LEVELS OF REPRESENTATION OF PROGRAMS AND THE ARCHITECTURE OF UNIVERSAL HOST MACHINES

B. Ramakrishna Rau

Coordinated Science Laboratory
University of Illinois, Urbana, IL 61801

Summary

The issue of high level language support is treated in a systematic top-down manner. Program representations are categorized into three classes with respect to a host processor: high level representations, directly interpretable representations and directly executable representations. The space of intermediate languages for high level language support is explored and it is shown that whereas the ideal intermediate language from the point of view of execution time is a directly executable one, the best candidate from the viewpoint of memory requirements is a heavily encoded directly interpretable representation. The concept of dynamic translation is advanced as a means for achieving both goals simultaneously; the program is present in the memory in a compact static representation, but its working set is maintained in a dynamic representation which minimizes execution time. The architecture and organization of a universal host machine, incorporating this strategy, is outlined and the potential performance gains due to dynamic translation are studied.

1. Introduction

1.1. Microprogramming and Interpretation

Microprogramming was originally conceived by Wilkes as a systematic means of implementing the control structure of a computer[1]. The microprogram, embedded in a read-only memory, interprets the instruction set visible to the programmer. In view of the permanence of the microprogram and its transparency to the user, the interpreted instruction set was, reasonable enough, thought of as representing the architecture of the machine. Accordingly, the emphasis was on the interpreted instruction set.

With the advent of writeable control store, the situation has changed and, yet, the perspective has remained much the same. Writeable control store is viewed as a means of providing a "soft architecture," i.e., one that can be changed dynamically to match the needs of the moment which might, for instance, entail the support of a high level language. Experience with the Burroughs B1700[2,3] and the work of Hoevel[4] has demonstrated the effectiveness of such a strategy. However, the emphasis still is on the interpreted instruction set. An artificial line is drawn upon which lies the conventional machine language. On one side of this line is the domain of high level languages, compilers, interpreters and main memory. On the other side lie the microprograms, nanoprograms, emulators and a host of other micro- and nano- entities. This viewpoint arises, in part, from the use of microprogrammable machines predominantly for the purpose of emulating the instruction sets of other machines. This classical concept of microprogramming tends to obfuscate the issue which may be phrased as follows: given a certain (open ended) set of high level languages, what is the nature of the host hardware that is best suited to supporting them and what is the process by which programs, written in these high level languages, are supported? A fresh perspective can be valuable; the host machine should be viewed as a special purpose machine designed to provide high level language support. The architecture and organization should evolve as the outcome of a top-down design process rather than as a carry-over from the classical view of microprogramming. This is the objective of this paper and so as to avoid any preconceived notions, the terminology of microprogramming is avoided as far as is possible.

The architecture and instruction set of a host is determined by the class of languages that are to be supported by it. If this class is restricted and consists of similar languages, the application of the host is fairly specific and the instruction set will contain powerful instructions which closely resemble the semantics of the high-level languages that are supported by the host. Several examples of high-level machine designs fall into this category[5-10]. On the other hand, if the class of languages is large and vague, commonality of semantics will exist only at a very low level and the instruction set of the host machine will be primitive. This provides generality and flexibility. A host of this type is termed a universal host machine (UHM). A number of examples of UHM's are available[2,11-14].

Given a host architecture and a high level language, one could either interpret the latter directly, compile it into the machine language or compile it into an intermediate language which is then interpreted. Hoevel derives conditions under which the last alternative is superior to the other two[15]. These conditions are generally satisfied for the types of universal host architectures that exist or are under consideration.

CH1411-8/78/0000-0067$00.75

The nature of this intermediate level is treated in some detail by Hoevel and Flynn[16]. The only true machine language involved is that of the host. However, to avoid confusion with the interpreted instruction set, which is traditionally viewed as being the machine language, the host's instruction set will be referred to as the host machine language.

1.2. Overview

This paper is an attempt to approach the problem of high level language support in a systematic manner. The basic assumptions are that, in the future, most programs will be written in a high level language, and that the number and variety of languages will not be constrained. In such an environment, the effectiveness of an architecture is judged not by the performance achieved with a particular language but by the performance over the spectrum of languages. Accordingly, the focus is on the architecture of universal host machines and the nature of the intermediate level language. It is worth emphasizing that the universal host machine is not seen as providing a faster way to interpret a single language such as the System/360. A specialized architecture will obviously perform better. The universal host machine is effective when the objective is to support a large number of languages with equal facility.

A top-down treatment must begin by identifying and evaluating the various representations of programs. In Section 2, three levels are defined: high level representations, directly interpretable representations and directly executable representations. Section 3 considers the space of intermediate representations into which a high level representation may be compiled. This space is shown to have two dimensions: semantic level and the degree of encoding. It is shown that the ideal intermediate representation varies depending on whether importance is attached more to the speed of interpretation or to the compactness of the representation. Section 4 introduces dynamic translation as a means of achieving the two goals simultaneously. The program is stored statically in the compact representation, but its working set is translated dynamically into the representation suitable for speedy interpretation. On the basis of the "principle of locality" it is possible for just a small fraction of the program to be in the dynamic representation and yet to find that the majority of instructions that are executed are in the dynamic representation. This dynamic representation of the working set is maintained in a dynamic translation buffer, the organization and management of which is outlined in Section 5. Section 6 discusses the architecture of a universal host machine incorporating a dynamic translation buffer. Finally, Section 7 attempts to evaluate the effectiveness of the dynamic translation buffer.

2. Levels of Representation of Programs

Central to the discussion of the various levels of representation of programs is the concept of binding. We follow Radin[17] in defining a program to be bound with respect to a given automaton if the automaton is able to execute this program correctly. If the automaton is unable to execute the program due to lack of information regarding the syntax and semantics of the program and data structures, then the program is unbound in those aspects with respect to the automaton. Binding is the process of supplying the missing information by augmenting or modifying the program so that the resulting program representation is bound with respect to the automaton.

The level of a program representation may be studied by examining the (virtual) machine to which it is bound. Higher level representations are bound to machines that are semantically more powerful. Although the levels of representation may vary over a wide range, three broad categories have particular significance in the context of high level language support:

2.1. Directly Executable Representations

Assuming that the universal host machine is specified, one may immediately define the lowest level; a directly executable representation, (DER), is one which is bound with respect to the architecture of the universal host machine. This is a well defined level since it is possible to test whether a program representation falls into this category by attempting to execute it on the universal host machine. However, as will be seen subsequently, it is possible to specify more than one DER for a program.

