two, or three dimensional binary codes (e.g. file, segment, offset). At the host level, identifiers are simply physical addresses. Typical source level objects are integer variables, program labels, and boolean flags; typical DEL level objects are words, bytes, and bits; and typical host level objects are registers, busses, and execution units.

Classification

Name spaces can be classified according to various abstract characteristics -- e.g., range, resolution, and homogenity.

Range and resolution refer to the maximum number of objects that can be specified in a name space and the minimum size of an object in that name space respectively. Traditionally, instruction resolution is no smaller than an 8 bit byte (frequently it is based on a 16 bit

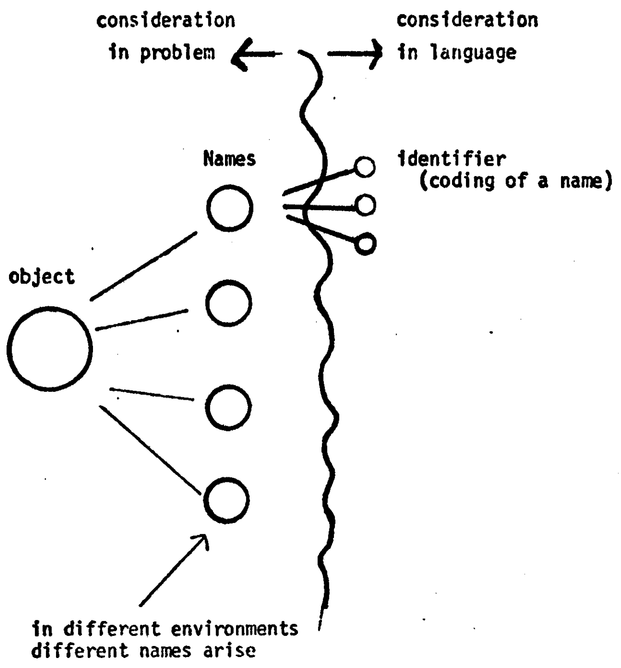

consideration in problem
consideration in language
Names
identifier (coding of a name)
object
in different environments different names arise

Figure 4: OBJECTS, NAMES AND IDENTIFIERS

-- or larger -- word), and range is defined as large as can be comfortably accomodated within a reasonable instruction and program size. Thus, the range of 2^{16} for minicomputers to 2^{24} for System 360/370 covers most common arrangements (Fig. 5).

The range of the name space directly accessable to a host is necessarily bounded, and access to large data bases must be provided. Note that I/O is a general mechanism for attaching external objects to the DEL name space. Usually, archived data must be physically moved into the current access environment before it can be operated on by

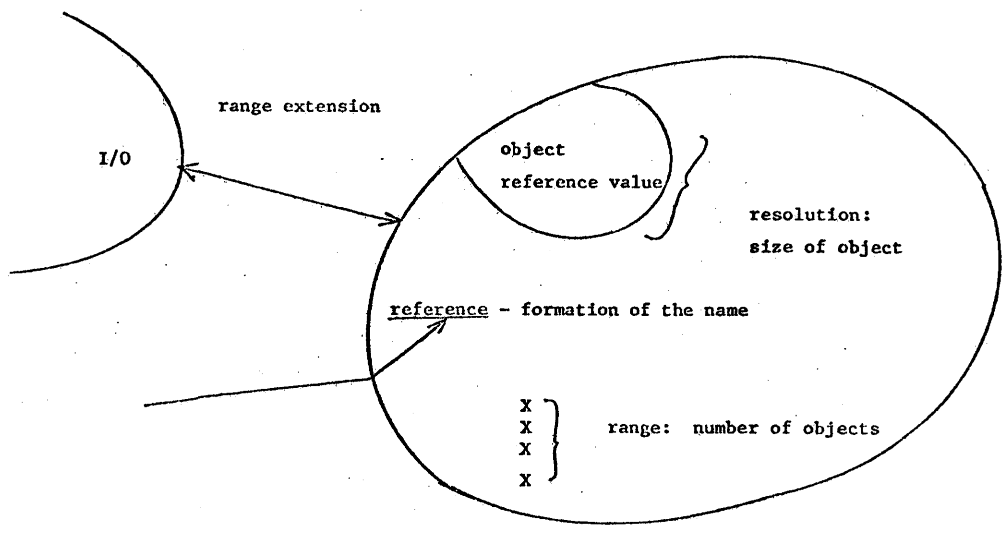


Figure 5: PROCESS NAME SPACE

DEL instructions. Commands performing this data movement must be carefully synchronized in order to maintain data integrity and avoid exceeding various internal limits (e.g., overflowing a buffer). Using a distinct, interpretive sub-system to manage I/O activities is one way to factor external range and resolution constraints out of the primary instruction stream. Additional performance may also be obtained by aggregating conversion and formatting routines within a single locality.

Name spaces may be partitioned in many different ways. Action rules do not generally treat all objects in the same way, and this in itself creates distinct classes -- registers, accumulators, and generic memory cells. Action rules are often applied in a non-symmetric manner: one argument must be a register, while the other may be either a register or a memory cell. The term homogeneity refers to partitions distinguished by the domain of the action rule. Partitioning is justified by performance -- the underlying assumption being that access to registers is faster than access to memory.

Many familiar machines have their name space partitioned into a register space and a memory space: e.g., the 360/370, PDP-11, and NOVA architectures. As the partitioning of the name space increases, its homogeneity decreases. The undesirable effects of a fragmented name space are not always obvious, since the name space for most simple examples -- i.e., small programs -- can be captured within the register set without great difficulty. It is only when the problems of automatic translation for large programs is considered that homogeneity becomes important, as in the specification of standard linkage and addressing conventions.

Scopes and Working Sets

The scope of an identifier is the largest program fragment over which it has a consistent interpretation. These fragments are usually highly correlated; indeed, for most high level programming languages, the term scope of definition is used to refer to a given lexical block (procedure, subroutine, function, or begin-end segment) associated with a given level of declarations.

At the machine level, however, the natural interpretation of a scope is as a range of instructions over which indexing registers remain effectively unmodified -- so that a given operand identifier always refers to the same program object within a given scope. The term is often used imprecisely, however. The "scope of a procedure" may refer either to the set of statements outside the definition of a procedure in which its own name is a valid identifier, or to the set of statements immediately inside a procedure definition in which identifiers local to that procedure have consistent interpretations.

