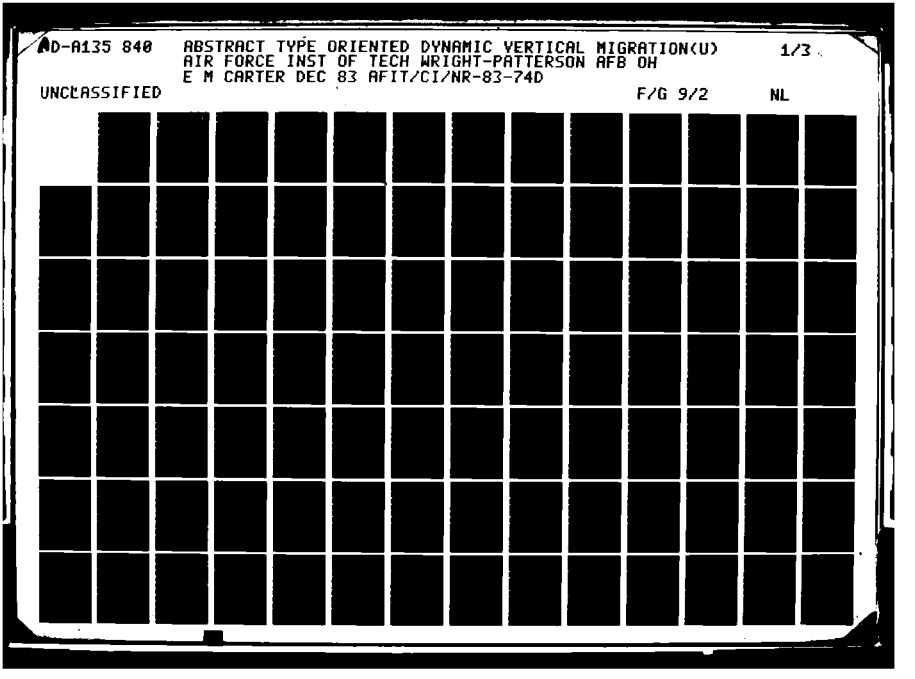
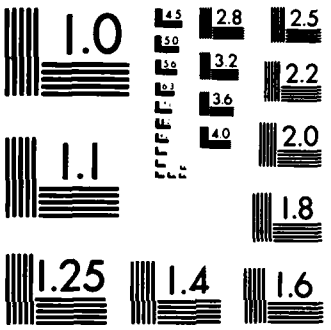

AD-A135 840 ABSTRACT TYPE ORIENTED DYNAMIC VERTICAL MIGRATION(U) 1/3
AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
E M CARTER DEC 83 AFIT/CI/NR-83-74D
UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ABSTRACT TYPE ORIENTED DYNAMIC VERTICAL MIGRATION

By

Edward M. Carter

Dissertation
Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of
DOCTOR OF PHILOSOPHY

in

Computer Science

December, 1983

Nashville, Tennessee

Approved:	Date:
[signature]	15 Nov 83
[signature]	15 Nov 83
[signature]	15 Nov 83
[signature]	15 Nov 83
[signature]	15 Nov 83

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1 REPORT NUMBER AFIT/CI/NR 83-74D	2 GOVT ACCESSION NO AD-A135840	3 RECIPIENT'S CATALOG NUMBER
4 TITLE (and Subtitle) Abstract Type Oriented Dynamic Vertical Migration		5 TYPE OF REPORT & PERIOD COVERED THESIS/DISSERTATION
		6 PERFORMING ORG. REPORT NUMBER
7 AUTHOR(s) Edward M. Carter		8 CONTRACT OR GRANT NUMBER(s)
9 PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: Vanderbilt University		10 PROGRAM ELEMENT PROJECT, TASK AREA & WORK UNIT NUMBERS
11 CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433		12 REPORT DATE 1983 13 NUMBER OF PAGES 185
14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15 SECURITY CLASS. (of this report) UNCLASS 15a DECLASSIFICATION DOWNGRADING SCHEDULE

16 DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17 DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18 SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-17

8 DEC 1983

Lynn Wolaver

LYNN E. WOLAVER
Dean for Research and
Professional Development

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

83 12 14 011

COMPUTER SCIENCE

ABSTRACT TYPE ORIENTED DYNAMIC VERTICAL MIGRATION

EDWARD M. CARTER

Dissertation under the direction of Professor R. I. Winner

The study of structured programming has shown that through data abstraction, program reliability and maintainability can be improved. At the same time, vertical migration has been shown to be an effective way to improve the performance of programs. Contemporary techniques, however, tend to address the needs of only certain classes of programs and therefore may overlook or even preclude certain optimization opportunities. Dynamic microprogramming can overcome the problem of applicability of a particular vertical migration by allowing the migration to be tailored for each particular application.

This research describes a study of a technique known as abstract type oriented dynamic vertical migration. This technique involves determining the needs of a program in differing execution environments and tailoring the architecture of the machine to support those environments individually. The paper describes foundation work in the areas of computer architecture, dynamic microprogramming, data abstraction, and vertical migration and describes how these can be integrated to form a computer architecture which is adaptable to user need while providing a means of encouraging modern programming practices without incurring the performance degradation as is often seen in current architectures.

The implementation of this architecture is discussed as well as the effects of modern programming languages and architecture implementation techniques on the proposed technique. The result of the research is that abstract type oriented dynamic vertical migration has been shown to be an effective technique which can be used to enhance the performance of programs while providing a degree of reasoning about the interaction of migrated function which has yet to be attained.

Author: Edward M. Carter
Rank: Captain
Branch: United States Air Force
Title: Abstract Type Oriented Dynamic Vertical Migration
Year: 1983
Pages: 185
Degree: Doctor of Philosophy in Computer Science
Institution: Vanderbilt University

ACKNOWLEDGEMENTS

I would like to acknowledge the efforts of many people in making this research possible. I first need to point out that without the endless support of my wife, Margaret, and encouragement from my three children, Aaron, Jason, and Steven, this work could not have been completed. I would like to forgive Steven for the month long pause he caused in my writing. Of next importance is the contribution of my advisor Dr. Robert I. Winner. His expertise and encouragement have brought me to this stage of my work. My time at Vanderbilt has been enjoyable for two reasons. First, the faculty has continually challenged and encouraged me to learn new concepts and to practice them. For this I am very grateful. Secondly, and no less important, are the friendships which I have shared while studying here. I thank Stan Thomas, Len Reed, Eric Roskos, Richard Johnson, Fred Nixon, Tom Wood and many others for being special colleagues with whom I have shared many a gripe and a great deal more laughs. The United States Air Force Academy made the time for my study possible and the Air Force Institute of Technology provided my support. Of most importance, I will acknowledge my Lord and Savior Jesus Christ in knowing Him, and being loved by Him as being the single greatest part of my life.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER I

INTRODUCTION

Subsequent to the introduction of computers in the early 1940's, much has changed in the way in which computers are used. These changes include broader application areas, vastly increased numbers of computers, and advances in the way in which computer programs are designed and implemented. Change has not occurred as quickly in the basic data processing facilities, known as the architecture, of computers. Newer and faster components have been introduced and miniaturization has brought the mammoth early computers down to the more manageable size of the personal computer of today with an increase in functionality. The basic architecture however, is still based on the von Neumann model of the early machines. The purpose of this research is to examine ways to improve this basic architectural schema.

In the following pages the reader will find an examination of computer architecture and how advances in the two disciplines of software engineering and firmware engineering can be combined to produce a computer architecture which better supports user needs than the simple von Neumann architectures of today. In chapter II the four areas of computer architecture, dynamic microprogramming, abstract

data typing, and vertical migration will be surveyed. Emphasis will be placed on how these may be adapted and integrated to provide a better computing environment which is oriented toward one particular problem or set of problems. Within this chapter will also be a summary of the need for this research and a brief description of the problem to be examined.

Chapter III will present a proposed solution for the problem, known as abstract type oriented migration (ATOM), and will describe some measures which may be used in evaluating its applicability. Chapter IV will look at some of the effects which modern programming languages have on ATOM. In particular the effect of lexical scope, derived data types, and generic data types will be examined. Chapter V will look at the effects which modern hardware implementation techniques, such as multiprocessing and pipelining, have on the the new architecture. Within this chapter is an examination of ways in which ATOM may affect future hardware decisions and capabilities of future architectures. Chapter VI will be an in-depth look at an implementation of ATOM and will study several examples of its application. The final chapter, chapter VII, will summarize the results of the research and describe directions which further efforts should pursue in advancing dynamic redefinition of computer architecture.

CHAPTER II

BACKGROUND

Overview

The purpose of this chapter is to survey the areas of computer architecture, abstract data typing, vertical migration, and dynamic microprogramming and to highlight the work which has been done in these areas. With this background the reader will understand the need for ATOM and will more readily appreciate its usage and constraints. The first section of this chapter will briefly focus on the process of problem solution and how the resulting solution is implemented on a computer through program design and implementation. The next four sections of this chapter are dedicated to the study of the four areas mentioned previously and how they affect the programming environment. The final three sections of the chapter will summarize the need for ATOM, precisely define the problem being solved, and describe the environment in which the research was conducted.

Problem Solving and Programming

When a person is confronted with a problem, an orderly problem solution process is often used to determine how best to meet the needs of the situation. The exercising of this process may be implicit in the case of simple problems or

explicit where greater complexity is involved. The following technique is a generalization of a large class of problem solving techniques.

The first step is to determine the factors which affect the solution; this step is often called data gathering. Once this process has occurred, several different solutions can be determined and a search begun for the best one. After this step has been completed the final move is to implement the problem solution in a way which will overcome the problem. This technique can be adapted to any problem area without regard for its complexity.

In solving problems on computers we very often follow this same process. The data gathering step involves collecting a set of specifications describing the constraints of the computing environment as well as the required input data and output results. The process of determining a solution involves describing a set of data structures and algorithms which access these structures to provide the transformations from input data to results. The final step, implementing the solution, is what is of interest in this research.

In this step we choose a programming language and describe our abstract problem solution in a more formal way. This formalism is bounded by the syntax and semantics of our chosen programming language. After the problem solution has been translated into the chosen programming language, it

must then be either interpreted by the computer or be translated to a lower form of language whose syntax and semantics are bounded not only by a formal description but by the architecture of the computer. This architecture is the set of facilities which the machine language programmer, hence the compiler's code generator, sees as the most basic tools which are available for problem solving.

In modern computers these facilities are the primitive data types and operations on these types which we have come to know as the instruction set. For example, we may know that a machine can support integer, character, and floating point objects. The problem solution, however, may need something more complex or even something more simple than what is provided in the architecture of the machine. In other words, the facilities required by the problem solution and the facilities provided by the computer's architecture may not agree. The result is that the semantics of the problem solution must be mapped onto those which are provided by the architecture in order for the solution to be implemented on a computer. The problem arises that the specific needs of the problem solution must be mapped onto an architecture which has been generalized to support a vast range of applications and which has been optimized for hardware realization.

The architecture of most computers today is based on the von Neumann model of the early 1940's. In this archi-

tecture, a primitive memory model is described which is the basis for the difficulties in mapping higher level structures onto the programming model of the architecture. The characteristics of the von Neumann model are :

1. A single, sequentially addressed memory.
2. A linear (one-dimensional) memory.
3. No distinction between instructions and data.
4. Meaning not being an inherent part of data.

On the other hand, a study of modern programming languages shows that in general the following characteristics are evident:

1. Storage is a set of discrete named variables.
2. Many data types are multi-dimensional.
3. Data and instructions are not the same.
4. Meaning is an inherent part of the data.

It is the disparity between these sets of characteristics which causes the difficulties in the mapping of abstract objects and operations to architecture.

The Semantic Gap

The idea of a gap between concepts supported in the architecture and those required by programmers has been described by several authors. An interesting treatment of the topic has been by Myers in [31]. The premise of Myers' book is that modern computer architectures have some serious shortcomings. He points out that these shortcomings are

based primarily on a "bottom-up" design methodology where a group of designers or computer architects decide what data types and operations to support. The design method is heavily influenced by tradition and may not be affected by the needs of the applications to be run on the machine. Myers defines the gap between the needs of the programmer and the facilities of the computer architecture as the semantic gap. This gap is primarily a measure of the difference between the concepts in high-level languages and the concepts in the underlying computer architecture used to provide these facilities. As we have seen before, there is also a semantic gap between the concepts in the problem solution and those which appear in the programming language. In actuality, there is a series of semantic gaps which start at the abstract problem solution and continue through implementation of the solution on a computer.

Myers points out several examples to illustrate the consequences of the semantic gap between the programming model described by programming languages and the concepts supported by contemporary architectures. For example, the IBM S/370 PL/I compiler generates 17 machine instructions occupying 62 bytes of memory for the following array assignment:

$$C(I,J) = A(I,J) + B(J,I)$$

If the optional SUBSCRIPTRANGE check is enabled the compiler generates 75 machine instructions for a total of 274 bytes

of memory. Note that this comparison is simply a static analysis of the problem and disregards dynamic behavior, such as the number of memory fetches required for data and instructions. An architecture which provided a primitive array type would have required fewer instructions since it would have known and supported the concept of arrays and subscripts. Furthermore, runtime bound checking could be implemented in the architecture as opposed to additional instructions being added to the program to provide this function.

The obvious results of the semantic gap are decreased machine efficiency and excessive program size. In addition, compiler and language complexity are increased to provide facilities which are not easily implemented in the architecture. Two additional problems arise, however, which are even more serious than those already mentioned. The first problem involves a drastic decrease in programmer productivity. In the von Neumann model, a programmer is forced to represent his model in terms of entities supported by the architecture. These entities may require a complex combination and interaction between more primitive structures supported by the architecture. The programmer is responsible for implementing these interactions and for maintaining them properly. A large portion of his programming efforts may then be spent in ensuring that these interactions and depen-

dencies are maintained resulting in less time being spent on more productive efforts.

An example can best serve to illustrate this point. Let us assume that we must represent a student entity on a computer. Since most computers do not have a student data type and instructions which manipulate items of this type, we must model the type by using the primitive types in the architecture. We may model the student entity as a record with some of the following components: a string of characters for a name, an integer phone number, an array of courses each consisting of a string for course name and a character for the letter grade. Let us also assume that there is an integer count of courses in this record. This type of record implies a dependency of the size of the record on the course count field. If the object is not supported in the architecture the dependency must be enforced by the programmer implementing the model. A change to one item, for example the course count, which should affect other items may be overlooked since there is no architecturally enforced relationship between them. The result of this error in the student record case may be the loss of new course data or the retention of deleted course data. The practice of forcing the model to fit the architecture is like giving a workman inappropriate tools for completing a task and expecting the job to be completed in an efficient and timely way. Imagine the increase in the effectiveness

of a programmer if he could work on an architecture which is capable of representing more closely the model which he is required to produce.

The second problem, software unreliability, is perhaps the most severe of those described. Consider once again the array case shown before. What is the result of using an array index which is outside the bounds of the array or, even worse, of using an array element which has yet to be assigned a value? Again, if the array type were implemented as a type of the architecture, then the supporting architecture could ensure that the selected element of the array was a valid one. These represent a class of problems which often arise and are a direct result of having to simulate operations on types which are not supported by the architecture.

The problems can be much worse when we consider other more complex types such as stacks, queues, and dynamic list structures. The point here is this: modern architectures support only a small number of data types and operations on these types, and this restriction causes unreliable software and excessive costs. The von Neumann architecture does not lend itself to supporting user defined data types. The problems with the von Neumann architecture arise from its primitive memory model. Excessive mapping of language structures to architecture is needed to match the language concepts to the von Neumann view of storage. The structure

of data consequently must be absorbed into the logic of the program. In addition, the von Neumann memory model is too general in that there is a need to provide the flexibility to use any word of memory for any purpose, i.e. instructions or data. This can result in the possibility of executing data or accessing instructions as data. Finally, the primitive storage concept of this model implies that the instruction set must support overly simplistic operations as its primitive operators. As a result of the von Neumann model, there exists the dichotomy that machine architects view data type as a property of operators while compiler writers view data type as a property of the data itself. Some attempts have been made to enhance the range of types supported in architectures; these are reported in the following sections.

Object Oriented Architectures

The object model as described by Jones in [25] is both a concept for design and a tool for implementing programs and systems. An object oriented architecture uses the model to change the perspective of the programmer. Instead of viewing the memory as a linear collection of storage cells, this architecture views memory as a heap which is used to allocate entities known as objects. Each of these objects is a self-identifying collection of items which share some common properties. For example, an executable program image may be an object which contains other objects known as seg-

ments. Each of these segments is in turn another object which contains machine instructions for carrying out the operations of the object. The data items to be operated on are also objects. Data structures as entities may also be objects. For example, an array of records may be implemented as a collection of record objects with the collection as a whole also being an object.

An object may also contain a list of other objects as well as access permissions for these. Most object oriented architectures today have implemented this capability-based system for controlling access to data. The strong point of object oriented architectures is that they provide a level of granularity which is adjustable to the needs of the programmer in the way objects are described. This level of granularity is useful further in providing protection domains for objects, the primary use of the object model in architectures today. Several architectures have been attempted based on this model including the iAPX-432 [38] and Hydra:C.mmp [68]. Both of these projects have suffered from a lack of efficiency caused by the overhead involved in maintaining the objects of interest. The object oriented architecture is plagued by this fault as noted by the lack of a widely accepted object oriented processor on the commercial market today.

The object oriented architecture is at the high end of a spectrum of complexity for modern computer architectures.

Another new architecture, which is heavily influenced by VLSI technology, is the so called reduced instruction set computer. The next section will briefly look at this proposed architecture.

Reduced Instruction Set Computers

The set of reduced instruction set computers (RISC) is a class of computers which have been designed to combat the decreased performance brought about by the semantic gap. The premise of RISC is that by providing a simple set of instructions which can for the most part be executed in one clock cycle, the architecture will provide an extremely fast model for implementing programming systems. The large number of instructions needed to model abstract data types will still be there and in fact will increase, but the mean time to execute one of these instructions will decrease. The burden of complexity is therefore shifted from the computer architect to the compiler designer.

Examples of these architectures include the Berkeley RISC-I [37], the Stanford MIPS [19], and the IBM 801 [39]. We can see that this solution does not address the complexity issue brought to light by the semantic gap but simply provides a solution to the efficiency problems which are encountered when excessive mapping from programming language structures to architecture is required. Although RISC machines provide a large increase in performance, the con-

comitant software complexity cost may not justify the adoption of the architecture.

The Software Dilemma

The problems mentioned above seem to present an insurmountable obstacle to efficient implementation of problem solutions. However, there is relief for this problem. Research in the areas of computer architecture, dynamic microprogramming, abstract data typing, and vertical migration have each contributed to improving the computing environment in their respective areas. A combination of these, however, can be seen as relief for many problems presented by the semantic gap. This integrated technique, abstract type oriented migration, will be described in detail in chapter III. In the immediately following sections, we will be looking at each of the contributing areas and work which has been completed in each.

Architecture

The programming architecture of a computer can be defined as the set of functions and interfaces which are available for use by programmers in typical programming languages such as assembler and procedure oriented languages. For our purpose, we can equate this to the set of operations provided by the instruction set and the data types on which these instructions operate. Since the advent of the von Neumann architecture in the late 1940's, little

has changed in the basic schema of computer architectures except that more complex data types and instructions are being provided. This set of types and operations is still established by the computer architect and restricts the applications which can be implemented efficiently to those that can be mapped efficiently onto this set.

In the last ten years increasing scrutiny has been given to this restricted set of data types and operations and to the effect it has on reliability and maintainability of software. Some measures have been attempted which have to date resulted in working, yet not efficient, architectures for solving this problem. For the most part, only simple modifications to the basic von Neumann architecture have been successful.

A computer architecture can be implemented in one of two ways. In the first way, sequential and combinational circuits can be built which provide the functions required when activated by the control circuitry of the machine. This way is called the hardwired approach. It is extremely fast, but is built to provide a single set of functions and can only be modified through hardware change. Figure 1 illustrates this type of architecture schematically.

The alternative method of implementing an architecture is through microcoding. In the microcoding approach the hardwired circuitry is replaced by a set of microinstructions which control the hardware resources and data paths of

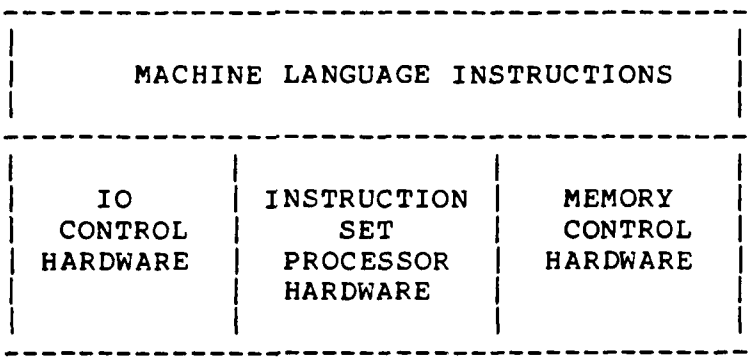


Fig. 1--Hardwired Architecture Implementation

the machine. Each bit of these microinstructions may control as small a resource as a single gate or as large a resource as a control line to a functional unit such as the ALU of the computer. In this approach one or more microinstructions may replace a sequential or combinational circuit of the hardwired approach. This approach yields a large degree of flexibility yet costs in efficiency of execution. Operations which were implemented in hardware are now dependent on one or more microinstructions which must be fetched, decoded, and executed from a control store. The control store, although implemented in fast memory, adds a degree of overhead not in the hardwired approach. The resulting payoff is that the instruction set of the computer and hence its architecture can be changed by simply loading the control store with a new set of microinstructions. Figure 2 illustrates the microcode approach to architecture implementation. For the rest of this paper, the terms machine instruction and macroinstruction will be used interchangeably. Similarly, macrocode and microcode will refer to programs consisting of macroinstructions and microinstructions respectively. Since the microprogrammed approach provides the ability to change the architecture without hardware change, we will concentrate on this method of implementing architectures.

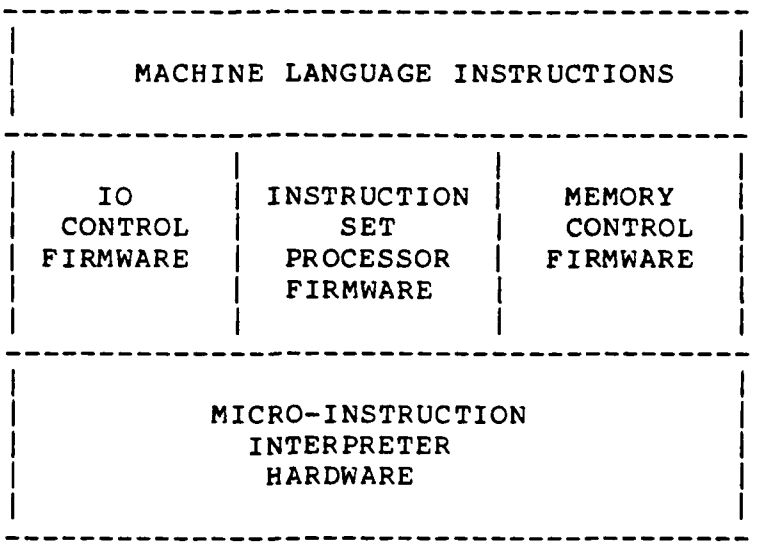


Fig. 2--Microcoded Architecture Implementation

Dynamic Microprogramming

For this paper, we will define dynamic microprogramming as the ability of a system user to place microcode dynamically into a writable control store, thus effectively extending the programming architecture of the machine. Architecture synthesis is the application for dynamic microprogramming in which this research is most interested. According to Stockenberg and van Dam in [50] there are two types of architecture synthesis, manual and heuristic. The most well known synthesis technique is manual synthesis and has been seen in numerous applications such as language accelerators, operating system assists, and migrations of specific user functions to firmware. This type of migration is very often quite complex and is usually restricted to only a small class of programs.

In heuristic synthesis an attempt is made to discover what portions of the program should be microcoded to yield the best performance improvement. This second type of architecture synthesis is of most interest to our research. The search for the best code to place in control store is made in an automated or semi-automated procedure so that the complexity is reduced and the applicability of the procedure can be greatly expanded. Earlier work by Abd-Alla [1] and El-Ayat [11] have both found that automated techniques can be implemented for dynamically redefining computer architecture.

A topic of concern at this point is how to create the microcode and load it into the control store once the functions to be placed there have been discovered. One approach to changing the architecture to match specific user needs is through language directed computer architecture. In this approach an architecture is synthesized by the compiler writer when the compiler is implemented. This architecture becomes the target or image architecture for which programs are compiled. The implication is that this architecture will be realized on the machine at runtime by loading the control store with an emulator for that architecture. This emulator is specific for all programs written in the target high-level programming language. This approach has been used in the design and implementation of the Burroughs B1700 and is described in [56] and [57]. A more general method involves describing a single high-level architecture for supporting all programming languages. Wortman in his dissertation examined the applicability of this approach and demonstrated such a system using a subset of PL/I [65]. This system is similar to contemporary architectures with the exception that the primitive operations are at a more abstract level. The decision is still made a priori as to what data types and operations are to be part of the architecture.

A more specific approach to synthesis is what we will call a problem oriented architecture. This technique will require a flexible way of describing architectures and emu-

lators. Gieser reports on techniques which can be used for describing microarchitectures in [18]. Other recent advances in microprogramming language research have brought to light the fact that high-level microprogramming languages can be used for implementing architectures without losing efficiency. Sint in [46] surveys a large number of these high-level microprogramming languages. Each of the languages described is associated with a microcode compaction facility which provides for an efficient flow of control among microinstructions. Common high-level programming languages have also been adapted for use as microprogramming languages. One effort with the language C is discussed by Ellison in [12]. Most hardware vendors which supply user microprogrammable hardware also provide at least an assembly language facility for user microprogramming. Some third party software vendors supply more extensive microcoding packages such as the VAX Automatic Microcode Generation facility discussed in [45].