2.2. High-Level Representations

At the other end of the spectrum is the high-level representation, (HLR). A HLR is one written in a high level language, a precise definition of which is difficult. It is more profitable to describe a HLR by its characterizing properties. Of relevance are those properties which affect the ease with which a HLR may be interpreted. These properties are best studied by describing the virtual machine to which the HLR is bound. Such a machine has:

1. an associative memory to allow a variable reference to be associated with the variable declaration statement which specifies the mapping to type and value. Similarly, an explicit transfer of control to a procedure or label requires an association with the procedure declaration or labelled statement;

2. a mechanism which qualifies the above association based on the scope rules of a language such as ALGOL[18] or APL[19];

3. a scanning mechanism which can match BEGIN's and END's in a block structured language. Such a mechanism is needed to skip over nested blocks which are not to be executed, e.g., in a CASE statement;

4. the ability to parse statements generated by a recursive grammar;

5. The ability to evaluate expressions in parenthesized infix notation obeying rules of precedence;

6. the ability to handle symbolic names of arbitrary length.

These features are designed into high level languages so as to aid the programmer in expressing the algorithm. They also complicate the task of the interpreter immensely. As a result, the idea of building hardware to directly interpret a HLR is unattractive and even so-called high-level language machines invariably translate the HLR to some other internal representation.

2.3. Directly Interpretable Representations

The purpose of compilation is to translate the HLR into a representation that is devoid of the problems that arise in directly interpreting a HLR. This representation should be bound to a virtual machine that more closely resembles the host than does the HLR's virtual machine. In particular:

1. the representation should be bound to a directly addressed memory;

2. variables and labels should be bound to memory locations to the extent permitted by the HLR;

3. the hierarchical, block structure of the HLR should be replaced by the familiar sequential form with explicit branches around code which is not to be executed;

4. the syntax should be context-insensitive and relatively simple;

5. expressions should be decomposed into a non-hierarchical form, preferably reverse Polish notation;

6. the size of operand and operation specifications should belong to a small, well-defined set of possibilities.

The resulting representation allows for a faster and more efficient interpreter. It also does not require a preliminary scan (to set up symbol tables and resolve forward references) before the program can be interpreted. Such a representation is termed a directly interpretable representation (DIR). The class of DER's is a special subset of the class of DIR's. Programs written in conventional machine languages, the S-languages of the B1700[3] and DELTRAN[4] are examples of DIR's.

3. The Space of Intermediate Representations

The intermediate representation that results from compiling a HLR could assume any number of forms. From the point of view of the speed of interpretation and the size of the representation, the space of intermediate representations may be parameterized in two-dimensions -- the semantic level and the degree of encoding.

3.1. One Dimension: The Semantic Level

The representation of lowest semantic level is obtained by compiling down directly to the host machine language. Such a DER is termed an S-DER. Since any computation performed by the host must eventually be performed in the host machine language, the S-DER might be expected to provide the shortest possible execution time in a host machine which has a single level of memory.

The manner in which compilers generate code (by substituting similar sequences of code for occurrences of the same terminal symbol) may be exploited by replacing every such sequence by a call to a generalized procedure which performs the same function. The arguements of the call are included in-line. The resulting procedurally structured DER is termed a P-DER. While being considerable more compact than an S-DER, the speed of execution of the P-DER suffers from having to pass parameters and from the loss of the local optimization based on context that must be sacrificed when using a procedure. The procedure calls are logically equivalent to more powerful opcodes and, so, the semantic level of the P-DER is higher than that of the S-DER. Nevertheless, the P-DER remains directly executable, the binding being performed explicitly via the procedure calls. The semantic level of the P-DER is determined by the functional complexity of the semantic routines.

Further economy of size may be achieved by combining sequences of P-DER procedure calls into single calls to procedures that perform the same function as the sequence of calls. Clearly, the inclusion of every possible sequence (of a given length) does little to compact the program representation and leads to a combinatorial increase in the number of semantic routines. Consequently, such a scheme, to be successful, must discard all but a selected set of sequences which are then combined. The resulting representation is constrained by this selection and is termed a C-DER. Unless the selection of the C-DER procedures is undertaken carefully, the loss of semantic flexibility manifests itself in the inefficient generation of code by the compiler and a resultant increase in execution time. The C-DER is logically equivalent to a representation in a machine language with a limited choice of instruction formats, e.g., a three-address format alone.

3.2. The Other Dimension: The Degree of Encoding

Every DIR which is not a DER may be viewed as being an encoded form of a DER. The simplest form of encoding is obtained with a P-DER consisting solely of procedure calls and no other host machine language instructions in-line. In such a P-DER the opcode portion of the procedure call is

redundant and may be discarded. The new representation consists of the addresses of the semantic routines followed by the arguements of the call. Such a representation has been termed "threaded code" by Bell[20]. The address of a semantic routine along with the arguements to the call form an instruction in the new representation with the address of the routine assuming the role of an opcode. The reduction in program size is accompanied by a significant disadvantage. The representation is no longer a DER but rather a DIR. The DIR must be interpreted by an interpreter which increases the execution time. (In Bell's threaded code, the interpreter consists of code in each semantic routine which increments the DIR program counter past the arguements and a final branch to the location pointed to indirectly by the DIR program counter).

Further code compaction is achieved by selecting the sizes of the DIR's opcode and arguement fields independently of the size of the smallest unit of memory access (e.g., a word or byte). The instruction fields are packed together with fields being allowed to overlap access boundaries or with multiple fields in one access unit. Most conventional machine languages fall into this category. The price paid is that the interpreter must extract these fields before performing the semantics of the DIR instruction.

An encoding based on contextual information may be used to reduce the size of each field. For instance, the scope rules of a high level language limit the number of variables that may be referenced from within a given contour[21]. The operand specification field need be no larger than necessary to select between these variables. This technique has been used in DELTRAN[4].

A more sophisticated Huffman encoding[22], based on the static frequencies of occurrence of the operators and operands, leads to further economics in program size, often up to 75% reductions in size[3,23-25]. A compromise between decoding time and encoding efficiency may be achieved by limiting field sizes to a small number of alternatives[3]. This frequency based encoding scheme may be generalized by considering the frequency of occurrence of pair, triples, etc., [23-25], and may be combined with contour-based encoding scheme. The S-languages of the B1700 are examples of the successful application of frequency based encoding techniques.

