FORMAL REPRESENTATION OF COMPUTER ARCHITECTURE

RICHARD LUNT NORTON, JR.
FORMAL REPRESENTATION OF COMPUTER ARCHITECTURES

BY

RICHARD LUNT NORTON, JR.

B. Indiv. Stud., New Mexico State University, 1974
M. S., New Mexico State University, 1975
M. S., University of Illinois, 1980

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1985

Urbana, Illinois
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

AUGUST 1984

WE HEREBY RECOMMEND THAT THE THESIS BY

RICHARD LUNT NORTON, JR.

ENTITLED FORMAL REPRESENTATION

OF COMPUTER ARCHITECTURES

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

Director of Thesis Research

Head of Department

Committee on Final Examination†

Jacob A. Abraham Chairman

Ed Davidson

Jane E.

† Required for doctor's degree but not for master's.
This thesis is on the subject of computer architecture, specifically in the area of architecture representation. The main results are that it is possible to represent computer architectures in a formal fashion and that there are theoretical and practical advantages to considering architecture in the abstract. The representation that we define and propose here is mathematically sound, and admits a very simple and direct translation into an executable program, which can then be used to simulate the machine allowing the evaluation of both the functionality and the performance of a target system.
ABSTRACT

This thesis is on the subject of computer architecture, specifically in the area of architecture representation. The main results are that it is possible to represent computer architectures in a formal fashion and that there are theoretical and practical advantages to considering architecture in the abstract. The representation that we define and propose here is mathematically sound, and admits a very simple and direct translation into an executable program, which can then be used to simulate the machine allowing the evaluation of both the functionality and the performance of a target system.
ACKNOWLEDGMENT

Many people provided essential support for this work. Foremost among them is Jacob Abraham who is, in my opinion, the perfect advisor and a dear friend. I am very grateful that Jacob Abraham, Ed Davidson, and Janak Patel have been so successful in making the Computer Systems Group at the University of Illinois such a rewarding intellectual environment. I am greatly indebted to Kent Fuchs, Jackie Ziemer, Ellen Oliver, and especially to Karen.
# TABLE OF CONTENTS

1. INTRODUCTION........................................................................................................... 1
   1.1. The Problem........................................................................................................ 1
   1.2. Prior Work.......................................................................................................... 3
   1.3. Advantages of a Representation....................................................................... 5
   1.4. Outline of Subsequent Chapters....................................................................... 7

2. A MODEL OF MACHINE BEHAVIOR......................................................................... 8
   2.1. The Need for a Model of Architecture............................................................. 8
   2.2. A Formal Model of Computer Architecture................................................... 9
   2.3. Interconnection................................................................................................... 14
   2.4. Graph Notation.................................................................................................. 15

3. INSTRUCTION SET INTERPRETERS.......................................................................... 18
   3.1. Definition of an Interpreter............................................................................. 18
   3.2. Instruction Set Level......................................................................................... 19
   3.3. Practical Interpreters....................................................................................... 27
   3.4. Comparing Interpreters................................................................................... 33
   3.5. Performance...................................................................................................... 36

4. EVALUATION TOOLS................................................................................................ 40
   4.1. LX0: A Testbed Language................................................................................. 40
   4.2. Ptrace................................................................................................................ 41
   4.3. LS1: A Concurrent Programming Language.................................................. 42
   4.4. Other Tools....................................................................................................... 47

5. CACHE POLICY AND A STATIC INTERPRETER......................................................... 48
   5.1. Multiprocessor Bus Structure.......................................................................... 49
   5.2. Cache Coherence and Performance Gain from Write-Back......................... 50
   5.3. Simulation Analysis......................................................................................... 55
   5.4. Effects of Cache Policy.................................................................................... 56
6. ANALYSIS OF AN ADAPTIVE INTERPRETER ................................................................. 67
   6.1. Direct Interpretation ......................................................................................... 69
   6.2. High-Level Language Behavior ....................................................................... 69
   6.3. Proposal for an Adaptive Interpreter ............................................................... 74
   6.4. Performance Evaluation Tools .......................................................................... 82
   6.5. Event-Driven Simulation Facility ....................................................................... 82
   6.6. Instruction-Level Instrumentation ..................................................................... 82
   6.7. Microcode-Level Instrumentation ..................................................................... 83
   6.8. Performance of the I-Cache .............................................................................. 84
   6.9. Matrix Multiplication ....................................................................................... 86
   6.10. Reuse of Addresses ......................................................................................... 86
   6.11. Set Size ........................................................................................................... 88
   6.12. Summary of I-Cache Results ........................................................................... 90

7. EXPLOITING HIGHER-LEVEL SEMANTICS ................................................................. 92
   7.1. Algorithm Issues ............................................................................................. 92
   7.2. Probabilistic Considerations ............................................................................ 95
   7.3. A Numerical Analysis Example ......................................................................... 97
   7.4. Comparison with MCPAM .............................................................................. 97
   7.5. Comparison with a Pattern Matching Program ............................................... 102
   7.6. Semantic Locality ............................................................................................ 102

8. CONCLUSIONS ......................................................................................................... 108
   8.1. Future Work ..................................................................................................... 109

REFERENCES .............................................................................................................. 110

VITA .............................................................................................................................. 113
1. INTRODUCTION

This thesis is on the subject of computer architecture, specifically in the area of architecture representation. The main results are that it is possible to represent computer architectures in a formal fashion and that there are theoretical and practical advantages to considering architecture in the abstract. The representation that we define and propose here is mathematically sound, and admits a very simple and direct translation into an executable program, which can then be used to simulate the machine allowing the evaluation of both the function and the performance of a target system.

1.1. The Problem

In designing a computer system, it is common that after the initial specifications of performance and function have been made, candidate architectures are evaluated and one is adopted. In the process, there is nothing explicitly guiding the generation of the candidate architectures other than experience. In this thesis we argue that it is useful to appeal to an architecture theory both to reduce the number of candidates and to evaluate those that are left. Key to this theory is a representation for computer architecture that is both general and executable, in the sense that a system so specified can be simulated with the representation as the definition of the semantics of the system.

Using a representation for architecture isn’t a revolutionary idea, in that designers commonly use many means to describe a system [1]. Even if a formal method, such as flowcharts or a functional simulation, are not used, the design is usually specified in prose, and is termed a “functional specification” [2,3]. Since modern hardware tends to be complicated, some kind of specification is essential just to allow multiple people to work on an implementation. A good representation is very useful even for a single implementor, since it allows one to selectively ignore various parts of a system and to concisely and accurately specify changes. Such principles of sound engineering design have been known and practiced for many years, and have recently been embraced by the software engineering community under the name of structured programming and top down design. Given that these techniques are
viable, one should consider whether they can be directly applied to the design of hardware. Unfortunately, there are three features of the hardware design environment that prevent these techniques from being directly applicable. The first is the multi-level nature of hardware design. It is common that almost any piece of hardware will have some state machine control. Typically, the designer is responsible for defining what amounts to an instruction set for this state machine, as well as the associated data paths, and the combinational parts of the design. This usually results in the designer also having to deal with a microassembler to allow the code for the state machine parts of the design to be coded conveniently. A fair analogy is that of software engineers defining the instructions that the compiler can generate code for, rather than having a fixed instruction set with which to deal. The second problem is that the software engineering environment is typically that of a single program in a sequential programming language. Those attempts to apply software engineering techniques to concurrent languages are as yet too embryonic to consider applying to hardware, even if the third problem was not an issue. The third major problem that prevents the direct application of software engineering techniques is that the hardware designer is typically very conscious of both the cost and the performance of the machine being designed. This is not the case with software. Hence, there is no mechanism to allow parameters of the design to be considered in light of performance issues while the design is being done. For these reasons, we abandon a simple translation of software engineering techniques to hardware.

The three most important requirements for a representation of architecture are that it be general, mathematically sound, and well integrated with other tools. By mathematically sound, we mean that the representation does not introduce spurious ambiguity into the semantics of the system being described. By well integrated, we mean that if at some point evaluation must be done by simulation, then the translation into whatever simulation system is being used should not introduce errors or undue delay. To make these ideas more explicit, we consider the manipulations that such a framework should provide the user.
To achieve the goals stated above, we have to provide a representation of the system being
designed, a notation so that the representation can be manipulated, a semantics in order to say what the
system does, and some kind of simulation facility so that those aspects of the system that don't admit
to analytical modeling can be evaluated. In this thesis, we rely on mathematics to provide the seman-
tics and notation. For the representation, we will use the concept of asynchronous processes, and for
simulation, we will use a process-based concurrent programming language containing abstract data
types via objects. Structurally, we say that the representation (communicating processes) is providing
concurrency semantics (i.e., what events and actions occur when) and the semantics of the processes are
specified functionally. Although it isn't yet obvious, this does not violate hierarchy in any way. We
are still able to construct submodels of an architecture that are themselves models in the sense stated
above. These details are addressed in Chapter 2.

1.2. Prior Work

There is a great deal of current literature that is related to this problem, and it comes from many
fields. As expected from the simple description above, we can look to the basic mathematics and nota-
tion of function theory, semantics of programming languages, simulation languages, and hardware
design languages. Since our task is to eventually infer the performance of the system, many of these
approaches aren't directly useful, since they either don't consider performance, or they consider it at
such a level that it isn't useful for comparing low-level effects. It is still useful, however, to briefly
consider how these areas relate to what we will present in Chapter 2.

Theory of computation, as presented in [4], is an example of a theory in which the performance
issues are considered, but are not at a really useful level. Constructs such as parallel Turing Machines
and abstract computation are useful, and have similarities to what we will present, but the lack of a
good treatment of performance prevents them from being the right thing to base a representation on.
Additionally, any abstract computation that doesn't admit an obvious translation into an executable
representation is inadequate for our purposes.
Another way to describe part of the character of a computer is to consider the level of its instruction set. The translation of a program among different levels of language representation naturally arises at this point, and there has been some interesting ad hoc work done in [5] on translating from various levels of representations of programs. These levels are described in terms of directly interpreted languages, or DILs, and directly executed languages, called DELs. Similar work has been done in [6] concerning an operational classification of machine instructions in terms of their function, overhead, etc. The work presented here differs in that we propose a much more rigorous formal basis for the instruction semantics as well as for the control semantics.

Associated with the concepts of DILs and DELs above is the idea of a machine that is a universal host [7]. Here, the design of a machine that is universal is considered. The universality of the machine refers to the property of it being able to execute any function (instruction) and to its ability to do so without a performance penalty, when compared to a machine designed for a specific instruction set. Relative to the work presented here, both the instruction set level work and the universal host machine work are special cases of computer architectures and are, therefore, instances of types of machines that our representation can express. One of the major problems with the universal machine approach is that it tends to force one to either throw away information that is known at compile time, or to be very expensive to build, in order to deal with special cases of information that will be useful at run time.

The field of computer hardware description languages, which is somewhat related to computer-aided design, is also strongly related to the concept of a formal representation for computer architecture. The work in this area ranges from languages that are intended to allow convenient representation and simulation of hardware [8] to formal verification of hardware structures [9]. Typical of these languages are ISP [10] and AHPL [11]. While systems such as these are an essential part of a design methodology, they do not by themselves constitute a solution to the problem of abstraction of principles of hardware design. These languages do not impose any structure on what is designed. The representation that we propose in Chapter 2 does provide some of that structure and, hence, is a useful
addition to the technique of simulation in languages such as these.