Data Abstraction and Encapsulation

One of the largest strides in programming methodology in recent years has been the drive toward the utilization of structured programming techniques. Perhaps the best known of these techniques is that of information-hiding. In information-hiding, presented in [36], the major goal is to "hide" the representation of data within a capsule or module

which provides controlled access to the data through a set of entry points. A programmer need only know how to call the entry point and what results to expect from the call in order to use the encapsulated data type.

The facility may be as simple as an integer type or arbitrarily complex. The simple facilities are normally provided directly in the computer architecture and we will refer to these as primitive types. The more complex facilities are specific to the application and we will call these abstract types. An abstract data type is simply a representation of a class of objects and the operations which can be performed on them. These operations define a set of invariant properties and must maintain these properties for the type. An example of one of these properties is that all of the data values must come from a well-defined set of allowable values. The operations which provide for assigning values to objects of the type will ensure that only allowable values are used. In this way only values from the set of allowable values can be given to items of the type. The presence of these invariant properties as part of the type description provides a useful tool for axiomatic verification of program behavior. This verification is necessary where types may be migrated into firmware and which effectively become part of the computer architecture for the particular problem under study. Further significant research has been accomplished which further addresses formal methods

for describing, implementing, and verifying the behavior of abstract data type facilities. These works include: Gannon [16], Thatcher [52], Jones [24], and Herlihy [20].

Some of the earliest work on abstract data types is described in [10] and [61]. In these reports Dijkstra and Wirth demonstrate that a program is simply a model of some physical or abstract system which is iteratively refined until it is implementable on a computer. In each of the iterative steps, some information about the model is simplified or deleted to yield a new abstract description of the model. This description is called an abstraction of the model. For the duration of the paper, model and abstraction are used interchangeably.

Horning in [22] lists some useful properties of data abstraction facilities. Each of these facilities can be seen to provide a useful tool for dynamic microprogramming. These properties are:

1. Avoidance of repetition of code sequences.
2. Modular program structure.
3. A basis for structured programming.
4. Conceptual units for understanding and reasoning about programs.
5. Clearly defined interfaces that may be precisely specified.
6. Units of maintenance and improvement.

7. A language extension mechanism.

8. Units of separate compilation.

The first property provides for a simple method of compacting programs by avoiding repetitive sequences of instructions. The entire sequence is replaced by a single instruction which causes the appropriate action to be performed on an object of the abstract type. The concepts of modularity and interfacing standards form the basis for a type mechanism by completely and unambiguously specifying the representation of the type and the operations which can be performed on objects of the type. The remaining concepts give us a syntactic basis for encapsulating a type such that controlled access to objects of the type, hence possible migrated code, can be enforced.

Data abstraction therefore, forms a new set of types; let us call this set the types of interest for this particular model. The components of the objects of the types of interest are formed by combining the primitive data types and implementing procedures to manipulate these components to implement the abstract type. It is the responsibility of the programmer to describe the operations on these objects as well as their representations. It is the job of the compiler to translate this representation and operations into a coherent set of directives or machine instructions which can be understood by the architecture of the host computer.

Data encapsulation is the process of grouping the object representations and operations of the abstract data type together. In modern programming languages this is done by describing a user defined data type or structure and the procedures which access the elements of this type. Another requirement of encapsulation is that the objects of the abstract type must be protected from unauthorized access. The only method of creating, updating, or deleting these objects must be through the provided routines. These routines maintain the invariant properties of the objects of the abstract type.

The concepts of data abstraction and encapsulation provide the single best opportunity of all current software engineering techniques for improving the maintainability and reliability of programs [4]. The associated cost is an increase in overhead for calling the procedures which encapsulate the objects of the abstract type. An architecture which directly supports data abstraction must account for this overhead and eliminate a large portion of it. The success of an abstraction oriented architecture may depend on its ability to overcome this overhead factor while still providing the facilities of encapsulation.

Vertical Migration

A computer system can easily be seen as a hierarchy of interpreters. From the end-user's perspective, the computer

may seem to be a transaction processing facility which operates on user requests. From a high-level language programmer's perspective, the system may appear to be a programming language interpreter which is based on the particular programming language being used. This hierarchy can be followed down through the levels of language until one eventually can view the computer as a microprogrammed engine which interprets macrocoded instructions. This type of system has been studied extensively by Stankovic in [47,48,49] and Stockenberg and van Dam in [50] and is further illustrated in figure 3. The collection of levels has come to be known as a multi-level software/firmware hierarchical system. Stankovic's research [47] has shown that moving functions from level to level can enhance the performance of the system as a whole. Furthermore, this same research has shown that the interaction between the migration of several objects can often result in less total improvement than the sum of the individual improvements gained from those migrated objects.

This technique, known as vertical migration, provides a generalized n-level hierarchy which can be used for tuning a computing environment. Note that the concept of vertical migration allows both an upward and a downward migration of functions as well as a horizontal migration of functions to the same layer of the interpretive structure. Several efforts including Tucker [54], Wilk [58], Wulf [66], and

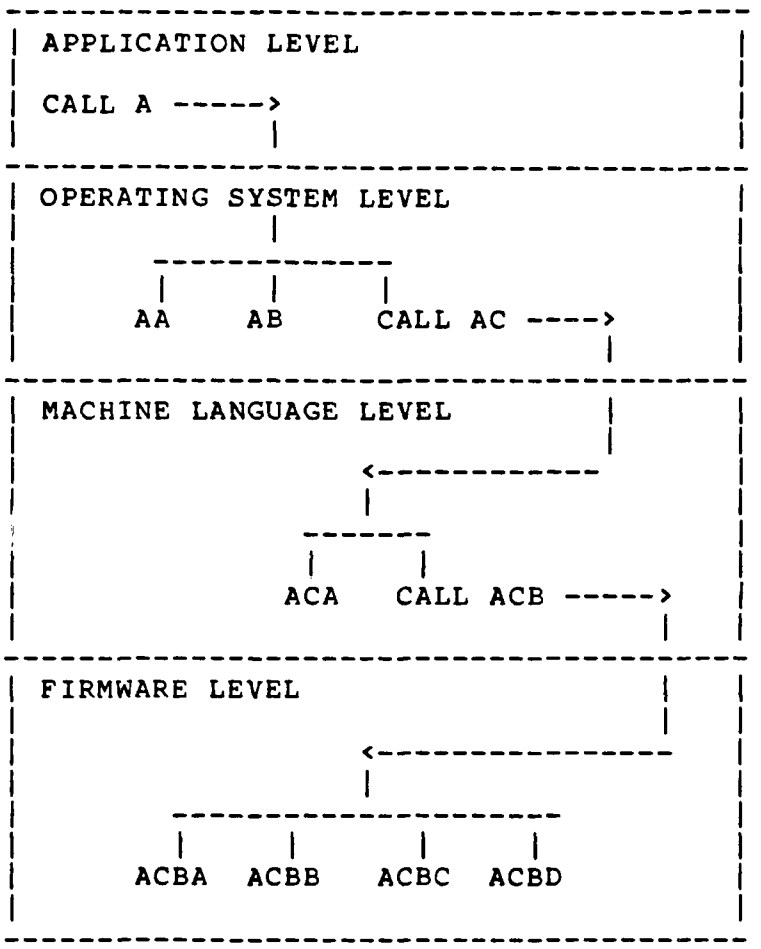


Fig. 3--Multi-level Interpretive Hierarchy

Baker [3] have shown that significant performance enhancement can be achieved through vertical migration by user microprogramming. In this technique the user microcode becomes a part of the architecture in that it is placed in the control store and is available to programmers like any other service of the architecture. Figure 4 illustrates how the machine language function named AC from figure 3 was migrated from the machine language level to the firmware level. Notice that the other machine-level functions which were part of AC were also migrated to preclude mapping back to the machine-level for these functions. Stankovic points out that this change of intra-level mapping to inter-level mapping for functions migrated together is a large source of the performance improvement which can be gained through vertical migration. He further states that the decrease in generality from a function at a higher level is another contributor to the performance improvement.

Included in vertical migration literature is a review of methodologies used for determining what functions to migrate. Rauscher [40] brought to light the gains that could be made through dynamic microprogramming coupled with vertical migration. He showed that contemporary function-based and instruction-based migration schemes could be adapted for dynamic vertical migration decisions by a compiler. With the exception of Rauscher's technique, most methods involve only static decisions and once made apply

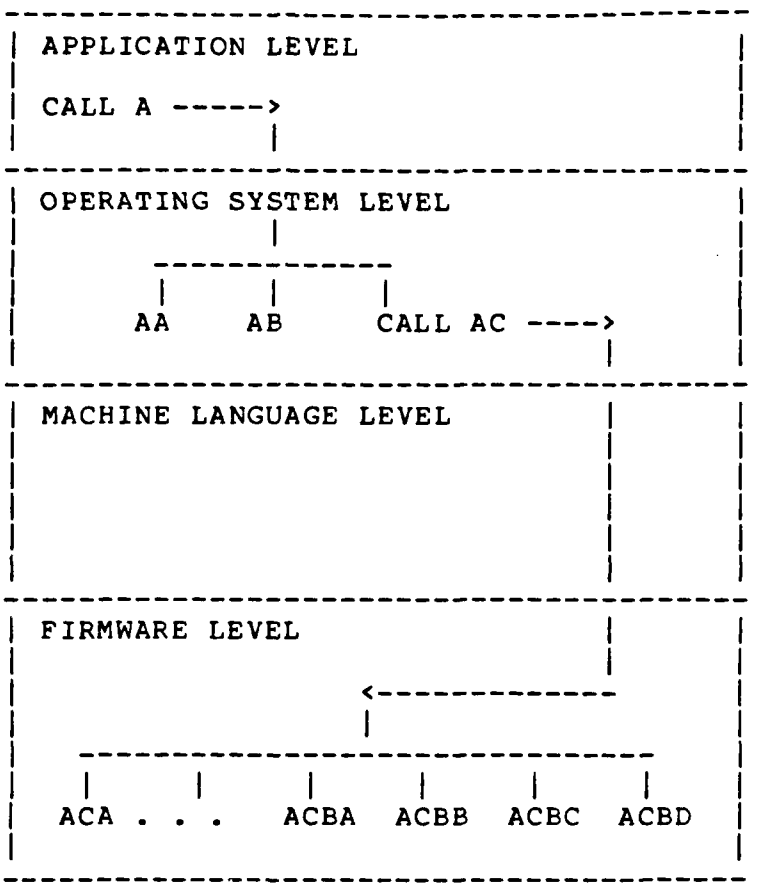


Fig. 4--Vertical Migration of AC

globally to all applications. Furthermore, sequences of instructions are often grouped solely by their physical proximity in the examined code and tend to be unrelated. As a result, similar sequences which differ by only a small degree will be migrated independently and will use more control store. Some of these sequences could avoid using additional control store by calling already migrated sequences and passing a set of parameters for accomplishing the desired goal.

The majority of the literature on vertical migration concentrates on two important migration classes. Holtkamp in [21] reports on these two classes. The first class consists of migrations which are function oriented. In this class the migrated objects are functions much like those in programming languages.

The second class consists of migrations which are instruction oriented. In this class sequences of instructions that occur more than once in a machine or intermediate language program or that are frequently executed are candidates for migration. Holtkamp further states that regardless of which of these these two classes is used for a migration strategy, the following four steps must be performed.

1. Identify suitable migration candidates.
2. Predict the performance improvement.

3. Migrate the objects.

4. Verify the system's behavior.

Research has been carried out in each of these areas. Rauscher in particular has investigated the correct choice of migration objects. Stockenberg and Stankovic have investigated performance prediction in depth. Several efforts have been applied toward automated migration of objects once the correct object has been chosen. Finally, much has been written about verification of program behavior outside of microcode and existing techniques seem readily adaptable to microprogramming as well. As can be seen by this discussion, the study of vertical migration has shown that simply moving functions in the hierarchy can provide performance improvement. The choice of objects to migrate however, is very important.

Motivation

From our background so far, we can see that there are indeed some problems with modern architectures which make them less than desirable for implementing modern systems. Flynn in [15] has shown that instruction sets on today's computers have been designed to optimize hardware realization rather than to aid the programmer in expressing his model in an efficient way. Furthermore, the data types which have been chosen for implementation as well as the operations on these types have been shown to be incomplete.

This incompleteness is due to the fact that the architect cannot determine the class of all possible data types which should be available to provide efficient implementation of all classes of applications. In an attempt to enlarge the class of data types which are implemented in instruction sets, some instruction sets have become overly complex. Due to this complexity such instruction sets are not good target architectures for compilation since the compiler has difficulty mapping to anything other than simple instructions and data types. Wulf in [67] also has noted that in an architecture with a large instruction set only a small subset of the instructions are actually used by high-level language compilers.

This argument leads us to believe that a problem specific architecture is most appropriate for efficient implementation of programs. However, the methods mentioned previously for creating these architectures are function or instruction oriented. This type of organization is inappropriate for modern software engineering practices where separate compilation and library routine usage are encouraged for enhanced maintenance and reliability. A technique of architectural synthesis whereby modern programming practices are encouraged and performance is enhanced is needed.

As a result of modern programming practices such as abstract data typing, a program now models a series of exe-

cution environments. Each environment is characterized by a set of data types and objects of those types which are available for operators to access. This state of being available for access is known as being in scope and can be applied to operations as well as to data objects. In previous techniques for automatic vertical migration, these environments were coalesced into a single global environment and the selection of objects to migrate was made from this single set. When control store became full, the decision on which objects to migrate was based on this single global context. In ATOM each of the local contexts is examined and a unique execution environment can be created for each of these contexts in a program. This results in a closer match of machine architecture to program model. The rest of this report concentrates on the ATOM technique and how it can provide these execution environments.

The result of this study can be a significant step in closing the semantic gap between programmers, languages, and architectures. Individually the four major areas of architecture, dynamic microprogramming, data abstraction, and vertical migration, have not, nor were they intended to, close this gap; they merely address symptoms of it. The combination of these techniques into a single coherent facility can significantly narrow this gap. This gap also is evident in other levels of a layered hierarchical system and

can be narrowed in these areas through proper encapsulation of system types such as processes, messages, etc.

The ATOM approach is not a revolutionary change but an evolutionary one. It seeks to combine successful programming methodologies with proven firmware engineering techniques to provide a model specific runtime environment. The system is flexible in that it allows the programmer to make architectural decisions for the machine on which the program will be run. An intuitive notion leads us to believe that if a programmer took the time and trouble to describe an abstract or user defined type then the objects of that type must be of some importance. The ATOM approach seeks to realize in the machine architecture the model which the programmer wants to simulate. The result will be a problem oriented architecture which is tuned for a specific application rather than a model which has been made to fit an inflexible architecture.

Problem Definition

It is the intent of this research effort to capitalize on the gains which have been made in both software and firmware engineering in order to produce a working facility for abstract type oriented migration. Our research goal then, can be summarized as follows.

> The goal of this research is to investigate the feasibility and applicability of migrating abstract data types into the architecture of the machine through dynamic microprogramming. Migra-

> tion will be automatic and will be oriented towards the implementation of abstract data types in the user architecture. The result will be a program which is supported by an architecture which more closely models the intended problem solution than current architectures. An auxiliary goal is to allow programs to be written using modern software engineering techniques without excessive runtime costs.

This research proposes to decide dynamically which functions should be migrated into firmware to provide the best opportunity for increased performance. The unit of migration will be the abstract data type. The programmer will directly influence the migration decision by describing abstract data types and using the operations provided in the abstract type facility. The migration decision may be affected by such factors as the lexical level at which the type is described, the number of objects created of the type, the number of times the operations of the type are used, etc. The result will be a system where the types of interest to the programmer will be the same as the types supported by the computer architecture. The programming architecture is changed to match the programmer's abstraction rather than the programmer being required to refine further his abstraction or model to gain an efficient implementation.

Environment

A description of the programming environment in which this research is conducted is important in understanding the

scope of ATOM. This study will first assume that all programming will be in high-level, procedure oriented languages. Furthermore, this study will assume that these languages are block structured and provide an identifier scope mechanism. The languages must also provide an abstract data typing facility. In accordance with contemporary programming practices, the languages should also allow access to a large library of standard functions which may include input/output facilities, mathematical functions, string functions, and other application dependent procedures.

The application programming environment will consist of diverse types of programs for many application areas. Programs which are processor intensive will be the primary focus of our study, however, we do not wish to preclude other programs from our discussion. Several classes of programs will be highlighted in subsequent chapters.

The hardware environment is characterized by a general purpose, microprogrammable computer system. This system should be supported by some generally available system software such as a compiler, assembler, and operating system. The microprogramming environment should be supported by at least a micro-assembler and compaction facility for horizontal architectures.

CHAPTER III

AN APPROACH TO ABSTRACT TYPE ORIENTED MIGRATION

Overview

This chapter describes one approach to ATOM. The first section examines previous techniques for problem oriented architecture redefinition and highlights the reasons why ATOM is different from previous approaches. The next section investigates in detail how ATOM works and describes why it is a useful method for performing dynamic redefinition of computer architecture. The next section looks at what language facilities are required to support the ATOM technique for architectural redefinition. The following section discusses the concept of binding as related to computer architecture synthesis and presents a characterization of contemporary architectures in this light. In the next section the performance measurement criteria which are to be used in subsequent discussion of the benefits of ATOM will be developed. These measures will then be used in the following section to derive the expected performance improvement which can be gained using ATOM. The final section will review the need for ATOM and summarize how it can be applied in solving contemporary architectural problems.

Previous Approaches to Architecture Redefinition

The process of architectural redefinition involves determining what objects should become part of a problem oriented architecture and then migrating those objects into the control store of a microprogrammable machine. As we have seen there are both manual and automatic methods for redefining architectures. For the remainder of this paper we will be looking only at automatic methods for architecture synthesis. The three techniques for automatically redefining computer architecture which we will be investigating here are the instruction sequence, function oriented, and hybrid method.

Instruction Sequence Method

The primary goal of the instruction sequence method is to save main memory while increasing performance. This method is based on the assumption that some instructions will naturally follow other instructions in code sequences. For example, branching instructions will often follow instructions which cause operands to be compared. These sequences, although evident in code created by assembly language programmers, are even more evident in code which has been created by compilers since this code has been created by an automated process. This process will repeatedly create the same machine code for similar high-level language constructs. The result of this phenomenon is that

a large number of similar code sequences will be created by compilers for frequently used high-level language constructs such as assignment, condition testing, and iteration.

The instruction sequence method is best described by Rauscher in [40] and is summarized as follows:

1. Analyze a representation of a program as produced by the compiler to find all sequences of machine or intermediate code instructions and the number of times each sequence occurs.
2. For each of these sequences determine the amount of memory which will be saved by microcoding the sequence. Note that this is actually the difference between the amount of memory which would have been used for the machine code representation and the amount of memory required to execute the microprogram from the macrocoded version of the program.
3. For each sequence calculate the total memory savings. This is the product of the memory savings computed in step 2 and the number of occurrences of the string found in step 1.
4. For each sequence calculate the relative space savings. This is the quotient of the potential savings calculated in step 3 and the size of the equivalent microprogram to implement the sequence.
5. Select as new instructions the sequences with the highest relative space savings.

The result of this type of migration is a savings in memory and a performance improvement in execution time. The savings in memory occurs when all of the similar sequences of machine or intermediate language instructions are coalesced into a set of microinstructions which form a microprogram which implements the sequence. The instruction sequence itself is replaced by an instruction which causes the microprogram to be executed. This process of discovering sequences to migrate and including them in the microstore continues until the microstore is full or until all sequences are migrated. The memory savings is then the difference between the size of the migrated sequence and the microcode calling sequence times the number of occurrences of the sequence.

The performance improvement comes as a result of eliminating memory fetches for instructions. When the machine or intermediate language instructions are migrated, the instructions which implemented the sequence are no longer needed and are replaced by a single instruction as mentioned before. The result is that there is no need to fetch and decode these instructions. The microstore functions in this case much like a cache memory in that the fetch is replaced by a much faster fetch from the microstore and a much smaller decode time.

Function Oriented Method

The primary goal of the function oriented approach is to improve performance by migrating into control store those functions which are executed most frequently. This technique effectively creates new instructions which are part of the instruction set of the newly defined architecture. This architecture is formed by a concatenation of the existing instruction set and the instructions created by migrating the most frequently executed code sequences. A method for performing function oriented migration is described by Olbert in [34] and is summarized as follows:

1. Select a benchmark or set of benchmarks to stress the system while examining its performance.
2. Analyze the system under loaded conditions.
3. Identify the areas of high software execution frequency.
4. Project the performance improvement attained by each of the areas of high frequency.
5. Migrate the functions of highest execution frequency.
6. Measure the resulting system performance to determine the amount of improvement.

The result of this technique is an architecture which is optimized for execution of certain functions. These functions are those which were judged to be most frequently executed. The result then is a system which has effectively extended the instruction set by adding new instructions for frequently executed functions. The performance improvement

comes from the decrease in instruction fetch and decoding time as described for the instruction sequence approach. One of the most frequent uses of the function oriented approach is in providing operating system assists.

An interesting variation of the function oriented approach is one in which the benchmark step is eliminated. In this approach, which provides a more dynamic redefinition procedure, an estimation is made by the compiler as to which portions of a program will be executed most frequently. This portion of code, which may be as small as a basic block or as large as an entire procedure or function, is then migrated automatically. The heuristics which are used for the frequency determination have received most attention.

Sammet in [43] points out that some of the earlier versions of FORTRAN included a FREQUENCY statement in which the programmer could specify relative branching frequencies. This information in conjunction with iterative language constructs could provide a clear picture as to which parts of programs were executed most frequently. Another suggestion was for the compiler to prompt the user for branching probabilities and to use these as in the previous approach to determine execution frequency. Perhaps the best known technique was described by Rauscher [40] and involved determining relative branching frequencies from structured programming constructs of modern programming languages.

Hybrid Method

Since the instruction sequence method seeks to minimize the size of programs and the function method seeks to minimize the runtime of programs, it would seem a logical step to combine these techniques into a single facility. Rauscher and Agrawala report on such a facility in [40]. In this technique both the instruction sequence method and a modified function method are used for affecting architectural redefinition. This technique is summarized as follows:

1. Using the function approach calculate the branch probabilities for the flow of a program within its basic blocks and save these in a vector.
2. Using the sequence approach, find all sequences which occur two or more times in a program and save these in a vector.
3. For each of the sequences, calculate a weighted number of appearances by finding the product of the number of occurrences and the associated probability vector found in step 1.
4. For each sequence, calculate the weighted savings by finding the quotient of the product of the weighted number of appearances times the savings of the sequences divided by the number of microinstructions needed to implement the sequence.

5. While there is still available control store, migrate the sequence with the largest remaining quotient found in step 4.
6. Implement the remaining operations in the program by standard macrocoded functions.

The result of this technique is a redefined architecture which is optimized for representation of the particular program which is being compiled. This architecture is tuned to minimize the program size while increasing its performance. To date, this technique has been shown to be the most effective in describing problem oriented architectures.

Abstract Type Oriented Migration

All three techniques mentioned for redefining architecture are good techniques which have found application in many programming environments. Modern programming languages and software engineering practices, however, have made these techniques become less desirable. In the case of the instruction sequence method, migration decisions are made based upon a static analysis of a program. This type of analysis ignores the dynamic behavior of programs and results in migration decisions which are based more on memory savings than on migration of the most frequently executed functions. In modern computer systems we find that memory costs are rapidly falling and the need to provide

memory savings is greatly overshadowed by the need to enhance performance.

The function oriented method was found to be useful in enhancing the performance of programs by migrating to the control store functions which were executed most frequently. This approach does provide performance improvement yet is constrained by the fact that a global decision is made as to which functions to migrate. A function which is used only locally inside of another function and which may provide a significant performance improvement may not be migrated if it does not contribute as much to the global performance improvement as some other function. The hybrid method also is weak in that functions selected for migration on a global basis with respect to memory savings may actually preclude locally significant functions from being selected as migration candidates. These weaknesses of current redefinition methods point to the need for a technique which can account for more localized program behavior. This technique is known as abstract type oriented migration.

Abstract type oriented migration provides the ability to redefine dynamically the computer architecture to match the model that the programmer is implementing on the computer. This redefinition directly supports the environment in which the program is currently executing and can be changed to model a different environment in other parts of the same program. Consequently, this architecture will not

just describe the global programming model but the local programming models as well. The result is that the architecture which supports the model can be redefined locally and can provide for performance improvement which is of most significance in a particular programming environment. This architecture will be formed by a union of a small kernel instruction set providing general support functions and a problem oriented instruction set for supporting the programmer's model. The resulting architectural schema is shown in figure 5.

Other researchers have been able to enhance the performance of systems through vertical migration, however these systems have used function oriented or instruction oriented techniques as the basis for migration decisions. In other words, the decision for vertical migration was made as a result of statically or dynamically examining a sequence of operations and determining which parts of the code should be migrated into microstore to provide the greatest degree of performance improvement. This type of analysis can often be very complex and may preclude analysis of a large number of application programs. In addition, most migration techniques seek to find new global primitives which can be applied across a system as a whole rather than for a particular program or application system.