3.3. Comparison of Intermediate Representations

If the host machine possesses only a single level of directly addressable memory, the space/time trade-offs are straightforward; an increase in the semantic level or in the degree of encoding generally decreases the size of the intermediate representation but increases the time needed to execute or interpret it. The fastest and least compact representation is the S-DER obtained by directly compiling into the host machine language.

The presence of a memory hierarchy leads to more interesting trade-offs. A P-DER or DIR partitions the memory space into two parts -- that consisting of the P-DER calls or DIR instructions and that consisting of the semantic routines and the interpreter. Each reference to the former part generates a large number of references to the latter part, which thus becomes a prime candidate for placement in the faster portion of the memory hierarchy. The average memory access time is reduced as compared to the average memory access time using an S-DER. Thus the use of a P-DER or DIR is beneficial both in space and in time and a well designed host machine must possess a memory hierarchy to capitalize on this property. In fact, all microprogrammable machines possess at least two levels -- control store and main memory -- which explains, in part, why traditional microprogrammable machines and universal host machines are equated.

The space of intermediate representations may be graphically represented as in Figure 1. The vertical dimension is a measure of the syntactic and semantic complexity of the representation. The horizontal dimension specifies the complexity of the encoding. A point in the space denotes a representation. In general, the size of a program decreases with increasing distance of the representation from the origin, but the size of the interpreter and semantic routines increases although by a smaller extent. Assuming a two-level memory hierarchy, the interpretation time may be expected to decrease in the vertical direction with increasing level. At the same time, the compile time will decrease since it, presumably, is easier to compile into a higher level intermediate level than it is to compile into one which is greatly removed from the HLR. As one moves to the right, both interpretation and compilation time may be expected to increase. The increase in compilation time is caused by having to compile first into an unencoded form followed by an encoding step.

If one is concerned only with the size of the intermediate representation and the interpretation time, the former consideration would indicate the use of a highly encoded DIR of a level as high as can be tolerated from the point of view of interpreter size. The latter consideration would indicate the use of a P-DER, once again of as high a level as the size of the semantic routines will permit. The size of the semantic routines and interpreter is important since they must fit into the faster, smaller levels of the hierarchy if high speed interpretation is to be achieved. In the next section, a method of simultaneously fulfilling these contradictory requirements of high speed interpretation and a compact intermediate representation of the program, is presented.

4. Dynamic Translation of Program Representations

A characterizing property of a compiler is that whatever binding it does persists over the entire period of execution of the program. The

interpreter must complete whatever binding remains. However, this binding persists only over the period of execution of an instruction and must be repeated each time that instruction is executed. From the point of view of persistence of binding, the compiler and interpreter are at opposite extremes. We introduce the notion of a dynamic translator, the persistence of whose binding lies in between that of the compiler and the interpreter. Once the dynamic translator binds an instruction (totally or partially), it remains bound over a period of time that spans a certain number of successive executions of that instruction. Such a strategy assumes, of course, that the program is not self-modifying -- an assumption that is valid when programs are written in high-level languages.

One could conceive of a hierarchy of representations each with a different level of binding and degree of persistence: the source program which exists until destroyed, the DIR which lasts until the source is modified, the link-edited version which exists for one execution of the program, possibly a number of lower levels, each increasingly bound and persisting for decreasing fractions of the program execution period and, finally, a completely bound representation of an instruction which only lasts for the duration of that instruction's execution.

The significance of the dynamic translator is that it raises the possibility of simultaneously achieving high speed interpretation and a compact static intermediate representation. Since the binding performed by a dynamic translator persists over a number of executions of an instruction, the time spent in binding is spread out over those instructions, thereby reducing the average time spent in binding per instruction executed. It is possible then to use a highly encoded DIR without increasing the interpretation time by very much if the binding is made to persist over a sufficient number of successive executions of the same instruction. This persistence of binding is effected by saving the bound representation of the instruction which will be less compact than the encoded DIR version. Attempting to retain this bound version for extended periods of time for a number of different instructions will entail the use of large amounts of memory. In fact, if the bound version were never discarded, one would soon obtain and have to provide storage for a translated version of the entire program, thereby defeating the purpose of using an encoded DIR.

The effectiveness of the dynamic translator hinges on the ability to save the bound representation for just a short period of time which, nevertheless, spans a large number of executions of the instruction. The existence of loops and recursive calls implies this ability. In fact, the more general "principle of locality" states that over any interval of time, the vast majority of memory references are concentrated on a small subset of the address space. This principle has been empirically validated over and again[26-28] and is the fundamental justification for the existence of cache memories[28-30] and virtual memories.[31,32] The fraction of the address space that is currently being referenced heavily is termed the working set.[27] The function of the dynamic translator is to maintain in the dynamic translation buffer (DTB) a representation of the instruction working set that is more tightly bound than the static representation. If the size of the DTB is reasonably large and if the contents of the DTB are selected carefully, it is possible to ensure that a large fraction of all instructions executed will be present in the DTB. This fraction is termed the hit ratio. When the hit ratio is close to unity, most instructions when executed will be found in the more tightly bound representation. The time penalty associated with binding will be experienced only rarely and will not be a major factor in determining the execution time. If, at the same time, the size of the DTB is small in comparison to the size of the loosely bound representation, the memory requirements will not have been increased substantially and the conflicting requirements of a compact representation and low execution time will be met simultaneously.

The concept of a DTB is related to that of the dynamic address translation mechanism provided with virtual memories. When addressing a virtual memory, the virtual address must be bound to a physical address. This involves indirection through one or more segment and page tables on each memory reference. This overhead is reduced by retaining in an associative array the mapping between the virtual and physical addresses for the pages which have been referenced most recently. The DTB may be viewed as a cache on a virtual memory in which the program is stored in the more tightly bound representation.

When the dissimilarities between the representations corresponding to minimum execution time and minimum storage requirements, respectively, are great, it is possible that a number of levels of dynamic translation will be required. However, in the rest of this paper, we shall concern ourselves with only one level of dynamic translation. Typically, three different representations are of interest: the HLR in which the program is written, the static (intermediate) representation into which it is compiled and the dynamic representation which is obtained by dynamically translating the static representation of the working set. Of these, only the latter two will be in the directly addressable memory during execution.