1.3. Advantages of a Representation

One of the features of a representation for architecture is the degree to which it is rigorous. Intuitively, this refers to how much the human using the representation has to impose an ad hoc interpretation on the representation to see what the semantics of the target system are. In this thesis we will refer to three kinds of representation, namely rigorous, formal, and ad hoc. Rigorous means that the semantics of the representation are well-defined mathematically, and it implies that the representation is very precise and that there is very little in the way of user interpretation of what the representation means. This style is most prevalent in mathematical foundations. Slightly less rigorous is what we will call a formal representation. A formal representation is one in which the model is well-specified, but in which many approximations to rigor are evident. For example, we will model combinational units in this representation as maps of the form \( f : \mathbb{R}^n \rightarrow \mathbb{R}^m \) where \( \mathbb{R} \) denotes the real numbers. This is not rigorous since any realizable function unit will actually be of the form \( f : \prod_{i=1}^{K} A_i \rightarrow \prod_{j=1}^{L} B_j \), where \( \forall \), \( A_i \) and \( B_j \) are finite contiguous sets of integers. Ad hoc implies that even more rigor is discarded, and essentially all of the burden of validity of the representation is on the shoulders of the user of the model. Parts of the translation of the representation into a simulation language are ad hoc. The representation itself is formal. This is in contrast to most representations, which are either formal but ignore performance [12], or are completely ad hoc, such as ISP [10].

In addition to these abstract approaches, machines have been built that exhibit differing levels of instruction set. Patterson [13] has developed a fairly low-level machine called the RISC-I, which he contrasts with available commercial machines such as the VAX-11/780 [14]. It is fair to say that the VAX has a higher-level instruction set than does RISC-I. Other machines have been built [15,16] which implement unconventional levels of interpretation either at the user program level, or at the operating system level. Even though there are real machines of this type in existence, performance evaluation of the architecture, as distinct from the implementation, remains a difficult problem. In particular, it is
not known whether one level of instruction set is significantly superior to any other for non-numeric processing. There is no known useful characterization of the kind of program behavior that is most well-suited to a given level of interpretation. Further, program behavior in the case of non-numeric programs has not been characterized relative to differing levels of interpretation.

Even though most current systems are designed in an essentially ad hoc manner, there are advantages to attempting to formalize the design process. This is primarily evident in considering the role of abstraction in machine design. If something that works on one system can be abstracted to a general technique, then that abstract principle can be applied to other designs. This is one of the advantages having a formal representation.

An example of an abstract principle is that of early binding, which refers to the compile time binding of values when that is possible, to avoid recomputation at run time. This concept has been applied at run time in [17], and will be discussed in Chapter 5. Recognizing that this principle has application in a setting other than that in which it was originally proposed is facilitated by the abstract representation of the system in that it makes binding issues more obvious.

Related to the binding issues just mentioned is the degree to which system performance is dependent on the data that is input. This is another area of architecture design that is aided by the representation proposed here. The concept of performance being data dependent can be illustrated by considering a matrix multiply program on one hand, and a matrix addition program on the other. Due to the nature of the primitives involved, the addition program can be expected to have performance that is independent of the values in the matrices involved, since implementations of addition typically execute in fixed time independent of the input operands. If the multiplication is carried out using, say, Booth's algorithm, then the multiplication steps are data dependent. Another example of performance being data dependent is pattern matching. This is an area where our representation makes the dependence more obvious.

We have stated some of the advantages of a formal representation. These are primarily concerned with facilitating abstraction compared to an ad hoc approach. We will now conclude this chapter with
a brief description of the structure of the rest of the thesis.

1.4. Outline of Subsequent Chapters

In Chapter 2 we will present the representation beginning with the mathematical foundations that will be used to define the semantics of the basic computational unit, called a basic process. This constitutes the formal definition of the representation and includes a description of an object-oriented language that was also developed for use in simulation of systems specified by the representation. This is followed in Chapter 3 by a specialization of the representation to those machines that are instruction set interpreters, and static, adaptive, and dynamic interpreters are defined. Chapter 4 describes some of the more interesting tools that were developed to support our design methodology. In Chapter 5 the model is applied to the study of preemption effects in write-through and write-back cache. This study contains a detailed simulation, at bus level, of the target system, and is an example where the representation gives only relatively little information about the details for which cache policy is superior for a given environment. Chapter 6 then discusses an adaptive interpreter as an implementation of a complex instruction set machine. Here the representation is more useful and indicates some useful directions that are verified as profitable via detailed simulation. Chapter 7 considers data dependence and exploitation of higher-level program behavior that does not admit to a simple static characterization of performance data dependence. This chapter also considers binding time. Chapter 8 states conclusions and possible future directions.
2. A MODEL OF MACHINE BEHAVIOR

In the field of performance evaluation of computer systems, simulation is often the most powerful tool available [18]. This tool is usually applied in an ad hoc manner, in the sense that a different simulation is constructed to consider each new system. As a result, a given simulation tends to have only a few parameters, and these are typically performance parameters, such as memory speed, rather than parameters that affect structure, such as cache replacement policy [19, 20, 21]. In addition, simulation is inadequate to deal with problems that are just too expensive, in terms of run time, to simulate in a detailed fashion. Alternatives exist which attempt to impose structure on design and evaluation [22], but these techniques are not general, and in the case of [20], are applicable only to a complete realization of the system.

2.1. The Need for a Model of Architecture

This situation would be improved if one were not forced to rely on engineering judgment and simulation to such a great degree. What is needed are ways to inject generality and objectivity into the design/evaluation cycle and to relate the sensitivity of the system performance to both structural and performance parameters.

This can be achieved by using a model for architectures in addition to simulation. The technique of proposing a model for a system, or a class of systems, doesn't give any more information about it than a detailed simulation, but it does organize the information in a way that will hopefully make inference about the behavior of the system easier. For example, a Petri net description of the synchronization properties of a collection of processes can allow one to infer whether deadlock is possible. Similarly, Karnaugh maps merely organize information in a way that allows inference, in this case, regarding function minimization, to more easily be made.

This is not to say that simulation is never necessary for performance evaluation. One of the aims of a model of computer architectures is to allow the user to determine in a structured fashion that part
of the system that has to be simulated in a detailed fashion. It should also make obvious other features of the proposed architecture that impact performance, without simulation. In order to do this, there are three main areas in which the model should excel.

**Representation**
There should be a way to represent any architecture in a way that is easily controlled, in terms of level of consideration or detail, by the user.

**Classification**
It should be possible to use this representation and model to precisely separate proposed architectures into classes that are consistent with what is currently known about these classes.

**Inference**
The model should allow the user to recognize and isolate features of different architectures that impact performance without having to simulate the detailed case.

In this chapter we will develop a model for machine behavior that satisfies these requirements. As indicated in Chapter 1, we have a specific application for this model, namely, the investigation of different levels of instruction set. Since the two are almost inseparable, we are also interested in what kinds of primitive operations are available. The primitives are still at the register transfer level, but given higher-level instructions, we expect that it might be possible to utilize multiple function units, or multiple data paths, to effect more powerful primitives. It would be useful if we could easily decide what kind of performance tradeoffs are involved in specific instances. This is strongly related to detecting where performance limitations, so-called bottlenecks, occur. Knowing this, we can propose alternative structures which address the problem of how to balance the system so that the performance is not dominated by a single subsystem.

### 2.2. A Formal Model of Computer Architecture

The model we present here is intuitively based on the idea that any activity can be described as a collection of communicating processes which are allowed to operate asynchronously. Accepting this premise, we can consider various characteristics of these collections of processes in the abstract. Examples of interesting attributes of such collections are the patterns of data flow from process to process, the idle time for a given process in the network, and the critical paths (bottlenecks) through the net-
work. In order to get to this point, however, we must first capture the crucial aspects of a single process. This must include items such as which other processes are signaled by a given process, how long it takes for that process to compute its result, when it begins operation, etc. Following this very intuitive line of reasoning, we will specify a process in terms of when it is scheduled, when all its inputs are ready, how to compute the result of the process, how long it takes, and finally, what events are signaled by the process on completion.

Although the motivation for all of this is clear, defining and working with a mathematical model can be very tedious. The reason is that the basic mathematical concept here, the function, has no explicit association with physical concepts, such as time. This means that in order to talk meaningfully of the performance or even the low-level behavior of a system, we have to impose a semantics on the mathematical model to relate it to what we normally call events in the real world. Doing this can be tedious, but the result is a precise notation, without which manipulation of the model would be impossible.

The model is developed in three stages. First, an abstract process is defined which is moderately general, but only hints at timing and event issues. This first step is then augmented by the definition of some auxiliary functions (derived from the abstract definition) that characterize the idea of an abstract process being run in four phases. These phases are being scheduled, awaiting inputs, computing a value, and communicating that result to other processes. At this point the process is still very abstract in the sense that there is no statement of how one would realize it. Further, the interpretation of the process beginning execution at some known time is semantics that the user is applying to the model. This isn't a weakness of the technique, but a statement of where the user has to expend effort in using the model.

The final part of the development of the model is to connect the processes so that they can communicate and synchronize in a meaningful way. In keeping with the development to this point, this is another specialization of the abstract process, since it begins to say how a given single abstract process could be realized as an interconnected collection of abstract processes. We will continue in this direction from abstract to concrete in this chapter and the next, finally relating all of this to some real
machine structures. We turn now to the formal development of this model, beginning with an explanation of the symbols that comprise the syntax of the notation we use to define and manipulate the representation.

The syntax of the notation that we use to represent processes relies on standard symbol usage from the mathematical theory of functions. The symbols we use will be defined now. The reals and the integers will be denoted by the usual $\mathbb{R}$ and $\mathbb{Z}$, respectively. Their finite Cartesian products will be denoted $\mathbb{R}^n$ or $\mathbb{Z}^n$. We have occasion to form product spaces denoted $\prod_{i \in I} A_i$, which is the Cartesian product of some collection $\{A_i\}_{i \in I}$ where $\forall i \in I. A_i$ is a set. We also use the $\prod$ symbol to indicate the projection function, which maps a vector onto one of its coordinate spaces. This is written $\prod_{i \in I} A_i \rightarrow A_n$ which is defined as taking a vector $X \in \prod_{i \in I} A_i$ to the value in its $n^{th}$ coordinate. We have just used the notation for defining a map, which is generally of the form $f : A \rightarrow B : a \rightarrow f(a)$ and is read "$f$ maps $A$ into $B$ via $a$ going to $f(a)$." The symbol $\cdot$ in general means multiply. When it occurs between two finite dimensional vectors, it indicates the usual vector dot product, and in context, the scalar product. The symbols for set membership, subset and composition are as usual. Other symbols will be defined as needed. We can now define a basic process.

Let $E = \prod_{i \in I} \{0,1\}$ where $I$ is some index set. This is the event space, and its value is the current state of all events. We define a basic process as a five-tuple $\{s,b,e,r,k\}$ where
where the support of \( s \) is compact and contains no isolated points, \( J \) is a proper subset of the index set, \( I \), and in the definition of \( e \), \( n \) and \( m \) differ for different \( e \)’s but are fixed for a given \( e \). To say that the support of \( s \) is compact means that \( s \) has value one only on closed bounded subsets of \( R \). Intuitively, we want to say that if \( s \) becomes one at some time, then it stays one for some non-zero length of time. Hence, we also exclude \( s \) being one on some isolated set of points. We refer to these functions as the scheduling function, the binding function, the execution function, the run time function, and the communication function. If \( \alpha = \{ s, b, e, r, k \} \) is a process, we will denote \( e \) as \( e_\alpha \) and, similarly, for \( s, b, r, \) and \( k \). Informally, the execution of a process consists of each of these functions “happening” in sequence.

As we have already remarked, the user of such a model has the task of relating it to the real world. To facilitate this connection, we define auxiliary functions which can be derived from the five above. In the following definitions of the derived functions, the first factor of the domain can be thought of as time.
\[ b':E \rightarrow \{0,1\}: (t,e) \rightarrow s(t)b(e) \]

\[ e':\mathbb{R} \times \mathbb{R}^m \rightarrow \mathbb{R}^m: (t,X) \rightarrow b(s(t))e(X) \]

\[ k':\mathbb{R} \times \mathbb{R}^m \rightarrow \prod_{i \in J \subset I} \{0,1\}: (t,X) \rightarrow r(t)k(X) \]

The idea is that the derived functions are zero if the part of the process to which they pertain is inactive. All of this machinery is solely for the purpose of making the performance and timing of the process explicit.