In order to provide this type of locally determined architecture, the level of granularity of migrated function

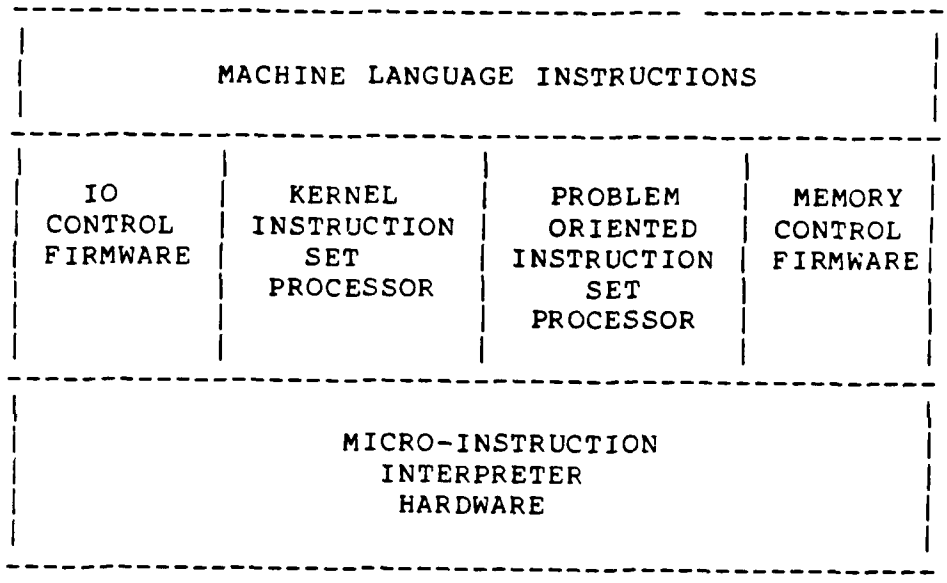


Fig. 5--Problem Oriented Architecture Implementation

which accounts for program behavior and which is well defined by the semantics of the programming language must be determined. As we have seen before, the abstract data type is a good unit which can be used to describe new data types as candidates for migration. These types include an encapsulation of the representation of the type and the operations on objects of the type. In a larger context, the locally determined architecture will be selected from those abstract types which are currently available to the programmer. In the same way as current instruction sets define an architecture by providing a set of data types and operations on these types, abstract type oriented migration will define an architecture by providing the data types and operations of interest to the programmer.

Abstract type oriented migration is a two phased approach to dynamic architectural redefinition. In the first phase there is a static, horizontal migration. This migration involves the programmer encapsulating the representation and operations of a user defined data type into an abstract data type. It is this phase which defines the types which are required to support the programmer's model. In the second phase there is the dynamic, vertical migration. This vertical migration is the step which effectively implements the architecture which is required to support the programming environment described by the programmer. In this two phased approach a problem oriented archi-

tecture can be described and synthesized in a joint effort by the programmer and compiler to create a model specific computer architecture.

For a system to provide the capability to define an architecture which is specifically defined for a particular programming environment we must consider several questions. The first question is how best to describe an abstract data type. An extension to this question is to examine if there are special requirements for describing these types to be used in abstract type oriented migration. The next section addresses these issues. A second question involves how switching from one local architecture to another within a program's execution can occur. This technique which we will refer to as architecture binding can be provided in the abstract type oriented migration scenario and is discussed in a subsequent section. The effect which modern programming languages will have on such an environment and the role which machine architecture will play in this scenario are discussed in chapters IV and V respectively.

Abstract Type Definition in ATOM

The purpose of an abstract data type in ATOM is to form the unit which can be used in automatically synthesizing problem oriented architectures. Since the purpose of ATOM is to provide a facility which can perform automatic architecture redefinition, it becomes obvious that the nota-

tion chosen for describing abstract data types must also be capable of describing data types which are not candidates for migration. Furthermore, the same language should also be used for specifying algorithms to access the objects of the types described. This leads us to the conclusion that the selected notation should be a readily available programming language rather than some special data description language. The programmer should be able to program all parts of the model in the same programming language without regard for what is being migrated by the supporting system software.

Language Elements Required for Abstract Data Typing

Of primary importance is the ability of the selected notation to describe the representation of a data type and the operations on objects of that type and to encapsulate the representation effectively. This encapsulation should include a specification of the entry points to access objects of the type but should not allow access to the representation of the type except through these entry points. This protection is necessary so that the set of invariant properties which defines the type cannot be violated. The encapsulation must be enforced for all routines attempting to access the objects of the type except those which are also part of the encapsulation and which implement the operations of the type.

Another requirement of the abstract type description notation is that it should make it easy for the programmer to derive new abstract data types from other abstract types which have been previously defined. For example, it should be simple to derive a weight data type from a floating point data type once the floating point type is available. Also, it should be simple to constrain the new type with additional restrictions not evident in the parent type. For example, we should be able to enforce the restriction that the objects of the weight data type mentioned before should fall within some reasonable range of the floating point parent type. In describing the derived type, the notation must be capable also of copying the operations of the parent type to the newly derived type. This transfer of operations should be controllable through the syntax of the notation.

Since the notation will be used by programmers, we can assume that there must be some way to translate the notation so that it may be executed. The requirements of protection and derivability place a large burden on the ability of the translator to parse the programs written in the selected notation. One example involves allowing access to the representation of the type by procedures of the encapsulated type and disallowing access to those procedures which are not part of the type. A larger problem involves the ability to implement operator overloading in a much more complex way than is evident in most programming languages today. The

translator must be capable of determining in an unambiguous way which of many possible operators of the same name should be applied in a particular expression. A simple example is whether to use the standard addition or a user defined addition when presented with two numeric operands.

Many different languages provide abstract data type facilities. These languages include: Clu [28], Alphard [44], Simula [33], Ada [23], an extension of C [51], Modula [62], Modula-II [63], and Mesa [17]. Each of these languages provides a facility for hiding the representation of data from the user and providing controlled access to that data. Each language however, adds its own language semantics including rules for name visibility and identifier scope. For these reasons it has been determined that the choice of programming notation is not significant to the ATOM concept, but the requirements mentioned previously for supporting abstract type semantics are critical.

Another consideration of the abstract type description notation is whether special features are required of the language to facilitate translation to microcode for redefining the architecture. A study of high-level microprogramming languages has shown that these languages do include special features not found in common high-level procedure oriented languages. These features however, are concerned with direct access of low-level hardware resources and are not required by the the high-level language programmer.

Since the abstract data type must be programmed in this high-level notation it becomes clear that these special high-level microprogramming language features are not only undesirable but are also inappropriate for use in ATOM.

Another consideration is to determine how programmers may identify abstract data types in the syntax of the language which is being used. One example is a special language construct for describing abstract types as we see in Alphard and Clu. Another method is to compile abstract types separately and to provide a compiler directive statement which identifies the module being compiled as an abstract data type. A final method is to define a special abstract data type description language for use in defining abstract data types and the operations on objects of these types.

Ada As an Abstract Type Definition Language

The Ada programming language was selected as the abstract type description facility of ATOM. The primary reason for selecting Ada as the abstract type description notation is that the language supports abstract data types in a comprehensive way through its package construct. The derivability characteristics described previously are also served well by the derived data type, subtype, and generic data type facilities of Ada. Ada type specification facilities go so far as to allow hardware representation specifi-

cations in the declaration of the type. Another consideration which has made the Ada facility desirable is the degree to which the language has been studied.

Ada has been examined in depth by a number of people from diverse backgrounds and has evolved in response to their comments. The language specification has matured to the point that a language standard has been adopted. Consequently, several implementations of the full Ada language have passed the Ada validation process and are commercially available. Ada also provides a compiler directive facility known as a pragma. It can be used to facilitate syntactic identification of an Ada package as an abstract data type. The description of an intermediate language for Ada known as DIANA [7] also makes Ada desirable for purposes of architectural synthesis.

A simple example of an abstract data type described in Ada follows. This example is used to familiarize the reader with the language facilities of Ada and not as an Ada tutorial. The reader who desires to study Ada in greater detail should see [23]. In this example we are describing an abstract data type for complex numbers. We are interested in ensuring that the type is encapsulated for security as well as being flexible enough to allow description of other similar types.

As we discussed before, the encapsulation of an abstract type requires that the user know the procedure

```
package COMPLEX_NUMBER is

     type COMPLEX is private;

     function "+"(A,B:in COMPLEX)return COMPLEX;

     function "-"(A,B:in COMPLEX)return COMPLEX;

     function "*"(A,B:in COMPLEX)return COMPLEX;

     function SETCOMPLEX(A,B:in FLOAT)return COMPLEX;

private

     type COMPLEX is record
          REAL_PART:FLOAT;
          IMAG_PART:FLOAT:
     end record;

end COMPLEX_NUMBER;
```

Fig. 6--Package Specification for COMPLEX_NUMBER

names for accessing objects of the abstract type and the results of the procedure, but need not know what the procedural code for implementing the prccedure is like. Furthermore, the user need not know how the objects of the type are represented. This interface information is provided to the user through the Ada package specification. Figure 6 shows the package specification for a complex number package, COMPLEX_NUMBER. In the specification it can be seen that there is a data type called COMPLEX and that there are four functions for accessing objects of that type. Furthermore, it can be seen that the representation of the type COMPLEX is a record containing a real part and an imaginary part. Although the names of the constituent parts can be seen, they cannot be accessed from outside the package because the type description of COMPLEX is declared to be private.

Declarations shown in the private part of the package specification are not available for use outside of the package specification and body. The package specification also names the four functions which can be used to access objects of the type COMPLEX and provides the data types of the input parameters and returned results. Note that although the package seems very simple, it still can provide the level of protection which is required for encapsulation. Also notice that the package only encapsulates a single abstract type. This practice is discussed in [5] and is useful in providing

access to only those abstract types that are required by a particular function.

The package body for COMPLEX_NUMBER is described in figure 7. Within the description of the package body is specified the procedural code for the four functions of the abstract type. Notice that three operation names which are possibly defined for other data types have been used. These operators, "+", "-", and "*" may be used either in infix or prefix notation. It is the job of the Ada compiler to determine when these operators are to be used in the place of other similarly named operators. If COMPLEX had been specified as a limited private type in the package specification, functions for equality and inequality tests as well as assignment would have to be defined since these are not automatically provided for limited private types.

Figure 8 shows a sample program which uses the complex number package. Note that the program starts with a with statement. In Ada this is known as a context specification which tells which package or packages are to be accessed in the program. In this case COMPLEX_NUMBER is the only library unit being used. In larger programs other abstract data types as well as other kinds of packages provided in the programming environment could be used. Each of these would be included in a context specification. The use statement is placed in the context specification to make

```
package body COMPLEX_NUMBER is

     function SETCOMPLEX(A,B:in FLOAT)return COMPLEX is
          RESULT:COMPLEX;
     begin
          RESULT.REAL_PART := A;
          RESULT.IMAG_PART := B;
          return RESULT;
     end SETCOMPLEX;

     function "+"(A,B:in COMPLEX)return COMPLEX is
          RESULT:COMPLEX;
     begin
          RESULT.REAL_PART := A.REAL_PART + B.REAL_PART;
          RESULT.IMAG_PART := A.IMAG_PART + B.IMAG_PART;
          return RESULT;
     end "+";

     function "-"(A,B:in COMPLEX)return COMPLEX is
          RESULT:COMPLEX;
     begin
          RESULT.REAL_PART := A.REAL_PART - B.REAL_PART;
          RESULT.IMAG_PART := A.IMAG_PART - B.IMAG_PART;
          return RESULT;
     end "-";

     function "*"(A,B:in COMPLEX)return COMPLEX is
          RESULT:COMPLEX;
     begin
          RESULT.REAL_PART := (A.REAL_PART * B.REAL_PART) -
                              (A.IMAG_PART * B.IMAG_PART);
          RESULT.IMAG_PART := (A.REAL_PART * B.IMAG_PART) +
                              (A.IMAG_PART * B.REAL_PART);
          return RESULT;
     end "*";

end COMPLEX_NUMBER;
```

Fig. 7--Package Body for COMPLEX_NUMBER

```
-- Context specification
with COMPLEX_NUMBER;
use COMPLEX_NUMBER;

-- A program is simply a procedure
procedure MY_PROGRAM is
     X:COMPLEX;
     Y:COMPLEX;
     RESULT:COMPLEX;
begin

     -- Normal call on SETCOMPLEX
     X := SETCOMPLEX(1.234,56.789);

     -- Aggregate assignment
     Y := COMPLEX'(56.789,123.3333);

     -- Prefix call of complex addition
     -- Fully-qualified reference
     RESULT := COMPLEX_NUMBER."+"(X,Y);

     -- Infix call of complex addition
     RESULT := X + Y;

     -- Infix call of complex multiplication
     RESULT := X * Y;

end MY_PROGRAM;
```

Fig. 8--Ada Program Using Package COMPLEX_NUMBER

naming simpler. If the use statement were not included, functions would have to be called by giving their fully qualified names such as COMPLEX_NUMBER.SETCOMPLEX or COMPLEX_NUMBER."+". Even though COMPLEX was described as a private type, aggregate assignment is still allowed as shown in figure 8 immediately following the call of SETCOMPLEX. The aggregate form is allowed because assignment was not prohibited in the package specification.

As can be seen from even this simple example, Ada provides useful facilities for specifying abstract data types. It has been shown that the ability to encapsulate a type is provided through the package construct of Ada. Furthermore, it has been shown how the abstract type can be used in a program through a simple context specification statement. The more complex features of Ada which provide derivability will be discussed in the next chapter in a discussion of programming language effects on ATOM. Although these facilities are provided by some other programming languages, Ada will continue to be used as the model of abstract typing in further discussions of ATOM.

Architecture Synthesis in ATOM

Saltzer in [42] describes binding as the process of choosing a specific lower-level implementation for a particular higher-level semantic construct. In the context of ATOM there are two bindings. The first binding is a high-

level one and is concerned with selecting which components of a program will become part of the problem oriented architecture. The second binding is a low-level one which is concerned with mapping the high-level binding onto the microarchitecture. The combination of these two bindings forms a problem oriented architecture for the current execution environment. Since this execution environment can easily change, ATOM must also be capable of changing the problem oriented architecture to reflect this change. From this requirement comes the dynamic binding property of ATOM. High-level bindings are static and are determined at compile time while low-level bindings are dynamic and are altered at execution time.

Binding Characteristics of Architectures

An understanding of the concept of binding and how it can be applied to architectures is essential in understanding ATOM. In this section the concept of architecture binding and how it can be accomplished will be considered. Three characteristics of binding, instruction set creation, instruction set binding to programs, and length of binding will be examined. This discussion will highlight how ATOM can provide a dynamically created and bound instruction set for each execution environment of a particular program. The following section will then explore in more detail the binding properties of ATOM.

Instruction Set Creation

Contemporary architectures and consequently instruction sets have evolved from the simple instruction sets of early machines. As was discussed in earlier chapters these instruction sets were primitive due to the simplicity of the von Neumann model of storage. Little has changed in these instruction sets except for the addition of more complex data types and operations supporting these data types. Currently, computer architects are responsible for making the determination as to what data types and what operations are supported. This decision is made when the machine is first designed and the microcode for supporting the instruction set is written. When the design stage has been completed, the computer architects have decided what problems will be solved by the computer and what tools are necessary and sufficient for solving these problems.

The tools which the architects provide comprise the instruction set of the machine. Wulf in [67] however, points out that compiler writers do not want computer architects to provide them with the solutions to their problems as embodied in complex instructions but what they really need are good primitive operators from which good solutions can be synthesized. This points to the fact that modern instruction sets are too complex and often are difficult for compiler writers to use efficiently. Winner argues in [59] that if the creation of the instruction set could be delayed

until the actual problem has been described, then an architecture which directly supports the problem solution could be described and consequently provide a better architecture for solving the problem at hand. These two extremes will be referred to as fixed and dynamic instruction sets. Some machines allow instruction sets to be changed more often than in the fixed case yet do not provide facilities for changing as often as the dynamic case. These type instruction sets will be referred to as variable instruction sets.

Instruction Set Binding

The purpose of a compiler is to translate programs written in high-level programming languages into a form which can be executed on a computer. During this translation process several levels of a language hierarchy are used including source language, intermediate language, and machine language. Myers [31] points out that although programming languages are far removed from the machine language, they are affected by the architecture of the machine as represented by the machine language. This can be traced to the translation process where the programming language must be mapped onto this instruction set. This mapping is actually a binding of high-level programming language semantics to the low-level semantics of the architecture.

In both contemporary and problem oriented architectures this binding occurs at compile time and determines the architecture for the problem at hand. An alternative would be for there to exist many instruction sets for a particular problem, each of which represented an architecture for supporting a particular execution environment of the program. This is the ATOM approach. The extremes represented by contemporary architectures and ATOM will be referred to as global and local binding. A binding technique which is supported by a control store cache mechanism can capture some of the sense of both global and local binding. This type of binding will be referred to as paged binding.

Length of Instruction Set Binding

Akin to the issue of global and local binding is the notion of how long the binding is in effect. In contemporary architectures with fixed instruction sets, the binding is for the lifetime of the machine since the architecture provides only one instruction set. Problem oriented architectures provide a binding which lasts for the life of the problem. ATOM on the other hand provides a set of bindings for a particular program, each of the bindings being in effect during one particular execution environment. If program control leaves one environment for another one, a new architecture may be realized for this new environment. Control may once again return to the environment which is being

left and the architecture of that environment may once again be realized. These extremes will be referred to as long and short bindings, respectively. Bindings which may change due to some other notion than program locality will be referred to as medium binding lengths.

Figure 9 summarizes the architectural characteristics of instruction set creation, instruction set binding time, and binding length. From this table it can be seen that ATOM provides a very flexible architecture which can adapt for many differing execution environments within the same program. For programs which exhibit highly localized execution behavior this can provide a performance improvement which may have been overlooked by contemporary problem oriented synthesis techniques. These techniques look only for global solutions and do not address individual execution environments as does ATOM. ATOM can also provide performance improvement for programs which do not exhibit localized behavior by providing an architecture which is closely related to problem oriented architectures which have been synthesized using the hybrid method discussed previously.

Architecture Binding in ATOM

In the architecture classes of figure 9 the last column describes the degree to which an architecture can change to accommodate a specific problem solution. For example, the von Neumann and high-level language computer

Architecture Class	Description
vN	Contemporary von Neumann Architecture
HLLCA	High-level Language Computer Architecture [9]
AISC	Adaptive Instruction Set Computer [59]
LSA	Language Specific Architecture [56]
POA	Problem Oriented Architecture [40]
ATOM	Abstract Type Oriented Migration

a) Architecture Definitions

Architecture Class	Creation Time	Binding Time	Length of Bind	Flexibility
vN	Fixed	Global	Long	Little
HLLCA	Fixed	Global	Long	Little
AISC	Variable	Paged	Medium	Some
LSA	Fixed	Global	Long	Some
POA	Dynamic	Global	Long	More
ATOM	Dynamic	Local	Short	Most

b) Architecture Classification

Fig. 9--Architecture Comparison

architectures are shown to have little flexibility due to their fixed/global instruction sets. The adaptive instruction set is shown to be more flexible because it presents a very large instruction set. The instruction set of the AISC can be changed but is less likely to change than that of POA or ATOM. AISC's paged control store however, with a third level of memory creates another level in the definition of binding much in the same way that virtual memory does.

The language specific architecture in the same way is shown to provide flexibility because it presents several different architectures dependent on the number of languages supported. The problem oriented architecture is shown to be even more flexible in that it changes to reflect the nature of the problem at hand, yet it still is a single, global solution and lacks the adaptability of the ATOM approach. The primary emphasis of ATOM can therefore be seen as an attempt at capturing the notion of program locality to a higher degree than previous architecture schemes. Other architectures such as AISC capture this idea at runtime by using paging schemes while ATOM uses the lexical structure of programs to gain the same information. In this section will first be shown the method which ATOM uses to select items to be included in a problem oriented architecture. The following section will address the dynamic binding issue of ATOM.

High-level Architecture Binding in ATOM

In deriving problem oriented architectures, current methods depend on the approaches for architectural synthesis which have already been discussed. These approaches, instruction sequence, function oriented, and hybrid method derive problem oriented architectures from the intermediate code generated by the compiler as it translates a program. A result of this type of analysis is that the synthesized architecture is a collection of new instructions which effectively extend the instruction set of the conventional machine with new functions which were determined to be the most frequently executed functions in the program. The result is that the new architecture is a global representation of the entire program without regard for the requirements of individual program units. The result is that there is no clear relationship between the items in the new architecture and subsequently no judgement can be made as to which items are of local importance and which are of global importance.

The ATOM method takes a different approach. In ATOM the unit of migration is the abstract data type. This unit is a closely related group of functions which have a scope of reference as defined by the block structure of the language and which have a well defined structure as defined in the abstract type definition. In the previous approachs, no judgement could be made as to the applicability of a par-

ticular migrated function for an execution environment. The result is that control store space occupied by a migrated function which is of no importance in the current execution environment, but which may be pivotal in subsequent environments, is wasted. The replacement of this function for the current environment with one of more local interest could provide an opportunity for performance improvement which had been precluded in previous methods. The key concept of ATOM therefore, is the logical method in which items are selected for migration. The result is a collection of candidates for migration where the relationship between the items are known and the point at which they are applicable in a program is well defined.

The architecture synthesis for ATOM is based on a procedure where the compiler collects information on abstract data types and uses this information to create an architecture which is tuned for a particular execution environment. An execution environment is defined as a collection of one or more program units which is defined by the types of objects which may be accessed in that collection. These program units are defined by the scope of reference of abstract data types. Figure 10 shows the algorithm which is used for architectural synthesis in ATOM.

Several points in the algorithm need clarification at this point. The first block refers to a collection of abstract type and scope information. This is a function

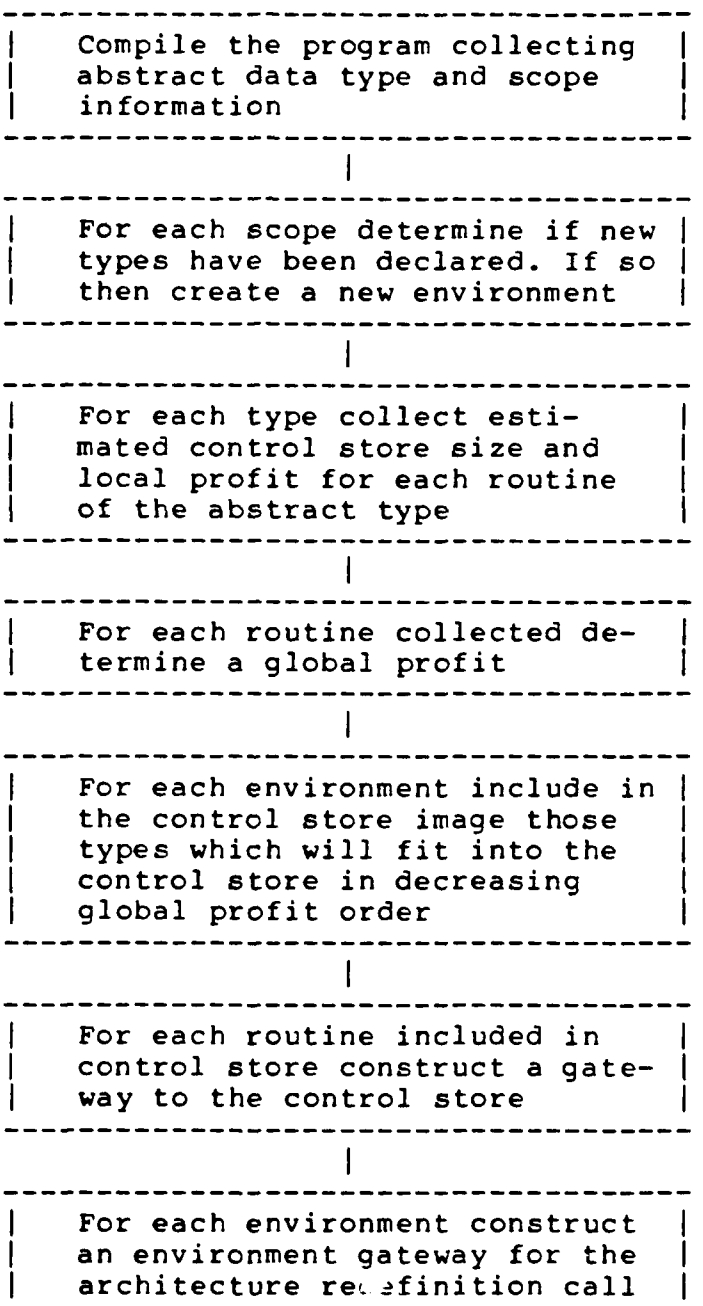


Fig. 10--ATOM Architecture Synthesis Algorithm

which is done by all compilers currently supporting abstract types. The scope information is simply symbol table information which is required for any translation process. The abstract type information need only be an indicator that the code being compiled is an abstract type. This can be supplied by a compiler directive statement if the programming language does not provide unique syntax for identifying abstract data types.