The use of dynamic translation permits the decoupling of the design decisions involved in selecting the intermediate representation. The static representation may be selected solely to minimize the size of the program. Ideally, it should be a high level, highly encoded DIR. The dynamic representation, on the other hand, should be selected to speed up execution and should, ideally, be a high level P-DER.

5. Organization of the Dynamic Translation Buffer

5.1. Memory Management

A factor that strongly influences the choice of the static/dynamic pair is the memory allocation policy in the DTB.

Variable allocation. Such a policy permits for great flexibility in the choice of the static/dynamic pair. The formats of the static representation need not be constrained to ensure that the corresponding dynamic sequence of instructions will fit into a pre-specified unit of allocation in the DTB. However, the replacement policy is constrained since the choice of what is to be replaced in the DTB is influenced by the size of the dynamic translation of the incoming instruction. Furthermore, the memory fragmentation[33] that results will require garbage collection which could degrade performance significantly.

Fixed allocation. While eliminating these memory management problems, a fixed allocation policy constrains the choice of the static/dynamic pair. The variability of the instruction formats in the static representation must be limited to permit the choice of a unit of allocation which will accommodate the longest dynamic instruction translation without wasting much space on the shortest one. The dynamic representation in turn should be semantically well matched to the static representation.

Variable allocation in fixed size increments. This represents a satisfactory compromise between the two previous alternatives. A dynamic translation that will not fit into the unit of primary allocation is assigned one or more units of allocation in a secondary overflow area. If the unit of allocation is shosen judiciously, the frequency of overflows may be kept low without space being wasted. This permits flexibility in the choice of the static and dynamic representations while reducing garbage collection to the chore of maintaining a list of available space in the overflow area.

5.2. Organization

The organization of the DTB is, for the most part, similar to that of a conventional cache.[28-30,34] It consists of four memory arrays (Figure 2). The first two, the associative tag array and the address array, are jointly known as the associative address array, one half of which contains the address of the DIR instruction (the associative tag), while the other half contains the address at which the P-DER translation is to be found. The third array, the buffer array, contains the PSDER instructions. This array occupies a predefined portion of the machine's directly addressable memory. As is normally the case, a degree of associativity of 4 is employed.[34] The DIR instruction address is hashed to select a unique set of four address array locations. These four are searched in parallel using the DIR address as the associative tag. If the required DIR to P-DER address mapping is not present, one of these four locations must be used to store the mapping. The one selected for replacement is that which was used least recently. The replacement array keeps track of the ordering of each set by recency of use. Set associativity of degree 4 has been found to be nearly as effective as full associativity.[28]

In cache organizations the pointer into the buffer array is implicit, i.e., the address at which the match is found in the associative address array is used to calculate the required buffer array address. Thus the second array (containing buffer array addresses) is not physically present. The presence of this array, and as part of the processor's directly addressable memory, is needed to permit the unit of allocation in the buffer to be altered to accommodate the needs of different HLR's. Variable and variable with fixed increments allocation policies, too, are supported by this feature. The contents of the address array are not altered after initalization if the policy used is either fixed allocation or variable allocation with fixed increments. The access time to the P-DER instructions might be increased (depending on the implementation) since two arrays must be accessed before the buffer address is obtained.

6. Architecture of the Universal Host Machine

The desirable architectural features of a universal host machine, (UHM), fall into two broad categories: those features that are generally useful in the task of interpretation, and those that are specific to a UHM that incorporates a DTB. The former category has been discussed at length elsewhere.[35] We shall concern ourselves in this paper with merely classifying these features into broad categories without dwelling on the implementational alternatives. Rather, we shall concentrate on the architectural implications of using a DTB.

6.1. General Features

An language, be it a HLR or DIR, makes certain assumptions about the virtual machine to which it is bound:

1. the ability to parse or interpret the syntax of the language,
2. the nature of the memory space, i.e.,
   a. the number of memory spaces, e.g., registers, control store and main memory,
   b. the type of memory - associative in the case of HLR's and directly addressable for DIR's.
3. the legal data structures, with respect to
   a. resolution - the smallest item of information,
   b. size - the relationship of other data structures to the unit of information,
   c. structure - the aggregation of simpler data types to form more complex ones.

4. semantic capability, i.e.,

   a. the permitted transformations upon the data structures,

   b. the facilities for specifying named objects, e.g., subscripted variables, record fields in PL/1 and base plus displacement addressing in conventional machine level languages such as System 360,

   c. procedural control structures such as subroutines, coroutines, IF-THEN-ELSE, DO WHILE constructs.

To cope with these assumptions, a UHM must have the following properties:

1. powerful shift and mask instructions which facilitate the extraction and examination of arbitrary bit strings.

2. associative memory or instructions that aid in the table look-up that is needed to simulate an associative memory,

3. i) high memory resolution, i.e., the ability to view the memory space as a bit string,

   ii) residual specifications of data structures to enable memory to be simultaneously viewed in a more structured fashion,

4. i) good functional resolution, i.e., primitive operations from which functions of arbitrary complexity may be synthesized,

   ii) high parallelism so that performance may be preserved despite the existence of a primitive functional capability,

   iii) structural resolution, viz., the ability to manipulate and reconfigure the data paths and interconnectivity of the functional units at a detailed level,

   iv) residual control over those aspects of the datapath structure which are relatively static.

The functional operations provided in the universal host machine should include those that can be thought of as forming the common denominator of the semantic capabilities that are encountered in all DIR's that the UHM may be called upon to interpret. Considering the diversity of existing and conceivable HLR's, the commonality of the corresponding DIR's will exist only at a rather low semantic level. Performance of the UHM may be retrieved by the provision of a number of primitive functional units which function concurrently. A shortcoming of most microprogrammable machines when reviewed as UHM's is that data, in the course of a register-to-register transfer, undergoes just a couple of elementary transformations (e.g., an add and a shift). The availability of a large number of busses and functional units and a powerful restructuring capability would permit the hardware to be configured, on a static or dynamic basis, to reflect the data flow graph of complex operators. As a result, more significant transformations could be performed in one register-to-register transaction. Thus, whereas the compiler binds the HLR down towards the hardware, the ability to restructure the data flow topology binds the hardware up towards the DIR.