Similarly, the working set of a process is normally defined to be the set of objects corresponding to the arguments or results of action functions applied over a relatively short span of state transitions in a computation.

The correlation between the notion of scope and working set is clear enough. The working set represents a series of spatial localities that have been recently referenced by a particular program. These localities of data references must be contained in the scope of definition for the portion of a program currently being executed, however, and thus the current scope of definition logically subsumes the current working set. Intuitively, the difference between the scope of definition and working set is small (there are few potentially referencable items that are not, in fact, actually referenced during execution). The working set is an implicit way of identifying data items, whereas scope is a logically explicit identification of (nearly) the same data items.

Level Mappings and Transparency

Users relate the observable (but low level) effects of executing a program to source level semantics through an association established between the source level name space and the host name space. The complexity -- and accuracy -- of this mapping determines the ultimate transparency of the system. In general, there is always a map from any higher level L to the next lower level L-1 that defines the way in which level L is realized at level L-1. There is also a dual map from level L-1 to level L that defines how an execution of a level L-1 program is to be visualized in terms of level L semantics. In a transparent system these maps are mathematical inverses (Fig. 6).

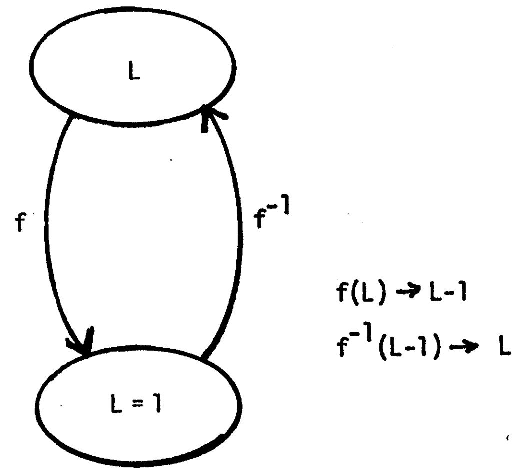


Figure 6: TRANSPARENCY BETWEEN A LEVEL REPRESENTATION AND A LEVEL L-1

CONVENTIONAL ARCHITECTURES

Much research has been focused on whether the best program representations are based on a stack, single accumulator, two address, or three address organization. The premise is that a single, "best" execution architecture can be uncovered by exhaustive statistical analysis of benchmark experiments. Unfortunately, slightly different assumptions lead to radically different conclusions and in general this type of analysis suffers without a unifying, theoretical justification. Each of the instruction organizations mentioned above, as well as heretofore untried combinations, will have some advantages in specific situations. The problem lies in using a single organization as the basis for program representation in a universal environment.

In order to quantify the degree of "overhead" forced into program representations in a universal environment, studies [3,9] have been made which attempt to separate instructions related directly to the original expression of an algorithm in the high level source language, and those which appear to have been introduced because of architectural constraints within the DEL. For example, load or store instructions are almost always included because of requirements imposed by the execution architecture. These commands merely move items around in the DEL name space -- which is not, in general, partitioned according to any sensible division of the name space for the original high level language representation of a program. This has led to the notion of three different generic classes of instructions:

1) Functional type (F-type) instructions that actually

operate on and/or transform data values. It is assumed that these instructions actually correspond to operations in the higher level language representation of a program.

2) Memory type (M-type) instructions that rearrange items without changing their value within a memory hierarchy.

3) Procedural type (P-type) instructions (branch and compare) that alter the sequencing rule during interpretation, but do not change data values.

If we assume that only the functional instructions are absolutely necessary to a program representation and define ratios of M and P type to F type instructions we arrive at some interesting overall statistics for current machines (Table 1). From this table, it is evident that two or three memory type instructions are required for each functional instruction. Notice that the introduction of general purpose registers into the 360/370 architecture (as opposed to the single accumulator, multiple index register complement of the 7090/7094) did not reduce this ratio, but rather these extra objects created more memory type instruction overhead.

Processor	"Ideal"	7090	360	PDP-10 [9]
M-ratio	0.0	2.0	3.9	1.5
P-ratio	0.0	0.8	2.5	1.1
NF-ratio	0.0	2.8	5.5	2.6

Table 1: OVERHEAD RATIOS

IDEAL MACHINES

An ideal program representation and corresponding machine uses minimum storage space and requires minimum compilation and interpretation time. This does not imply a linear tradeoff between space and time; relative weights can only be determined within a given user context.

The notion of environment, however, is fundamental to the discussion of space time optima. Like many concepts dealing with computer systems, environment may be viewed as a hierarchial concept. The chosen higher level language itself represents the highest level, while a specific program would represent a lower level, and individual statements within a program would represent a still lower level. Environment includes all residual control information needed to interpret objects; i.e., all information implicitly available to an interpreter, as distinct from information supplied directly and explicitly by the specific coding of a statement or instruction. Thus, the environment of a traditional machine language would include a program counter, address registers, interrupt status, etc.

Associated with each level is a property of stability. Stability of environment at any level is probably best measured in terms of the number of statements or objects that must be interpreted before the environment is changed. A change of environment is caused by anything that disturbs the interpreter or prevents it from completing the interpretation of a program, statement, or operator. Instabilities arise from one of two basic causes. First, exogenous events outside the current environment -- a time sharing system, for example, may time slice over short intervals, and each program switch would necessitate a change in environment. Second, internal events associated with the nature of a program itself may cause a change in environment -- for example, a single user program may involve several languages, or may defer binding between names and values until very late (as in the binding of actual to formal parameters upon entry to a subroutine). In this case, the act of binding the interpreter to a given language, or of binding a name to a given value, will also cause a change of environment.

For each level of environment there is an associated scope of definition. Syntax and semantics are the scope of definition for high level languages. The scope of definition of a program includes its various subroutines, variables, etc. The scope of definition of an individual statement consists of just those variables and operators involved in its evaluation.

All objects within an environment have names, whether these objects are a language, a program, a statement, an operation, or an operand. The ideal machine model assumes that a compiler will make at least one pass over its input to identify these names, and associate them with some static interpretation. As much information about a source program as is easily available is to be extracted during this (and other) pass of the compiler, and may be used to optimize the program at either the source or DEL level.