Figure 11 shows a simple Ada program which uses the complex number abstract data type which is implemented in the Ada package COMPLEX_NUMBER of figures 6 and 7. In this program the compiler must determine if there is one or two execution environments. The determination is based on the fact that the package, ANOTHER_PACKAGE, which is contained in the separately compiled unit INNER_SCOPE may contain other abstract data types. Figure 12 demonstrates that there are two scopes in the program. The first scope contains the items declared in the program CALL_MICRO excluding those items in INNER_SCOPE. The second scope contains all of the items in CALL_MICRO which have been declared prior to the specification of INNER_SCOPE and all items in INNER_SCOPE. If this second scope contains new abstract types and includes references to objects of this type, then a new environment is formed. Otherwise only one environment exists.

```
with COMPLEX_NUMBER;
use COMPLEX_NUMBER;

-- Most global program level

procedure CALL_MICRO is
     X:COMPLEX;
     Y:COMPLEX;
     RESULT:COMPLEX;
     -- include a separate compilation unit
     procedure INNER_SCOPE is separate;

begin
     -- Normal call on SETCOMPLEX
     X := SETCOMPLEX(1.234,56.789);

     -- Scope change
     -- But not always an architecture change
     INNER_SCOPE;
     -- Return to original scope
     Y := COMPLEX'(56.789,123.3333);
end CALL_MICRO;

-----------------------------------------------
           Separate Compilation Unit
-----------------------------------------------

separate(CALL_MICRO);
with ANOTHER_PACKAGE;
procedure INNER_SCOPE is
     MY_RESULT:COMPLEX;

begin
               .
               .
               .
     MY_RESULT := COMPLEX_NUMBER.SETCOMPLEX(1.0,2.0);
     MY_RESULT := ANOTHER_PACKAGE.XXX(X);
               .
               .
               .
end INNER_SCOPE;
```

Fig. 11--Inner Scope in CALL_MICRO

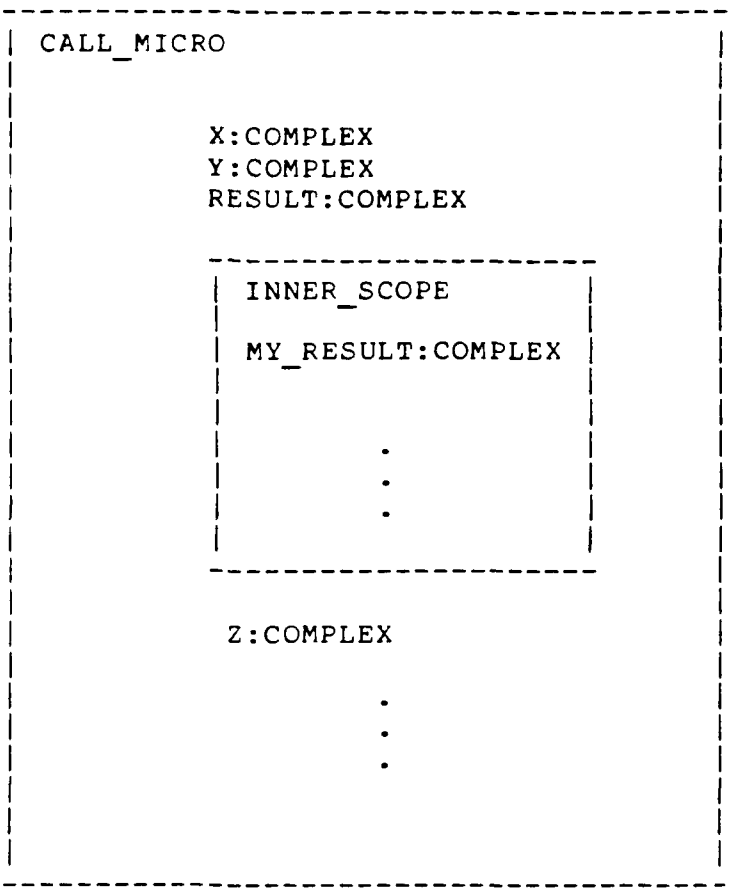


Fig. 12--Scopes in CALL_MICRO

The estimated control store size of the abstract type can be derived by first compiling the macrocode version and then taking as an upper bound the equivalent size of the microcode in the emulator which implements the conventional machine language. The local profit can be derived by examining the potential saving derived by an execution of the routine in control store. The next section examines these potential savings in more detail. This profit is effectively a measure of the local execution behavior of an abstract data type. The calculation of the global behavior can be done in several ways. The first way uses the program structure method of Rauscher [40] to predict the usage of each routine of the abstract type. This method, however, ignores the dependency which can be applied due to derived data types.

A more applicable method is to use heuristics which account for the interdependency of derived data types and for the program structure. For example, data types which are more global may be more profitable in determining global program behavior. On the other hand, localized program behavior may be best determined by migrating the least global type in a scope. These issues will be discussed in more detail in chapter IV. Regardless of the method used, global profit will be a function of the local behavior as described by the local profit and the global behavior as described by the program structure. The implementation of scope gateways

and architecture redefinitions will be discussed in the next section.

Low-level Architecture Binding in ATOM

The facility to redefine the architecture of a computer dynamically to reflect the current execution environment is what makes ATOM unique among problem oriented architectures. The process of changing the architecture is invoked when a program either enters a new execution environment or returns from an execution environment which it entered previously. Within the environment, ATOM must be capable of causing a function to be executed from either control store or from the conventional machine level.

The architecture redefinition call or environment gateway is the routine which is invoked at the entry to any execution environment which effectively causes the supporting architecture to be changed. This call is created by ATOM when it creates an executable program. It is responsible for loading the microcode image which has been created using the algorithm in figure 10. This call is also responsible for setting up global indicators which are used in the subsequent gateway routines to determine if a routine is currently implemented in the control store or is implemented using the facilities provided by the architecture at a higher-level. Subsequent to returning from the new environment, the environment gateway is responsible for reestab-

lishing the calling environment, including control store image and global indicators.

The gateway for each routine is also created in the ATOM architecture synthesis algorithm of figure 10. The responsibility of the gateway routine is to act as a switch. The switch determines where a function is currently implemented and vectors the caller to the appropriate level. This routine is also responsible for returning from the call to the appropriate location in the caller. This is a complex decision since the call may have come from control store or from a higher-level function. Figure 13 shows the interaction involved in utilizing these gateways.

In figure 13 the dependency of each execution environment on the previous environment can be seen. Each environment is responsible for ensuring that the proper architecture is bound when it returns from a call into that environment. The dynamic architecture binding calls are CALL_MICRO_ENVIRONMENT and INNER_SCOPE_ENVIRONMENT. As discussed before, these calls are responsible for realizing the architecture for supporting the current execution environment. The call labeled RESET_ENVIRONMENT is responsible for replacing the architecture which was in existence at the time the current execution environment was entered. The call labeled GATEWAY is the call which is responsible for determining if the routine which is being called is implemented in the problem oriented architecture or is mapped to

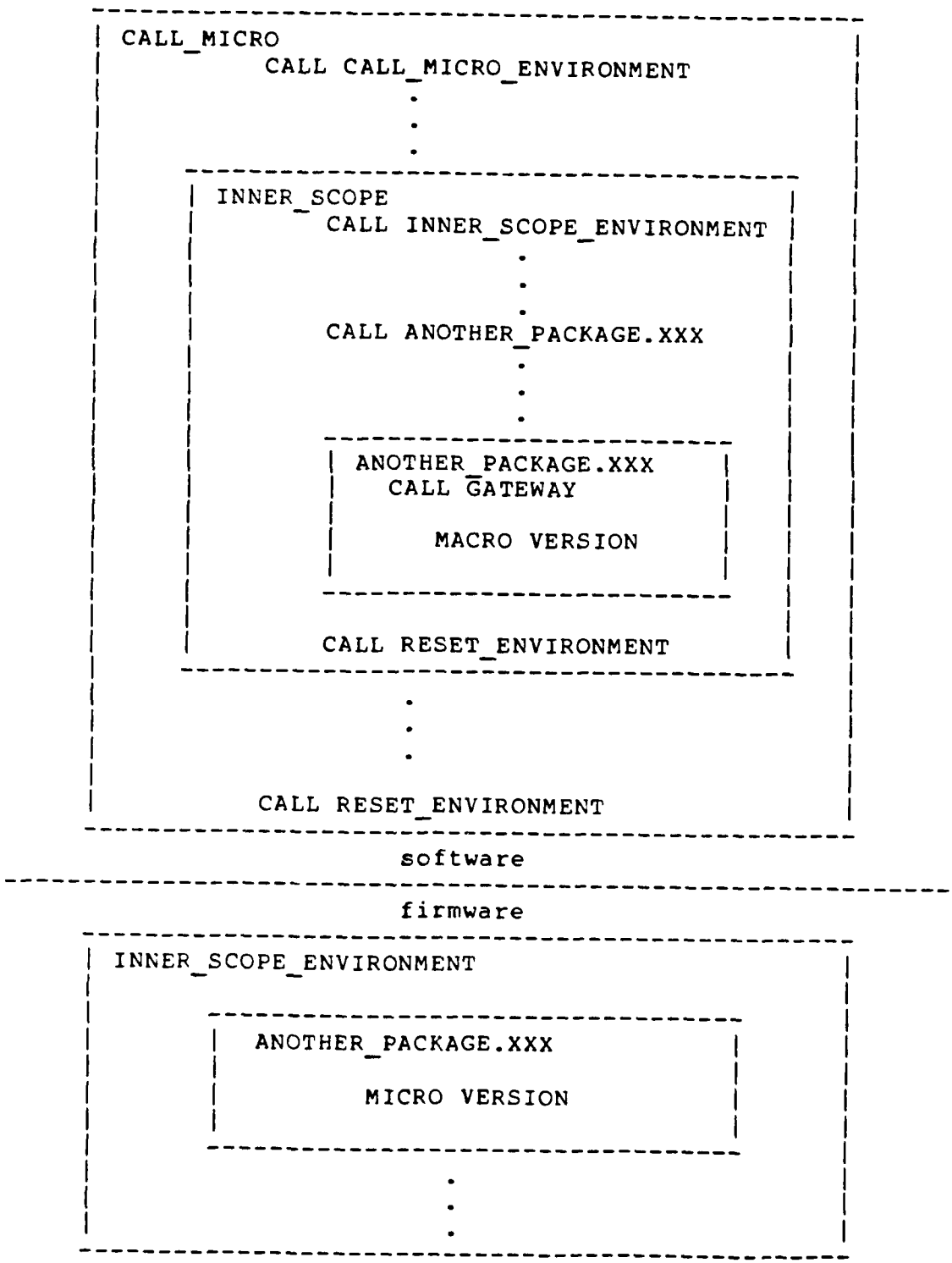


Fig. 13--Low-level Architecture Binding in ATOM

the architecture supported by the kernel instruction set. These versions of the routine are labeled MACRO VERSION and MICRO VERSION, respectively.

It should be noted that for all programs which can be translated to run on a particular machine, a microcode image can be derived to represent every environment in that program. The proof lies in the fact that the total architecture is a combination of both the macrocode emulator and the problem oriented architecture. If the program can be translated so that it can run on the conventional machine, then it can be described in at least one image, that image being the macrocode emulator.

Performance Measures

In determining the performance of a computing system, many variables are relevant. For complex systems these can include queuing time, service times at various components, transmission time of data and many others. With ATOM however, a simpler level than normally studied in performance evaluations is of interest. Since ATOM effectively changes the architecture of a computer, it is the method used in fetching and executing the instructions of application programs and consequently the performance of that method in relation to more conventional techniques with which we are concerned. For this analysis the time that it takes to fetch instructions, decode them, and to execute the sequence

of microinstructions which implements the macroinstructions will be the primary interest. The following equation summarizes the time required to execute a single macroinstruction:

TIME = FETCH + DECODE + MICROINSTRUCTION EXECUTION

As a result the total time for execution of a program will be the sum of the independent instruction times for all of the instructions executed in that program. The effects of cache structures and pipelining are discussed in chapter V.

Fetch Time

Fetch time is the component of execution time which measures how long it takes to make the instructions of a program available for execution in the processing unit of a computer. Total fetch time is a function of the speed at which individual instructions are accessed and the number of instructions which must be fetched. The access time factor is dependent upon the speed of the memory unit from which the instruction is being accessed. For example a memory access from cache memory is faster than an access from main memory since the cache memory is implemented with faster memory devices. Some machines include a memory interface unit which is responsible for fetching instructions at the same time other instructions are being executed. In this analysis we will not be concerned with this overlap.

Decode Time

Decode time is the component of execution time which measures how long it takes to determine what instruction has been fetched and to begin executing the microcode which implements the instruction. The decode operation usually involves inspecting the operation code and performing either a table search or a hashing type function to determine where in the control store the microcode to implement the current instruction is located. The result of the decode operation is that the control store address mechanism is initialized so that instruction emulation can begin. Some machines implement instruction decoding with additional hardware to support this activity while other machines utilize the same hardware which executes the microcode for performing the decoding operations.

Often included in the decode phase is the calculation of the memory address for any main memory operands. Decoding time is dependent on the complexity of the instruction set of a computer. For example some computers implement complex instruction sets where part of the operation code signifies a class of operations and another portion signifies which operation within the class is to be performed. Other architectures provide an operation code which is simply a vector to a location in control store which contains an instruction which branches to the beginning of the emulation routine for the decoded instruction.

Microinstruction Execution Time

Microinstruction execution time is the component of execution time which measures how long it takes to perform the microoperations which implement each macroinstruction. A simple instruction such as a register transfer may take only one microinstruction while a complex instruction such as a memory block move and translation may require hundreds of microinstructions. Some machines implement interruptible instructions which can be interrupted by the hardware of the machine and resume execution at the point the interrupt occurred. We will not be concerned with this type of instruction.

Performance Improvement Expectations

In the previous discussion it was shown that there are three components to instruction execution time and consequently there are three opportunities for performance improvement in ATOM. Some of the performance improvement opportunities which will be discussed are opportunities which are enjoyed by all problem oriented architectures. However, with ATOM more opportunities will be available for utilizing performance improvement techniques.

Fetch Time Decrease

Atom can realize a decrease in fetch time in three ways. The first technique is concerned with the memory access time as described above. When a problem oriented

architecture is defined, a control store image is created and is placed in secondary storage. When the architecture is to be realized the control store image is loaded into the control store and execution is begun. The performance improvement is a result of the difference in time it takes to access the control store and the main memory access time. The microinstructions which are contained in control store exhibit a cache-like behavior in that they are loaded into control store only once and are then available for access from control store rather than from main memory. In problem oriented architectures the control store will remain loaded with this image until the program which loaded it is complete. In ATOM this is not the case since many programs may be using their own, unique, problem oriented architecture. Further control store changes will result from reloading caused by changing execution environments.

The second way that ATOM can decrease fetch time is by eliminating a large number of main memory fetches. When an architecture is defined in ATOM a gateway is created from the routines of abstract data types to the control store. When the gateway decides that a routine is currently implemented in control store it enters the control store and continues execution there. By entering control store, the routine has bypassed the main memory fetches required to execute the instructions from main memory as opposed to execut-

ing the microinstructions from control store. This again can be related to the cache-like behavior of the control store.

The third way of reducing fetch time is by reducing the number of microinstructions which are fetched due to decreased generality of the executed code and a decrease in the amount of mapping actions which must occur. Weidner and Stankovic [55] point out that functions provided in a higher level of a multi-level interpretive hierarchy provide a level of functionality which is more general than that provided at lower levels. This is due to the fact that these functions can accept a variety of different inputs and provide a similar transformation on these inputs. When migrating *abstract data type* routines into control store, the generality can be reduced since the inputs are constrained by the encapsulation of the abstract data type. In the same way the mapping overhead is reduced. This overhead is caused by code being required at each level of the hierarchy to provide for calls between levels. If the entire abstract type is migrated, then the calls will be within the same level and again generality can be reduced. The ATOM approach however, does not preclude calls between levels as is the case in the work described by Weidner and Stankovic.

Decode Time Decrease

The decreases in decode time stem from the same factors discussed for fetch time. The decrease in generality

requires that fewer microinstruction be decoded. The result is that overall decoding time is decreased. In the same way as fetch time, the cache-like behavior provides a performance improvement by trading microinstruction decoding for macroinstruction decoding. When routines are migrated to the control, they are effectively being decoded by the compiler. The only decoding required is that required by the microinstructions which implement the routine. In other words, a set of macroinstruction decoding actions have been traded for a set of microinstruction decoding actions. The microinstruction decoding actions are much faster than the decoding actions for macroinstructions. The decrease in generality also requires that there are fewer microinstructions decoded in executing migrated routines than are required in executing the microinstructions to implement an equivalent set of macroinstructions.

Microinstruction Execution Time Decrease

Again, the decrease in generality experienced in changing levels in a functional hierarchy provides improvement by executing fewer microinstructions. This decrease is largest in architectures which provide an opportunity for compaction of microcode thereby yielding more concurrency in the execution of functions to implement the routines of the abstract data type. This point will be covered in more detail in chapter V. Even in architectures which do not

support concurrency, the ability to decrease the number of microinstructions executed is evident. This can be observed by first determining the macroinstructions required for implementing a particular function. The microcode which will be executed by the instruction set emulator for these macroinstructions can then be compared with the microcode which has been directly generated for the function without regard for the conventional machine architecture. The result will demonstrate the decrease in microinstruction execution which is attributed to the decrease in generality.

For example, an access variable reference in Ada may be compiled as the loading of a register in one machine instruction and the accessing of the item referenced through an index plus offset addressing scheme in the next instruction. The microcode used to implement the register load and index plus offset addressing instruction must be general enough to provide features like condition-code setting, use of any register as source and destination, and possibly even error checking. This is not required in the same type of operand reference in abstract data types and can consequently be deleted from the code generated for a problem oriented architecture.

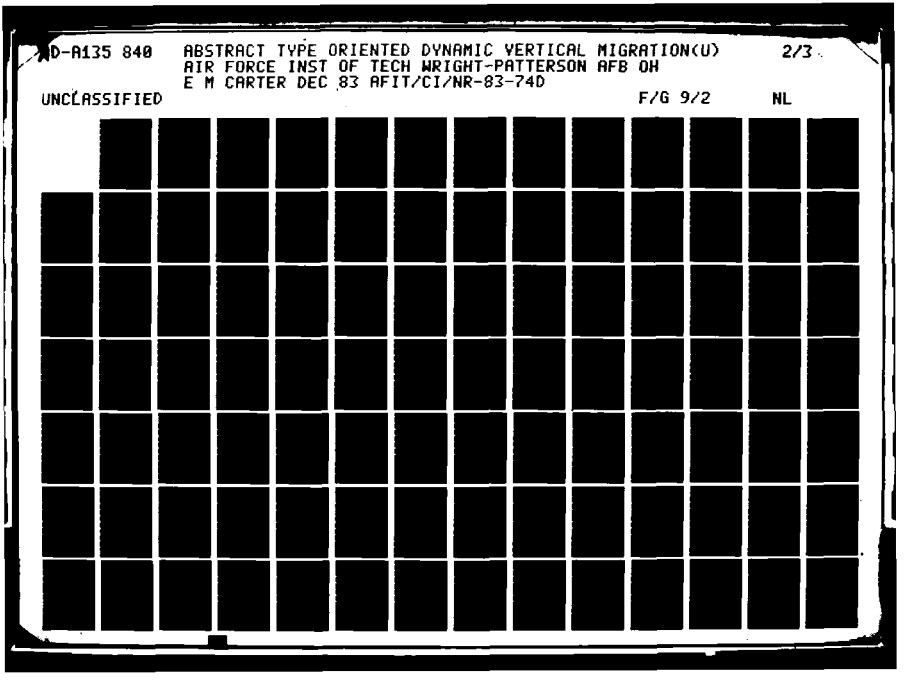
Performance Improvement Estimation

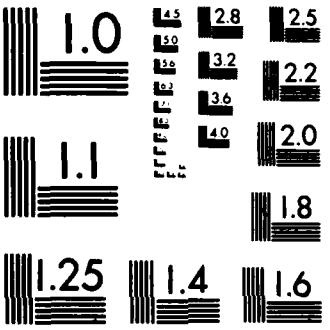
Identifying those routines which provide the greatest opportunity for performance improvement is not a difficult

task. By identifying these routines and determining which abstract data types they implement, one can determine which types to migrate. A performance improvement estimation model has been implemented and is used for determining local profits for architecture synthesis as described in figure 10. The model uses the algorithm of figure 14 for predicting performance improvement for a particular set of data.

The performance of the model is strongly influenced by the choice of the data set used in collecting statistics. The result is a data dependent estimation of how the program will perform with ATOM. This however, presents no difficulty in the technique since the results are used only as a initial estimate of local profit. Figure 15 shows the result of using the algorithm with the complex number program of figure 8.

In the performance estimation model, the decoding time is a function of the machine on which the model is being run. For example, this run of the model was performed on a Perkin-Elmer 3220. This machine implements decoding in special hardware rather than in the more general purpose microinstruction execution hardware. The resulting decoding times are derived from the decoding hardware timing charts. Similarly, the number of microinstructions executed by the emulator is derived from examining the emulator source code and determining the number of microinstructions which


AD-A135 840 ABSTRACT TYPE ORIENTED DYNAMIC VERTICAL MIGRATION(U) 2/3
AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
E M CARTER DEC 83 AFIT/CI/NR-83-74D
UNCLASSIFIED F/G 9/2 NL



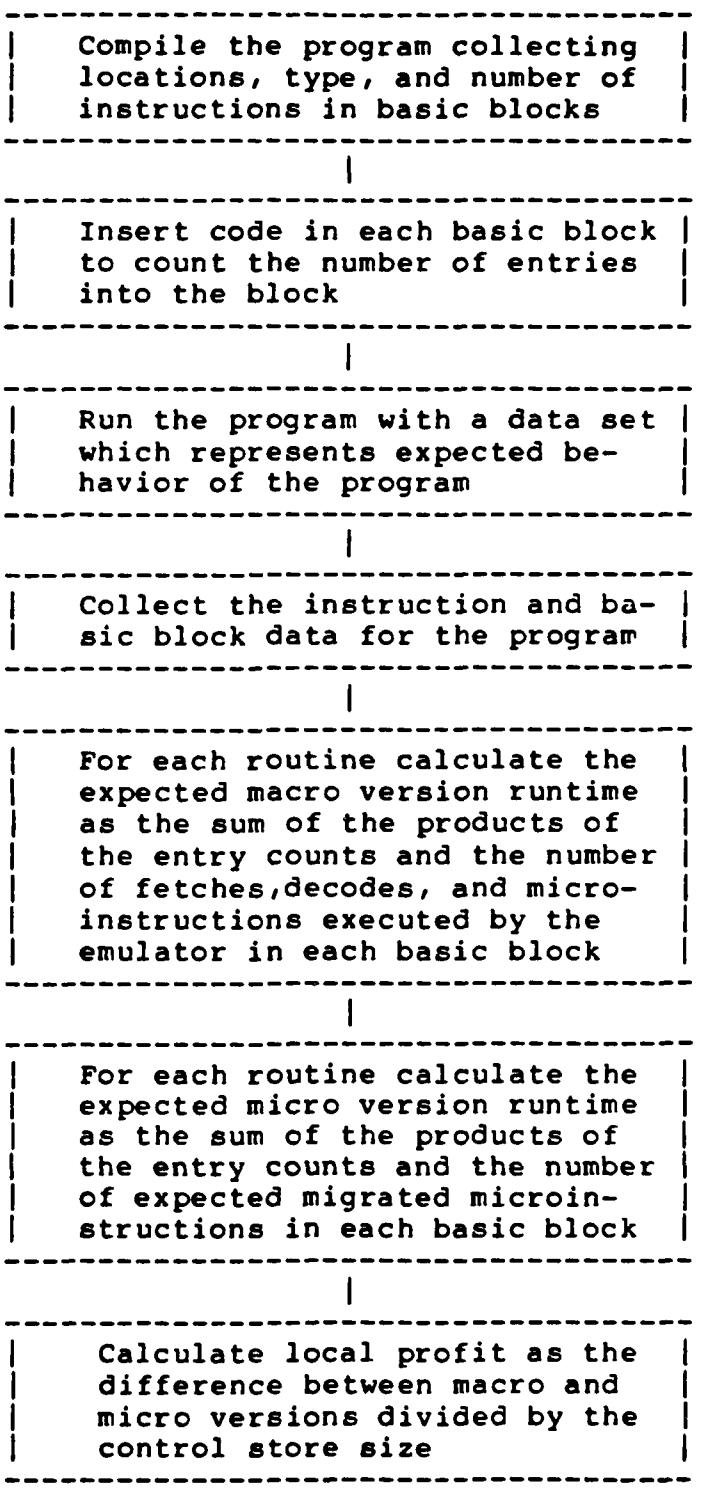


Fig. 14--Performance Improvement Estimation

```
For file: complex.out
There are 16 blocks
For routine: setcomplex
There were 2 calls
     There were       36 instructions executed
     Decoding time is     0.000043 seconds
     Instruction fetch memory accesses is       46
     Executed emulator microinstructions is      248
     Migrated image size is       88 instructions
     Executed migrated microinstructions is      358
     Calculated macro runtime is     0.000093 seconds
     Calculated micro runtime is     0.000073 seconds
     Potential savings is     0.000019 seconds
     Savings per microword used is 0.00000018 secs/word
For routine: plus
There were 2 calls
     There were       40 instructions executed
     Decoding time is     0.000044 seconds
     Instruction fetch memory accesses is       50
     Executed emulator microinstructions is      262
     Migrated image size is      107 instructions
     Executed migrated microinstructions is      372
     Calculated macro runtime is     0.000096 seconds
     Calculated micro runtime is     0.000076 seconds
     Potential savings is     0.000020 seconds
     Savings per microword used is 0.00000017 secs/word
For routine: minus
There were 0 calls
For routine: times
There were 1 calls
     There were       32 instructions executed
     Decoding time is     0.000035 seconds
     Instruction fetch memory accesses is       41
     Executed emulator microinstructions is      175
     Migrated image size is      183 instructions
     Executed migrated microinstructions is      262
     Calculated macro runtime is     0.000070 seconds
     Calculated micro runtime is     0.000054 seconds
     Potential savings is     0.000016 seconds
     Savings per microword used is 0.00000008 secs/word
```

Fig. 15--Performance Estimation Algorithm Results

executed for each macroinstruction. Note also that a 2.25% overhead cost has been added to the execution time of the migrated microcode to account for worst-case control store loading for architecture redefinition. Chapter VI discusses the use of the model in more detail.

Why Abstract Type Oriented Migration ?

In previous sections we have examined the need for a problem oriented architecture to present an architecture which is more closely related to the model which programmers are attempting to represent. The result of such an architecture will be more efficient programs due to the elimination of excess mapping overhead incurred in mapping from the programmer's model to an architecture which supports a set of data types and operations which may not allow an efficient mapping. At the same time we must also realize that reliability and maintainability of programs which can be gained through modern programming practices should not be compromised. These efforts should be encouraged through architectures that support their use and perform more efficiently as a result. ATOM addresses these goals.

The significance of ATOM is that it provides an opportunity for increased reasoning about problem oriented architectures. The emphasis is no longer focused on unrelated program units such as instruction sequences or collections of frequently executed functions. In ATOM the emphasis is

placed on logically related units which form new programming environments by directly supporting abstract data types which form the programming model for a particular problem solution. As a result ATOM directly supports the modern programming practices of abstract data typing, library usage, and separate compilation and reward the programmer through increased program performance for their use.