Primitive operations, a certain amount of parallelism and a limited restructuring capability are found in horizontally microprogrammable machines. Residual control over these abilities can allow for shorter instructions without much sacrifice of power. Although elementary operations are necessary for the synthesis of arbitrary functions, this does not preclude the presence of very powerful features aimed specifically at the task of interpretation. Two examples have been noted above - powerful shift, mask and extract instructions and instructions which support table look-up. To be discussed next are architectural features built around the presence of a dynamic translation buffer. We note that many of the features listed above as desirable are present to a greater or lesser extent in recent microprogrammable processors.[2,11-14] Consequently, any one of these machine architectures could be used as the basis for a UHM architecture that includes a DTB.

6.2. Features Specific to the Use of a DTB

The organization of a universal host machine incorporating a dynamic translation buffer is shown at the block level in Figure 3. It is best viewed logically as two machines in one, a semantic processor which executes the semantic routines, and a procedural processor which executes P-DER code and steers control between the semantic routines. Both machines share certain common resources. This specialization of function permits each processor to be designed to best fit its task. The semantic processor is obtained by ignoring the portion within the broken lines. The semantic routines and the dynamic translation routine reside in the level-1 (fast) memory. Instructions from the level-1 memory are presented to Instruction Unit 1 (IU1) which generates the control word which specified the control points. The instruction set recognized by IU1 should reflect the desirable features listed in the previous section. Since detailed control must be exercised over the data paths, the instruction format is of the horizontal type, i.e., long and unencoded. Residual control may be used to shorten the instruction.

The procedural processor consists of Instruction Unit 2 (IU2), the DTB and certain other resources of the UHM. The sole function of the procedural processor is to call semantic routines and to pass parameters to them. Consequently, IU2 recognizes just four instructions: CALL, PUSH, POP and INTERP. In view of the obvious advantages of a stack in managing the entry and exit from procedures, it is assumed that one exists for stacking return addresses and operands. The small instruction repertoire requires a short opcode field which is desirable since space in the DTB is at a premium. The CALL, PUSH and POP instructions each have three variations to allow the operand to be specified immediately, directly or indirectly.

The most important short format instruction is the INTERP instruction which exercises the DTB. The operand of this instruction is the address of a DIR instruction in the DIR address space. The INTERP instruction causes this address to be presented to the associative address array of the DTB. A hit indicates that the P-DER translation of the DIR instruction is present in the DTB and control is transferred to that sequence of P-DER instructions. The last instruction in this sequence is another INTERP instruction which transfers control to the P-DER version of the next DIR instruction which is to be executed. When the next DIR instruction is known unconditionally (i.e., the sequential successor or the target of an unconditional branch) the operand of the INTERP instruction is supplied immediately. When the next DIR instruction address has to be computed, the result may be left on the operand stack for use by the INTERP instruction. The INTERP instruction, therefore, must come in two flavors depending on whether the operand is specified immediately or left on the stack. Overall, the short format instruction set is similar to a simple one address plus stack instruction set except, of course, for the INTERP instruction.

If the hit ratio in the DTB were unity, as it will be while the DIR program is in a tight loop, the execution of one sequence of P-DER instructions would lead directly to the execution of the next sequence. The UHM would spend all its time in performing computation related to the semantics of the DIR program instead of performing overhead tasks such as parsing, informatic theoretic decoding and binding which constitute computation that is not inherent in the algorithm of the DIR program but is the result of the mismatch between the representation of the program and the hardware.

Averaged over the entire execution of a program, the hit ratio will, of course, be less than unity. The sequence of actions that result when a miss occurs is as follows (Figure 4): the INTERP instruction presents to the DTB a DIR address for which a match is not found in the associative address array. This causes a trap to the dynamic translation routine, the pointer to which is maintained in a dedicated register, DTRPOINT. Simultaneously, the replacement logic of the DTB chooses the location into which the P-DER translation is to be placed, stores the DIR address in the associative tag array and makes available to the dynamic translation routine the pointer to the location in the DTB at which the P-DER translation is to be stored. The dynamic translator fetches the DIR instruction, decodes and parses it, generates the P-DER translation which it then stores in the DTB at the selected location. Lastly, it resumes normal operation by transferring control to the first instruction in the P-DER translation. The dynamic translator does slightly more than a conventional interpreter in that it performs analysis (of the DIR instruction) as well as code generation (of the P-DER sequence) instead of merely transferring control to the semantic routines. In this respect, it has something in common with a compiler. However, since the mapping from DIR to P-DER is almost one-to-one, the extra computation is not significant and is easily masked by the number of times that the task of decoding and parsing is avoided.

The control word is specified by one or the other of the instruction units depending on which one currently possesses control. When IU2 encounters a CALL to a semantic routine (expressed in long format instructions), control is handed over to IU1. The last instruction in the semantic routine causes a return to the dynamic translation of the DIR instruction and automatically returns control to IU2. IU2 only executes instructions fetched from the DTB. The Instruction Fetch Unit (IFU) decides which instruction unit is to be active depending on whether the instruction it is fetching is from the DTB or elsewhere.

The DTB is shown in Figure 3 as a separate resource, but the address array and the buffer array actually form part of the level-1 memory. Similarly, although the two instruction units are shown as separate resources, common portions may be shared unless it is desired to have concurrent operation of both units in an overlapped mode of execution.

7. Performance Analysis of the Dynamic Translation Buffer

In this section, expressions are derived for the average DIR instruction interpretation rate as a function of the parameters that affect the performance to the greatest extent. These parameters are:

Hardware dependent

- τ_1 - the level 1 access time
- τ_2 - the level 2 access time
- τ_D - the access time to a DTB or cache (nominally $2\tau_1$)

Language dependent

- d - average decode time per DIR instruction
- g - average time to generate and store the P-DER version of a DIR instruction (after decoding has been performed)
- x - average time to perform the semantics of a DIR instruction
- s_1 - average number of level 1 memory references to access the P-DER version of one DIR instruction
- s_2 - average number of level 2 memory references to access one DIR instruction

Program behavior dependent

- h_C - the average hit ratio in a cache (of stated capacity) used to buffer DIR instructions
- h_D - the average hit ratio in a DTB (of stated capacity).

Three cases are of particular interest; the performances of a conventional UHM and that of a UHM equipped with a DTB provide a measure of the

benefit derived from a DTB. However, the comparison is not quite valid since a UHM with a DTB has more resources than a UHM without one. Therefore, the case of a UHM with an instruction cache on the level 2 memory will be studied, too. For the sake of simplicity, it will be assumed that the instruction fetch and decode are not overlapped and that no instruction prefetch is active. Overlap between operand fetch and other computation is permitted since it is all lumped into the parameter x and is common to all strategies.