Aspects of Program Representation

A program representation is in fact an abstract machine that usually corresponds to either higher level language (as in DELs) or to host machine (as in traditional instruction sets) entities. In attempting to define an ideal representation three criteria form the premises for our discussion:

1. Transparency -- that the program representation correspond closely to the source representation.
2. Size -- that the program be as concise as possible, i.e. a minimal encoding.
3. Number of Required References -- that the program representation requires as few host machine state transitions as possible, both for its creation (another aspect of transparency) and for its interpretation.

It is useful then to explore these criteria in a variety of environments. The distinction between traditional machine languages and the DEL approach will become clear by this examination.

Within an environment the basic issue of definition is: to what do identifiers refer. In general they may refer to:

1. objects in a particular program,
2. objects in a higher level language, and
3. resources or objects in a host machine.

Clearly, when identifiers correspond to objects in a particular program a very special purpose machine is defined. Identifiers corresponding to objects in a higher level language define a language oriented machine and identifier corresponding to host machine entities define traditional machines. From our point of view it is clear that we wish to choose a correspondence as close as possible to higher level language objects in the specific programs.

Correspondence Environments

The transparency requirement defines two correspondence environments, one for operations and one for names.

An "operation" in an executable program representation could be defined by (i.e. correspond to):

(1) program

(2) subroutine

(3) HLL operation

(4) host operations.

For higher level language oriented interpretive representation HLL operations are the most interesting correspondence environment. Note that traditional machines usually employ host operations as their correspondence environment, while special purpose machines select certain subroutines or even a program for their operation correspondence.

Name Correspondence Environment

To what should an operand or label identifier correspond? The choice here is between HLL name objects and objects named in the host. Again, for an ideal HLL machine the name object should correspond to the objects named by the higher level language including an exact correspondence in data structure as well as occurrence. This also implies that instructions may not introduce additional name identifiers to contain temporary values. This is an interesting restriction and will require a powerful format set for realization.

Thus, an ideal HLL machine would have exactly one instruction (transparency requirement) for each nonassignment operation used in the HLL source. Associated with each operation would be no more name identifiers than those which were used in the HLL source. This intuitive notion will be refined later.

<u>Size Environment</u>

Conciseness of representation is important not only as a measure of the cost of storage for the representation but also a measure of secondary effects on time for interpretation or translation into a program representation. An ideal representation must be concise in its coding of identifiers yet not so concise that it exacerbates interpretation. The number of identifiers in a program representation is defined by the correspondence environments. Thus, the total program size is in reality determined by the size selected for the three basic kinds of identifiers:

(1) operators,

(2) operands,

(3) labels (the operator for a procedural operation).

The issue of identifier size is the determination of the number of objects that must be distinguished by a particular identifier since $\lceil \log_2$ (number of objects)$\rceil$ determines the field size. This ignores variability since an identifier may be:

(1) fixed in size across all or many environments,

(2) variable by environment,

(3) variable by frequency within an environment.

In this work we will not further consider frequency encoded identifiers since:

(1) frequency statistics must be a priori available,

(2) given such statistics the techniques for taking a nonfrequency encoded scheme and transforming it into a minimal encoding is straightforward and treated elsewhere [5,8],

(3) minimal encoding by frequency requires serial inspection of the bits within a field, thus, increasing the complexity of interpretation and, hence, the interpretation time.

Further, while these disadvantages may in fact be outweighed by the advantages of more concise representation, the introduction of minimal encoding would needlessly complicate this discussion. There is really no distinction between fixed and variable approaches since a fixed identifier size implies merely that the environment is fixed .

Again the environment may be determined by either the program, the HLL or the host. Thus, for each class of object (operation, operand, or label) any of the following may be used to define the number of entities which will determine the identifier size:

(1) Operands and labels are not usually bounded in number by a language syntax in a meaningful way. However, the number of operations are usually a priori limited and the number of HLL objects allowed in the HLL definition as the

operation size is a possibility.

(2) HLL Objects Used in a Program -- i.e., the total number of distinct operations, operand names or labels used in a program, could form the domain of an identifier definition.

(3) HLL Objects Used in a Subroutine -- it is assumed here that a program consists of a number of subroutines. Each subroutine has its own scope of definition. Thus, this might be an interesting size environment for operand and label identifiers.

(4) The HLL Statement -- The even lower level concept of a statement for an environment might also form the basis of a size environment. However, since entry into and exit from an environment requires interpretation time, the statement may be at too low a level to provide an optimum space time tradeoff. That is, at the level of the statement the setup time required may not offer a worthwhile space time tradeoff since the size of the identifiers increase as a log function while the interpretation time is linear in the number of statements to be interpreted.

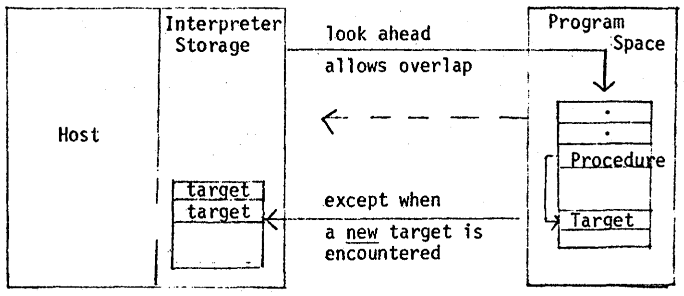
Note that at the subroutine level the set up time can be regarded as being relatively small compared to the interpretation time for the overall subroutine while at the statement level this may not be true.

(5) All host objects may form the basis of the domain of the identifier environment. In fact, traditional machines commonly use this as their basis. Since this environment does not change, the identifier containers are also fixed. The identifier may be either one dimensional or multidimensional. In the one dimensional case all operations or operand names, etc. form the basis of the domain which will define the identifier field. This is most frequently true in identifying operations. In the multi-dimensional situation the identifier name is decomposed into several component identifiers, e.g. for an operand name index, base, and displacement and even for operations one might have format, function, data type.