To summarize to this point, we model entities called basic processes as requiring one setup phase called binding, and an execution phase which occurs after all inputs are ready and which takes some non-zero amount of time to complete. At the completion of the computation of the result, the processes are signaled. In Figure 1, this is illustrated by a timing diagram which shows when each phase of the abstract process is active.

![Timing Diagram](image_url)

Figure 1. Timing of a basic process execution.
Now that we have some machinery in place, we can begin to manipulate the model. The most fundamental manipulation we can do to a process is to get a result from it. This is defined by the map \textit{eval}, which when given a basic process and an appropriate input, yields a result. This can be formally stated after we define an appropriate input. Let \( P = \{ p : I \rightarrow R \times R^n \} \) where \( \prod_{i=1}^{n} p(t) \) is a monotone increasing continuous function from \( I \rightarrow R \). Any \( p \in P \) will be called an input path, and it gives enough information to say what a process does. The function \( \prod_{i=1}^{n} p \) models time, and that part of the path will be used by the derived functions and, hence, will determine when the various phases of the process are active. The other part of the map which relates \( I \) to \( R^n \) provides the input values to the execution function, \( e \). Hence, we can say
\[
\text{eval}: P \times A \rightarrow (I \rightarrow R^m): (p, \alpha) \rightarrow e_{\alpha}(p(t))
\]
where \( A \) is some set of abstract processes.

2.3. Interconnection

To this point we have been considering only a single basic process. We now want to model networks of processes. We define two ways of composing a pair of basic processes into another process. A pair of processes, \( \alpha \) and \( \beta \), are denoted \( \alpha \circ_k \beta \), for \( k \)-composition and \( \alpha \circ_p \beta \) for \( p \)-composition. The first corresponds to letting the output from one process be the input to the next. Further, the control for the second process is provided by the \( k \) function of the first. The second method, the \( p \)-composition, differs from the first only in that the scheduling of the second process in the chain is controlled by a third process, which has as its only task the control of \( p \)-composed processes. This second method reflects the common case of a controller and a data path.

More formally, we say that
\[
\alpha \circ_p \beta
\]
if there exists \( \gamma \) such that \( s_{\gamma} \) is a function of \( k_{\alpha} \) and \( s_{\beta} \) is a function of \( k_{\gamma} \). The \( k \)-composition is the
trivial p-composition, i.e., $s_\gamma = k_\alpha$ and $s_\beta = k_\gamma$. In both cases the result of combining these processes will itself be a process. The construction of a process in this fashion is a result of composing the execution function in the mathematical sense. Hence, we will refer to the interconnection of two processes as the composition of those two processes.

2.4. Graph Notation

In any abstract representation of a process that actually reflects the behavior of a real system, there is a graph structure due to the control and data flow dependency in the network. Often the data and control flow graphs are sufficient to represent the structure of a system that is modeled in this fashion. Since the nodes of these two graphs are the processes, it is natural to draw two graphs with the nodes being the processes and the links indicating either the data flow or the control dependency. We will represent systems in this way, with the data flow being indicated by solid arcs, and the control dependency by dashed arcs. An example is shown in Figure 2.

A representation such as this can be simulated either deterministically or probabilistically. We will discuss these options in Chapter 3. There is also another means of considering the dynamic state of a system described in this fashion. We can define an abstract measure of the degree to which the system is staying in a given mode of operation based on its behavior in terms of which of the basic processes is active at a given time. This is a situation that is very similar to characterizing a linear system by its frequency response. One can take that approach here by considering the states of the model to be values of some discrete signal, and then we can apply Fourier analysis to this signal. If the model is cyclic, in the sense that some state is returned to infinitely often, then there is also some periodic structure that will be evident. In this case, however, there is no linear structure to appeal to; hence, it is not useful to mimic exactly the approach taken there. Instead, we will consider a simple averaging method that will allow us to quantify the locality of the model and the rate at which it is changing.

To do this, we define the abstract state of a process. In the case of a single process, such as one of the constituent processes in Figure 2, we merely look at some window of the state trace and compute a
moving average. This allows any locality of the state graph to be detected independently of the state assignment. Hence, while we may not be able to infer anything from nonconstant regions of the graph of this moving average, regions of constant average state can be related to areas in a program that can be exploited either by adapting to local behavior or, in some cases, predicting at compile time when
these events of constant behavior will occur.

The algorithm for forming this function is given now, in the context of having already defined a process network that we wish to analyze.

1. To each basic process in the system, assign a distinct integer, denoted \( w_i \).

2. For each basic process \( \alpha \), form the activity function \( a_i(t) = w_i \cdot (s_i(t) \cdot (1-r_i(t))) \).

3. Define the function \( \Lambda(t) = \prod_{i=1}^{n} a_i(t) \). Note that \( \Lambda(t) \) is piecewise constant.

4. Define the moving average state, \( A \) with window \( w \), as \( A_w(t) = \int_{x=-\infty}^{x=t} \Lambda(x) dx \). Since \( \Lambda \) is piecewise constant, there is no problem forming the integral in 5; hence, \( A_w(t) \) is well-defined.

Some remarks are in order. First, note that the function \( A_w(t) \) has the property that if the system being analyzed has no overlap and is periodic, \( A_w(t) \) is constant. Further, if the \( w_i \) are chosen to be distinct primes, then the converse holds, and distinct constant values of \( A_w(t) \) correspond to distinct cycles of the system. We will use this method to compare two programs in Chapter 7.

The formalism that we have developed so far allows us to model very accurately many concurrent systems. We have also defined some characteristics of these processes that ease the task of evaluating the performance of these systems. From a pragmatic standpoint, it is important that the connection between these models and the implementation be natural.

To aid in this connection and to fully utilize the design style that we are advocating here, one has to at some point evaluate some realization of the system being designed. This is typically done by simulating the system at a very detailed level, although that need not be the case. Even the computation of the abstract state as described above is done in this fashion. In the following chapter we will discuss instruction set interpreters, which are a specialization of the formalism given here. In Chapter 4 the tools that are required to carry out the low-level evaluation will be presented.
3. INSTRUCTION SET INTERPRETERS

In this chapter we will specialize the representation from the last chapter to consider instruction set interpreters. This will allow the representation to be applied more easily to some common problems of machine design, such as how the level of an instruction set affects the performance of a system. We will introduce some graphical notations to describe interpreters and we relate some of the common machine structures, such as overlap and pipelining, to the form they take when represented using this technique.

3.1. Definition of an Interpreter

One way to think of a computer system in operation is the language and translator model. A user program is translated from some high-level language into loader format by the compiler. This representation is then translated into machine instructions by the loader. Eventually, microinstructions are executed which implement the machine instructions that form the so-called executable program. This last translation, from machine instructions to microinstructions, occurs at run time, each time the program is run. This fact distinguishes this translation phase from the others, and is the salient feature of those translators that are interpreted rather than being directly executed. In general, we say an interpreter translates an input token stream (the instructions) into some output token stream (the microinstructions) based on the current state of the interpreter. This current state can be large and complicated, as in a LISP interpreter, or it can be very simple, as in a machine instruction interpreter, such as the VAX-11/780. In this latter case the only context available is the user registers.

We are primarily concerned with interpreters that translate and execute what we commonly think of as machine instructions. To usefully represent these objects, we have to deal with performance and functionality issues at the level of an implementation of the interpreter. Hence, common features of conventional interpreters, such as the prefetch of instruction tokens, should be accurately modeled. Since it is so common to implement conventional machines as a data path and a controller, that structure will play a central role in our representation.
The formal properties of such a model are particularly interesting, if one wants to maintain a firm connection with the implementation of the system. One obvious path is to mimic the denotational semantics approach, where each object (in our case instruction set interpreter) is a member of some domain [12]. This results in useful semantics, but doesn't deal at all with the implementation structure or the performance of the interpreter. What we have chosen instead is motivated by the observation above that most instruction set interpreters are naturally realized as a controller and a data path. We will use the concept of p-composition of processes, from the last chapter, to model this structure.

Recall from the last chapter that the p-composition of processes C and D was a composite process where for all \( \alpha \in D, s_{\alpha}, b_{\alpha}, \) and \( k_{\alpha} \) were functions only of elements of C. That is, the data path part, D, synchronizes and communicates only with the control part, C, rather than with subprocesses of itself. If we make the further restriction that for all \( \alpha \in D, \lim \infty(b_{\alpha} \circ s_{\alpha})^{-1}[1] - \lim \infty(r_{\alpha}[1]) \in R. \)

This merely states that the controller assumes that there is a fixed bound on the amount of time required to carry out the execution phase of any process in D. This corresponds to clocking a function at its worst-case delay time, rather than trying to shorten the cycle for inputs that are known to be processed faster.

3.2. Instruction Set Level

The statement above, that any function that the data path executes has a fixed bound on its execution time, is closely connected to the concept of the level of an instruction set. The instruction set for a general purpose computer can be described in terms of attributes such as number of registers available, addressing modes, and the degree of support for high-level language constructs such as structure reference and complex expression evaluation. Considering all of these together, we can talk informally of the level of the instruction set, which is roughly the complexity of those operations that are directly supported as instructions. As we have stated previously, our performance model is an execution of a higher-level language code; hence, this instruction set need not be "humanly usable," and ergonomics
can be sacrificed in the interest of performance and cost.

Machines to support high-level languages can be built with instruction sets of different degrees of complexity. Systems that have been implemented range from efforts such as the SYMBOL computer [15], which directly supports many high-level language and operating system functions, to very low-level microprocessors such as the RISC [23] machine, which requires the compiler to explicitly generate codes for functions and addressing modes that are built in to higher-level machines. Intuitively, one would suspect that the former approach incurs less overhead and results in higher performance, but this is not obvious, since there are aspects of simplified instruction sets that allow them to be more efficiently handled.

Closely associated with the idea of the level of an instruction set is the set of resources available to the code generator of the compiler. This leads to taxonomies such as stack machine, register-based machine, and vector machine. This situation can be explained in general by considering that the machine is capable of directly interpreting some kind of language, called its instruction set. This language, called a DIL for directly interpreted language, is interpreted by a series of directly Executable language tokens, or DEL tokens. Given that the higher-level language is expressed in a language different from the DIL or DEL for the machine, there must be some translation from the higher-level language code to the DIL. There may, in fact, be several intermediate DILs and some translations may be done at compile time, while others are done at run time. This model for execution was proposed by Hoevel and Flynn [24]. A concrete example of this kind of translation is shown in Figure 3.

In Figure 3, the translation is from a fragment of C code into machine instructions, and then into microinstructions. The DIL2 indicated between the movl and the microcode is an additional phase of translation that is carried out at run time by the VAX-11/780 implementation of the VAX instruction set architecture. The translation from C code to DIL1 is carried out at compile time.

The meaning of the word "direct" in DIL and DEL is illustrated in the translations of Figure 3. The original C program is a simple assignment statement and the act of doing the assignment is translated into a single machine instruction. However, the information as to how to compute the
int \ x;

register int y;

y = x;

movl 4(fp), r10

fp + 4 \rightarrow\ latch
read \rightarrow r10

Figure 3. Translation of C to VAX code.

locations of the variables that are being assigned is contained in the declarations of \( x \) and \( y \). The information as to where these variables are, i.e., four bytes past wherever the frame pointer points, and register ten, is explicit in the instruction labeled DIL1. This token is directly Executable since it doesn’t depend on other instruction for information as to how it should be carried out. Another implication of the word direct is that if there is no dependence on other instructions, and we can hope that, given
enough hardware, the instruction can be executed in a single minor cycle in the case of the lowest-level DEL. At the next level we have the interpreter which issues directly Executable language tokens to effect the semantic actions of the DIL. These tokens are the primitives of the machine and are typically executed by asserting values on busses and passing them through function units. This level is interpretive because the current state of the machine can affect the actual directly Executable language tokens that are issued in response to a given directly interpreted language token. It is also possible to make multiple translations into several successively lower-level DILs rather than to a single directly interpreted language as described here. To relate these levels to real situations, consider the subroutine of Figure 4(a), which is a function written in C that returns the product of five and its argument.