1. Conventional UHM

$$T_1 = s_2\tau_2 + d + x.$$

The average instruction interpretation time is composed of three components: the instruction fetch time, the time to decode it and the time spent in the semantic routines.

2. A UHM equipped with a DTB

$$T_2 = s_1\tau_D + (1-h_D)s_2\tau_2 + (1-h_D)(d+g) + x.$$

In this case, the normal instruction fetch time is given by the first term, but, on the occurrence of a miss in the DTB, level 2 memory must be accessed for a DIR instruction (the second term). This DIR must be decoded and translated, which accounts for the third term.

3. A UHM equipped with a cache

$$T_3 = h_C s_2\tau_D + (1-h_C)s_2\tau_2 + d + x.$$

The first two terms account for the average instruction fetch time. Every DIR instruction interpreted must be decoded. For the same capacity for the cache or DTB, h_C will be closer to unity than will h_D since the DIR representation is more compact.

Two important figures-of-merit for the DTB strategy are given by F_1 and F_2 where

$F_1 = \frac{T_3-T_2}{T_2} \times 100$ i.e., the percentage degradation caused by using the DTB as an instruction cache

and

$F_2 = \frac{T_1-T_2}{T_2} \times 100$ i.e., the percentage degradation caused by not using a DTB.

The evaluation of F_1 and F_2 is hampered by the lack of suitable statistics. A number of the parameters are very dependent upon the type of program, the static and dynamic representations and the architecture of the host machine. The figures of merit would have to be evaluated for each specific case. We shall, however, calculate F_1 and F_2 for representative and plausible values of the parameters.

The unit of time is taken to be the access time of the level 1 memory which is also assumed to be equal to one machine instruction execution time. Therefore, $\tau_1 = 1$. τ_D is assumed to be $2 \times \tau_1 = 2$ and τ_2 is assumed to be $10 \times \tau_1 = 10$. g is set equal to $1.5 \times d$ and S_2 is taken to be 1 and S_1 is chosen to be 3. Thus the dynamic representation of one DIR instruction is assumed to be three times as long on the average as the DIR instruction. A study of the literature on cache memories[28,30,36] indicates that a choice of 0.9 for h_C is reasonable for a cache size of 4096 bytes. The effective DTB size is 4096/3 bytes since $S_1 = 3S_2$. A reasonable value for h_D, then, is 0.8. Substituting these values into the above equations gives

$$F_1 = \frac{0.4+0.6d}{8+0.4d+x} \times 100$$

and

$$F_2 = \frac{7.4+0.6d}{8+0.4d+x} \times 100$$

where d, the average number of instructions spent in decoding a DIR instruction, and x, the average time per DIR instruction spent in the semantic routines are yet to be specified.

The parameter d is very dependent upon the extent of encoding and the hardware features of the host machine. The provision of powerful field extraction instructions reduces d. However, the use of frequency based encoding increases the number of levels of decoding needed. For each field, for each level of decoding, at least two instructions are needed; the first one extracts the field (or a portion thereof) and increments the program counter by that amount, causing a CASE STATEMENT type of branch to a list of branch instructions. The selected branch instruction must then be executed, thereby transferring control to either a semantic routine or to another routine which continues decoding at the next level. Thus even with a powerful host architecture, d could easily be equal to 10. For simpler host architectures, d might well be twice as large if not more. The parameter x can vary greatly depending on the nature of the DIR and the architecture of the host.

Tables 1 and 2 list F_1 and F_2 for various values of d and x. The figures demonstrate that the DTB does have the potential to improve performance significantly. The actual values of F_1 and F_2 for any given situation must, of course, be evaluated for the specific values that the parameters assume in that particular case. In general, the figures-of-merit decrease as d decreases or as x increases. Thus the DTB is not particularly effective if the task of decoding is trivial or if the time spent in the semantic routines is much greater than the time that would be spent in decoding. This would be true, for instance, in machines with vector instructions which are heavily used.

8. Conclusion

The architecture and instruction set of a processor is determined by the class of language that will be executed (interpreted) by it, either directly or following compilation. If this class is restricted, the application of the processor

is fairly specific and the instruction set will be at a high level and closely matched to the single or small number of high-level languages that are supported by the processor. The several examples of high-level machine designs fall into this category.

On the other hand, if the class of languages is large and vague, commonality of semantics will exist only at a very low level and the instruction set of the universal host machine will be primitive. Under such circumstances, the high-level language and the machine language are extremely dissimilar and it is more efficient, both in space and time, to interpose an intermediate level, a directly interpretable level, into which the program is compiled and which is interpreted by an interpreter written in the machine language. An intermediate language is characterized by its position in a two-dimensional space of which one dimension is the semantic level of the language and the other dimension is the degree of encoding.

However, the choice of the intermediate language is complicated by the fact that it is possible to trade-off execution time against the size of the intermediate language program representation. The concept of dynamic translation has been introduced to overcome this dilemma. The dynamic translator permits the program to be present in a compact, static representation but maintains the working set in a dynamic representation that lends itself to speedy execution. The dynamic translator dynamically translates instructions from one representation to the other as they enter the working set. Expressions were derived for two figures-of-merit of this scheme and they were evaluated for certain typical values of the relevant parameters, demonstrating the potential performance benefits of this scheme.

The decoding overhead of a universal host machine may be reduced either by providing powerful hardware aids to the decoding process or by the use of a dynamic translation buffer which decreases the number of instructions that need be decoded. The former approach requires the addition of random logic whereas the latter approach relies on the use of memory. This fact may influence the relative cost-effectiveness of the two schemes. Future research will be aimed at gathering statistics which permit a more quantitative evaluation of the cost-performance of various combinations of intermediate representations and universal host machine architectures, with and without dynamic translation buffers.

Acknowledgments This paper has benefitted greatly from the long discussions between the author and Michael Schlansker and from the very constructive criticism offered by George Rossmann and Michael Flynn. Any residual deficiencies are to be credited solely to the author.

References

1. M. V. Wilkes, "The best way to design an automatic calculating machine," Manchester Univ. Comput. Inaugur. Conf., 1951, p. 16.