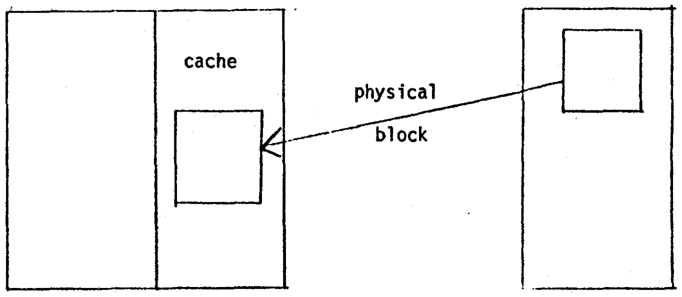
From the above, an ideal HLL language has a size environment whose domain is related to the number of HLL objects used in either the representation, the program, the subroutine or the statement. Because of implied interpretive overhead, the statement level does not seem to be a good choice and the entire HLL definition limitation seems equally inefficient in its lack of conciseness. Intuitively it appears that the subroutine level is a natural space time optimum.

Referencing Environments

While the effects of referencing activity on interpretation time are implied by the number of identifiers in the program representation, these effects should be considered in a more direct manner. References to a program name space arise from either an instruction or a name reference. Accesses in the former case reference the instruction space while the latter reference the program name space. The reference environment corresponds to the definition objects or their characteristics which require an access to program space. Depending on the sophistication of the interpreter and homogeneity of this space, a number of sub-referencing environments may also be defined (e.g. references to register space) (Fig. 7).



(a) Instructions - DEL Oriented



(b) Instructions - Host Oriented

Figure 7: REFERENCING ENVIRONMENTS

Instruction Referencing DEL Oriented

(1) The Instruction Operation -- the most obvious environment would correspond to a reference per operation (op-code) interpreted.

(2) Procedural Operations -- less obvious, though frequently used in an equivalent way, is a look-ahead mechanism of traditional machines which proceeds to fetch instructions in anticipation of their interpretation until a procedural operation is encountered. Under such a regime only procedural instructions force an unanticipated additional reference since all others lie in sequence and their reference is overlapped with the interpretation of the intervening instructions.

(3) Destination Capture -- An extension of (2), the first incidence of a procedural instruction forces a reference; however, the target instruction of the branch is captured (i.e. stored) in a buffer for subsequent use, avoiding additional references to program storage.

(4) The Subroutine -- allows one reference per entry to or from a subroutine. The subroutine represents a natural locality, and block transfer mechanism currently available with host oriented cache could be used to capture the program representation for a subroutine. Since the program representation is presumed to be reentrant only one fetch

would be required presuming that the subroutines are somewhat restricted in size.

(5) The program -- higher level referencing environments (such as physical access the whole program) do not seem to be meaningful in familiar terms.

<u>Instruction referencing - Host Oriented</u>:

(1) physical instruction word in the host frequently defines an instruction fetch in traditional machines. If the physical word is less than the instruction unit, multiple fetches may be required; if it exceeds the instruction, less than a single fetch is required per instruction.

(2) since instruction prefetching can continue to a branch point, prefetching is frequently employed in higher speed machines at least up to the point that a conditional branch is decoded. At that point some penalty is envoked akin to a reference to a program space. While a number of strategies have been discussed to minimize this penalty, they are not completely successful and beyond the scope of our treatment here.

(3) branch target buffer - As in the DEL case the target of a branch instruction can be stored in an associative buffer (e.g. MU-5) avoiding additional referencing activity.

(4) the program reference locality, when reference is made to

a location in program space in physical host oriented system with a cache, the target instruction is fetched together with the block in which it is contained. This is similar to our subroutine referencing in HLL oriented representations. Clearly the ideal HLL machine should have a referencing environment related to the HLL. Depending upon the nature of the higher level language the reasonable choice of ideal program representation would be based on either an allowance of a reference per procedural operation or per subroutine.

In a DEL oriented ideal machine use of (2), procedural operations, as a referencing environment would correspond to assuming the capability of a simple host. A more sophisticated host could implement either (3) or (4); however, since (3) - destination capture - seems somewhat more modest in its host requirements, we will rather arbitrarily regard it as preferable. This strategy also avoids excessive transfer requirement from program store to interpreter store.

<u>Name Space Referencing Environment</u>

As before, referencing environment can be oriented toward the higher level language source representation of the resources of the host.

<u>Name Referencing -- DEL Oriented</u>

(1) One reference per unique identifier. If an identifier is used as both the source and the destination then a read

and write must be allowed into the program name space. Implied operands are assumed to be captured in a high speed (no access required) storage.

(2) One reference per subroutine/scope. Just as in cache based traditional systems, locality is an important attribute in HLL machines. If the entire name space of the subroutine can be captured in a high speed buffer storage with a block oriented transfer mechanism then it seems reasonable that only an entry into the subroutine would require the overhead of a name space reference. An additional reference must be made to restore to the program name space. While this is conceivable for referencing names whose values are known on entry to the subroutine, it cannot be used to reference names whose values have not yet been computed and, an additional reference must be allowed to account for each computed name reference.

Data Referencing -- Host Oriented

(1) Host physical word. An object in the representation may point to a physical memory as in traditional machine languages. An effective address defines an object whose contents is the value and the entire word is referenced for interpretation.

(2) Host localities. Each access is to a block of data containing the required object. The referenced block of data

consists of a number of physical words which are brought into a buffer, thus, minimizing the need for additional accesses to the much larger (and presumably much slower) program name space. As in the DEL case, computed names may cause additional references since these lie outside the selected blocks.

In a DEL oriented ideal machine the second alternative seems preferable. This means that one reference is allowed per read, one read reference and one write reference is allowed per entry into a subroutine, and one additional reference is allowed per computed name. This corresponds to an idealization of the traditional cache used in conventional machines.

Canonic Interpretive Forms

The following is a proposed measure called the Canonic Interpretive Form, or CIF. It is an attempt to define the behavior of an ideal, DEL oriented machine. The transparency requirement defines a correspondence property affecting both space and time. Program size is determied by transparency and environment. From the earlier discussion the subroutine or lexical scope is the most natural environment for this measure. Interpretation time depends both on the number of objects interpreted and the number of unanticipated references to program space. In selecting reference activity environments, the intent is to parallel the state of the art in traditional design.

1:1 CORRESPONDENCE PROPERTY

<u>Instructions</u> -- one CIF instruction is permitted for each functional operator used in the source representation.

<u>Name Space</u> distinct -- one CIF identifier is permitted for each unique identifier used in a source statement.

LOG$_2$ SIZE PROPERTY

<u>Operators</u> -- CIF operator identifiers are of size $\lceil \log_2(F) \rceil$, where F is the number of distinct operators used in the operator naming environment. The operator naming enviornment is taken to be the scope: i.e. the subroutine or function level.