The compiled version of this program, expressed in the directly interpreted language of the VAX, is shown in Figure 4(b). The next phases of interpretation differ significantly for different implementations of the VAX instruction set architecture. The VAX-11/780 implementation of the VAX has a write-through cache and a prefetch buffer, which is used not only to prefetch instructions, but also to decode them. Hence, for this particular implementation, the execution of the assignment statement in Figure 4(a) involves a translation from the form shown in Figure 4(b) to a lower form in which the opcode is replaced by the address of the first microinstruction used to interpret this instruction. This constitutes a true translation into a lower-level directly interpreted language since the prefetch hardware is responsible for making this substitution and since the processor cannot interpret the instruction until this translation is complete.

We now turn to a more concrete model of the actions of a given level of DIL. This model is based on the parse trees which are common in compiler theory. Consider the C assignment statement,

\[ a := b \times c; \]

This assignment can be realized with the instruction

\[ \text{mulfl3} -4(fp),-8(fp),-12(fp) \]

The parse tree for this expression, as a statement in C, is shown in Figure 5. We can also parse the instruction that realizes this assignment relative to the usual DIL for the VAX. This tree is shown in
\begin{verbatim}
f1(k)  
int k;  
{  
    int j;  
    
    j = 5 * k;  
    return(k);  
}
\end{verbatim}

(a)  
Function returning five times its argument.

.globl f1
subl2 $4, sp
f1:  
    subl2 $4, sp  # space for the local j
    mull3 $4(ap),$5,-4(fp)
    movl -4(fp), r0
    ret

(b)  
Generated macro code.

Figure 4. DIL for VAX-11.

Figure 6. These trees express a way to execute the statement in the sense that if the node operations are carried out in the order they are seen in a post-order traversal, then the statement is correctly executed. As soon as we can construct a parse tree for a given level DIL, we can apply to this structure certain optimizations that are common from the field of code generation in compilers. In Chapter 6 we consider in detail the case of common subexpression elimination relative to reducing recomputation of frame relative references in programs such as these.
This rather intuitive discussion can be considered more formally by introducing the concept of the order of an instruction set interpreter. We want to express those features of an implementation that correspond to the level of its instruction set in a formal fashion. In this way, we can abstract principles that can be exploited in hardware to improve performance. To motivate this, consider the instruction

```
addl2 -4(fp),(R1)+
```

On most machines we consider this a complex, or high-level instruction, since it implements the statement

```
*p = *p + x;

p = p + 1;
```
where \( x \) is an integer local and \( p \) is a pointer to an integer array. We can analyze this instruction further by examining the parse tree for the instruction, as in Figure 7.

A conventional computer would carry out this instruction by issuing a microinstruction for each of the internal nodes in the tree. For such a machine, we typically have about six minor cycles, assuming there is no additional delay due to memory latency.
Figure 7. Parse tree for addl2 -4(fp), (R1)+.
An alternative would be to more directly implement the algorithm for this instruction in hardware. Presumably, that would result in faster execution, since we could, in principle, execute sub-trees A and B together, then C, then D, and finally E and F, for a total of four cycles. We say that this instruction is a higher-level instruction in the first scenario, since it is further from being directly executed. This lets us informally relate the performance of the machine to the level of the instruction set and we say the instruction is a lower-level instruction on the latter hardware.

In relating the idea of complexity and level to instruction interpretation and performance, there are two interesting parameters. These are the number of subgraphs of the DAG representation that are active at any time, and the point at which the interpreter executes primitives, as opposed to compound actions. We say that a serial instruction interpreter is a pair of processes \((C,D)\) where

\[
\begin{align*}
    b_d(t) &= k_c(t) \\
    e_d(t) &= f(e_c(t)) \\
    e_c(t) &= g(i,k_d(t)) \\
    s_c &= 1 \cap s_d = \emptyset
\end{align*}
\]

The last condition says that the control and data paths are not concurrently scheduled.

The utility of this representation is that it allows us to consider precisely the effect of instruction set level on performance. In Chapter 6 we will return to the issue of instruction set level, and show those cases in which complex instruction sets can be exploited for a performance improvement. We now move from consideration of the interpretation of a single instruction to networks of processes arranged as an instruction set interpreter.

### 3.3. Practical Interpreters

There are many detailed characteristics of real instruction set interpreters that we haven’t discussed yet. For example, how do common structures such as loop buffers, cache, and prefetch buffers fit into this representation? We now want to examine some abstract attributes of instruction set interpreters and relate this to these kinds of concepts. Further, we want to do this in a setting that is
representative of what one would consider an interpreter for more than a single instruction.

In formally considering instruction set interpreters, we are faced with the problem of making the representation sufficiently practical that real implementations are possible, and at the same time retain the generality that a formalism should provide. In representing conventional machines, there is no problem, since there are typically a finite number of fixed purpose resources that are connected in a fixed communication and control network. However, these resources are typically reused if iteration is part of an instruction's interpretation, and a given resource, such as an ALU, is typically programmed by a control field for a given function on a per minor cycle basis. From a mathematical point of view, changing any aspect of a network of such resources makes it a different network. We are thus faced with the user of such a model being responsible for making the final connection between the model and reality.

In considering the conventional machine alluded to above, one would probably accept that the algorithm to interpret a given instruction, as typically realized in microcode, is fixed. If the algorithm can take one of two different paths in execution, we would like to characterize the system as somehow being dynamic in terms of different algorithms being used for a given instruction. To capture this difference, we define two classes of interpreters. These are static and dynamic. Intuitively, a static interpreter corresponds to the conventional setting described above, in that regardless of context, when a given instruction is encountered, it is executed by the same algorithm. A dynamic interpreter differs in that the context can dictate that for the same instruction, a different algorithm, with potentially different performance characteristics, can be used in place of the original, and that decision is made at run time. Formally, we define these as follows. Given an interpreter $(C,D)$ and an instruction $i$, we say that $(C,D)$ is a dynamic interpreter relative to $i$ if $ord(C,D) / D \neq i$, is not constant. Static is analogous. The idea of dynamic interpreter can be implemented via residual control in microprogramming. It is also present, at least in principle, in cache memories.

We are now ready to develop some practical interpreters. Our goals are to be able to represent a program in an architecture independent way and then to derive different architectures from that
representation. In particular, we will consider the difference between a stack machine and a register machine. This process of generating an architecture from the program representation is a kind of abstract computation model, and the architectures so derived are expressed by the now familiar abstract process formalism.

Since all of our effort here is aimed at executing higher-level language, it is natural to begin with a representation for programs that is compatible with compiled code. A likely candidate for this representation is found in compiler theory [25] in the form of directed acyclic graphs, or DAGS. The more usual form of DAG is simply a parse tree. In compilers, these are used to represent the evaluation of basic blocks, which by definition have no embedded control structure. We have need of control structures, however, so we will extend the DAG representation to include control. This extension can also be used in compilers, when it is desirable to represent control flow in a machine independent fashion.

The expression part of this graph representation has nodes which generally have one parent and some fixed number of children nodes. An interior node is nonatomic, i.e., nodes with children are operators whose children are their arguments. This restricts us in two ways. Functions are viewed as having only one return value, and they can't have a variable number of arguments. These features are useful and exist in many variants of LISP [26], but the extension to include them doesn't alter the structure of what we'll do here. Specifically we will only consider the programming model defined by C [27]. This means we can have variables that are global, or local, and that there is no true block structure in terms of up-level addressing.

The expression defined by the tree in Figure 6 can also be interpreted by a stack machine. The meaning of the claim that this parse tree naturally represents code for a stack machine is that we can generate code for the expression by a simple post-order traversal of the tree. The fragment in Figure 8 will generate code to evaluate the expression of Figure 7. In Figure 9, the generated code is shown.

If we now extend the form of these parse trees by adding a control node, we can represent programs in a meaningful way. The extension consists of a conditional node, which has one argument
; Walk the expression tree L emitting code to evaluate it.
(defun gen (L)
  (if (left L) ; See if left child is there
      (gen (left L)))
  (if (right L)
      (gen (right L)))
  (selectq (nodetype (car L))
    (LOCAL (emit "pushlocal" (nodename (car L)))))
    (GLOBAL (emit "pushglobal" (nodename (car L)))))
    (OPERATOR (emit (nodename (car L)))))

Figure 8. Parser to emit code from an expression tree.

            literal   -4  ; pushes a constant
            push     fp   ; the frame pointer
            add      ; compute the address
            load     ; load from that address
            literal  -8
            push     fp   ; as before
            add      ; form the product
            load     ; address computation
            mult     -12
            literal  ; for the
            push     fp   ; result
            add      ; store

Figure 9. Generated stack code.

that can be either true or false. Corresponding to these two possible outcomes, there are two more links
from the conditional node. Hence, to represent the conditional expression in Figure 10, we have the
graph in Figure 11.

    if (a > 38 + (5 * x))
    f(a);
else
    x = 3 + (6 * y);

Figure 10. A conditional.
We can now generate code for a stack machine from the parse graph of the program. This can be taken one step further by associating a basic process with each nonconditional node in the graph. This allows us to express the generation of a network of processes, which is a behavioral representation of a stack machine, in the form of an abstract computation machine that instantiates processes freely whenever they are needed to reflect the behavior of nodes in the parse tree. Finally, we can allow these
processes to be reused if they correspond to operators, and discarded after use if they are atomic (i.e. locals, globals, and constants). This structure is a stack machine architecture in the sense that we can model the behavior and performance of a stack machine at a very detailed level by assuming appropriate values for the constituent functions in the basic processes. The generation algorithm, which takes as input a parse tree, is carried out in a post-order traversal of the tree, and at each node, we construct a process satisfying

\[ s(t) = \Lambda \{ \alpha | \alpha is a child of this process \} \]

\[ b(t) = 1 \]

\[ e(t) = \begin{cases} x(t) & \text{if this is a leaf} \\ op(e_\alpha, e_\gamma) & \text{otherwise} \end{cases} \]

where \( \alpha, \gamma \) are child processes.

\[ r(t) = \begin{cases} 0 & \text{if instantiated} \\ \tau & \text{if previously uninstantiated} \end{cases} \]

\[ k(t) = U(r(t)) \]

If we carry this construction out for some parse trees, then we get an abstract process which computes the correct result, and which has performance and timing characteristics determined by \( r(t) \) and by the characteristics of the tree. This then represents one execution of the parse tree. We can extend this to reflect the more realistic situation in which such expressions are executed multiple times.

In the algorithm above, we freely create processes when needed as the tree is traversed. This is convenient mathematically or even as a means of specifying a network of processes that affect a given algorithm. It is not sufficient, however, for guiding the implementation of an instruction set interpreter. Hence, we make the following modification to the instantiation phase. At any node, before we instantiate a process to effect that node, we examine the set of already constructed nodes, and if any of them are currently unused instantiations of the process we need, we use them. Now we can show an
example of the computation above being carried out by this algorithm, but need slightly more notation. We’ll indicate that the processes \( s, \beta, e, r, k \) and \( s, \beta, e, r, k \) are \( k \)-composed by an arrow from one to the other. For example,

\[
\begin{array}{c}
\text{CA} \\
\overrightarrow{\text{CB}} \\
\overrightarrow{\text{CC}} \\
\overrightarrow{\text{CZ}}
\end{array}
\]

indicates that \( e_z = \rho_z(e_A, e_B, e_C) \) and \( s_z(t) = s_A(t) \Lambda s_B(t) \Lambda s_C(t) \). In prose, this is nothing more than the statement that \( s, \beta, e, r, k \) executes only after all of its argument processes are ready, and its inputs are exactly the outputs of those argument processes. Stepping through the algorithm above, we get the succession of states in Figure 12. The states we indicate are obtained if we assume that at instantiation time we will retain operator nodes, but we discard the atomic nodes corresponding to the local and global variables as soon as we’ve used them. To model the effect of reuse of some kinds of nodes and the immediate discard of others, we introduce the concept of lifetime of a process. The other two parameters that we have at our disposal are the order of instruction (subtree) we’re willing to retain and the number of resources, in terms of processes, that we are willing to keep around. In the next section we will consider the effect of varying some of these, so that the algorithm used here generates a register machine for the same parse tree.