Library units as well as separately compiled units can be maintained with their equivalent microcode images. These images can be optimized as a result of use by many programmers. The local profit as described in figure 10 can then be tuned through empirical data. The result is that a migration decision can be made with more complete data. Because of its dependence on logical program structure, the issue of interdependence of functions can be handled in a logical way. Reasoning about these dependencies can then be made using the semantics of the programming language and the semantics of the problem being modeled as represented in the program.

Another aspect of ATOM which makes it unique is that migration decisions are made on a local program unit basis. In previous techniques, global behavior was the unit of decision making. When migrating functions, it often is the case that control store will become full and not all desired functions may be migrated. In earlier techniques, those functions which had greater global significance were

migrated before other functions. If control store became full, then the remaining functions were not migrated. This resulted in loss of performance improvement opportunities for programs which exhibit local execution behavior. In ATOM, these execution environments can also be supported in control store and consequently can provide a problem oriented architecture for each execution environment and not just a single statically bound architecture. This technique allows for architecture binding to occur at the latest possible moment. As a result, programs are more closely represented in the architecture on which they are running.

Previous techniques for architecture synthesis often used benchmarking as a technique to determine what functions to migrate. This technique can lead to migration decisions which are data dependent. Also, once these decisions are made, they are reflected in the architecture without regard for changes in data and consequently without regard for changes in program behavior. In ATOM this is not the case. Program structure, not data determines the units to be migrated. As a result, heuristics can be implemented which can account for changes in program behavior and can subsequently cause dynamic architecture rebinding. This can not occur using previous techniques where there was no enforced relationship between execution units.

CHAPTER IV

PROGRAMMING LANGUAGE EFFECTS

Overview

This chapter further examines how modern programming language concepts can affect ATOM. The first concept to be examined will be lexical scope and its effect on control store allocation. The use of scope and visibility rules for determining and limiting execution environments will be described. The following section will examine the unique use of derived data types in ATOM. This discussion will concentrate on the design and implementation of primitive types and how these types may be used for program tuning. Next, the concept of generic types and their implementation in ATOM will be examined. The following section will discuss considerations which should be made in designing abstract data types in the ATOM environment. Finally, a short discussion of the special role of compilers in ATOM will be presented.

Scope Effects in ATOM

The ATOM technique for dynamic vertical migration requires that there be a method for determining parts of programs which exhibit localized program behavior. These program parts must also provide a description of the data

types accessible during their lifetime. In chapter III a unit which could provide this function was introduced. This unit, called an execution environment, was defined as a collection of one or more program elements defined by the types of objects which may be accessed in that unit. The program elements are constructed from programming language constructs which are responsible for defining names of data types, objects, and operations and which apply limits to how these may be used. These limits on usage are also called scopes of reference or more simply scopes. The programming language elements in which ATOM is most interested are those which are capable of describing and implementing the routines of abstract data types. It is the scope of these names which define execution environments.

As discussed in chapter II there are many languages that support abstract data typing and consequently there are many different rules concerning the scope of names of objects and data types. For this reason, the primary effect of programming languages on ATOM will be to limit the ability to determine scopes and therefore limit the ability to differentiate between execution environments. Some languages will provide a great deal of flexibility in defining scopes while others will limit the extent of scopes. This limitation in the worst case can result in there being only one global scope for routine names. More flexible languages which allow nesting of procedure declarations and

embedded program blocks will provide more scopes and consequently can describe execution environments more precisely. The architecture synthesis algorithm of figure 10 can be used without regard for which language is used as long as scopes and consequently execution environments can be determined statically. It is the number of execution environments and therefore the number of generated control store images which will change. Following are some examples in popular programming languages of how scope may affect the creation of execution environments.

The first example is figure 16 which shows a program written in Ada. This program is similar to figure 11 except that it has declared an abstract data type ANOTHER_TYPE in an inner package called ANOTHER_PACKAGE. The flexibility occurs in that the placement of the declaration of ANOTHER_PACKAGE determines whether ANOTHER_TYPE is included in the execution environment of INNER_SCOPE or not. For example, if the declaration of ANOTHER_PACKAGE had been placed prior to the procedure specification of INNER_SCOPE then ANOTHER_TYPE would be included in the execution environment of INNER_SCOPE. If it follows the specification, then ANOTHER_TYPE is not in the execution environment of INNER_SCOPE.

A problem exists in that the compiler cannot determine if ANOTHER_PACKAGE is an abstract type or whether it is just another package which is part of the resources required for

```
with COMPLEX_NUMBER;
use COMPLEX_NUMBER;
procedure ENVIRON1 is
          .
          .
          .
     -- package specification for ANOTHER_PACKAGE
     package ANOTHER_PACKAGE is
          type ANOTHER_TYPE is private;
               .
               .
               .
     private
          type ANOTHER_TYPE is ...
     end ANOTHER_PACKAGE;

     -- package body for ANOTHER_PACKAGE
     package body ANOTHER_PACKAGE ... end ANOTHER_PACKAGE;

     -- include a separate compilation unit
     procedure INNER_SCOPE is separate;
          .
          .
          .

begin
          .
          .
          .
     INNER_SCOPE;
          .
          .
          .
end ENVIRON1;
```

Fig. 16--The New Execution Environment of INNER_SCOPE

proper execution of ENVIRON1. This problem can be solved for individual languages by utilizing the facilities provided by the programming environment which supports the language. In Ada, for example, there is a compiler directive statement called a pragma. This statement could be used to specify that a compilation unit is an abstract type and that it belongs in the program library. Subsequent context specifications which include the new abstract type could then know that it is an abstract data type and could use this information in determining execution environments. This technique assumes the presence of a standard programming environment which accompanies a programming language. Unlike many languages, Ada presupposes a complex programming environment known as the Ada Programming Support Environment which provides a program library facility.

An interesting addition to modern programming languages is the ability to declare concurrent execution units. Both Ada and Modula provide this capability in units called tasks. In Ada, a task is declared in much the same way as a procedure. Within the task specification is a list of entry points for the task and within the task body is a sequence of code to implement these entries. There also may be a sequence of statements executed at task initiation time. A task can be declared as either a single task object or as a member of a task data type. The scope of reference of a task is specified in the same way as that of a pro-

cedure, therefore any two tasks declared together or which are members of the same task type will share the same scope and consequently the same execution environment. These tasks therefore could share the same control store image and could avoid the overhead involved in repeatedly loading the control store in task scheduling. This issue rightfully belongs in a discussion of scheduling techniques in operating systems.

The scope rules of Pascal provide a lesser degree of flexibility for tailoring execution environments. Although Pascal does not provide the facilities for enforcing encapsulation and hence is not adequate for supporting abstraction in ATOM, an examination of the more limited scope of Pascal is useful. Figure 17 shows a program which is modeling the complex number abstract type from the previous example. The program also includes a declaration similar to ANOTHER_TYPE of figure 16.

The point to note is that the limited scope rules of Pascal are a result of the language syntax which requires that all types for a particular program or procedure must be declared together in a single type statement. This prohibits the ability to limit the scope of a procedure and thereby restricts the ability to limit its execution environment. The result is that large execution environments will be produced by the architecture synthesis algorithm and fewer abstract types will be placed into the con-

```
program environ2(input,output);
type complex = record
                   real_part : real;
                   imag_part : real
               end;
     another_type = ...
var
          .
          .
          .
procedure innerscope;
begin
          .
          .
          .
end;
function setcomplex(a,b : real):complex;
begin
          .
          .
          .
end;
function plus(a,b : complex):complex;
begin
          .
          .
          .
end;
function minus(a,b : complex):complex;
begin
          .
          .
          .
end;
function times(a,b : complex):complex;
begin
          .
          .
          .
end;
begin
          .
          .
          .
          inner_scope;
          .
          .
          .
end.
```

Fig. 17--Pascal Implementation of Abstract Type Complex

trol store when contention occurs. This type of scope rule causes ATOM to function in a way similar to other problem oriented architecture synthesis techniques. A possible solution is to modify the algorithm of figure 10 to verify that objects of a particular type are being accessed in a scope before considering that type for inclusion in the execution environment.

Other languages provide even more restrictive scope rules. C, for example, does not allow nested procedure declarations. Fortran in a similar way requires that there is only one global level of scope which applies to function calls. These languages provide a rigid set of scope rules and consequently will not provide as great an opportunity for ATOM to enhance performance. This restriction on execution environments causes ATOM to choose those data types which have the most global influence on performance for migration. Smaller program units which would have been migrated by other techniques such as the instruction sequence method are overlooked and performance, although better than a non-migrated version, is less than that of a version migrated by global techniques. ATOM can therefore be seen to depend on the ability of a programming language to provide flexible scope rules for its ability to ensure performance improvement.

Derived Data Types in the ATOM Environment

A derived data type is a new data type which is formed by assigning the attributes, values, and operations of one type, called the parent type, to the new type, called the derived or child type. In this way a new data type is formed which, although similar to the parent type, is an autonomous type with its own set of invariant properties which must be maintained independent of the parent type. In Ada a derived type inherits and overloads all of the operations, attributes, values, literals, and aggregates of the parent type. It also derives and overloads any visible subprograms that have a parameter or result of the parent type or subtype. In the process of deriving a child type from a parent type, many languages allow certain constraints to be applied to the new type including range, accuracy, index, and discriminant constraints. Ada even allows a representation specification to be applied to the new child type.

The concept of derived types although complex, is very useful in providing type descriptions which match the programmer's model. For example, a new type called WEIGHT may be derived from the type called FLOAT by a single statement:

```
type WEIGHT is new FLOAT;
```

A range constraint can be applied by simply specifying it in the statement which derives the new type as follows:

```
type WEIGHT is new FLOAT range 0.0 .. 500.0;
```

Both of these examples have used Ada syntax for the derivation but the facility is similar in many languages which support derived types.

Of primary interest to ATOM is the manner in which these new data types are implemented. When we look at derived types in detail, what we see is much akin to the situation faced by compiler writers today in mapping from high-level abstract types to the lower-level data types provided in the architecture. In this case however, both the high-level and low-level types are abstract types provided by the programmer and not by the machine architecture. A complex type is formed by mapping the facilities required of the new type onto the facilities already provided by the programmer in the collection of user defined types.

In the case of the simple derivation, there is a one-to-one mapping while in the case of the constrained derivation there is a more complex mapping which may involve additional range checks, index checks, or maybe even a complete new physical representation of the type. We assume that the operations on items of the new type are derived by first performing the new constraint checks, then coercing the object of the new type to the parent type using the default conversion procedure. The operation is then performed on the coerced object using the operation of the parent type. The results are then coerced back to the child type using the default type conversion procedure.

In ATOM this situation raises the question of whether both the parent and child types should be migrated into the same execution environment. Since the goal of ATOM is to enhance performance this decision can only be made in the light of the architecture synthesis algorithm of figure 10. The results of this algorithm are the sole determination of what should and what should not be migrated. The dependency of the child type on the operations of the parent type however, raises an interesting question.

This question involves determining what effect the dependency should have in calculating the profitability of migrating a particular abstract data type. As discussed before, the operations of the child type depend upon the operations of the parent type. The calling of one of the operations of the child type then is tantamount to calling the constraint checking code, performing the coercion, calling the operation of the parent type, and then coercing the results to the child type. A call of a child type operation should therefore involve increasing the global profit of the parent type as well as a calculation of local profit for the code which calls the parent operation. In this way data types which are often used as parent types will be migrated. The ability to migrate the additional operations of the child type in place of or at the same time as the parent operations is provided by applying the profit calculation of

figure 10 and then migrating those functions deemed appropriate for migration by the algorithm.

Generic Data Types in the ATOM Environment

The necessity of requiring strong typing in languages which support abstract data typing has been discussed in chapter III. The concept of strong typing however, often makes programming inconvenient. For example, a programmer may design and implement an excellent sorting algorithm. This algorithm can only be programmed to sort objects of one particular data type because all actual parameters passed to the sorting procedure must match the formal parameters with regard to data type. If the routine has been designed to sort arrays of integers, an attempt to call it to sort an array of floating point numbers will fail. Generic program units can be used to describe a class of algorithms which will differ only in the types of data used. The concept of generic program units has been described to make programming easier in languages which support strong typing.

Generic program units are simply templates which can be used to form other program units at compilation time. In other words, new packages and subprograms can be created from the generic description given in the generic package or subprogram declaration. For example, several different kinds of complex number data types which can use varying data types for the real and imaginary parts may be required.

```
generic
     type C_TYPE is private;

package COMPLEX_NUMBER is

     type COMPLEX is private;

     function "+"(A,B:in COMPLEX)return COMPLEX;

     function "-"(A,B:in COMPLEX)return COMPLEX;

     function "*"(A,B:in COMPLEX)return COMPLEX;

     function SETCOMPLEX(A,B:in C_TYPE)return COMPLEX;

private

     type COMPLEX is record
          REAL_PART:C_TYPE;
          IMAG_PART:C_TYPE:
     end record;

end COMPLEX_NUMBER;
```

Fig. 18--Generic Specification for COMPLEX_NUMBER

Figure 6 describes a COMPLEX_NUMBER package which is constrained to using real and imaginary parts of type FLOAT only. The generic package of figure 18 on the other hand can be used to create different packages which have different types for the real and imaginary parts.

The following Ada syntax, called generic instantiations, cause two new packages and consequently two new abstract types to be created:

```
package COMPLEX_FLOAT is new COMPLEX_NUMBER(FLOAT);
package COMPLEX_DOUBLE is new COMPLEX_NUMBER(DOUBLE);
```

Each of the two new types has a complete set of operations, values, and attributes. Unlike derived types, there is no dependency between the new types and any types from which they were formed. In Ada, as in other languages supporting generic program units, parameters may be included in the generic instantiation which tailor the newly created unit for its particular needs. There are three classes of generic parameters in Ada, type parameters, value and object parameters, and subprogram parameters. Each of these effectively causes a generic template to be transformed into a completely new program unit which is independent from both the template and other program units derived from the template.

In ATOM generic abstract types are handled like any other type which is a candidate for migration. The generic instantiation of an abstract type introduces a new type and therefore, expands the current execution environment and the

execution environment of any procedures declared in the scope of reference of the new type. The effect of generic types in the ATOM environment is therefore minimal and is the same as the introduction of new abstract types.

Abstract Data Type Design in ATOM

Booch in [5] points out that there are four software design needs which data typing meets:

1. Maintainability: the need to describe objects with a factorization of properties.
2. Readability: the need to say something about the properties of objects.
3. Reliability: the need to guarantee that properties of objects are not violated.
4. Reduction of complexity: the need to hide implementation details.

As we have seen before, these characteristics are embodied in abstract data types. Previously, we have seen that programmers paid a performance price in using techniques like abstract data typing because of the overhead involved in procedure calling. ATOM on the other hand seeks to overcome this overhead while still providing the four characteristics above.

In modeling intricate systems, programmers may find it necessary to describe some very complex abstract data types. If these abstract types are selected for migration a very

large amount of control store may be used. Figure 19 shows a large, complex data type comprised of a record containing other user-defined types. Due to the organization of the declaration as a single unit, the entire abstract type must be migrated if any of the operations are to be included in the control store. Since control store is a fixed size resource, the selection of a large abstract type for migration may preclude the migration of other types which could contribute to a significant performance improvement. Wise design practices however, can stop this from occurring.

In current computer architectures, it is the responsibility of the compiler designer to map high-level constructs to lower-level implementations of these constructs. This is required since the architecture only supports a limited number of data types. If control store contention becomes a problem due to the large number of abstract data types, the problem may be solved in much the same way as compiler writers do today. The technique involves factoring larger, more complex data types into smaller abstract data types. Figure 20 shows the individual types of figure 19. Figure 21 shows the complex type of figure 19 implemented using a factored representation.

This technique will provide a larger number of data types in a particular execution environment, but will provide a smaller level of granularity in migration. This

```
package TOO_BIG is
     type NEEDED_TYPE is private;
     type MORE_STUFF is private;
     type BIG_TYPE is private;
          .
          .
          .
private
     type NEEDED_TYPE is ...
     type MORE_STUFF is ...
     type BIG_TYPE is record
                          X : NEEDED_TYPE;
                          Y : MORE_STUFF;
                      end record;
end TOO_BIG;

package body TOO_BIG is
          .
          .
          .
end TOO_BIG;
```

Fig. 19--Complex Type Description

```
package TYPE_MORE_STUFF is
     type MORE_STUFF is private;
          .
          .
          .
private
     type MORE_STUFF is ...
          .
          .
          .
end TYPE_MORE_STUFF;

package TYPE_NEEDED_TYPE is
     type NEEDED_TYPE is private;
          .
          .
          .
private
     type NEEDED_TYPE is ...
          .
          .
          .
end TYPE_NEEDED_TYPE;

package body TYPE_MORE_STUFF is
          .
          .
          .
end TYPE_MORE_STUFF;

package body TYPE_NEEDED_TYPE is
          .
          .
          .
end TYPE_NEEDED_TYPE;
```

Fig. 20--Factored Types

smaller level of granularity will allow other data types to be migrated to control store while still providing some degree of the performance improvement provided by the larger data type. The performance improvement of the program as a whole however, may be greater as a result of the factorization.

The factoring of larger data types into smaller ones amounts to finding the primitive types of a model. This technique can be used for tuning programs which use ATOM by allowing the programmer explicitly to influence migration decisions. The programmer influences the decision by finding the most primitive types and implementing these as abstract types. Since other types will be created using these types the global profit will increase and consequently the probability of being migrated will also increase. Where this technique was not provided in earlier methods for describing problem oriented architectures, it can prove to be very useful in ATOM.

Another interesting effect arises when we examine languages which provide low-level constructs for interfacing with the supporting machine architecture in a direct way. Ada, like other modern system programming languages, provides a representation specification for describing the actual format to be used in mapping data types to the supporting machine architecture. For example, objects of a set data type may be mapped as bit vectors ic may be packed

```
with TYPE_MORE_STUFF;
use TYPE_MORE_STUFF;
with TYPE_NEEDED_TYPE;
use TYPE_NEEDED_TYPE;
package TOO_BIG is
     type BIG_TYPE is private;
          .
          .
          .
private
     type BIG_TYPE is record
                          X : NEEDED_TYPE;
                          Y : MORE_STUFF;
                      end record;
end TOO_BIG;

package body TOO_BIG is
          .
          .
          .
end TOO_BIG;
```

Fig. 21--Complex Type Using Factored Types

into a single memory word. Without ATOM the degree to which the mapping can be specified is bounded by how efficiently the mapping may be applied on the machine architecture. With ATOM however, the efficiency of the mapping may no longer constrain the types of mappings which can be accomplished. In the previous example the size of the set data type would be bounded by the size of a memory word. At the microcode level, which is the target architecture for ATOM, memory word size is an artificial boundary and would no longer constrain the mappings from abstract to primitive type. The point is that through defining new types using representation specifications, new primitive types can be described which are as efficient as the primitive types supported in the machine architecture.

Compilers in the ATOM Environment

Earlier in this chapter it was shown that using high-level programming languages which support abstract typing was necessary to provide the scope flexibility needed by ATOM. These languages by necessity are supported by software systems which include at least a compiler. These compilers are complex programs which are responsible for ensuring that language semantics are enforced and consequently that programmer's models are implemented as described. Software which implements the ATOM technique can be programmed with little effect on current compilers.

Although no additional requirements are levied on compilers, some non-traditional usage is made of the products of compilation. The symbol table which results from compilation is a useful tool for the software which actually implements the architecture synthesis algorithm of figure 10. This information, as a result of normal compilation, contains scope information. This information is used in synthesizing the execution environments and subsequently in building the control store images required by ATOM. Compilers can also be easily changed to insert the environment and routine gateways as required by ATOM. This function however, can also be handled easily by pre-processors or linkage editors [64].

The intermediate code prepared by the compiler can also be a useful tool in creating control store images. This can be used as the representation of the algorithms being implemented by the abstract types and can then be used to create the microcode to implement them for routines which are chosen for migration. The intermediate code is a closer representation of the programmer's actual problem solution than is the macroinstructions created in the final compilation phases. This occurs because the final code is yet another mapping from the intermediate code to the actual facilities of the machine level architecture. In other words the intermediate code has not yet obscured the programmer's problem solution with the architecture provided

by the computer architect. For this reason it can be used in more precisely defining the problem oriented architecture for the particular problem at hand. This intermediate code will also be useful in determining program structuring for use in calculating global profits.

The generated control store images can be placed into a program library for subsequent use. Other programs which include these routines with context specifications will simply be able to access them directly without recompilation. Each of these routines should have stored with it a calculation of local profit as determined by examining the intermediate code. These profit figures may be tuned manually or automatically by heuristics and data gained through empirical use of the routine.

CHAPTER V

MACHINE ARCHITECTURE EFFECTS

Overview

This chapter will examine the effects introduced by the interactions between ATOM and other parts of the computing environment which are common to modern architecture implementations. The first section will be an investigation of the interaction between ATOM and other programs running in a multiprogramming environment. Following this will be a discussion of the effect of ATOM in a multiprocessor configuration. The next section will turn attention to internal processor issues such as cache and pipelining and will investigate the effect of these implementation techniques on the expected performance improvement of an ATOM environment. Microprogramming architecture and its effect on ATOM will be the next topic to be examined. Of primary interest will be a discussion of the effect of vertical and horizontal microarchitectures on the expected performance improvement using ATOM. The final section will look at how the adoption of ATOM as an architectural model will affect the design of future microprogramming architectures. This section will be looking at such issues as protection, handling of interrupt structures, and subroutine calling.

ATOM in a Multiprogramming Environment

The discussion of ATOM to this point has concentrated mainly on the implementation of abstract data types in a single program without regard for the other programs which may be running concurrently with it. In a production environment ATOM will be operating with other programs, all of which will be interested in improving their individual performance to as great a degree as possible. In this environment we will be concerned with two issues. The first issue involves a policy for determining how the control store will be managed when many programs are attempting to load problem oriented architectures. The second issue entails an examination of the effect of image sharing between programs with the same execution environments.

In the ATOM environment the control store plays a major role. The entire architecture redefinition process is based on the ability to load the control store with micro-code which implements the architecture required in the current execution environment. The control store therefore must become like other devices in a multiprogramming system in that it must be sharable and must submit to a centrally controlled scheduling policy. This sharing and consequently the scheduling policy will greatly influence the performance improvement which can be gained by programs in the ATOM environment. During this research the control store was

managed like any other preemptable device and was implemented with the same scheduling policy as the processor.

The control store is loaded when required through a demand loading policy. This loading is accomplished when the process currently running on the processor attempts to enter the control store. Prior to entering the control store the only requirement is that the program notify the operating system of the name of the control store image file. Several techniques were used in managing this image including maintaining the image on secondary store, maintaining the image in primary store as part of the process image, and managing a cache of control store images within the operating system kernel. The work of supporting the control store as a virtual device is documented in [60] and was used as the facility for implementing a modified ATOM as described in chapter VI.

Sharing control store images is a possibility in ATOM if centralized management of the control store is provided. ATOM will provide the best opportunity for performance improvement in an application environment which is characterized by processor intensive activity. Many of these types of environments such as flight simulation, weather prediction, and process control are characterized by a small number of data objects but a great deal of repetitive processing such as data sampling, interpretation of data, and device control.

For example, a process control application may only be concerned with one type of sensing device and one type of device to control, but the majority of its processing will be spent in sampling the sensor, examining the data, and reacting to it. If each of these entities were represented as an abstract data type and if there were several programs running concurrently which could share an image, control store loading overhead could be reduced by sharing the control store image. The compilation of programs could also be simplified if these abstract types were defined in a program library thereby avoiding recompilation. As mentioned before the local profit of these routines could be adjusted through empirical data to predict as closely as possible those routines which provide the greatest benefit for improving performance in migration.

Multiprocessing Environment

A multiprocessing environment can be characterized by two or more processors functioning concurrently with a shared primary memory. Within such a multiprocessing environment there are two interesting effects which are brought about through ATOM. The first one is a result of the decreased usage of main memory as is described in chapter III. The second effect is actually a side-effect of ATOM when used in a multiprocessing environment with individually configurable control stores for each processor.

The first effect is caused by the elimination of the fetch portion of the fetch-execution cycle of conventional machines. As described in chapter III, this decrease in fetch time is a result of two factors, the first being a result of the difference between the memory speeds of primary memory and control store. The second factor however, is the one in which we are now most interested. This factor is brought about by a decrease in the number of memory accesses which are required for fetching instructions from primary memory.

As was described before, the control store functions like a cache in that machine level instructions are no longer fetched from main memory. In the place of these machine level fetches, microcode which implements the machine level instructions, is fetched directly from control store. This latter fetch is often done on a private processor bus while the former is done on the shared memory bus. The result of using ATOM with its decrease in fetch time is a decrease in the usage of the main memory bus and consequently an increase in excess memory bandwidth on the bus. In a configuration with a large number of processors or with a large amount of DMA traffic, this factor can result in a large degree of performance improvement.

The performance estimation model discussed in chapter III examines the performance improvement which can be gained by this decrease in the number of instruction fetches. Fig-

ure 15 shows that for a single call to the complex number multiplication routine over forty memory accesses can be saved. If this routine were included in a matrix multiplication routine and if this matrix routine were included in an iterative algorithm then the performance improvement will be greatly enhanced.

The second effect brought on by ATOM in a multiprocessing environment is also very interesting. This effect is due to the ability of many multiprocessing environments to support individually configurable control stores. In this configuration, each processor in the machine has its own private control store. Each of these control stores can be loaded with a different architecture as created by ATOM.

The problem of processor scheduling is complicated by the presence of ATOM. For maximum utilization of processor potential, the scheduling algorithm must account for control store loading overhead. If a process can be scheduled on a processor which is currently running the same image which it requires then a control store load can be avoided. For further performance improvement, it may also be worthwhile to schedule a class of programs on one processor if these programs all use the same control store image. The result of this type of scheduling algorithm is that the performance improvement can be greater in a multiprocessor environment if control store utilization is taken into account in scheduling decisions.