<u>Operands</u> -- CIF operand identifiers are of size $\lceil \log_2(V) \rceil$, where V is the number of unique variables used in the operand naming environment. The operand naming environment is also taken as the lexical scope.

REFERENCE ACTIVITY PROPERTY

<u>Instructions</u> -- one CIF reference allowed for each program control (procedural) operator encountered during execution.

<u>Name Space</u> -- one CIF reference allowed per entry into and one reference per exit from a lexical scope; also one reference is allowed per computed name within the scope.

Space is measured by the number of bits needed to represent the static definition of a program; time by the number of instructions (operations) and references needed to interpret the program. Source programs to which these measures are applied should themselves be efficient expressions of an optimal abstract algorithm -- so as to eliminate the possible effects of algorithm optimization during translation -- such as changing "X = X/X" to "X = 1."

Generating canonic program representations should be straight forward because of the 1:1 property. Traditional three-address architectures also satisfy the first part of this criteria, but do not have the unique naming property. These instruction sets are of the form $\underline{OP}$ $\underline{X}$ $\underline{Y}$ $\underline{Z}$ -- where OP is an identifier for a (binary) operation; X the left argument; Y the right argument; and Z the result.

For example, the statement A = A * A + A * A contains only one unique variable, and three functional operators (*,*,+). Hence, it can be represented by three CIF instructions consisting of only one operation identifier and one operand identifier. The three address representation of this statement also requires only three instructions, but it would consist of twelve identifiers rather than the six required by the CIF.

There may be some confusion as to what is meant by an "operation". Functional operators (+, -, *, /, SQRT, etc.) are clear enough; however, allowance must also be made for selection operators that manipulate structured data (i.e. a name computation). For instance, one could view the array specification "A(I,J)" as a source level expression involving one operator (two dimensional qualification) and as three operands (the array A, and its subscripts I and J). The canonic equivalent of "A(I,J) = A(I,J) + A(I,J)" would then require two instructions -- the first to compute the proper array element, and the second to compute the sum. Thus:

```
Example 1:    X = X + X                      +    X

Example 2:     A(I,J) = A(I,J) + A(I,J)      @    A    I    J   A_IJ
                                             +   A_IJ
```

The operator "@" computes the address of the doubly indexed element "A(I,J)", and dynamically completes the definition of the local identifier "A_{IJ}". This identifier is then used in the same manner as the identifier "X" in the first example.

We count each source level procedural operator, such as IF or DO, as a single operator. The predicate expression of an IF must, of course, be evaluated independently if it is not a simple variable reference. Distinct labels are treated as distinct operands , so that:

```
Example 3:     IF (X-Y) 10,20,30             -    X    Y
                                             IF   10   20   30
```

The only references required in examples 1 and 3 are associated with subroutine entry and exit. Example 2 involves the computation of a name A(I,J) and, hence, a subsequent access reference. No references are needed for either example just to maintain the instruction stream, since the order of that such reference activity can be fully overlapped execution is entirely linear. The 1:1 property measures both space and time, while the $\log_2$ property measures space alone, and the referencing property measures time alone.

The 1:1 property defines transformational completeness -- a term which we use to describe any intermediate language satisfying the first canonic measure. Translation of source programs into a transformationally complete language should require neither the introduction of synthetic variables, nor the insertion of non-functional memory oriented instructions. However, since the canonic measures described above make no allowance for distinguishing between different associations of identifiers to arguments and results, it is unlikely that any practical DEL will be able to fully satisfy the CIF space requirements.

Comparison of CIF to Traditional Machine Architectures

The following three line excerpt from a FORTRAN subroutine, taken from [4], illustrated the CIF.

```
1     I = I + 1
2     J = (J-1)*I
3     K = (J-1)*(K-I)
```

Assume that I, J, and K are fullword (32 bit) integers whose initial values are stored in memory prior to entering the excerpt, and whose final values must be stored in memory for later use.

CANONIC MEASURE OF THE FORTRAN FRAGMENT

Instructions

```
Statement 1 -- 1 instruction   (1 operator)
Statement 2 -- 2 instructions  (2 operators)
Statement 3 -- 3 instructions  (3 operators)
              --------------
Total         6 instructions   (6 operators)
```

Instruction Size

Identifier Size

Operation identifier size = $\lceil \log_2 4 \rceil$ = 2 bits
(operations are: +, -, *, =)

Operand identifier size = $\lceil \log_2 4 \rceil$ = 2 bits
(operands are: 1, I, J, K)

Number of Identifiers

```
Statement 1 -- 3 identifiers  (2 operand, 1 operator)
Statement 2 -- 5 identifiers  (3 operand, 2 operator)
Statement 3 -- 7 identifiers  (4 operand, 3 operator)
             --------------
Total        15 identifiers   (9 operand, 6 operator)
```

Program Size

```
6 operator identifiers x 2 bits = 12 bits
9 operand identifiers x 2 bits  = 18 bits
                                  -------
Total                             30 bits
```

References

```
Instruction references -- 1 reference

Operand references     --
     (i)  Subroutine environment -- 1 load
                                    1 store
                                    0 computed names

     (ii) Identifier environment -- 9 loads
                                    3 stores
```

The following listing was produced on an IBM System 370 using an optimizing compiler[1]:

```
1   L     10,112(0,13)
    L     11,80(0,13)
    LR    3,11
    A     3,0(0,10)
    ST    3,0(10)

2   L     7,4(0,10)
    SR    7,11
    MR    6,3
    ST    7,4(0,10)

3   LR    4,7
    SR    4,3
    LCR   3,3
    A     3,8(0,10)
    MR    2,4
    ST    3,8(0,10)
```

A total of 368 bits are required to contain this program body (we have excluded some 2000 bits of prologue/epilogue code required by the 370 Operating System and FORTRAN linkage conventions) -- over 12 times the space indicated by the canonic measure. Computing reference activity in the same way as before, we find 20 accesses to the process name space are required to evaluate the 370 representation -- allowing one access for each 32 bit word in the instruction stream.

The increase in program size, number of instructions, and number of memory references is a direct result of the partitioned name space, indirect operand identification, and restricted instruction formats of

---

[1]FORTRAN IV level H, OPT = 2, run in a 500K partition on a Model 168, June 1977.

the 370 architecture.

The table below illustrates the use of ratios for the foregoing example.