3.4. Comparing Interpreters

The stack machine of the last section was generated by letting lifetimes on the atoms be zero and the lifetimes on operators be infinite. Now consider what happens if we let the lifetimes on the atoms be infinite, just as the operator node lifetimes are. If we do this, we get the succession process networks
in Figure 13. This network represents a conventional register machine just as the previous figure is a stack machine.

We alluded previously to conventional system taxonomies which use attributes such as number of registers, or number of operands in an instruction. In our method of representing systems, there is no direct analog to these classifications except via explicit implementation, since objects such as registers are viewed as resources that are manipulated by the instruction set interpreter and are, therefore, more a data structure associated with the algorithm that implements the instruction set.
Figure 13. States of the abstract register machine.

The kind of interpreter used to implement an instruction set, whether static or dynamic, is orthogonal to issues such as number of registers or whether evaluation occurs on a stack, since we can implement any of these architectures with a static or dynamic interpreter. This is not to say that we can't compare these other systems. Rather, the canonical way to do so is to begin with the higher-level language representation of the target system and examine the overhead incurred in interpreting the higher-level language level into the primitives of the machine. Using the abstract process model for this task is appropriate, but it is one more point at which the user has the burden of correctness in terms of making the model reflect reality. In some cases the association is obvious, as in the following examples.

To relate the process model to conventional structures, consider a pipelined data path. We have two options for describing this structure, namely, via a k-composition or a p-composition of basic processes. These two styles correspond to two common means of pipeline construction [28]. In the k-composition form, a given segment of the pipe is communicating the fact that an operation is done
directly to the following stage. The control for such a pipe is distributed in the sense that a given segment relays data and control only to the next stage. An alternative is to have centralized control by one process which organizes each segment so that it is functionally equivalent to the k-composition model. Practically, the first form is simple to implement but less flexible, since it can exhibit only one mode of operation, namely pipelined. In the p-composition case, there is more flexibility since the system can tolerate more variation in the segment times. A good example of this type of pipeline, more correctly called overlap, is seen in prefetch units. Prefetch buffers have constraints that preclude their being considered ridged pipe machines. The prefetch buffer is typically fetching instructions from the same place that data is fetched from, and here it must synchronize on this access. The prefetch buffer is required to synchronize with the instruction stream only when the instruction stream branches. Hence, it is often the case that the prefetch buffer is idle waiting for either an instruction stream flush, due to a jump, or a refill due to having been emptied below some threshold. For these reasons, it is inappropriate to implement this structure as a k-composition of processes. The p-composition of such a buffer is seen in Figure 14, using the graphical notation presented in Chapter 2. A graphical representation of a ridged pipe is given in Figure 15. The main difference is that the p-composition in Figure 14 is actually controlled by the process labeled Resource Control.

3.5. Performance

One of the goals of a representation such as this is that it offers insight as to potential problems with the structure of a system without requiring a detailed simulation. The model in Figure 14 can provide some insight as to the performance of the prefetch buffer it represents. For example, the resource control section implements a critical section. We should, therefore, be concerned about the control dependency indicated by the dashed arrow at node A1. These arrows indicate a way in which E3, the phase of execution which requires the data, can be caused to wait for A1 either for access to the buffer, or for the buffer not being empty via C1. From the point of view of the execution unit, delay due to the buffer can occur when trying to get the op-code, at E4, or the instruction operand at E2. We can continue in this ad hoc fashion and get a good feel for the operation of the system.
Figure 14. Network model of a prefetch buffer.
Figure 15. Network model of a ridged pipeline.
In the case of a more complicated system, it is more difficult to notice in detail what parts of the system are problems in terms of performance. It is at this point that one has to resort to simulation, and there are basically two ways that this fits in with our formalism. These are via statistical simulation and detailed deterministic simulation.

The simplest means of determining the performance of a system such as the one described in the previous section is to associate with each of the nodes a waiting time and with each of the branches a transition function which may or may not be a constant probability. This technique is ideal if the parameter of interest is a steady-state average, as is often the case.

If the parameters in question do not fall into the above class, then one must simulate at a detailed level, so that the behavior of the target system can be inferred. An example of a case that requires detailed simulation is the question of cache design that we address in Chapter 5. The question there is how much performance improvement can be expected from using a write-back rather than write-through policy, in an environment where context switches are potentially frequent. Here the performance is intimately associated with the transient response of the cache system. For this reason we must resort to a very detailed simulation of the target system.

In the next chapter we will mention some of the more interesting tools that we have developed for carrying out evaluation of these systems. All of the abstract design of the system structures presented in Chapters 5 and 6 was done using the abstract process model. All of the performance evaluation is carried out at bus cycle level, or below. This affords us confidence that the low-level simulation does reflect the behavior of the real machine, and at the same time allows us to mentally manipulate the design at the abstract level. Essential to this style of design is a good simulation language. A description of this language and its implementation are the major parts of the following chapter.
4. EVALUATION TOOLS

In the previous two chapters we have concentrated on the abstract aspects of representing architecture. We now turn to the concrete task of evaluating systems via simulation and instrumentation. This is related to the abstract work in that, as we have stated, simulation is often the court of last resort in terms of evaluation. These tools are mentioned in context in Chapters 5, 6, and 7, in studies that use them. They are presented here for completeness and to make the statement that, methodologically, it is often critical that a proposed system be simulated in detail in order to evaluate it. The tools include a very simple programming language, some sophisticated instrumentation tools, and a powerful concurrent programming language based on C [27] that is used to implement all the simulations in later chapters.

4.1. LX0: A Testbed Language

In the following chapters we will compare architectures via benchmark. We take a well-defined problem, code it in a higher-level language, implement that code on each of the systems being evaluated, and compare the results. An alternative is to compare the architectures at an algorithm level, in which case we could write a different program on each machine. Presumably this would result in two programs, each of which was optimized for the hardware it was to run on. This approach has not been taken, since we feel that higher-level language is the only viable environment for code development, and, hence, we want to detect cases in which proposed hardware requires excessive effort on the part of a compiler writer to support a higher-level language.

The disadvantage to this approach is that we then have the task of implementing a programming language for any machine we propose. This problem has been addressed by a compromise language that we will refer to as LX0. This language is sufficiently interesting to be representative of higher-level language and sufficiently simple that it is not unduly complicated to implement. Our approaches are to generate code for the machine being considered and to write the benchmark programs in LX0.
The LX0 language is simple. It has no data structuring capability beyond arrays and very simple input and output. This is important from an implementation standpoint, and the entire compiler is less than two-thousand lines of Pascal. The language does have multidimensional arrays, subroutines with arguments, floating arithmetic, and true block structure. This makes it a viable vehicle for algorithms, such as matrix inversion and linear system solution, the main ones we will use in Chapter 5.

4.2. Ptrace

We will refer to the technique of instrumenting a system so that performance or behavior information can be obtained as embedded instrumentation. Instrumentation, in general, requires some level of cooperation from the system being instrumented. An example of this technique is the use of specialized microcode to make a trace of some aspect of system behavior. The only caveat here is that the instrumentation code cannot be allowed to bias the results due to its changing the performance of the system.

Another powerful instrumentation technique that is common on modern computers is a single-step facility. In single-step mode, control is passed to some kind of instrumentation routine before the execution of each instruction. Typically, a program is run in this mode if one wants to find out exactly what the program execution path is. Since the single-step mechanism causes user-level programs to be invoked at each instruction execution, timing information cannot be directly gathered. However, there are some things that this method is ideal for, such as the generation of data address reference traces. An alternative to single-step for address trace generation is to instrument the backplane of the machine with a logic analyzer, or to embed microcode to gather the address reference stream. This is tedious at best and one still has the problem of being sure that the addresses obtained are the virtual addresses for the program being instrumented, and no other.

Even though the single-step method is simpler than the brute force methods, it is still complicated by the fact that the addresses have to be inferred from the current instruction and the current register values and, perhaps, memory contents. Consider the instruction

```
movl -4(fp),r3
```
To determine what address this instruction references for the read, we have to know the current contents of the frame pointer register. In addition, we have to know that the semantics of the movl instruction are that the first operand generates an address (if it isn't a register addressing mode) that is used for a long word read. Hence, a program to generate the address stream must look at the instruction operation code, its addressing modes, and the current state of the registers. In general it may also have to know the current state of the instrumented program's data segment.

We have implemented all of this machinery, for the VAX-11/780, using the ptrace mechanism of the Berkeley variant of the Unix [29] operating system. Ptrace is a system call that allows one process to single-step a sibling process and to have access to the registers and address space of the traced process. The dynamic address and instruction traces that we generate with this system are used to infer cache hit rate for different cache organizations in later chapters. The programs that are instrumented in this fashion are in no way aware of the fact that they are being instrumented, so that there is no modification to them whatsoever.

4.3. LS1: A Concurrent Programming Language

For implementing the representations mentioned in Chapter 2, it is essential to use a process-based concurrent programming language. An additional advantage is realized if this language is object based, since this allows the dynamic instantiation of objects and for simple protocols for communication paths, such as busses. In terms of existing languages, Simula [30] is probably the closest to appropriate for this task. Since Simula is not a widely available or supported language, the alternative of implementing our own was chosen.

The method we chose for implementing the language was to extend both run-time semantics, via a collection of run-time routines, and the syntax, using a preprocessor. Roughly, the run-time routines deal with the concurrency features of the language and the preprocessor implements objects, in the sense of object-based programming [31]. The structure of the overall system is shown in Figure 16.
There are many advantages to this style of implementation. The passive advantage is that this method is sufficiently general that any language can be implemented. The active and practical advantages lie in the ease with which the implementation can proceed. In fact, the two parts, run-time system and preprocessor, can be done independently. It makes more sense to implement the run-time system first, however, since the language is more useful with concurrence in the absence of objects than vice versa.

On the negative side, the user must be aware of the two new sources of error in programs, namely, preprocessor errors, and the run-time system. As with any preprocessor system, the real problem is error containment. Typical of many preprocessors, and in particular of this one, is the problem of assuming that the source it doesn't recognize is a legitimate source for the compiler, and it is quietly
passed on. This results in poor error containment. Preprocessor-based systems also demand more of the user in that, typically, the mechanics of the translation process are more complicated. None of the these problems is more than an annoyance, however, and we recommend this technique very highly for laboratory systems such as this.

We chose to implement the language around the C programming language, since it is a reasonable language for expressing the sequential parts of the processes we model with. We will now explain what each piece of the system does.

The run-time system consists of two major modules. At the lowest level is the kernel code, which deals with manipulating multiple stacks (one per process), deciding which process is ready to run, and context switching among the processes. This part of the code is written in VAX-11/780 assembler in order to take advantage of those instructions that are well suited to the task of process management, and which aren't generated by the C language. An example of these is the queue manipulation instructions.

The lowest level of synchronization that occurs among processes is the signaling or awaiting of an event. In our system the user creates events, which are of the form <event class> <event bits>. There are currently 4 bits in the event class field and 28 in the event bits, allowing up to 488 different events without requiring more than 32 bits to represent an event. An encoding scheme such as this is essential to provide both a large number of events and at the same time allow a simple implementation of awaiting multiple events simultaneously.