2. W. T. Wilner, "Design of the B1700," AFIPS Conf. Proc., 1972 FJCC, 41, 489-497, Montvale, NJ, AFIPS Press.

3. W. T. Wilner, "Burroughs B-1700 Memory Utilization," AFIPS Conf. Proc., 1972 FJCC, 41, 579-586, Montvale, NJ, AFIPS Press.

4. L. W. Hoevel, "DELTRAN Principles of Operation: A Directly Executed Language for FORTRAN-II," Tech. Note No. 108, Digital Systems Laboratory, Stanford Univ., Stanford, CA, March 1977.

5. J. P. Anderson, "A computer for direct execution of algorithmic languages," Proc. EJCC, 1961, 184-193.

6. Y. Chu, "Introducing the high-level language computer architecture," Tech. Rep. No. TR-227, Comput. Sci. Center, Univ. Maryland, College Park, Maryland, 1973.

7. H. M. Bloom, "Design and simulation of an ALGOL computer," Tech. Rep. No. 70-118, Comput. Sci. Center, Maryland, College Park, Maryland, 1970.

8. T. R. Bashkow, A. Sasson and A. Kronfeld, "System design for a FORTRAN machine," IEEE-TEC, Aug. 1971, 485-499.

9. M. Sugimoto, "PL/1 reducer and direct processor," Proc. ACM, 1969, 519-538.

10. R. Rice and W. R. Smith, "SYMBOL - A major departure from classic software dominated von Neumann computing systems," SJCC, 1971, 575-587.

11. E. W. Reigel, V. Faber and D. A. Fisher, "The interpreter - a microprogrammable building block system," AFIPS Conf. Proc., 1972 SJCC, 40, 705-723, Montvale, NJ, AFIPS Press.

12. H. W. Lawson and B. K. Smith, "Functional Characteristics of a Multilingual Processor," IEEE-TC, 20, July 1971, 732-742.

13. Nanodata Corp., "QM-1 Hardware Level User's Manual," Second Edition, August 1974.

14. M. J. Flynn, C. J. Neuhauser and R. M. McClure, "EMMY - an emulation system for user microprogramming," AFIPS Conference Proceedings, 1975 NCC, 85-89.

15. L. W. Hoevel, "'Ideal' directly executed languages: an analytic argument for emulation," IEEE-TC, 23, 8, 1974, 759-767.

16. L. W. Hoevel and M. J. Flynn, "The Structure of Directly Executed Languages: A New Theory of Interpretive System Support," Digital Systems Lab. Tech. Rep. No. 130, Stanford Univ., March 1977.

17. G. Radin, "A note on the concept of binding," IBM Thomas J. Watson Res. Rep. No. RC 3287, Yorktown Heights, NY, March 1971.

18. P. Naur, (Ed.), "Revised report on the algorithmic language ALGOL 60," CACM 6, Jan. 1963, 1-17.

19. K. E. Iverson, "A Programming Language," Wiley, New York, 1962.

20. J. R. Bell, "Threaded Code," CACM, 16, 6, June 1973, 370-372.

21. J. B. Johnston, "The Contour Model of Block Structured Processes," Proceedings of the SDSPL (SIGPLAN Notices, Vol. 6) Feb. 1971, 55-82.

22. D. A. Huffman, "A method for the construction of minimum redundancy codes," I.R.E., 40, 9, Sept. 1952, 1098-1101.

23. C. C. Foster and R. Gonter, "Conditional Interpretation of Operation Codes," IEEE-TC, Jan. 1971, 108-111.

24. E. C. R. Hehner, "Computer design to minimize memory requirements," Computer, 9, 8, Aug. 1976, 65-70.

25. E. C. R. Hehner, "Information Content of Programs and Operation Encoding," JACM, 24, 2, Apr. 1977, 290-297.

26. B. S. Brawn and F. G. Gustavsen, "Program Behavior in a Paging Environment," AFIPS Proceedings, 33, FJCC, 1968, 1019-1032.

27. P. J. Denning, "The working set model for program behavior," CACM, 11, 5, May 1968, 323-333.

28. K. R. Kaplan and R. O. Winder, "Cache-based Computer Systems," Computer, 6, 3, March 1973, 30-36.

29. D. H. Gibson, "Considerations in Block-Oriented Systems Design," Proc. SJCC, 1967, pp. 78-80.

30. R. M. Meade, "Design Approaches for Cache Memory Control," Computer Design, 10, January 1971, 87-93.

31. T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Summer, "One-level Storage Systems," IRE Trans. Elec. Comp., 11, 2, 1962, 223-235.

32. P. J. Denning, "Virtual Memory," Computing Surveys, 2,3, 1970, 153-189.

33. B. Randell, "A Note on Storage Fragmentation and Program Segmentation," CACM, 12, 7, July 1969, 365-369.

34. C. J. Conti, "Concepts for Buffer Storage," Computer Group News, 2, March 1969, 9-13.

35. S. H. Fuller, V. R. Lesser, C. G. Bell, and C. M. Kaman, "The Effects of Emerging Technology and Emulation Requirements on Microprogramming," IEEE-TC, 25, 10, Oct. 1976, 1000-1009.

36. W. D. Strecker, "Cache Memories for PDP-11 Family Computers," Third Annual Symposium on Computer Architecture, 1976, 155-158.

x \ d	10	20	30
5	37.7	59.1	73.6
10	29.1	47.7	61.3
15	23.7	40	52.6
20	20	34.4	46
25	17.3	30.2	40.9
30	15.2	27	36.8

Table 1. Percentage increase in the average DIR instruction interpretation time due to using the DTB as a cache on the level-2 memory.

x \ d	10	20	30
5	78.8	92.4	101.6
10	60.9	74.6	84.7
15	49.6	62.6	72.6
20	41.9	53.9	63.5
25	36.2	47.3	56.4
30	31.9	42.2	50.8

Table 2. Percentage increase in the average DIR instruction interpretation time due to not using the DTB.