COMPARISON FOR THE EXAMPLE

	370 FORTRAN-IV (level H extended) optimized	370 FORTRAN-IV (level H extended) non optimized	CIF
No. of Instructions	15	19	6
M-type Instructions	9	13	0
F-type Instructions	6	6	6
M-ratio	1.5	2.7	0
Program Size	368 bits	604 bits	30 bits
Memory References	20	36	3 or 13 (see below)

Of memory References, 13 are required using "identifier" as operand reference environment while 3 are required for "subroutine" environment.

CIF AND THE MEASUREMENT OF ARCHITECTURES

Three important questions remain: (1) Are the statistics developed in the above example valid over a larger body of program material? (2) Is 370 a uniquely ill-suited architectural representation for programs and perhaps some other host oriented architectural representations can provide a significant improvement? (3) Can we use the notion of a CIF as an ideal program representation form to actually create a useful architecture which will provide close to CIF measures?

This third question is by far the most complex and while the answer is positive we will defer its detailed discussion to a companion paper on the synthesis of directly executed languages (DELs). To address the first two questions we have selected a widely used benchmark called the Whetstone benchmark. This was selected on several bases: it is widely known and in a series at the Stanford Emulation Laboratory the Whetstone does not appear to generate profoundly different statistics than most other benchmark materials.

We have measured the CIF on the Whetstone and then compared these measures to traditional architectures: IBM System 370 and Honeywell Level 66, and DEL architectures: Burroughs S-Fortran (B1700) and a CIF based "ideal" DEL called DELtran. This latter language is described in a companion paper [6].

It can be seen from the accompanying table that conventional architectures are not optimum program representations. Particularly impressive is the static program size especially when compared to the

WHETSTONE COMPARISON TABLE

Architecture (0 = optimized / No = non optimized) Comparison

	Source	CIF	370-0	370-NO	H66-0	H66-NO	Burroughs S-FORTRAN	DELtran
Static Size	4.26	1	16.75	12.83	7.53	7.09	5.87	1.32
Instruments Executed	0.71*	1	4.09	6.64	2.87	3.00	3.50	1
Memory References								
I = Branch Target Capture + D = scope environment	-	.00024**	†	†	†	†		
I = no capture + D = name environments	-	1	7.11	11.38	5.51	6.13	3.80	1.46

* statements interpreted

** relative to number of instructions executed; corresponds to "perfect buffer" miss rate

† typical cache miss rates .1 to .01 per instruction executed depending on cache size, block size, etc

original source program. Source names are mnemonic and could not be regarded as even attempting to provide an efficient representation yet most host oriented architectural representations significantly expand the amount of space required for program representation. Most attempts at specific language DELs also seem to miss the mark in terms of the efficiency of the program representation. Two great obstacles to achieving efficient representation seem to be the need for a 1:1 instruction to source operation correspondence and a $\log_2$ container size referenced to the scope of definition of the language. In our companion paper we will use these two observations to synthesize DELs roughly corresponding to CIF measures.

The accompanying table describes some of this in terms of overall architectural comparisons. The size comparisons are fairly straightforward. Neglected in the size comparisons are preamble and epilogue code for subroutine or function entry/exit. Only routine code bodies are compared in our analysis. Excluding this overhead the program size is limited by one:one, i.e. transparancy requirements, and information theoretic environmental naming limitations in the absence of frequency encoding.

Analysis of interpretation time is necessarily more complex involving both the number of instructions to be interpreted and the amount of referencing activity: as to which of these in a practical host system will dominate the program interpretation time, the answer, of course, is host dependent. In systems that must make reference to program store for each instruction to be interpreted and reference to name space for each operand used the referencing activity will tend to dominate the total program interpretation. A host without a cache is largely memory limited. A host with a cache is largely interpretation time limited. The exact balance point depends a great deal on the physical parameters of a host and its memory access time. "Cache" per se is exactly what is <u>not</u> modelled by an ideal DEL with scope referencing enviornments. What is modelled is a <u>logical</u> <u>locality</u> not physical memory addressing space locality. All logically related <u>predictable</u> aspects of an environment are acquired on scope entry, each non predictable aspect requires an additional reference. It is interesting to note that this type of logical "cache" has a very low miss rate.

On the Whetstone, the referencing activity is almost completely captured by branch target capture for instruction and scope capture for operands. The hit and miss ratios when compared to simple DEL instruction reference environments are:

miss rate = references required to capture scope and target

hit rate = .99976

These rates represent a "<u>perfect</u> <u>buffer</u>" hit/miss rate and are significantly better than conventional cache statistics.

Note that this does not imply that total DEL program execution time could be correspondingly reduced. Rather as referencing activity is driven to a very small percentage of instruction interpretation activity, the instruction interpretation will necessarily dominate and the 1:1 (or transparency) requirement on HLL operations/instructions will become the program execution time limitation.

CONCLUSIONS

The traditional computer architectures (i.e. program representations) are created about objects, actions and/or capabilities presumed to be present in a physical host computer -- thus, simplifying the interpretation process. This is done, however, at the expense of compilation, storage space requirement and number of items to be interpreted.

An alternative is presented, created about the notion of a directly executed language (DEL); an architecture in close correspondance to the high level language that was used to originally represent the program. Various DEL possibilities have been considered with an "ideal" form defined as the Canonic Interpretive Form or CIF. The CIF is actually a measure against which architectures can be compared in their representation space requirements and interpretation time requirements.

Traditional architectures are significantly inferior to CIF measures (by a factor of from 3 to 10), while DEL's specifically designed to retain the CIF measures are able to come rather close (within 1.3) to that indicated.

APPENDIX

The Whetstone

A derivative of the Whetstone benchmark is used as the basis for architectural comparison. Originally developed as a research tool at the National Physical Laboratory, the Whetstone is now a well established commercial standard; both Data General and Digital Equipment have conducted numerous evaluations of mini-computer FORTRAN systems in scientific environments using this benchmark.

The traditional Whetstone contains twelve loops, three of which emphasize transcendental operations; these were deleted to avoid focusing on a few specific functions much to the advantage of the ideal form. Each of the remaining loops tests a different aspect of the FORTRAN language:

Loop 1: floating point arithmetic over scalar variables;

Loop 2: the same operations over elements of linear arrays;

Loop 3: again the same operations, but invoked as a subroutine;

Loop 4: conditional branching;

Loop 5: two-dimensional array manipulation;

Loop 6: integer arithmetic on scalars and array subscripts;

Loop 7: in-line polynominal evaluation;

Loop 8: subroutinized polynomial evaluation;

Loop 9: swapping array elements, as in a sort or shuffle.