To support communication among processes, semaphores, and synchronization via queues, there is a collection of utility routines written in C. These routines also implement a simple debugger and state trace facility, as well as routines that associate events with their printable names, and some initialization code.

The other major part of the system is the preprocessor, which implements an object-oriented style of programming [31]. An object consists of state and semantics which say how the state is manipulated. The semantics are in the form of routines that are invoked when the object is accessed. The
routines are defined by the user when the object is defined. This provides the user insulation of the object from the access method, and it allows arbitrary code to be used to effect abstract data types. As an example, one might define a bus as an object. In addition to the usual read and write transactions, we might also have an enroll transaction that informs the bus that we are now connected and which returns a slot address. The semantics of the read and write transaction are that the object, via the concurrency mechanism we've already presented, can signal whatever process has enveloped and has as its bus slot address the slot number that is provided in the read.

The preprocessor parses the grammar in Figure 17. The major portions of an object are its state, which follows the keyword "state," and its semantic routines, which are indicated by the keyword "o_code." In addition, there is an initialization routine that is invoked whenever an object is created. All objects are dynamic in the sense that the data structures associated with an object are defined, but not instantiated at preprocessor or compile time. At that time, a routine is also defined to create an object and return a pointer to the object. To use the object, this pointer is used as one of the arguments to "eval," which is a macro defined by the preprocessor, and the mechanism by which an object is invoked. An example of the code generated to effect the object is given in Figure 18. The invocation

\[
x = \text{eval}(\text{optr}, \text{BuSREAD}, \text{addr})
\]

effects the read just mentioned.

The preprocessor generated moderately complicated code due to the dynamic nature of objects. This desire for dynamic objects is the reason that each o_code routine has as its first argument a pointer to the current object state structure. The implementation of this functionality relies on the standard C preprocessor heavily, and to the extent possible generates macros rather than routines in the interest of run-time efficiency. An unfortunate result is that the generated code can be quite unreadable, but this is hopefully not a concern to the user, since it should not be necessary for the generated code to be examined. The advantage over the comparatively simple concurrent C is that the process interconnection, in the style of those shown in Chapter 3, can be easily and reliably implemented.
Figure 17. L1 grammar.
struct a,sb,sc;
#define eval(optr,func) (optr->entries[func])
typedef int func_ptr();
#define fixbus bus f i x b u s  f i x b u s();
#define fix enroll bus enroll
#define read busread
#define write buswrite
int enroll(), read(), write();
struct bus *
mk_bus()
{
    struct bus *p;
    p = malloc(sizeof(*p));
    /* Now initialize the message vectors */
    p->entries[0] = busenroll;
    p->entries[1] = bus read;
    p->entries[2] = buswrite;
    p->entries[3] = bus fixbus;
    return (p);
}
#undef enroll
#undef read
#undef write
    /* end object */
struct gorp__parms
{
    int x,j,k;
}
gorp__args;
struct a *sa,*sb,*sc;
#define gorp__arg_count 6
#define gorp__arg_block (&gorp__args)
#define LAUNCH_gorp(x0,x1,x2,x3,x4,x5) gorp__args.x = (x0);
gorp__args.j = &(x1); gorp__args.k = &(x2);
gorp__args.sa = (x3); gorp__args.sb = (x4);
gorp__args.sc = (x5);
launch(gorp__arg_count,gorp__arg_block,gorp,2,100,"gorp")
int gorp();

Figure 18. C code generated for an object.

4.4. Other Tools

There are many other tools that were used in the studies presented in following chapters. These include routines that generate microinstruction traces by single-stepping a VAX-11/780 at minor cycle level and a microcode disassembler to analyze those traces. These will be discussed briefly in context in those chapters. The main point one should be left with from this chapter is that serious design and evaluation of computer architecture can be greatly helped by having an attitude of being willing to create useful and capable tools.
5. CACHE POLICY AND A STATIC INTERPRETER

In this chapter we will consider certain global effects on a static instruction set interpreter. Specifically, we consider a simple serial processor which has a cache between it and memory. The effect of cache policy is global in the sense that any instruction that references memory is affected by it. As always, we are interested in the performance of the system. One of the more interesting effects in this study is the consideration of the effect that context switch rate has on the performance of the system. Since this requires that the transient behavior of the cache be accurately modeled, we are unable to calibrate a statistical (queuing) model of the system without actually experimenting with the equivalent of a detailed simulation (such as this one) to obtain reliable statistics. For this reason there is very little of the formalism that we have proposed in this chapter. The tools mentioned in the previous chapter are used heavily, however, and the style of detailed evaluation that is evident here is such a critical component to the success of this method of system design that it is worth leaving the formalism for the time being.

In considering the performance of this system we are concerned with several factors. Among these are the effective memory bandwidth, the interaction of time sharing with cache policy, and the amount of traffic that is present on the bus. This last point is especially important in the case of multiprocessor systems. In the context of a multiuser multiprocessor system with private cache, we will consider the policies of write-through versus write-back for main memory update under the assumption that the processor is being time shared. This implies that the cache has to be occasionally flushed to allow a different process to run. The write-back policy has the advantage that the bus traffic is reduced compared to the write-through policy. It is usually assumed that the coherence problems of write-back require hardware such as global directories to detect potential coherence problems. For this reason a write-through cache is usually used which provides coherence for all transactions.

In this chapter, we will first present an analysis of this static interpreter and memory system under the framework developed in the previous chapter, we will suggest ways to avoid coherence
problems altogether in the user code, and we will examine the potential savings due to being able to use a write-back rather than a write-through cache, in terms of bus traffic. This topic is interesting by itself and, in addition, will give us an opportunity to demonstrate some techniques of analysis related to the graphical process model we have already defined. Most importantly, we will see an example of a case in which the run-time behavior of a program is such that analysis of the bus traffic situation by assigning fixed probabilities to such a process model is inadequate to describe the physical situation. This fact makes the point that simulation is an essential part of any performance evaluation effort.

5.1. Multiprocessor Bus Structure

The shared bus approach to multiprocessing is very attractive since it is simple to implement and easy to use in a multiuser time-sharing environment. Standard busses such as the Multibus, Versabus, and S-100 bus all have provisions for multiple processors on the bus [32, 33].

Larger computer systems such as the VAX-11/780 have been converted for shared bus multiprocessor operation [34].

Modern operating systems that are process based (VMS, UNIX, AOS) are particularly well-suited to such an environment [35, 36].

The obvious disadvantage of the shared bus approach is that the bus (and memory), being the only shared resource, is a bottleneck. In [34], a dual processor VAX is described and it is reported that bus saturation occurs somewhere between two and three processors. To ease this problem, multiple paths to memory, which can be switched to allow concurrent access by multiple processors, are provided. A great deal has been written on this approach [37, 38]. In the case of relatively few processors, however, it is convenient to avoid the complexity of cross bar or delta switches and attempt to connect the processors on a single bus. One can then consider connecting this substructure to others through various switching networks, as in Cm* [38]. In a shared bus system the number of processors that can be supported depends on the bus bandwidth available; hence, it is important to consider ways of reducing the traffic on the (single) memory bus. One method is to use a private cache for each processor. The effectiveness of cache memories in improving performance of computer systems [39, 40] is well-
known. The most obvious advantage of the cache is the reduced access time for cache relative to that of main memory. The use of a private cache can also reduce traffic on the memory bus. While this is of secondary interest in uniprocessor systems, it is of critical importance in bus coupled shared memory multiprocessor systems.

In this chapter we consider methods of cache organization which offer reduced bus traffic compared to the methods commonly used in existing uniprocessor designs. We consider a general purpose time-sharing system for which a detailed bus transaction level simulation has been constructed. Under realistic assumptions, we find that the bus traffic can be reduced in the typical case by a factor of greater than 2, and for some systems by a factor of greater than 8, by employing these techniques. This surprising result directly translates to having more than twice as many processors in the system at a given level of bus saturation. These techniques deal mainly with the update policy of main memory. This improvement is made with no degradation of common or desirable operating system functionality. In particular, neither interprocess communication nor symmetric multiprocessing is precluded.

5.2. Cache Coherence and Performance Gain from Write-Back

The two major categories of cache organization are shared and private. Examples of these structures (in the uniprocessor case) are commercially available. The VAX-11/780 from DEC uses private cache while the Data General MV/8000 uses a shared cache [14,41]. The ATU (address translation unit) is shown between the CPU and the cache, indicating that the virtual addresses which are issued by the CPU are translated into physical addresses which index the cache. There are advantages to placing the ATU between the cache and the system bus (the cache is then indexed by virtual addresses) and this organization is under study now. For the remainder of this chapter we will assume the usual case of translating the addresses before indexing the cache, as illustrated in Figures 19 and 20.

As in any hierarchical memory system, the question of coherence among multiple copies of logically identical data items (e.g., a cached item and its copy in memory) must be resolved. The shared cache in Figure 20 has no coherence problem, since there is no device that modifies the memory without going through the cache. This allows the use of a write-back rather than a write-through policy for main memory update. This is the organization used in the MV/8000.
In the private cache structure of Figure 19, there is potential for cache coherence problems even in the uniprocessor case, since DMA I/O can modify cached data. In the VAX-11/780, which uses this structure (with a single CPU), the cache monitors the bus for writes to locations that it has cached. When it detects one, it marks the corresponding cache slot empty so that the next access will be forced to read the modified value from memory. Writes to cache can be immediately passed on to main memory, and the memory system is able to queue write requests, so that the processor can continue without waiting for the write to complete. Read access to the cache, of course, requires no transaction on the system bus; hence, the private cache saves bus traffic over the shared bus cache.
There is a third alternative, namely, using a write-back policy in a shared cache. Since cached values are written only on a cache fault that requires a replacement into memory, and at context switch time, cached values may be modified in cache more times than they are written to memory. This represents a potential savings in bus traffic over the write-through case. In addition, the complexity of queued writes to main memory can be avoided. The disadvantage is that a "modified" bit must be maintained in the cache, and at context switch time, any modified words (or blocks) must be written out. This causes bus traffic to be related to context switch rate. The main problem with write-back is the coherence problem, which we will consider now.
A user process as modeled in Figure 21, executing in a time-sharing environment, will typically do all of its I/O via system calls and in the usual case will be doing blocking I/O. Interprocess communication will also be done via system calls (as opposed to directly writing shared memory). It seems then that the user code need not worry about coherence, so that any write-through operation from a

Figure 21. States of process execution.
user process represents an unneeded bus transaction. This is the motivation for considering how much traffic is used for write-through, and whether it can be avoided. In a process-based operating system, which is typical of what is run on the systems considered here, a process can be blocked, ready, or running. We will assume that a ready process can execute on any of the processors in the system, and that when a processor is ready to run a process, that process is taken from a central queue in an atomic operation which is not susceptible to races among processors. This is not a difficult objective to achieve, and it allows the operating system to be largely independent of the number of processors that are connected to the bus. Figure 21 illustrates the major states that a process can be in, and some conditions under which transitions occur. This simple model is not at all unrealistic for consideration of the execution phase of a process. We will ignore process initiation and termination, since these are boundary conditions during which operating system control of cache can be assumed.

If the following three rules are observed, we can at least ignore the coherence problem for a non-system code.

1. The process does no I/O itself. This does not restrict the operating system from initiating DMA I/O into the process address space.

2. When a process is in the blocked or ready state, there are no values from the process address space in the cache.

3. When the process communicates with another process, it does so via a system call, as opposed to (for example) writing into physical memory that the receiving process is expecting to use for communication.

While it is beyond the scope of this chapter to treat operating system implementations relative to cache policy, we have considered the problem. Suffice it to say that these rules do not preclude services such as nonblocking I/O, multiple event wait, and interprocess communication, which we feel are essential in any multiprocessing system. Having confined the coherence problem to the operating system, we appeal to the fact that the system can be aware of when coherence problems can arise. In the case of interprocess communication, it is possible to implement message passing by mapping a block of memory into the receiver's address space. Since the pages were previously unmapped (not in the
receiver's address space), they certainly are not in cache, so there is no coherence problem. A less elegant alternative which has been used in DEC-10 dual processor systems under SMP (symmetric multiprocessing) is for senders to cause a cache flush in the receiver's cache. There are more intricate hardware solutions in the literature as well [42].