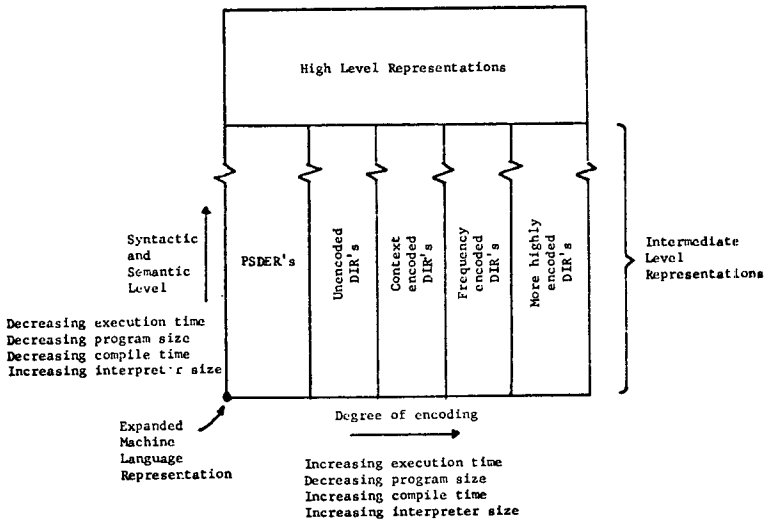

Figure 1. The space of program representations.

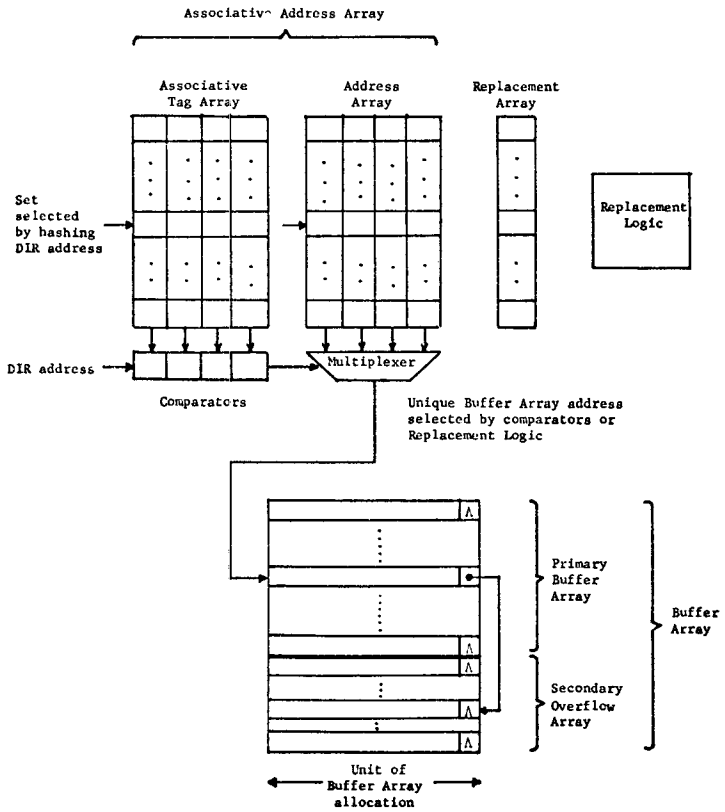

Figure 2. Organization of the Dynamic Translation Buffer.

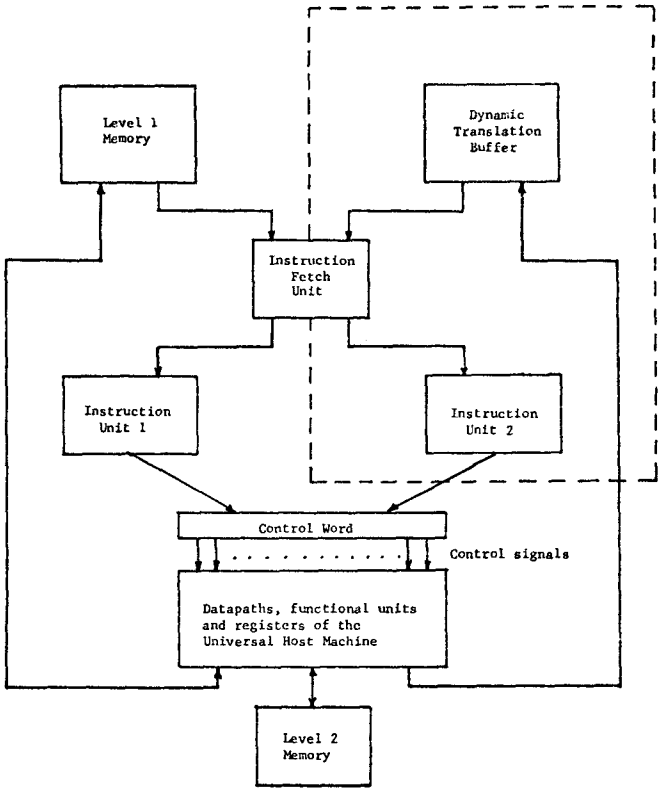

Figure 3. Organization of the Universal Host Machine.

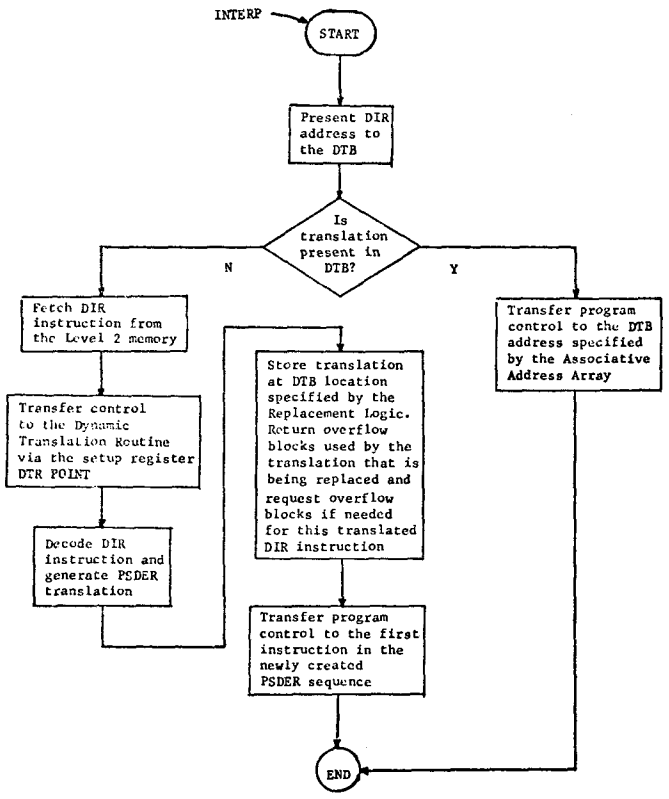

Figure 4. Flow diagram for the INTERP instruction.