The number of iterations per loop is determined by multiplying a weighting factor by an overall repetition count. Weighting factors reflect the dynamic behavior of typical user programs; in practice, they are adjusted based on installation specific trace-tape data. The benchmark is executed for two different repetition counts, and the difference in execution times taken to eliminate the effects of initialization, I/O operations, monitor functions, etc.--so that only the looping portions are significant. The average values of the weighting factors used by Data General and Digital Equipment were used to fix the number of loop iterations in the experiments described below, along with a repetition count difference of 100.

The Measures

The canonic measures are computed for each loop: size of representation (in bits); number of instructions (dynamic); and number of references (in main store accesses -- also dynamic). Some details concerning the way this is done may be of interest, since an attempt has been made to exclude everything not pertaining directly to executable code bodies.

First, with respect to space, only those sections of executable code that correspond to statements in the body of a source program are counted. Prologue/epilogue linkage, dynamic save/restore areas, imbedded constants (usually address constants and FIX/FLOAT data masks), and operating system or program library service routines (GET-MAIN, SIN, READ, etc.) are not included. This was done as it is difficult to apportion such spatial costs across the various Whetstone loops. Similarly, the interpretive store required to support the Burroughs S-language (which includes the stack and descriptor tables) is also not included in the space measure. This type of overhead is generally high for the 370 architecture; but the ratio of improvement may not be as high as it is within pure code segments for small programs with many variables.

Secondly, with respect to time, the executable instructions in prologue/epilogue linkage -- as well as the instructions and references needed to establish dynamic displays -- are counted. Only external library or system routines are excluded. Reference counts include the number of memory accesses needed to maintain the instruction stream itself, assuming each fetch brings in 32 bits.[2] Register accesses are not counted as references in the 370 architecture, and stack accesses are not counted as references in the Burroughs architectures. Accesses to either the program or data store are counted as references in all cases, however; i.e. the reference environment. For the CIF, the reference activity is computed with both types of reference environment name space and scope.

[2]The Honeywell Level 66 architecture is 36 bits wide, and the Burroughs B1726 is only a 24 bit machine; such differences have been normalized in this comparison.

CIF ARCHITECTURAL MEASURES

	Size	Instr	Referencing* Instr	Referencing* Data
loop 1	216^b	1,161	2 60	2 1762
loop 2	318^b	16,245	2 981	6 8414
loop 3	356^b	17,081	4 1260	4 23,242
loop 4	209^b	17,253	5 1726	2 13,800
loop 5	294^b	990	2 141	11 1435
loop 6	241	46,201	2 2101	2 56,702
loop 7	110	2,561	2 321	2 3842
loop 8	144^b	89,990	4 35,960	4 161,822
loop 9	135	55,441	4 24,641	7 36,960
	2023	246,923	27 67,199	40 307,981

* For referencing instructions, the first entry corresponds to target compare: i.e. only the first incidence of branch incurs a reference, the second entry corresponds to a reference for each branch instruction.
For data references, the first entry corresponds to scope as a referencing environment. Only scope entry/exit and computed names require a reference. The second entry corresponds to a reference per name.

IBM System 370 Statistics

Two different levels of optimization were employed using the standard IBM extended compiler for FORTRAN-IV. Compilations were performed on a Model 168 in a 300K byte partition under the VS operating system in March, 1977. The results of a hand analysis of these compilations are shown below.

Loop	Bits	Instructions	References
1	2,096	3,193	6,306
2	5,552	135,383	269,646
3	5,232	127,123	255,646
4	1,488	165,603	307,056
5	4,708	20,863	40,326
6	2,832	186,903	359,106
7	1,072	9,923	19,526
8	1,280	584,353	1,330,526
9	1,696	406,563	917,846
	------	---------	---------
Totals:	25,956	1,639,907	3,505,984

<u>370 Performance</u> (<u>No Optimization</u>)

Average instruction size without optimization is 31 bits; average instruction size under full optimization is 30 bits. Clearly, the Whetstone does not defeat the optimization strategies employed by the IBM compiler -- indeed, a factor of 2 in space and 1.6 to 1.8 in time is observed. This raises the question of which level of optimization should be used as a standard of comparison.

Loop	Bits	Instructions	References
1	1,504	1,871	3,031
2	1,968	26,905	48,626
3	2,256	28,984	53,628
4	1,024	31,062	48,323
5	2,096	2,990	5,969
6	1,728	115,508	197,414
7	880	7,685	14,410
8	1,120	512,437	1,159,724
9	1,072	283,364	659,128
Totals:	13,648	1,010,806	2,190,253

370 Performance (Full Optimization)

Honeywell Statistics

Compilation of the Whetstone into the Honeywell Level 66 architecture was performed on a Honeywell Model 6680 (by Honeywell personnel) in January 1978. The small difference in performance between optimization levels is attributable to the straightforward nature of single accumulator code (in general).

The anomalous behavior of loop 3, for which optimization appears to degrade both execution time and space, is the result of a known compiler bug. Not all compilers are perfect, however, and this illustrates one of the inherent dangers of an execution architecture requiring non-trivial program conversions in order to achieve high performance.

Loop	Bits	Instructions	References
1	1,368	1,780	3,430
2	2,124	37,803	76,866
3	2,040	40,040	117,980
4	1,224	55,203	89,704
5	3,816	10,951	20,497
6	1,944	109,202	327,604
7	756	5,762	11,203
8	1,136	306,667	791,120
9	828	172,480	449,680
	------	-------	---------
Totals:	15,236	739,888	1,888,084

<u>Honeywell</u> <u>Performance</u> (<u>No</u> <u>Optimization</u>)

Loop	Bits	Instructions	References
1	1,368	1,780	3,430
2	2,088	37,383	76,026
3	2,196	40,600	119,000
4	1,188	44,853	69,004
5	2,916	7,031	12,657
6	1,908	111,300	203,700
7	828	5,446	9,612
8	1,136	306,667	791,120
9	720	154,000	412,720
Totals:	14,348	709,060	1,697,269

Honeywell Performance (Full Optimization)

S-Language Statistics

Wilner [15] observes a spatial improvement factor of two for the Burroughs S-Language for FORTRAN over 360 machine language. The same version of the Whetstone used to develop the 370 statistics presented in this section was also compiled into this language in July 1977, using a B1726.