In the later sections we will consider the amount of bus traffic that can be saved by using a write-back cache in various system configurations. In what follows we assume that the problem of coherence is dealt with as suggested above. We now discuss the simulation system used.

5.3. Simulation Analysis

There are many ways of analyzing a complex system such as the one in Figure 19. These range from the stochastic approach of characterizing a system in terms of a small number of statistical parameters to the empirical investigation of a realization of the system. We have chosen to simulate the system at a fairly low level, i.e., instruction timing and bus conflict behavior are faithfully replicated, but those phases of operation which are not directly of interest relative to cache performance, such as instruction decode details, are not included. The simulation will accurately reflect, for example, alternating bus access by the processors. The system is driven by execution of a target system code.

An alternative to simulation in situations such as this is to model the system under question. In this case we would be interested in the amount of bus traffic the two cache organizations result in. In Figure 22, we have presented a graph model of the behavior of a subset of the multiprocessor system, namely, the cache on one of the systems. Nodes 2 or 6 can follow node 1 depending on whether the cache was hit or missed, respectively. The shorter path (via node 6) is taken for a hit. Using fixed probabilities on the branches of this diagram, we can predict the steady-state average memory access time, and even the traffic due to cache activity. However, for our purposes the steady-state behavior is not sufficient to imply which of the two organizations, write-through or write-back, will result in less traffic for a given time slice. This can be seen by graphing the traffic rate of the cache as a function of time as in Figure 23. In the figure, $T$ is the period of the (assumed constant) time slice. At each context multiple, the write-back cache makes a burst of bus transactions as it updates those locations that were
modified during the time slice. The write-through cache has no such need, and merely enters a cold condition similar to the cold behavior of the write back cache. Analytically modeling this situation is difficult since the program behavior is so strong an influence on the rate at which the cache reaches steady state. In addition, the program behavior can influence the choice of which update policy is preferred. If the program behavior is such that there is a long period of essentially random data access, as during garbage collection, then the write-through cache is preferred, since there is essentially no steady state reached. The point of this is that, in these situations, simulation is by far the most believable tool for analysis. As mentioned in the previous chapter, this system was simulated using the simulation system described there. Part of the benefit of using the simulation environment is that the code can be shared. For example, there is only one copy of the code which implements the CPU element, and there are two independent processes that execute this code. In this sense the simulation system is modular. To increase the number of processors on the bus we merely invoke a third copy of the CPU process by changing two lines of the code in the simulation. Changing the design of the target in this fashion is quite simple and allows the designer to evaluate several different system configurations in a matter of a few hours.

5.4. Effects of Cache Policy

Within the context of a system such as that in Figure 19, there are many parameters which can be varied without altering the basic structure. We consider the following:

1. Cache policy: write-back vs. write-through
2. Length of time slice
3. Timing parameters such as bus speed and behavior with and without cache

Our main result deals with the amount of bus traffic that can be saved by using a write-back rather than a write-through policy. For the write-back case, the length of time that a process runs without a context switch (and attendant write-back) is also examined. We refer to this time as the time slice.
As we have mentioned, the simulator used here is a low-level deterministic simulator. To evaluate a given design parameter, a test program is run on the (simulated) target. In our case the LM0 language described in Chapter 4 was used to program the target system. This language was chosen for convenience since some support software for its execution was already in existence. The hypothetical
Figure 23. Dynamic traffic rate due to cache.
target processor was chosen because it is architecturally interesting and straightforward to implement. There is nothing inherent in our analysis technique that precludes evaluation of existing real machines. We have, in fact, done so in evaluating a similar system incorporating PDP-11 processors. Since we are concerned here mainly with the issue of bus traffic and cache effects, the detailed issues of the processor architecture are largely irrelevant, as long as the processor used is similar in its address reference behavior to conventional machines. Our simulation has been so designed.

![Average Bus Utilization](image)

**Figure 24.** Bus utilization versus bus speed.
To illustrate the role cache plays in reducing bus traffic, a simple program was run on the system and bus speeds were varied. The program sorts elements in a matrix by calling several subroutines. This program was used because, in contrast to the Gaussian elimination program used later, it depends heavily on subroutines.

Figure 24 is a plot of average bus utilization versus bus speed for a two processor system with private cache. The center curve is for the case in which the cache is turned off completely. Note that saturation occurs at a much slower bus speed for the no cache case, indicating that the cache is effective in keeping the processors off the bus. It should be noted that the statistic given (bus utilization) is not a useful measure of performance since the amount of time a processor spends waiting due to conflicts is not indicated. It is not difficult to obtain conflict statistics from the simulator, but for this study it suffices to examine the execution time of the various configurations. A more interesting example is shown in Figures 25 and 26 which show traffic and execution time, respectively, as a function of time slice. The analysis of the cache behavior has been carried out on several variations of the following configuration.

1. The cache is two way set associative and can hold 1 Kilobyte. Both code and data are typically cached, and the block size is 4 bytes.

2. The cache is private to the processor, one cache per processor, as in Figure 19.

3. The memory is simplistic in that no requests are queued; if a word is written on a write-through cycle, the processor waits until that transaction is complete before proceeding.

4. The bus is relinquished at the end of each cycle, so that if contention occurs, processors will alternate bus cycles.

The code in this case is a 30 X 30 Gaussian elimination program. The cache size is 256 32 bit words. This is intentionally somewhat small compared to the size of the code plus data for the program, which totals about 5600 bytes. The hit rate for this program is typically 95%.
The Gaussian elimination program does no I/O and is inherently free of subroutine calls. To simulate the effect of operation in a time-sharing environment, a clock tick interrupt occurs at regular intervals corresponding to a transition from the running to the ready state of Figure 21. It is assumed that on return to the running state the cache appears empty and has to be demand loaded. While this is not necessarily the best way to design a system, it is common practice.

At context switch time, the write-back cache has to write any modified location into main memory. The write-through cache has no such requirement since written values have already been updated. Hence, we expect that a very high context switch rate will cause the write-back to suffer. As can be seen in Figures 25 and 26, this is indeed the case.

![Graph](image)

**Figure 25.** Bus traffic versus timeslice.
Figure 26. Execution time versus timeslice.
However, even going to the extreme of a context switch every two thousand instructions, the write-back strategy is superior by a factor of 1.64 in terms of bus traffic and by a factor of 1.62 in execution time, for this program. While the times as shown in Figure 26 are accurate relative to the assumed speed of the components of the system, they depend on both the speed of the processor relative to the cache and the times for the instructions executed. The bus traffic in Figure 25 is independent of processor speed and we can state that assuming a context switch every eight thousand instructions we need to support 2.57 times as many bus transactions if write-through is used.

On a large VAX system, the context switch rate under load is in the vicinity of sixty context switches per second [29]. With the timing used in our experiments, this corresponds to about 16000 instructions per time slice, which will give an even greater advantage to the write-back policy. Furthermore, increasing the number of processors for a given multiprogramming load will decrease the context switch rate, further reducing overhead bus traffic. In the limit, assuming processes run to completion, the write-back policy requires fewer bus transactions by a factor of 8. If the processor architecture is primarily memory to memory, as in the Intel IAPX 432 [16], the need for cache to reduce bus traffic is even greater. Under these assumptions, and assuming run to completion processing, we find that the write-through cache issues more bus transactions than the write-back cache by a factor of 18 for this problem. This illustrates the need for careful cache design in such systems.

We have also investigated smaller programs and different cache organizations. Figure 27 is a graph of execution time versus bus speed for the matrix sorting problem. The cache in this case has a blocksize of 16 bytes, and we assume that any write-back transaction must write a 16 byte block. Write through, of course, requires only a single transaction. The average bus utilization in this case ranges from 0.36 to 0.59, even though the bus speed is 2.2 microseconds, which corresponds to approximately twice the instruction time for this processor. When the processor speed was changed to 130 ns for all instructions, with a 100 ns cache, the bus utilization was still only 0.9 at a bus cycle time of 1.2 ns. Thus, adding a write-back cache effectively more than doubles (700 ns versus 2000 ns at the 0.55
Figure 27. Execution time versus bus speed for matrix sort.
saturation level) bus bandwidth as measured by the average saturation. In Figure 28, the bus traffic for this program for the write-through and write back cache is plotted. The two are equal at a point well below realistic levels of context switch activity. As in the other cases, the write-back policy is superior for reasonable context switch rates, though in this case, the improvement is only a factor of 1.3.

In conclusion then, we have shown that for a shared bus multiprocessor, organization one can significantly increase the number of processors a given bus can support by using a write-back rather than a write-through policy for main memory update from cache.

For small to medium size machines, for which the cost of a processor is small compared to the cost of the rest of the system, this is an especially attractive means of improving multiuser performance without adversely affecting the efficiency or the functionality of the operating system.

There are many more parameters and design tradeoffs that we have not considered here. One that is interesting is the width of the path to main memory. Assuming that the path from cache to memory is very wide and that all address resolution occurs in the cache, we can take advantage of large bandwidth to the memory via parallelism, and still retain the advantages of high address resolution. If we assume that the I/O devices are reasonably intelligent, it is possible to move a large part of the file system and operating system services into the device controllers (the file and I/O handlers) and into the bus interface (interprocess communication and synchronization). Current CPU chips are quite impressive in that they have reached the level of being viable for support of useful operating systems. VLSI techniques applied to considerations, such as cache design and operating system support, as described here, will allow the construction of extensible systems that can be configured for an extremely wide range of performance while maintaining component commonality.

In this chapter we have presented some interesting statistics on relative performance of various cache configurations. This has served to illustrate our methodology and some of the tools we have developed. Relative to classification of interpreters, this information is most useful in that a cache model is presented that will be used in a different form in the next chapter, in which we proceed from the memory bandwidth problem to the question of instruction set level.
Figure 28. Bus traffic for write through versus write back.
6. ANALYSIS OF AN ADAPTIVE INTERPRETER

In Chapter 3 we considered various classes of instruction set interpreters. One of the promising concepts developed there was that of an adaptive interpreter. In this chapter we will analyze an adaptive structure for instruction set interpretation in terms of the network model we have developed. The question is whether we can architecturally exploit the greater amount of context information inherent in a complex instruction in order to improve performance. Since we assume that essentially all programming will be done in a high-level language, we are not constrained to make the instruction set "humanly usable," and ergonomics can be sacrificed in the interest of performance and cost.

As stated in Chapter 3, machines have been built which assume differing answers to the question of what is a reasonable level for an instruction set. On one hand, the SYMBOL computer [15], directly supports many high-level languages and operating system functions. The antithesis of this approach is the RISC [43] machine, which requires the compiler to explicitly generate code for functions and addressing modes that are built in to higher-level machines. Intuitively, one would suspect that the former approach incurs less overhead and results in higher performance. On the other hand, it is typically easier to pipeline instructions that have little mutual dependence, and which are of a single fixed length. These facts typically favor simplified instruction sets.

In this chapter we are concerned primarily with a level between these two extremes. This level is typified by the VAX-11/780 [14] which is commonly referred to as a "complex instruction set" machine. This instruction set has twelve addressing modes and supports many common instructions having from zero to three explicit operands. There are also instructions with as many as five operands. An instruction set such as this can, in principle, be implemented in such a way that overhead for instruction fetch and decode is amortized over several micro-operations, thus reducing overhead compared to simplified instruction approaches. This machine will serve as a vehicle to compare the proposal for adaptive interpretation with current medium size machines.
Since the issue here is the choice of instruction set level, it is important to avoid penalizing an approach for the deficiencies of a particular implementation. We do not accept that the increased bus traffic demanded by simplified instruction sets is a significant problem, since a reasonable cache design will restrict this traffic to a private bus from CPU to cache, so that multiprocessing on a shared memory is not degraded. Virtual memory likewise is assumed to be supported by a translation unit between the CPU and cache, or between the cache and memory. Hence, we assume that the CPU deals with virtual addresses and that memory management is zero cost. Another example of an implementation problem is evident in the VAX calls instruction. This instruction is significantly affected by the fact that the cache is write-through and the memory can only buffer four writes before blocking the processor. A write-back cache is a reasonable solution to this problem.