Processor Implementation Issues

In the discussion of performance improvement with ATOM one of the primary methods used was to decrease the fetch and decode time for each instruction. ATOM and the use of microprogramming however is not the only way to achieve gains in these two areas. Two important advances outside of microprogramming which have added to performance improvement opportunities are cache memories and pipeline processors.

In machines with cache memory, instruction speed-up is achieved by decreasing fetch time. The fetch time is decreased by keeping the most frequently executed instructions in a memory which is faster than primary memory. As each instruction is fetched, the cache is examined. If the instruction is in the cache then the fetch is complete, otherwise the fetch is performed from primary memory. The average fetch time for each instruction is therefore decreased and results in faster instruction execution as a whole. As has been shown, control store acts much like a cache in that it also holds frequently executed instructions. These instructions however, are already decoded and a further performance improvement accrues.

Cache memory depends heavily on being able to capture the correct set of instructions in attempting to model the local execution environment. This notion of locality in cache memories is related solely to the addresses from which

instructions are fetched. If proper locality can be captured the results will yield higher cache hit ratios. Higher cache hit ratios result in greater system performance improvement as a whole. The result is that systems which have cache memory will not witness as great a performance improvement with ATOM as systems without cache.

Figure 22 shows the results of running the performance estimation model with the same data as for figure 15 but with a cache memory and an 85% probability for a cache hit in instruction fetch. In figure 22 the sole difference for each routine is in the decrease in decoding time. This decoding factor includes the time necessary to fetch as well as the time necessary to decode the macroinstructions. Note that it is this decrease which accounts for all of the performance improvement in the cache implementation. Note also that the machine which is being modeled shows more executed microinstructions being executed in the migrated version than in the macrocoded version. This is a result of three factors.

The first factor involves the decoding mechanism of the modeled machine. The machine being modeled is supported by special hardware for decoding. Cache results are therefore optimized since no microinstructions are executed in decoding. The second contributing factor is that the modeled machine is a vertical microarchitecture. As is discussed in the next section, the fact that the microarchitecture is

```
For file: complex.out
There are 16 blocks
For routine: setcomplex
There were 2 calls
     There were       36 instructions executed
     Decoding time is     0.000029 seconds
     Instruction fetch memory accesses is       46
     Executed emulator micro-instructions is     248
     Migrated image size is       88 instructions
     Executed migrated micro-instructions is     358
     Calculated macro runtime is     0.000079 seconds
     Calculated micro runtime is     0.000073 seconds
     Potential savings is     0.000006 seconds
     Savings per microword used is 0.00000005 secs/word
For routine: plus
There were 2 calls
     There were       40 instructions executed
     Decoding time is     0.000029 seconds
     Instruction fetch memory accesses is       50
     Executed emulator micro-instructions is     262
     Migrated image size is     107 instructions
     Executed migrated micro-instructions is     372
     Calculated macro runtime is     0.000081 seconds
     Calculated micro runtime is     0.000076 seconds
     Potential savings is     0.000005 seconds
     Savings per microword used is 0.00000004 secs/word
For routine: minus
There were 0 calls
For routine: times
There were 1 calls
     There were       32 instructions executed
     Decoding time is     0.000023 seconds
     Instruction fetch memory accesses is       41
     Executed emulator micro-instructions is     175
     Migrated image size is     183 instructions
     Executed migrated micro-instructions is     262
     Calculated macro runtime is     0.000058 seconds
     Calculated micro runtime is     0.000054 seconds
     Potential savings is     0.000004 seconds
     Savings per microword used is 0.00000002 secs/word
```

Fig. 22--Performance Estimation - 85% Hit Probability

vertical in nature prohibits compaction. This inhibits the ability to optimize performance of the migrated routines.

The final factor concerns the estimation of executed microinstructions for the migrated routines. Weidner and Stankovic in [55] point out that there are two sources of optimization in vertical migration. The first opportunity involves the speed gained in moving to a faster layer of the interpretive hierarchy. This movement is reflected in the performance estimation model. The second opportunity, omitting excess generality, however, is not included. In the performance estimation model the estimate of executed microinstructions is based on a worst-case analysis which uses the emulator microinstruction count plus some overhead for operand fetching. The result does not account for performance improvement which can be gained by eliminating excess generality as was shown in chapter III. The resulting performance improvement as shown by the performance estimation model is therefore a lower-bound on the improvement which can be expected.

Again as shown in chapter III, another performance improvement opportunity comes from a decrease in the amount of time required to decode instructions once they are fetched by the processor. In ATOM it was shown that a decrease in the decoding time can be gained by not having to decode as many instructions. This occurs since the macrocode which implements the migrated routines no longer needs

to be decoded. Also the number of microinstructions actually decoded and executed is less in ATOM than for the same macrocoded version.

In order to improve system performance, the ability to overlap decoding with instruction fetch and execution has been realized by pipeline processors. In these processors several stages of instruction fetching, decoding, operand fetching, and instruction execution may be going on simultaneously for various instructions. The result is that although there is not a decrease for the amount of time to execute a single instruction, the cumulative effect of overlapping instruction fetch and decode with instruction execution yields greater throughput for the processor in instruction execution. Again the result is that in systems with pipelined processors, performance improvement will not be as great with ATOM as in systems without pipeline implementation. It is not clear however, that decode and fetch time improvements are the dominating factors for performance improvement in systems which provide a horizontal microarchitecture. One can imagine situations in which compaction would be the most important factor in performance improvement in vertical migration. This type system is discussed in the next section.

Microprogramming Architecture Effects

In previous discussions of computer architecture the emphasis has been placed on the view which a conventional machine language programmer or compiler code generator will have of the computer. This discussion will focus on a more elementary level known as the microprogramming architecture or microarchitecture for short. It is this level which is seen by the instruction set emulator designer and programmer. Furthermore, it is this level which is the architecture upon which ATOM will base its implementation of problem oriented architectures for supporting execution environments.

Agrawala and Rauscher in [2] state that in comparing the microarchitecture of various machines one must consider both the functional components of a machine including the data paths and interactions and the design of microinstructions. Of these, the former describes the basic facilities which are available for the microprogrammer. The latter however, describes the way in which the microarchitecture will function and as such provides the information which is needed to examine the performance of the machine. For this reason microinstruction design is of interest in ATOM.

In previous discussion of the performance aspects of ATOM, it was shown that there are three components to the total execution time of a program, fetch time, decode time, and microinstruction execution time. It has been further

shown that there are significant effects on fetch and decoding time which are brought about in cache and pipeline systems. The third aspect of performance, microinstruction execution time, is of most interest when we consider microinstruction design. A microinstruction is a collection of one or more microoperations which can be performed at the same time. The microarchitecture determines how many microoperations can be packed into a single microinstruction.

A classification of microarchitectures usually divides machines into two types. Machines which allow only one microoperation in each microinstruction are generally called totally vertical microarchitectures. In contrast machines which allow several microoperations in each microinstruction are referred to as horizontal microarchitectures. Like many classifications the line between vertical and horizontal architectures is not well defined. The result is that there are many microarchitectures which fall between the two extremes. The distinction, however is sufficient for discussing the effect on ATOM of microarchitecture.

Vertical microarchitectures have microinstructions which resemble classical machine language instructions. They often contain one operation and one operand. Horizontal microarchitectures on the other hand have several operations and possibly many operands defined, either implicitly or explicitly. Vertical machines are typically simple to pro-

gram while horizontal machines are usually difficult to program efficiently. This difficulty is attributed to the ability of horizontal microarchitectures to control several resources concurrently. It is from this ability to control multiple resources that the potential of horizontal microarchitectures for improving performance arises.

Since vertical microarchitectures typically control only one resource at a time, there is little beyond contemporary compiler optimization techniques which can be done to increase the efficiency of vertical microprograms. Horizontal microarchitectures on the other hand offer a unique opportunity for improving performance. Given a set of *microoperations* which must be performed, one can build a minimal collection of microinstructions which contains all of the microoperations. The process in which these microoperations can be packed together so that all of the microoperations are performed without interfering with one another is called compaction.

In the process of compaction, the types of interference which can occur are data dependency, resource dependency, and instruction format dependency [53]. Data dependency deals with the proper placement of microoperations such that no operation occurs before the intended operands are in the correct state. Resource dependency is concerned with scheduling the functional components of the microarchitecture as well as the connecting data paths to ensure that

no two microoperations are attempting to use them at the same time. Instruction dependency involves making sure that two microoperations may be placed in the same microinstruction without interfering with the encoding of either operation. Violation of any of these dependencies can result in incorrect results from a sequence of microinstructions.

Conventional machine languages are implemented in emulators which are created using the microarchitecture of the machine. These machine instruction sets form a general purpose architecture which can be used in mapping operations from high-level programming languages to the low-level microarchitecture. In implementing these instruction sets microprogrammers take great care in compacting the microprograms which implement each individual machine instruction. However, since the machine language programmer can place the machine instructions in any order and consequently the microprograms may be executed in any order the emulator cannot be compacted across instructions.

The equivalent to compacting across instructions can be gained in ATOM. When each architecture for an execution environment is created the result is a series of microoperations which must be performed. These microoperations can then be compacted to form the microprogram to implement the architecture for the execution environment. The compaction can occur across the entire problem oriented architecture description since the complete ordering of microinstructions

is known. Currently, there are two classes of techniques which have been used for compacting microprograms.

The first technique is called local compaction and involves compacting only within individual basic blocks of a microprogram. This technique as documented in [27] can provide a performance improvement in a microprogram by a factor of 2 to 3. The second technique involves not only individual basic blocks but also the flow of control between these blocks. This technique is called global compaction. Recent advances by Fisher [13] have shown that through careful examination of execution paths of microprograms near optimum compaction can be determined. This technique is called trace scheduling. Nicolau and Fisher further state in [32] that if certain types of dependencies such as indirect memory references and conditional branches can be eliminated from programs, performance improvement can be enhanced even more. Studies of this technique have demonstrated performance improvements by a factor of 3 to 1000 in scientific and numerical applications.

The result of this analysis is that in horizontal microarchitectures ATOM can provide a significant speed-up in microinstruction execution time. Vertical microarchitectures on the other hand are limited to speed-ups which can be derived using common compiler optimization techniques. The technique of compiling from a high-level programming language to vertical microcode has been demonstrated by

Ellison in [12]. This same paper presents an algorithm which further converts the vertical microcode to equivalent horizontal microoperations and compacts them to a near optimal microprogram. More evidence as to the feasibility of using high-level language to microcode compilers in cooperation with microcode compaction is documented by Fisher in [14].

Microprogramming Architecture Requirements

The contemporary usage of control store has been as a static device which contains programs which are deemed to be well tested and have proven to be reliable. This stems from the fact that control stores have typically been used for instruction set emulators, language or application accelerators, and operating systems assists. The advent of problem oriented architectures however, has raised the level of use of control store from a relatively static to a dynamic device. The implementation of ATOM will further elevate control stores from their unique position into being just another, and perhaps more visible level in the programming architecture. In support of this environment the microarchitecture will be required to be more flexible and to provide more protection than is common in microarchitectures today. This section looks at several areas which will affect microarchitectures which implement ATOM.

Maekawa in [29] has examined some of the requirements which microarchitecture must meet in order to support a wide spread exploitation of user microprogramming. These requirements form a nucleus for implementation of the ATOM facilities. Within these requirements are listed control store protection domains, multiple execution modes and privileged microinstructions, and microcode trap and supervisor calls. These facilities on the machine architecture level provide the means of implementing common operating system functions on current computing systems. Facilities like these will allow implementation of new user support features in the microarchitecture of machines implementing ATOM.

The implementation of protection domains through the use of privileged instructions is an interesting aspect of the requirements of new microarchitectures. In vertical microarchitectures this task will be relatively simple, however in horizontal microarchitectures the degree of parallelism and the low-level of microinstruction encoding will make it difficult to verify protection facilities. In contemporary architectures, protection facilities are implemented in microcode since it is usually a protected resource. In ATOM the microarchitecture will simply become another programming domain and consequently protection facilities will need to be relocated. Multi-levels of control store may become more frequently implemented to support

new protection facilities. New microarchitecture protection domains should be capable of implementing both separation of users as well as controlled sharing of control store.

As discussed earlier in this chapter, there exists a need to make control stores appear less cumbersome to use. In this way users should not have to be concerned about other users of the control store nor any of its physical attributes such as size, timing considerations, or scheduling. Some method must be found which allows the ATOM user to ignore the possibility that part of the currently running program may be migrated to control store. Making the control store appear as a virtual device will relieve the programmer from any hardware considerations relative to control store. This requires a cooperation of hardware and software. In hardware the protection facilities mentioned before must be implemented while in software operating system support must be provided in scheduling as well as in using the control store.

Changing the contents of control store in conventional architectures is an action which does not occur frequently. Therefore, little has been done to examine the overhead involved and to implement hardware to improve the performance of control store loading. In ATOM the practice of processor context switching will possibly require changing the contents of the control store. The result will be a large increase in the frequency of control store loading and

unloading. Consequently, methods for increasing the performance of control store context switching are required.

One possibility is to implement control stores with paging capability. In this architecture some low-level support software will be required for managing control store page faults. The result however, is that only control store pages and not the full control store will be changed during a context switch. Foundation work in this area has been accomplished in implementation of the B1800 [35] and analysis of the adaptive instruction set computer [59]. The performance improvement would be similar to the improvement gained in machine architectures when paging is used rather than swapping. A simpler method would be to implement multiple control stores with context switching being affected by simply changing the designation of the current control store. This technique would allow preloading of the control store before the actual context switch was required.

Perhaps the most outstanding requirement under ATOM will be the need for larger user programmable control stores. If users are to migrate entire abstract data types, it will become important to provide enough control store space to support them. The paging solutions above for implementing virtual control stores can also be applied to this problem. Larger physical control stores also are more feasible with the marked decrease in memory cost and large increase in speeds and densities of semiconductor memories.

The implementation of ATOM as described will reduce the large complex instruction sets of current architectures to smaller, simpler instruction sets. These instruction sets, which we have referred to as kernel instruction sets, will provide a high-level architecture for use in implementing functions which cannot be migrated. The facilities provided will therefore be similar to contemporary instruction sets but will be simpler with fewer data types and operations. The result of this action is that that more control store space will be available for user microprogramming without actually enlarging the control store. A side-effect of this organization is that the operations of the kernel instruction set will be very simple. They will implement subroutines which will be used in writing programs which cannot fit into control store. As a result the kernel instruction set will actually serve as a cache of service routines which can be called from user programmable control store as well as from the machine architecture. These routines can then be used as utility routines from the microarchitecture as well as an instruction set emulator from the machine level architecture.

CHAPTER VI

IMPLEMENTING ATOM

Overview

In this chapter a partial implementation of an ATOM environment is examined. The first section looks at the difference between the proposed ATOM and the implemented environment. The topics which are examined include programming language, microarchitecture, and procedural differences. The following section highlights the language processors involved in the implementation with a short discussion of how they are *implemented*. The next section looks at the performance estimation model and examines its use in the implemented ATOM environment. In the last section a case study of ATOM migration is presented. Shown in this section are the effects of various computing environments on the migration of a matrix abstract data type.

Proposed and Implemented ATOM

As stated above, the implementation of ATOM which is discussed in this section is a sample implementation which has been developed to demonstrate the feasibility of the ATOM approach. The major differences between the implemented and the preceding proposal are simulation of abstract data types in the programming language, a vertical microar-

chitecture, and manual involvement in the ATOM architecture synthesis algorithm. The following sections address these differences.

Programming Language

In the description of the ATOM environment it was shown that a comprehensive abstract data type facility is required. This implementation of ATOM was accomplished on a Perkin-Elmer 3220(PE-3220) using UNIX.[1] At the time of this implementation, there was no compiler available which could directly support abstract data typing and the class preprocessor for C [51] was also unavailable. As a result the programming language C [26] was used to simulate many of the language features described in chapter III. Figures 23 and 24 show the equivalent C programs for the Ada COMPLEX_NUMBER package body and program MY_PROGRAM of figures 7 and 8, respectively.

The lack of a comprehensive scope facility in the C language prohibited implementation of an automated architecture synthesis algorithm as described in figure 10. The algorithm was modified to allow manual intervention and is shown in figure 25. The flexibility of the language and the availability of the data used in the compiler's symbol table however, made up for this deficiency. Consequently,

---

[1]UNIX is a trademark of Bell Laboratories.

```
struct cmplx {
     float real_part;
     float imag_part;
          };
typedef struct cmplx *complex;
complex setcomplex(b,c)
float b;
float c;
{
     complex result;

     result->real_part = b;
     result->imag_part = c;
     return(result);
}
complex plus(a,b)
complex a;
complex b;
{
     struct cmplx result;
     result.real_part = a->real_part + b->real_part;
     result.imag_part = a->imag_part + b->imag_part;
     return(&result);
}
complex minus(a,b)
complex a;
complex b;
{
     struct cmplx result;
     result.real_part = a->real_part - b->real_part;
     result.imag_part = a->imag_part - b->imag_part;
     return(&result);
}
complex times(a,b)
complex a;
complex b;
{
     struct cmplx result;
     result.real_part = (a->real_part * b->real_part) -
                  (a->imag_part * b->imag_part);
     result.imag_part = (a->real_part * b->imag_part) +
                  (a->imag_part * b->real_part);
     return(&result);
}
```

Fig. 23--C Implementation of package COMPLEX_NUMBER

```
#include "complex.c"
main()
{
     complex x;
     complex y;
     complex result;

     /* Call environment entry */
     set_mu("complex.image",1);

     /* Make calls as before */
     x = setcomplex(1.234,56.789);
     y = setcomplex(56.789,123.3333);
     result = plus(x,y);
     result = plus(x,y);
     result = times(x,y);
}
```

Fig. 24--C Implementation of program MY_PROGRAM

abstract data types were simulated rather than being directly supported by the language facility. The consequences of this action include simulation of derived and generic data types as well.

Microarchitecture

The microarchitecture provided on the PE-3220 supports a control store made up of 4k 32-bit words each containing a single microinstruction. The first 2k words are dedicated to the instruction set emulator as well as the microcode support of channels, console, and error recovery. The remaining 2k words form a writable control store which can be loaded with user microprograms. The microarchitecture supports vertical microinstructions which are generally in a one-to-one correspondence with the microoperations. Memory accessing can be accomplished in parallel with arithmetic operations. The processor however, is placed in a wait state if an attempt is made to access the memory data register before the memory access is complete. The result of the vertical nature of the architecture is that little compaction can be achieved and thus little performance improvement can be gained by reducing the number of microinstructions executed.

To support the ATOM environment, it is necessary to implement the control store as a virtual device which can be shared among many users. Winner and Reed in [60] describe the technique which was used to provide this capability

under UNIX. The technique involves treating a control store image as an extension to the currently running process and managing the image as part of the normal machine level executable image. The control store is demand loaded upon the initial entry of any scheduling quantum if a fault occurs.

The interrupt facility of the microarchitecture provides the capability to test for the presence of an interrupt. Upon detection of the interrupt, the program is responsible for saving the current state of both the microarchitecture and the macroarchitecture and to relinquish the processor and control store to other processes. In the current ATOM implementation, interrupts are checked only when a microcode branching instruction is being executed. If the branch is to an address less than the current control store address then the interrupt status is checked, otherwise processing continues regardless of the interrupt status.

Memory protection is provided through the memory access controller (MAC) of the machine level architecture. In the microarchitecture memory accesses can be either privileged or protected. In the privileged mode addresses are considered to be physical addresses and are not checked for addressing error until the physical memory access is attempted. In the protected mode memory mapping is performed by the MAC. This mapping includes bound checking as well as virtual to physical address mapping. In ATOM all memory

accesses from microprograms are performed in protected mode. In this way memory protection is not compromised in the ATOM environment.

Implemented ATOM Synthesis Algorithm

The major difference in the proposed and implemented ATOM environments involves the architecture synthesis algorithm. The implemented algorithm is shown in figure 25 and is contrasted with the ideal algorithm of figure 10. As can be seen the new algorithm includes some manual intervention in the process of synthesizing the problem oriented architecture for each execution environment. The intervention involves reducing performance estimation data and determining for each execution environment which data types are most appropriate for migration. From this determination the control store images can be built using the created microcode source programs.

The programmer is responsible for inserting calls to the execution environment gateway, set_mu. The parameters required are control store image file name and type number. The control store image file is created by the programmer using the source form of the microprograms as generated by the microprogramming version of the C compiler. The programmer is responsible for collecting the correct source statements for the abstract types to be migrated for an execution environment into a single source file. This file is

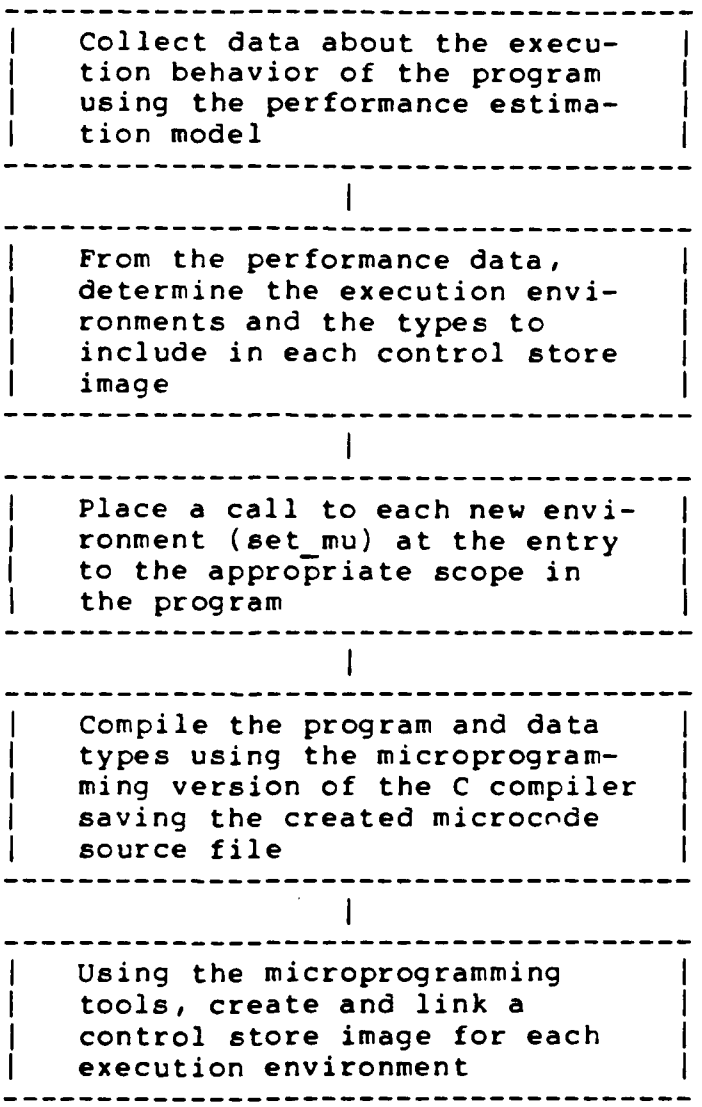


Fig. 25--Implemented ATOM Synthesis Procedure

then compiled and linked by the microprogramming tools to produce a loadable control store image. The type number used in the call to set_mu is produced by the logical OR of all type numbers included in the control store image file. These type numbers are assigned and reported by the microprogramming version of the C compiler. When exiting from an environment, another call to set_mu must also be used to cause the previous architecture to be reloaded for return from the current environment.

Language Processors

As discussed in the previous section, the implementation of ATOM requires several language processors. The purpose of these translators is to create microcode source files and to compile and link them so that they form control store images for representing problem oriented architectures for each execution environment. The programs contained in the language processing environment come from four sources. The C compiler (cc), assembler (as), and loader (ld) are standard UNIX programs and are documented in [30]. The assembly language pre-processor (premicroc) and the microcode creation program (microc) were created to support ATOM research and are described in [8]. The remaining tools, microassembler (mas) and microlinker (mulnk), are discussed in [41] and [64], respectively. Figures 26 and 27 show the interaction between the functional elements of the language

processing portion of ATOM. Note that in these figures broken boxes represent programs, boxes made of "#" characters represent files, and boxes made of "*" represent manual processing.

In the implemented environment emphasis was placed on speed of implementation rather than on performance of the resulting system. As a result maximum use was made of existing software. As can be seen from figures 26 and 27, conventional UNIX compilers and language processors are used where possible. The C compiler is initially used to create assembly language source for the abstract data types. Each abstract type is identified by being placed in a separate source code file. These files are then identified by a naming convention so that the language processors can identify the code which was generated by translating them.

Premicroc is responsible for scanning the assembly language generated by the compiler and inserting gateway routines and identifying certain constructs which are difficult to translate to microcode such as branches and switch statements. The result of premicroc is an annotated assembly language program for use in creating the microcode source program. The annotations consist of labels which follow a predetermined naming convention so that they may be handled in special ways by microc.