The statistics corroborate the results of Wilner's experiments under the assumption that no optimization is employed during 370 compilation. This indicates that OPT = 0 is the level of optimization that should be used when comparing different machine architectures.

Although disallowing optimization puts the 370 architecture (and indeed any traditional mono- format architecture) at a disadvantage, there are reasonable arguments for doing so. First, transparency is better preserved; second, there is usually little optimization performed when compiling into higher level DELs (the only optimization in the Burroughs version of the Whetstone is substitution of non-destructive stores for pop-push pairs), so that allowing optimization would tend to place these higher level DELs at a disadvantage (N.B. it is a true disadvantage however). Third, disallowing most substantial optimization strategies tends to equalize the time and space requirements of compilation, thus, eliminating a troublesome variable in performance comparisons. Fourth, prohibiting optimization tends to ensure that one compares two versions of the same algorithm -- if global program transformations (especially strength reduction) are permitted, it is possible to end up evaluating compiler performance rather than architecture performance.

The Burroughs FORTRAN DEL is typical of several other stack oriented architectures--such as McClure's DEL for Basic FORTRAN [10], Weber's Euler [14], and Wortman's DEL for Student PL/1 [16]. Indeed, the number of instructions and memory references required for execution should be the same for both the McClure architecture and FORTRAN versions of Weber's reverse polish string language and Wortman's high level intermediate language. The McClure and Weber machines are also roughly equivalent in space -- the average instruction size being about 28 bits for both architectures; Wortman's machine could require up to 50% less space if "short addresses" (one byte identifiers) are used for most operands.

Loop	Bits	Instructions	References
1	1,102	2,991	3,433
2	2,392	62,021	100,103
3	2,346	96,104	113,961
4	755	62,104	82,811
5	1,919	9,101	15,963
6	1,151	117,601	149,103
7	524	8,321	10,243
8	811	332,631	386,573
9	880	172,481	308,003
Totals:	11,880	863,577	1,170,193

S-Language Performance (Some Optimization)

DELtran Statistics

In the example cited in a companion paper we develop a language called DELtran [6] which is a CIF derived DEL for the FORTRAN language. It largely achieves the CIF size and measure achieving $\log_2$ container size, however requiring an additional 5 bits per instruction for format information. Otherwise, the number of instructions interpreted corresponds to CIF measures. The referencing environments in

DELtran are adjusted to the actual implementation of the DELtran interpreter which is developed on the Stanford Emulation Laboratory system, the EMMY. Since block access techniques on the EMMY system (from main memory to interpretive store) were not implemented at the time the DELtran language was developed, DELtran uses a 32 bit physical word as the referencing environment for both instructions and data. DELtran as a language will achieve the CIF measures in referencing environments in so far as the host allows it to, i.e. supports particular referencing strategies with respect to environments.

Loop	Bits	Instruction	References
1	362	1,161	2,343
2	458	16,245	32,770
3	451	17,081	34,023
4	214	17,253	31,053
5	380	990	2,266
6	348	46,201	79,803
7	153	2,561	5,763
8	165	89,990	161,823
9	141	55,441	98,563
	-----	-------	-------
Totals:	2,672	246,923	448,407

<u>DELtran</u> <u>Performance</u> (<u>No</u> <u>Optimization</u>)

REFERENCES

[1] Chu, Yaohan (Ed.), High Level Language Computer Architecture, Academic Press, New York, New York, 1975.

[2] Elson, M., and Rake, S. T., "Code-Generation Techniques for Large-Language Compilers," IBM Systems Journal, Vol. 9, No. 3, 1970, pp. 166-88.

[3] Flynn, Michael J., "Trends and Problems in Computer Organizations," IFIPS Congress, Stockholm, Sweden, August 1974, North Holland Publishing Company, 1975, pp. 2-10.

[4] Flynn, Michael J., "The Interpretive Interface: Resources and Program Representation in Computer Organization," Proceedings of the Symposium on High Speed Computers and Algorithm Organization, University of Illinois, Champaign, Illinois, (Pub. Academic Press) April 1977.

[5] Hehner, Eric C. R., "Information Content of Programs and Operation Encoding," Journal of the ACM, Vol. 24, No. 2, April 1977, pp. 290-97.

[6] Hoevel, L. W. and Flynn, M. J., "A Theory of Interpretive Architectures: Some Notes on DEL Design", Technical Report No. 171, Computer Systems Laboratory, Standford University, Stanford, California, February 1979.

[7] Hoevel, Lee W., and Flynn, Michael J., "The Structure of Directly Executed Languages: A New Theory of Interpretive System Support," Technical Report No. 130, Digital Systems Laboratory, Stanford University, Stanford, California, March 1977.

[8] Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes," IRE, Vol. 40, No. 9, September 1952, pp. 1098-101.

[9] Lunde, A., "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," Communications of the ACM, Vol. 20, No. 3, March 1977, pp. 143-52.

[10] McClure, Robert M., "CUC Basic FORTRAN Description," private working notes, 1970.

[11] McKeeman, W. M., "Language Directed Computer Design," _Proceedings of the Fall Joint Computer Conference_, Vol. 31, Fall 1967, pp. 413-17.

[12] Reigel, E. W., with Faber, U., and Fisher, D. A., "The Interpreter -- A Microprogrammable Building Block System," _Proceedings of the Spring Joint Computer Conference_, Vol. 40, Spring 1972, pp. 705-23.

[13] Sethi, Ravi, and Ullman, Jeffery D., "The Generation of Optimal Code for Arithmetic Expressions," _Journal of the ACM_, Vol. 17, No. 4, October 1970, pp. 715-28.

[14] Weber, Helmut, "A Microprogrammed Implementation of EULER on IBM System/360 Model 30," _Communications of the ACM_, Vol. 10, No. 9, September 1967, pp. 549-58.

[15] Wilner, W. T., "Burroughs B1700 Memory Utilization," _Proceedings of the Fall Joint Computer Conference_, Fall 1972, pp. 579-86.

[16] Wortman, Daniel B., _A Study of Language Directed Computer Design_, Ph.D. Thesis, Stanford University, Stanford, California, 1973.