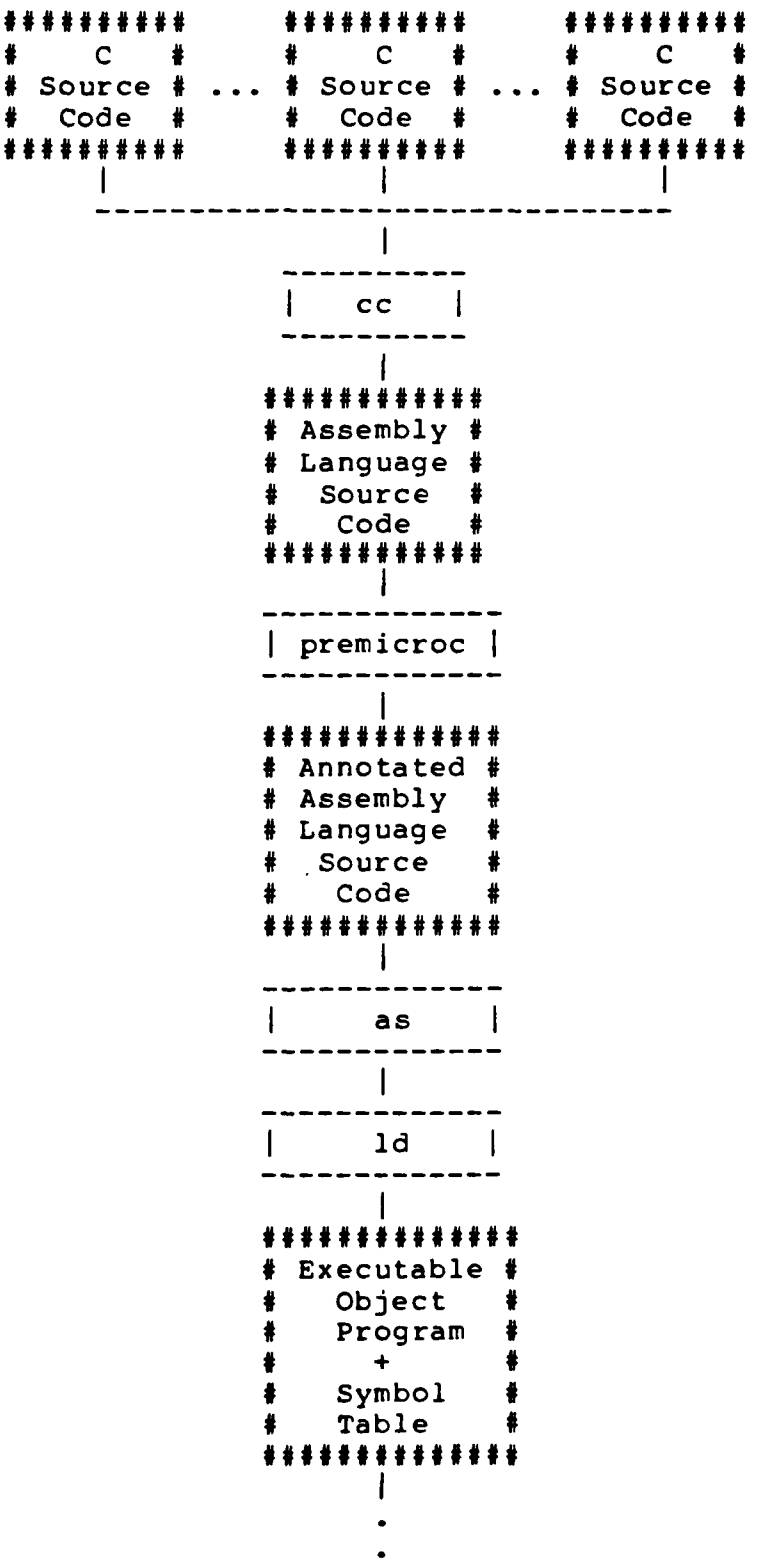


Fig. 26--ATOM Language Processors (Part 1)

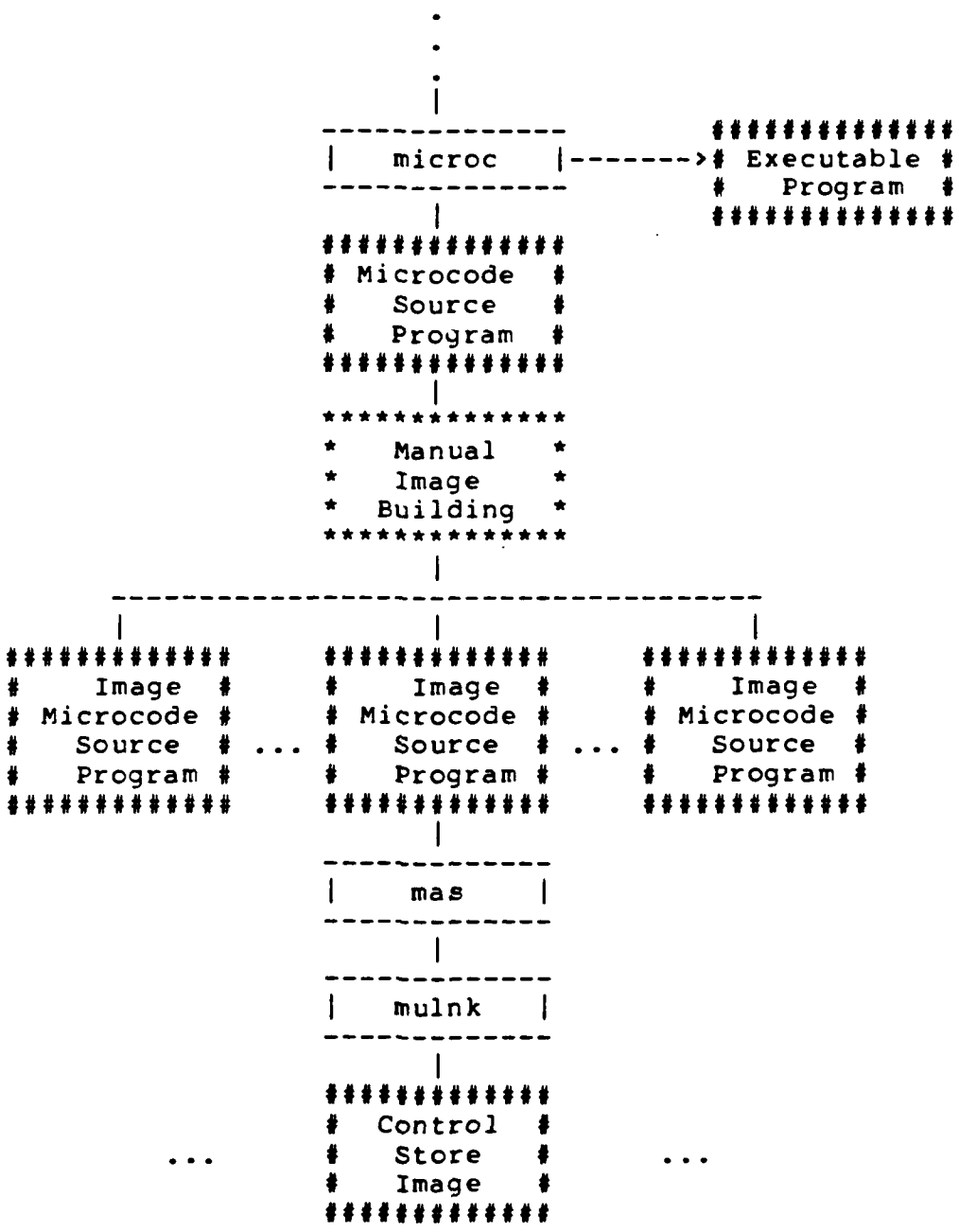


Fig. 27--ATOM Language Processors (Part 2)

The annotated assembly language is compiled and linked to create an executable program image. This image also contains the symbol table information which is used by the UNIX loader and debugging tools. Microc uses this same information in microcode generation. Note that in the implemented environment a programmer is responsible for inserting calls to the environment gateway. This routine is called set_mu and is described in the previous section of this chapter. Microc examines the executable program and using the symbol table information, which contains the labels inserted by premicroc, generates the microcode source for the identified abstract data types. The source code created at this time includes all identified abstract types as well as the entry point logic required for entering and returning from the control store. This code also contains logic for exiting microstore for handling interrupts and returning to control store when interrupt processing is completed.

The final stage of the language processing follows manual manipulation of the created microcode source by the programmer. The programmer is responsible for creating microcode source files for each of the identified execution environments. Each of these files must contain the control store entry, exit, and interrupt handling code previously described. After these files are created the programmer then assembles them with the microassembler and links them with the microlinker. At this time the images are ready to be

executed when loaded by the set_mu calls and entered by the programmers macrocode program.

Performance Estimation Model

One of the requirements for determining which abstract types to migrate in ATOM is to have some knowledge of resulting performance improvements which result from the migration. In chapter III it was shown that the performance improvement of ATOM is a function of decreases in fetch, decode, and microinstruction execution time. These times can be calculated by carefully examining a program to determine what instructions are executed and how frequently this execution occurs. This estimation can be made for both a conventional program and a program which contains abstract data types which have been migrated by ATOM. The programs which implement the performance estimation model were created to support the ATOM environment and include an assembly language preprocessor (asblock), an object code scanning program (objblock), a data collection program (blockcnt), and a data reduction program (model). Figures 28 and 29 show a schematic diagram of the performance estimation model and follow the same conventions as the previous figures.

The first stage of the performance estimation model involves the programs asblock and objblock. In this stage a program is broken into its basic blocks. The basic blocks

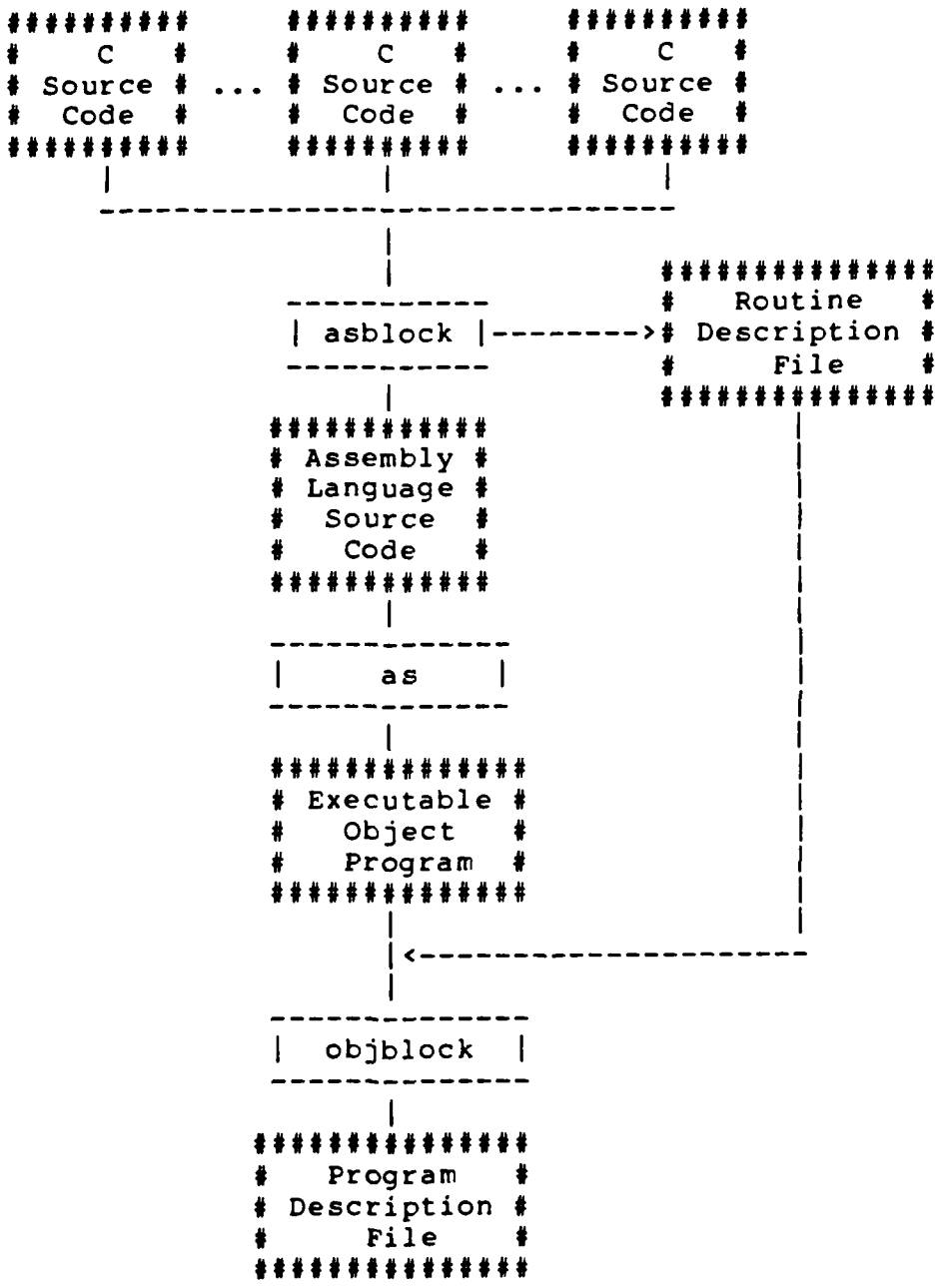


Fig. 28--Performance Estimation Model (Part 1)

are determined by examining the assembly language code created by the C compiler for each identified routine. Basic blocks are formed by dividing the assembly language program into segments of code which have only one possible entry point and one exit point. Each of these segments is a basic block. Code is inserted at the entry point of each block to count the number of times the block is entered. Code is also inserted after the block exit to determine how many times the conditional branch at the end of the block is not taken. This is called the fall-through count. Note that if the block does not terminate with a branch then the fall-through count will always equal the entry count. The routine *description* file produced in this phase lists the basic blocks and the instructions which occur in these blocks. Objblock scans the executable image and determines the addressing modes and locations for each of the instructions. This data is necessary for calculating correct fetch and decode timings. At the end of this first stage a complete program description has been made and is included in the program description file.

The second stage of the performance estimation model is the running of experiments. The data collection program, blockcnt, runs the executable image of the program to be analyzed. When the program is complete, it signals blockcnt that execution is finished and the resulting counts are ready. Blockcnt extracts the block entry and fall-through

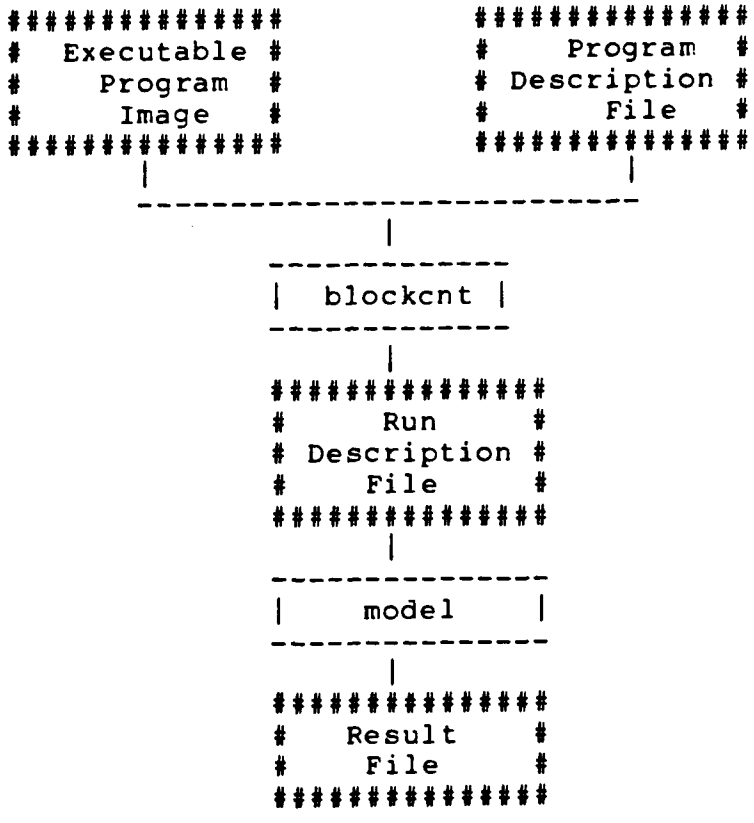


Fig. 29--Performance Estimation Model (Part 2)

counts from the program and incorporates these with the program description file resulting in a run description file. The result is a complete description of the experiment including basic block, instruction, and block entry count data. This is the information which is used for describing the performance improvement which can be expected in migrating abstract types.

The third and final stage of the performance estimation model is to run the data reduction program, model. Model accepts the data collected in the previous phase along with a cache hit probability supplied by the user and estimates the performance of the migrated program. The first step of this estimation is to determine analytically the runtime of the macrocode version of the program. This is done by summing for each basic block the amount of time spent in fetch and decode as well as the number of microinstructions executed in implementing the instructions of the block. These results are then multiplied by the number of entries for each block of the routine. The sum of the resulting counts is then used in calculating total macrocode program execution time.

The result of the above procedure is a complete picture of execution time including fetch, decode, and microinstruction execution time. The fetch, decode, and microinstruction execution count are machine dependent and can be contained in tables passed to the performance estimation

model. Machine dependent fetch and decode times may be passed either as absolute times or as memory fetch counts and microinstruction counts for these operations. In the current implementation the fetch and decode times are coded in the program while the instruction information is passed through a table.

Model's second step is to estimate the performance of the migrated program version. This is done in much the same way as the macrocoded version. The runtime of each routine is initially set to the calculated value of gateway overhead. This involves the number of fetches, decodes, and microinstructions required to determine if the routine being entered is in microcode. Added to this is the number of microinstructions required to implement the routine. This number is estimated by determining the number of microinstructions needed to fetch the operands according to the addressing mode of the instruction. This number is added to the number of microinstructions required to implement the instruction in the instruction set emulator. This sum forms an upper-bound on the number of migrated microinstructions to be executed. The estimated performance of a routine is then formed by multiplying the number of block entries for each block times the fetch, decode, and microinstruction count for each block. A worst-case control store loading factor of 2.25% of total execution time is then added to

each routine. This number was determined through empirical analysis of control store loading on the PE-3220 [6].

The expected performance improvement for each routine can be calculated by the difference between the total estimated runtimes for the macrocode and microcode versions of the program. Figure 15 of chapter III shows the resulting output from running the model for the complex number abstract data type. Figure 22 shows the results of the same run but with an 85% cache hit ratio. The model also calculates a parameter which illustrates the amount of potential savings for each control store word used. This parameter can be used as a local profit estimation for the architecture synthesis algorithm of figure 10. Note that in the implemented performance estimation technique, pipelining is not modeled. Statistical models however, can account for pipelined behavior and can be used in enhancing the model. Cache size is modeled as a function of the cache hit probability parameter.

Case Study: Matrix

In the preceding sections an implementation of the ATOM environment has been presented. In this section a case study of migrating a matrix abstract data type is examined. The abstract type being migrated is a matrix containing double precision floating point numbers. The matrix is dynamically allocated from the C heap by calls to the runtime sup-

port routine malloc. The storage is returned to the heap by a call on the runtime routine free. These routines are part of the C library and call the operating system only when additional memory is required for expanding the heap.

The matrix package is implemented using the language processors described in this chapter. The entry points of the matrix data type are mopen, assign, retrieve, mclose, and multiply. Mopen and mclose allocate and free the storage required to store an array of the given size. The numbers of rows and columns are passed to mopen as parameters. As mentioned before, mopen and mclose interface with the C library as well as UNIX for maintaining the C supported heap. The function of assign and retrieve are to store and retrieve data from the matrix. Assign is called with three parameters which describe the value to be inserted into the matrix and the row and column of the designated item. Retrieve is a function which returns the value indicated by its row and column parameters. Multiply uses mopen, assign, and retrieve to create a new matrix formed by the product of the matrices passed as parameters. Multiply returns a pointer to this newly created array.

Seven programs were used in the experiments listed in figure 30a. The first set of three programs used the matrix abstract data type but were not concerned with environment switching. The second set of three programs also used the matrix data type but in addition changed execution environ-

Run Number	Control Store Contention	Control Store Sharing	Environment Switching
1	No	No	No
2	No	No	No
3	Yes	No	No
4	Yes	Yes	No
5	No	No	Yes
6	Yes	No	Yes
7	Yes	Yes	Yes

a) Experiment Description

Program Name	Rows	Columns	User Time	System Time	Elapsed Time
micromat	10	10	.1	.3	:02
	20	20	.6	.4	:02
	30	30	2.0	.4	:05
	40	40	4.6	.4	:10
	50	50	8.9	.4	:19
	60	60	15.4	.4	:32
	70	70	24.3	.5	:50
	80	80	36.1	.5	1:14
	90	90	51.3	.5	1:44
	100	100	70.2	.5	2:22
macromat	10	10	.1	.1	:01
	20	20	.7	.1	:02
	30	30	2.3	.1	:05
	40	40	5.4	.1	:11
	50	50	10.4	.1	:21
	60	60	17.9	.1	:37
	70	70	28.3	.2	:58
	80	80	42.1	.2	1:25
	90	90	59.8	.2	2:01
	100	100	81.9	.3	2:46

b) Run #1 Results

Fig. 30--Matrix Test Runs (Part 1)

ments. This included loading a new architecture and restoring the previous architecture on return from the called environment. The additional architecture which was needed for environment switching was of the same size and complexity as the matrix abstract data type. The first set of programs create a new matrix and perform a single matrix multiplication storing the results in a newly created matrix. The programs involved were macromat, micromat, and micromatu. These programs represented the macrocode, migrated, and migrated but unshared microcode programs, respectively. All three programs accomplished identical processing.

The second set of programs included macromatenv, micromatenv, and micromatenvu. These programs represented the macrocode, migrated, and migrated but unshared microcode programs, respectively. These three programs accomplished identical processing. The major difference in this set of programs and the former set is that environment switching was included in the latter and was not included in the former. The final program, looper, was used for artificially loading the system to simulate interaction with other programs running concurrently with the test environment. In the figures depicting experimental results, user and system time are reported in seconds and are actual processor utilization times. Elapsed times are reported in minutes and

seconds and are a measure of actual program run time. Each of the seven experiments will be disussed individually.

The purpose of the first experiment was to evaluate the performance of an ATOM migrated program in relation to a macrocode program which performed the same processing. In this experiment simple matrix multiplication programs were used for varying matrix sizes. Figure 30b illustrates the results of the experiment. As would be expected the performance improvement for both processor and elapsed time is small where the matrix size is small. This is due to the overhead of loading the control store and the limited opportunity for performance improvement caused by small matrix sizes. The performance was degraded for very small matrix sizes since control store loading overhead was not regained. In general the performance improvement increases as processor activity increases and seems to approach a limit of approximately 14%. Also note that the system time is always higher for the migrated version. This factor is due to the additional control store management overhead involved when running ATOM programs.

Experiment two was designed to evaluate the influence of background processing when running ATOM programs. In this experiment the same matrix multiplication programs were run as in the first experiment, but this time the background program looper was also run. Figure 31a presents the results from running the macrocode and migrated versions of

Program Name	Rows	Columns	User Time	System Time	Elapsed Time
micromat	10	10	.1	.4	:01
	30	30	2.0	.4	:06
	50	50	8.9	.4	:34
	100	100	70.1	.5	4:40
macromat	10	10	.1	.1	:01
	30	30	2.3	.1	:07
	50	50	10.4	.1	:38
	100	100	82.0	.2	5:26

a) Run #2 Results

Program Name	Rows	Columns	User Time	System Time	Elapsed Time
micromatu/ micromat	100	100	70.8	.6	7:04
macromat/ macromat	100	100	82.1	.2	8:07

b) Run #3 Results

Program Name	Rows	Columns	User Time	System Time	Elapsed Time
micromat/ micromat	100	100	70.4	.2	6:58
macromat/ macromat	100	100	82.0	.2	8:07

c) Run #4 Results

Fig. 31--Matrix Test Runs (Part 2)

the matrix program in the presence of background processing. Note that once again small matrix sizes resulted in a performance degradation rather than performance improvement. Performance improvement for larger matrix sizes however, once gain appeared to approach the 14% figure. Also notice that as expected, processor time did not change but elapsed time increased due to the presence of background processing.

Experiment three examined the interaction between two ATOM programs running concurrently without sharing the control store. In this case both programs were performing similar processing however, their control store images although supporting the same execution environment, were not sharable. In addition the looper program was also running to provide additional background processing. The results of this test are shown in figure 31b. The processor times are close to the values determined in runs one and two but the elapsed times differ greatly. This is due to the contention of the two programs in the scheduling of the control store.

Run four is basically the same experiment as run three but with sharable control store images. The results of this test are shown in figure 31c. Note that the sharing of images provides some degree of performance improvement, although relatively small. The major difference in these two runs involves the decrease in the amount of system time in the two programs. The decrease comes from decreased overhead in managing two separate control store images when sharing

is allowed. The percentage performance improvement in both cases is near the 14% factor mentioned before. The processor and elapsed time performance improvements were similar in both cases, 12.8% processor and 12.9% elapsed for the unshared case and 14.1% for both processor and elapsed time in the shared case.

The final set of tests were designed to examine the overhead involved in managing the control when execution environments frequently change. The following programs perform matrix multiplication but for each program invocation ten environment changes are executed. After each environment change one entry into the control store is made to ensure that demand loading occurs. When the control store is exited the matrix execution environment is restored and another matrix multiplication occurs. This process is repeated ten times. The macrocode version performs the same processing with the exception that no environment switching overhead is incurred.

Run five demonstrates the overhead involved in performing this architecture switching. The results of this run are shown in figure 32a. In this run only the macrocode and migrated versions are running with no additional background processing. The system time increase in the migrated version is the expected result of additional control store management overhead. Once again with smaller matrix sizes performance improvement is small, while at larger matrix

sizes performance improvement improves. For matrices with 50 rows and columns, the performance improvement with environment switching is only slightly less than in the non-switching environment, 11.2% as opposed to 11.4%. Elapsed time improvement however, is greater with 11.8% performance improvement in the switching case and only 9% performance improvement in the non-switching case. This difference is a result of added overhead in loading the new architecture and restoring the old architecture on exit from the new execution environment.

Runs six and seven investigate the effect of sharing control store images with environment switching in the same way that runs three and four investigated this effect with no switching. The results shown in figures 32b and 32c are similar. Elapsed times remain approximately equal with a decrease in system time as a result of the decrease in control store management overhead. An interesting phenomenon is the decrease in the system time from run five to runs six and seven. In run five no background processing occurred while in runs six and seven looper was running. It appears that this is an artifact of the UNIX time accounting algorithm which causes the high system time for the single program run. Again a marked increase in elapsed time was evident in moving from a single program to a multi-program, shared environment.

Program Name	Rows	Columns	User Time	System Time	Elapsed Time
micromatenv	30	30	19.8	3.1	:47
	40	40	46.3	3.1	1:40
	50	50	89.6	3.1	3:08
macromatenv	30	30	23.0	.2	:47
	40	40	53.8	.2	1:50
	50	50	104.2	.2	3:31

a) Run #5 Results

Program Name	Rows	Columns	User Time	System Time	Elapsed Time
micromatenvu/ micromatenv	30	30	19.9	1.9	2:07
	50	50	90.0	2.0	9:08
macromatenv/ macromatenv	30	30	23.0	.2	2:14
	50	50	104.2	.2	10:23

b) Run #6 Results

Program Name	Rows	Columns	User Time	System Time	Elapsed Time
micromatenv/ micromatenv	30	30	19.9	1.5	2:05
	50	50	89.6	1.7	9:02
macromatenv/ macromatenv	30	30	23.0	.2	2:14
	50	50	104.2	.2	10:25

c) Run #7 Results

Fig. 32--Matrix Test Runs (Part 3)

These seven tests indicate that the ATOM facility as currently implemented can provide some degree of performance improvement. There are two interesting points which must be emphasized. In the implemented environment microcode is being generated from an examination of the executable machine language program. As a result the generated microprogram is a mirror image of the machine language. The problems which degrade performance for machine language programs also remain for a microprogram generated in this way.

The generated microprogram must include additional microcode for fetching operands since on the implemented machine this is done by the decoding hardware for machine level instructions. The result is that a larger number of microinstructions is executed for each machine level instruction than is executed by the instruction set emulator. The vertical nature of the microarchitecture also prohibits microcode compaction and therefore eliminates another source for performance improvement. The resulting performance improvement can therefore be seen as only a lower-bound to the amount of performance improvement which can be expected in a more complete implementation of ATOM.

In the analysis and implementation of ATOM other abstract data types were also examined. A stack was the first abstract type studied. Two stack operations, pop and push, were migrated for a stack of integers. These operations were by nature very simple and each involved only two

C source language statements of executable code. In the stack implementation no performance improvement was noticed until a very large number of calls had been issued. This was due to not overcoming the control store loading and management overhead until a significant amount of time had been spent in the control store.

The effect of algorithm design once again became evident during this study. When the code which controlled the iteration for the test program was migrated the performance improvement rose significantly. This was due to two factors. The first factor involved the control abstraction for iterative processes. To implement the iteration in control store meant that not only was the body of the loop migrated but also the code which caused the loop to occur. Since the loop body was small, the relative amount of code migrated with the **for** statement was large.

The second factor, which was the more significant, involved trading many control store entries for one entry. In the original migration, for each iteration a control store entry was made for each call on one of the stack procedures. In the fully migrated version, just as many procedure calls were made but this time the calls were within microcode rather than between the macrocode and microcode levels. The lesson to be learned is that in migrating functions in ATOM, one should be aware of the implications of

logically grouping statements into procedures which will become candidates for migration.

Another study migrated the entire C string library. In a program which called each of the six routines of the library once for a variable number of iterations, performance improvement started to be evident with a much smaller number of iterations than in the stack case. This was due to the fact that each of the string routines consumed more processor time than either of the two stack routines. The control store loading overhead was overcome early in this way.

In order to examine the effect of migrating more complex data types, two of the previous data types, stack and matrix, were merged to form a stack of matrices. For large 50x50 matrices the same performance improvement as was generated by the simple matrix example seen in the case study was produced. In the light of the stack results this was not surprising. The contribution of the stack routines to the overall performance improvement was negligible and consequently the results were similar to the matrix example taken alone.

CHAPTER VII

CONCLUDING REMARKS

Summary

As stated in chapter II, the goal of this research is to investigate the feasibility and applicability of migrating abstract data types into the architecture of a microprogrammable computer through dynamic microprogramming. A secondary goal is to allow the use of modern programming practices without excessive runtime costs. The reason for performing this research is to find better ways to overcome problems in mapping high-level programming languages onto the architectures of conventional computer systems. The method used for enhancing this mapping is to form problem oriented architectures which are more closely related to the problem being described by programmers than are conventional architectures.

In order to accomplish this research a background study was accomplished and is reported in chapter II. This background analysis examined four related areas which affected this study: computer architecture, dynamic microprogramming, data abstraction and encapsulation, and vertical migration. It was shown that an integration of key aspects of each of these areas into a single composite

facility can provide a vehicle which can be used to allow abstract data types to be used as the unit of migration in forming problem oriented architectures. A detailed outline of the technique for using abstract data types was then given. A first step in this discussion was to examine other techniques which had been used to form problem oriented architectures. Three techniques, instruction sequence, function oriented, and a hybrid of these two, were discussed.

In the discussion of these migration techniques, it was shown that they are oriented to global solutions for individual programs and ignore the possibility of applying problem oriented architectures to execution environments within programs. These techniques use only a small part of the available information concerning program behavior provided by programming language semantics when creating problem oriented architectures. The ATOM technique was shown to be a more complete method which utilizes information provided by the programmer through language constructs which are related to data abstraction. For these reasons ATOM was shown to be a more informed method for describing problem oriented architectures for programs which exhibit a high degree of execution locality.

Another concept which was considered in the discussion of ATOM is the notion of binding and how it relates to computer architecture. In this discussion it was shown that the process of architecture binding is performed in all com-

puter architectures. The binding process for conventional architectures however, is performed only once when the machine architecture is being designed. This binding is fixed and all programs must be mapped to the architecture resulting from it. In ATOM however, this binding is made at two levels; decisions at each of these levels are made with information concerning the specific problem being solved by the programmer.

High-level binding was shown to be the process of mapping execution environments to the architecture which could most closely represent the problem. This architecture, unlike that of conventional machines, is not fixed but is synthesized to support directly the program being modeled by the programmer. The low-level mapping was then shown to be the method used in implementing the high-level architecture synthesized in the previous step. Again, this binding is program specific and avoids the excessive generality which causes decreased performance in instruction set emulators.

Since a secondary goal of the research involves performance issues, an analysis of the performance improvement expectations of ATOM was the next topic covered. In this analysis a set of performance criteria was established and a study of the effect of ATOM on these was undertaken. It was shown that ATOM can provide a degree of performance improvement due to decreases in fetch, decode, and microinstruction execution time. This performance improvement depends on the

degree to which the program exhibits execution locality. For programs which enter a locality and remain there for a long period of time, the performance improvement will be greater than for programs which do not establish strong localized behavior.

In the original analysis of ATOM, it quickly became evident that a conventional programming language which supported abstract data types should be used. This is motivated by the desire to make the problem oriented architecture synthesis process as transparent as possible to the user. Furthermore, it is desirable to allow both migrated and non-migrated functions to be described in the same way to facilitate program reliability and maintainability. As a result of the decision to use conventional programming languages, an analysis of their effect on ATOM was undertaken. In this analysis it was shown that language scope facilities directly affect the ability of ATOM to synthesize problem oriented architectures.

Languages which have complex scope rules can describe to a greater degree the abstract data types which are of most interest in each program locality than can languages which support only limited scopes. An example of the differences between Ada and Pascal scope rules illustrated this point. The two concepts of derived and generic data types were also examined in light of how they may affect ATOM. It was shown that derived types can be implemented in ATOM and

that a reasonable distinction can be made between the applicability of migration of both derived and parent types. Generic data types were shown to provide new types for consideration by the ATOM architecture synthesis algorithm.

The effect of machine architecture was the next topic presented. This study highlighted the effects of different machine implementation considerations on ATOM. The first areas covered were the system-level architecture considerations, multiprogramming and multiprocessing. It was shown that ATOM can reduce the number of memory accesses caused by instruction fetching and in this way can increase the amount of excess memory bandwidth. It was also shown that sharing of control store images can further improve the performance of ATOM in multiprogramming environments. Also briefly discussed was the problem of scheduling in a multiprocessing environment where each processor in the system has an independently loadable control store.

The low-level implementation issues were then discussed. It was shown that processor implementation techniques such as pipelining and cache memories can decrease the amount of performance improvement which can be expected with ATOM. It was also shown however that in certain cases, ATOM can still improve performance significantly by reducing the amount of generality required in program execution and as a result reduce the number of microinstructions executed.

The effect of microarchitecture on performance improvement expectation was also discussed.

It was also shown that horizontal architectures, due to their inherent parallelism, provide a greater opportunity for performance improvement than vertical architectures. The effect of compaction in horizontal machines was shown to be the major source of performance improvement opportunities. A further discussion derived the requirements for future microarchitectures in support of ATOM-like architectures. It was shown that features similar to those which allowed multiprogramming at the machine architecture level are required in any microarchitecture which supports widespread user microprogramming. These facilities include control store sharing and protection, microcode trap and supervisor call facilities, and privileged microcode execution modes.

To support the analysis of ATOM, a partial implementation was described. This implementation includes a modified architecture synthesis procedure which makes use of a performance estimation model and microprogramming tools. The performance estimation model was implemented to support ATOM and describes in detail the behavior of a macrocode program and estimates the performance behavior of the same program when implemented with ATOM.

The microprogramming tools used in this implementation include a microcode compiler for the C programming language

and a linker/loader facility for a microprogramming environment. A detailed case study was shown which describes the behavior of a matrix abstract data type in several execution environments. This abstract type was migrated to control store using the implemented ATOM facility. A performance analysis showed that even in the absence of complex migration tools and in an entirely vertical microarchitecture a significant performance improvement could be expected with ATOM.

The Significance of ATOM

The major effect of ATOM is that it provides a new method for describing units of vertical migration. ATOM exploits the intuitive notion that if a programmer describes a data type and operations on that type then there must be something special about the objects of that type. The ATOM technique assumes that the programmer is describing those data types which are the primitive types of the problem solution. In conventional architectures the decision as to what data types should be supported in the architecture is made by the computer architect. In ATOM this decision is influenced by the programmer who knows what data types are actually needed for the particular problem at hand.

In other techniques for architectural redefinition, the unit of migration is dependent upon program function. In the techniques shown in chapter II, instruction sequences

are shown to be the unit which is migrated. There is no logical interaction described between these instruction sequences. The result, therefore, is a collection of independent program elements which have been selected for migration based solely on their appearance in a program in a certain order. ATOM on the other hand uses the programming language semantics, used by the programmer in describing the solution to a problem, as a basis for migration decisions. The resulting decision therefore, is a closer representation of the problem to be solved. This results in easier mapping to the synthesized architecture. It also provides a firm basis for reasoning about the program and its corresponding problem oriented architectures.

Since ATOM utilizes programming language semantics, it can exploit other modern programming practices such as libraries and separate compilation. These migrated routines can be maintained in both macrocode and microcode form and can be migrated simply by environment gateway switching. This is in contrast to other techniques which must recompile entire programs to determine the problem oriented architecture to be implemented. An architecture created in this latter way is a single, global solution to the mapping problem. ATOM on the other hand provides multiple problem oriented architectures where needed for different execution environments of a single program. ATOM also can account for dependencies of migrated routines by using the semantics of

the programming language to determine where dependencies lie and to what degree this dependency can be exploited.

The major significance of ATOM, therefore, is that it provides a new method of reasoning about vertical migration units. This reasoning can be applied when making migration decisions, tuning applications, or even when designing programs. ATOM opens the way for more formal techniques to be applied to the analysis of vertical migration decisions. The same techniques which have been used to verify and analyze program behavior can now be applied to programs which utilize vertical migration.

Directions for Further Research

In describing the basic ATOM technique, several areas were discovered which require further analysis. The first of these areas involves determining if a program exhibits the proper locality behavior required for performance improvement in ATOM. As shown in the implementation study of chapter VI, a performance degradation can actually occur if control store management overhead cannot be regained. This is a function of the complexity of the migrated routines and the number of times the routines are called. Further analysis of program structure can possibly determine a better measure of local profit. Local profit could then be examined to determine if the migration candidate meets a

minimal performance improvement opportunity before it is migrated.

The potential for library use in ATOM raises the possibility that repetitive use of a package can result in more complete information concerning local behavior of the package. Since this information is an indicator of local profit, and since local profit is used in migration decisions, a more effective migration decision can be made if this empirical data can be collected. Heuristics can be described which allow a program to tune its own execution environment based on its behavior. This same information can also be applied to library routines to allow other programs to benefit from this performance analysis. Certain artificial intelligence techniques can then be applied to assist static migration decisions. Expert systems can be described to utilize this information in moving the migration decision closer to optimal results.

In chapter IV a discussion was presented which looked at data type design for implementation in an ATOM environment. In this discussion the notion of data type factoring was shown to be a method for determining data types which are more primitive than those originally designed by the programmer. If ATOM could perform this factoring, the performance improvement for each execution environment could approach the performance improvement gained if the execution environment was migrated using the instruction sequence

approach. Primitive type detection could be carried on in much the same way as the instruction sequence method determines migration units. In this case each instruction sequence could be examined to determine if it actually implements a routine which operates on objects of a new primitive type. If this were found to be the case, the same reasoning applied to programmer described abstract data types could also be applied to these newly discovered primitive types. Iterative application of this procedure could therefore find and implement the true primitive data types for the problem solution being implemented.

One of the most interesting of the problems encountered in implementing ATOM was control store management. In ATOM one of the most difficult decisions to make is which of the abstract types for a particular execution environment to migrate. The technique described in this research requires that a static decision be made concerning the most appropriate types to migrate for an execution environment. This decision is required since a control store is a fixed size resource and can hold only a fixed number of microinstructions. If all data types for an execution environment could be translated to microcode and if the control store could be managed like a cache, such as in the adaptive instruction set concept [59], a completely dynamic binding could be realized. The static decision would no longer be required. The combination of ATOM and the adaptive instruction set

concept could provide for dynamic problem oriented architectures even in languages such as SNOBOL which have dynamic scopes.

LIST OF REFERENCES

1. A. M. Abd-Alla and D. C. Karlgaard, "Heuristic Synthesis of Microprogrammed Computer Architecture," IEEE Transactions on Computers Vol. C-23, pp.802-807 (1974).

2. A. K. Agrawala and T. G. Rauscher, Foundations of Microprogramming -- Architecture, Software, and Applications, Academic Press, Inc., New York, New York (1976).

3. H. G. Baker and C. Parker, "High Level Language Programs Run Ten Times Faster in Microstore," ACM SIGMICRO Newsletter Vol. 11(3&4), pp.171-173, MICRO-13 Proceedings Thirteenth Annual Workshop on Microprogramming (September-December 1980).

4. G. Booch, "Object-oriented Design," ACM SIGPLAN Ada Letters Vol. I(3), pp.I-3.64-76 (1982).

5. G. Booch, Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California (1983).

6. R. L. Booth, Cost To Load and Enter the WCS Under UNIX With No WCS Cache Configured, Vanderbilt University, Nashville, Tennessee (August 1983). (Unpublished)

7. K. J. Butler, et. al., Revised DIANA Reference Manual, Tartan Laboratories (June 1982).

8. E. M. Carter, A Microprogramming System Based on the C Programming Language, Vanderbilt University (November 1983). Tech. Report CS-83-07

9. Y. Chu and M. Abrams, "Programming Languages and Direct Execution Computer Architecture," Computer Vol. 14(7), pp.22-32 (July 1981).

10. E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, New Jersey (1976).

11. K. A. El-Ayat and J. A. Howard, "Algoritms for a Self-tuning Microprogrammed Computer," ACM SIGMICRO Newsletter Vol. 8(3), pp.85-91, MICRO-10 Proceedings Tenth Annual Workshop On Microprogramming (September 1977).

12. R. Ellison, MicroC: A High Level Microprogramming Language, Southern Illinois University (July 1978). Master's Thesis

13. J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers Vol. C-30(7), pp.478-490 (July, 1981).

14. J. A. Fisher, Very Long Instruction Word Architecture and the ELI-512, Yale University (April, 1983). Research Report YALEU/DCS/RR-253

15. M. J. Flynn, "Computer Organization and Architecture," pp. 17-98 in Operating Systems An Advanced Course, Springer-Verlag, New York, New York (1979).

16. J. Gannon, et. al., "Data-Abstraction Implementation, Specification, and Testing," ACM Transactions on Programming Languages and Systems Vol. 3(3), pp.211-223 (July 1981).

17. C. M. Geschke, et. al., "Early Experience with Mesa," Communications of the ACM Vol. 20(8), pp.540-552 (August 1977).

18. J. L. Gieser and R. J. Sheraga, "On Horizontally-Microprogrammed Microarchitecture Description Techniques," IEEE Transactions on Software Engineering Vol. SE-8(6) (September 1982).

19. J. Hennessy, et. al., MIPS: A VLSI Processor Architecture, Stanford University Departments of Electrical Engineering and Computer Science (November 1981). Technical Report No 223.

20. M. Herlihy and B. Liskov, "A Value Transmission Method for Abstract Data Types," ACM Transactions on Programming Languages and Systems Vol. 4(4), pp.527-551 (October 1982).

21. B. Holtkamp and H. Kaestner, "A Firmware Monitor To Support Vertical Migration Decisions In the Unix Operating System," ACM SIGMICRO Newsletter Vol. 13(4), pp.153-162, MICRO-15 Proceedings Fifteenth Annual Workshop on Microprogramming (December 1982).

22. J. J. Horning, "Some Desirable Properties of Data Abstraction Facilities," ACM SIGPLAN Notices Vol. 8(2), pp.60-63, Proceedings of Conference on Data: Abstraction, Definition, and Structure (March 1976).

23. J. Ichbiah, "Rationale for the Design of the Ada Programming Language," ACM SIGPLAN Notices Vol. 14(6) (June 1979).

24. A. K. Jones and B. H. Liskov, "A Language Extension for Controlling Access to Shared Data," IEEE Transactions on Software Engineering Vol. SE-2(4), pp.277-285 (December 1976).

25. A. K. Jones, "The Object Model: A Conceptual Tool for Structuring Software," pp. 7-16 in Operating Systems An Advanced Course, Springer-Verlag, New York, New York (1979).

26. B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

27. D. Landskov, et. al., "Local Microcode Compaction Techniques," ACM Computing Surveys Vol. 12(3), pp.261-294 (July-September 1980).

28. B. Liskov, et. al., CLU Reference Manual, Springer-Verlag, Berlin (1981). Computer Science Lecture Series No. 114

29. M. Maekawa, et. al., "Firmware Structure and Architectural Support for Monitors, Vertical Migration and User Microprogramming," ACM SIGARCH Computer Architecture News Vol. 10(2), Proceedings Symposium on Architectural Support for Programming Languages and Operating Systems.

30. M. D. McIlroy, UNIX Programmer's Manual, Bell Laboratories (September 1978). Seventh Edition

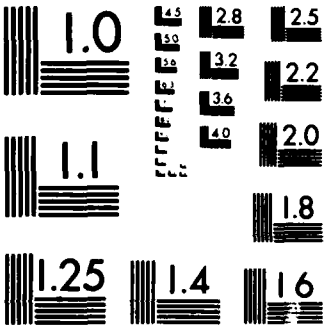
31. G. J. Myers, Advances in Computer Architecture, Wiley-Interscience, New York, New York (1982).

32. A. Nicolau and J. Fisher, "Using an Oracle to Measure Potential Parallelism in Single Instruction Stream Programs," ACM SIGMICRO Newsletter Vol. 12(4), pp.171-182, MICRO-14 Proceedings Fourteenth Annual Workshop on Microprogramming (December 1981).

33. M. Ohlin, The CLASS and Pointer Concepts in SIMULA, Swedish Research Institute of National Defense, Stockholm Sweden (31 July 1975). C10045-M3(E5)

AD-A135 840 ABSTRACT TYPE ORIENTED DYNAMIC VERTICAL MIGRATION(U) 3/3
AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
E M CARTER DEC 83 AFIT/CI/NR-83-74D

UNCLASSIFIED F/G 9/2 NL



34. A. G. Olbert, "Crossing the Machine Interface," ACM SIGMICRO Newsletter Vol. 13(4), pp.163-170, MICRO-15 Proceedings Fifteenth Annual Workshop on Microprogramming (December 1982).

35. E. Organick and J. Hinds, Interpreting Machines, North Holland Publishing Co., New York, New York (1978).

36. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM Vol. 15(12), pp.1053-1058 (December 1972).

37. D. A. Patterson and D. R. Ditzel, "RISC I: A Reduced Instruction Set VLSI Computer," ACM SIGARCH Computer Architecture News Vol. 9(3), pp.443-457, Proceedings Eighth Annual Symposium on Computer Architecture (1981).

38. F. Pollack, et. al., "Supporting Ada Memory Management in the iAPX-432," ACM SIGARCH Computer Architecture News Vol. 10(2), pp.117-131, Proceedings Symposium on Architectural Support for Programming Languages and Operating Systems (March 1982).

39. G. Radin, "The 801 Minicomputer," ACM SIGARCH Computer Architecture News Vol. 10(2), pp.39-47, Proceedings Symposium on Architectural Support for Programming Languages and Operating Systems (March 1982).

40. T. G. Rauscher and A. K. Agrawala, "Dynamic Problem-Oriented Redefinition of Computer Architecture via Microprogramming," IEEE Transactions on Computers Vol. C-27(11), pp.1006-1014 (November 1978).

41. J. E. Roskos, Microprogramming Tools for Perkin-Elmer 3220 UNIX, Vanderbilt University (December 1981). Masters Thesis

42. J. H. Saltzer, "Naming and Binding of Objects," in Operating Systems, Springer-Verlag, Berlin (1978). Lecture Notes in Computer Science, Nr. 60

43. J. E. Sammet, Programming Languages: History and Fundamentals, Prentice-Hall, Englewodd Cliffs, New Jersey (1969).

44. M. Shaw, ALPHARD: Form and Content, Springer-Verlag, New York, New York (1981).

45. R. J. Sheraga, User Microprogramming for the VAX 11/780, JRS Research Laboratories, Orange, California (December 1981). Presented at 1981 Fall DECUS Symposium

46. M. Sint, "A Survey of High Level Microprogramming Languages," ACM SIGMICRO Newsletter Vol. 11(3&4), pp.141-153, MICRO-13 Proceedings Thirteenth Annual Workshop on Microprogramming (September-December 1980).

47. J. A. Stankovic, "The Types and Interactions of Vertical Migrations of Functions In A Multilevel Interpretive System," IEEE Transactions on Computers Vol. C-30(7), pp.505-513 (July 1981).

48. J. A. Stankovic, Structured Systems and Their Performance Improvement through Vertical Migration, UMI Research Press (1982).

49. J. A. Stankovic, "Good System Structure Features: Their Complexity and Execution Time Cost," IEEE Transactions on Software Engineering Vol. SE-8(4), pp.306-318 (July 1982).

50. J. Stockenberg and A. van Dam, "Vertical Migration for Performance Enhancement in Layered Hardware/Firmware/-Software Systems," IEEE Computer Vol. 11(5), pp.35-50 (May 1978).

51. B. Stroustrup, "Classes: An Abstract Data Type Facility for the C Language," ACM SIGPLAN Notices Vol. 17(1), pp.42-51 (January 1982).

52. J. W. Thatcher, et. al., "Data Type Specification: Parameterization and the Power of Specification Techniques.," ACM Transactions on Programming Languages and Systems Vol. 4(4), pp.711-732 (October 1982).

53. M. Tokoro, et. al., "Optimization of Microprograms," IEEE Transactions on Computers Vol. C-30(7), pp.491-504 (July, 1981).

54. A. B. Tucker and M. J. Flynn, "Dynamic Microprogramming: Processor Organization and Programming," Communications of the ACM Vol. 14(4), pp.240-250 (April 1971).

55. T. Weidner and J. A. Stankovic, "Vertical Migration," in The Microprogramming Handbook, ed. S. Habib. To Appear

56. W. T. Wilner, "Design of the Burroughs B1700," pp. 489-497 in 1972 Fall Joint Computer Conference Proceedings, AFIPS Press, Montvale, New Jersey (1972).

57. W. T. Wilner, "Burroughs B1700 Memory Utilization," pp. 579-586 in 1972 Fall Joint Computer Conference Proceedings, AFIPS Press, Montvale, New Jersey (1972).

58. P. F. Wilk and G. M. Bull, "A Strategy, Method, and Set of Tools For A User, Dynamic Microprogramming Environment," pp. 54-61 in Systems Architecture: Proceedings of the Sixth ACM European Regional Conference, Westbury House, Surrey, England (1981).

59. R. I. Winner, "Adaptive Instruction Sets and Instruction Set Locality Phenomena," in Proceedings of IEEE International Workshop on Computer Systems Organization (March 1983).

60. R. I. Winner and L. B. Reed, An Overview of Sharing the Control Store Under UNIX, Submitted to SOFTWARE: Practice and Experience, August 1983.

61. N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, Englewood Cliffs, New Jersey (1976).

62. N. Wirth, "Modula: A Language for Modular Multiprogramming," Software - Practice and Experience Vol. 7(1), pp.3-35 (January 1977).

63. N. Wirth, Modula-2, Institut fur Informatik ETH, Zurich, Switzerland (December 1980). Nr. 36

64. T. M. Wood, A Linker and Librarian for Vertical Migration, Vanderbilt University (December, 1983). Master's Thesis

65. D. B. Wortman, A Study of Language Directed Computer Design, Computer Systems Research Group, University of Toronto (December 1972). Technical Report CSRG-20

66. W. A. Wulf, "Reliable Hardware-Software Architecture," ACM SIGPLAN Notices Vol. 10(6), pp.122-130, Proceedings International Conference on Reliable Software (June 1975).

67. W. A. Wulf, "Compilers and Computer Architecture," IEEE Computer Vol. 13(7), pp.41-47 (July 1981).

68. W. A. Wulf, R. Levin, and S. Harbison, Hydra: C.mmp: An Experimental Computer System, McGraw-Hill Book Company, New York, New York (1981).


FILM
1-8