In the following sections the level of instruction set will be considered using the VAX as an example of a moderately complex instruction set. This level of instruction set will be evaluated as a vehicle for executing code from existent high-level language compilers. It is clear that there are many ways to improve the VAX instruction set architecture to ease implementation or increase performance, but we are not trying to decide whether the VAX instruction set is better than lower-level instruction sets. Rather, we will show that architectural techniques exist for implementing an interpreter for instruction sets at the level of the VAX, which improve performance over both the low-level approach and the VAX-11/780 implementation of the VAX instruction set. These techniques assume no change in cycle time. Using simulation and hardware monitoring techniques, we find that applying these techniques to the VAX instruction set improves performance of compiled high-level language programs by at least a factor of two. The compilers used are unmodified. These techniques are not applicable to lower-level instruction sets, hence we conclude that very simple instruction sets as typically implemented are currently at their architectural limits and only improvement in technology (faster cycle time) can improve their performance. Complex instruction sets as implemented in machines such as the VAX are very far from their architectural limits and a more sophisticated (and, hence, more costly) interpreter for this instruction set can improve performance significantly.
6.1. Direct Interpretation

Saying that an instruction executes in a single cycle also implies that the required hardware to support it is in place. This being the case, the only way to speed up this particular instruction is by increasing the cycle speed. This is not the case for multiple cycle instructions, where the algorithm which is used to interpret the instruction is accessible to the architect. Consider for example the instruction

\[ \text{add13} \quad -4(fp), -8(fp), -12(fp) \]

which can be used as a realization of the expression

\[ a := b + c; \]

Two options for carrying out this instruction are sequentially

- Compute address \( a \), fetch, compute address \( B \), fetch, compute address \( c \), store sum

in parallel

- Compute addresses, then fetch, then store sum.

Assuming a dual ported cache and multiple adders for address computation, the second method executes in less time than the first but it is significantly more expensive. Another disadvantage of trying to make complex instructions execute more directly is that the control for such instructions can be complicated. The design that we propose here is motivated by the concept of an adaptive interpreter as defined in Chapter 3.

6.2. High-Level Language Behavior

The statement that we will support high-level languages has two significant implications. The first is that the run-time environment required by the definition of the language must be provided. The second is that since humans will not be expected to code in the machine language of the system, the assembly-level language does not have to be humanly usable.

The languages we are considering here are the so called "ALGOL-like" languages. These include Pascal, ALGOL, C, and Ada, all of which have similar behavior relative to local and global variables, recursion and (except for C) block structure. We will not consider support of languages such as LISP.
and PROLOG [44]. These languages assume a much richer run-time environment than do the ALGOL-like languages and, hence, are very amenable to specialized support, but they are less mainstream and should be considered separately.

One computational activity that can be easily cast as a complex instruction is expression evaluation. In a simple machine, the arithmetic and logical operators act only between pairs of registers. Hence, all computation of expressions has to occur within the register set. Complex instruction set machines may be able to execute an assignment statement from a high-level language in a single instruction. Figure 29, for example, lists the code for the expression \( a = b \times c \) for a complex and a simple machine where \( a, b, \) and \( c \) are local variables on the stack. Figure 30 lists the code for \( a, b, \) and \( c \) being register variables on both machines. Figure 31 lists the code for the assignment statement

\[
a[i] = b[j] \times c[k],
\]

on a simplified instruction set machine and on the Intel IAPX 432 [16].

The code density advantage for the complex machine is evident, but this is not necessarily an issue, since we are mainly interested in performance. This example is offered only to give an idea of

\[
\text{mull3 \hspace{1cm} } -4(fp), -8(fp), -12(fp) \hspace{1cm} a = b \times c
\]

VAX code, 7 bytes.

\[
\text{ldl \hspace{1cm} } -4(fp), r0 \hspace{1cm} \text{; pick up } b
\]

\[
\text{ldl \hspace{1cm} } -8(fp), r1 \hspace{1cm} \text{; and } c
\]

\[
\text{mul \hspace{1cm} } r0, r1 \hspace{1cm} \text{; perform mult}
\]

\[
\text{stl \hspace{1cm} } r1, -12(fp) \hspace{1cm} \text{; store it}
\]

Simplified instruction set, 16 bytes.

Figure 29. Code for \( a = b \times c \) with variables on stack.
mull3  r0,r1,r2  ; similar to above

VAX code, 4 bytes.

ldl  r1,r2  ; r1 has b, assume a is in r2
mul  r0,r2  ; r0 has c

Simplified instruction set, 8 bytes.

Figure 30. Code for \( a = b \times c \) with variables in registers.

shiftl  j2,r1  ; mult by object size
add  b,r1
ldl  (r1),r3  ; r3 has \( b[j] \)
shiftl  k,2,r1  ; \( j \) and \( k \) are indices in regs
add  b,r1  ; r1 has address of \( b[k] \)
ldl  (r1),r4  ; r4 now has \( b[k] \)
mul  r3,r4  ; form the product
shiftl  i,2,r1
add  a,r1  ; compute address for store
stl  r4,(r1)  ; store value

Simplified instruction set, 40 bytes.

\[ a[i] = b[j] \times c[k] \]

IAPX 432 code. Essentially direct, approximately 14 bytes.

Figure 31. Code for \( a[i] = b[j] \times c[k] \).

the level of complexity of the different instruction sets. One cannot infer from such trivial examples that one approach is better than the other without statistics on actual program behavior relative to such expressions. Some programs are dominated by a single kind of computation, but many are not. We have found for example that in a Gaussian elimination program operating on a thirty by thirty system,
approximately seventy percent of the entire run time for the program is accounted for by two lines of high-level language code. Both lines are complicated expressions in assignment statements which are typical of such matrix manipulation programs. Other programs are not so well-behaved, and so general techniques of analysis are needed to evaluate where optimization efforts should be directed.

In comparing the fragments in Figures 29, 30, and 31, we want to consider which instruction set level will result in higher performance when implemented using equal minor cycle times. One of the potential advantages of the single complex instruction approach is that the fetch-and-decode is only done once, whereas using a simplified instruction set, several instructions have to be fetched and decoded. Another is potential parallelism within an expression. Figure 32 is a diagram of the parse tree of one of the assignment statements that was mentioned above from the Gaussian elimination program. The entire right-hand side, which contains at least seven arithmetic operations, is known to be without side effects and, hence, can be computed concurrently, which represents a substantial speedup of the program if, as is the case here, the expression represents a large proportion of the run time of the program. This parallelism is not available in the simplified instruction set case, where the expression interpretation is viewed in the lower-level context of computing memory addresses, adding indices, computing a destination location and storing a result. Here, the architect of the system has little flexibility in optimizing the execution of this expression. This is not necessarily bad, since the burden is merely moved to the compiler, but the lack of flexibility is still inherited by the compiler. The last potential advantage of complex instruction sets that we will consider arises from the observation that the usual method of local variable access in high-level languages is via indexing the frame pointer register. Since the indexing of a register is one of the standard addressing modes, it is typical that in a subroutine each access to a local involves the adding of the frame pointer to the offset to determine the virtual address of the local. The frame pointer is typically only changed by subroutine calls and by returns. During the execution of a routine, it is known that the virtual address of locals will not change, so that adding the offset of a local to the frame pointer and thereafter using that value saves recomputation of the address. This technique is certain to work for C programs, since even allocation of locals in an inner "block" is done at routine entry time. ALGOL, with true dynamic allocation, may
Figure 32. Parse tree for an assignment statement in Gaussian program.

pose some difficulties relative to this technique.

In summary then, we will try to exploit the following aspects of complex instruction sets relative to high-level language execution.
(1) Amortize fetch-and-decode time over several low-level operations.

(2) Exploit parallelism inherent in high-level language statements.

(3) Appeal to invariance of the virtual address of locals during subroutines.

Before turning to our proposal for realizing these improvements, we should point out that we have not mentioned many of the features that are present in some complex instruction sets. The addressing modes we have mentioned and those that typically occur in code are the relatively mundane indexed, register indirect, and immediate. The complex instruction set features that are most likely to be useful in conjunction with the techniques we are about to propose are those which offer a more direct execution of common higher-level language idioms, such as multiple operand assignment instructions. The results we will present make no use of instructions such as polynomial evaluate or case, which are present in the VAX instruction set. Effective use was made, however, of multiple operand instructions, many of which had their operands in memory.

6.3. Proposal for an Adaptive Interpreter

The approach we take for improving performance primarily involves using a somewhat intelligent cache for instructions which we refer to as the I-cache. Each instruction in the I-cache corresponds to a single instruction in memory, but the version in the I-cache is translated into a more easily executable form. This translation at cache fill time is an example of what has been called dynamic interpretation [7], but since the micro-operations carried out for a given instruction are invariant during execution of the program, we prefer the weaker characterization of static adaptive interpretation. The I-cache structure will be detailed shortly.

We will assume that the instruction stream does not modify itself, primarily as a convenience. Since we cache a translated version of each instruction, allowing modifications to the instruction stream would have to be detected so the cache could be flushed. The default on most systems is for the code to be read only, so this poses no problem. We also assume that all addresses which index the I-cache are virtual so that no translation to physical addresses is necessary.
Since one of the important aspects of high-level language support is efficient expression evaluation, the design of a data cache to efficiently execute multiple operand expressions can be critical. In particular, a two-ported cache can be used quite naturally by expressions such as that in Figure 32. A two-ported cache implemented in the most obvious fashion will be roughly twice as expensive as a single ported one, since the address and data sections must be replicated for the additional port. Here we will assume that the cache is single ported as in the VAX-11/780. This is done to decrease the complexity of our design and to show that even without going to the two-ported cache dramatic improvement can be realized just from amortizing address computations from frame pointer references and from avoiding decode overhead. Since we will be comparing this system to the VAX-11/780, we are also guided by the assumption of a fixed two-hundred nanosecond microcycle time. All performance data presented is relative to a two hundred nanosecond minor cycle time.

Before presenting the design of the instruction decode cache, we will discuss the way the VAX instruction set is interpreted in the 11/780 implementation. We assume the reader is somewhat familiar with the instruction set and the fact that three operand instructions are available which work with almost all addressing modes. Some instructions are optimized in the VAX and are executed fairly directly. This can be accomplished by doing much of the work in the instruction decode microinstruction. Since many instructions may use the contents of a pair of registers, the registers indicated in what is the register specifier field for register-to-register instructions can be read during this cycle. If this turns out to be a register-to-register instruction, then the next microinstruction can use the values. If not, the values that were read can be ignored. This works as long as no state is modified. We refer to this as anticipatory operand generation. Figure 33 lists a generic microcode routine for the interpretation of a two operand register-to-register add, a three operand register add and a two operand local to local add. These instructions are representative of instructions that exist on the VAX 11/780. For the most part, the instructions are interpreted sequentially.

In Figure 34, a block diagram is given for a general machine that could be used to interpret the microinstructions indicated above. This structure is typical in that it has a two-ported register file and
; Instruction Micro operations

add12  r0,r1  ; read r0 & r1; set alu=add; NA = 2 operand op.
   ; write result to r1; NA = instruction decode.

add13  r0,r1,r2
   ; Jump to 3 operand routine
   ; read r0
   ; read r1; set alu=add
   ; write sum to r2; NA = instruction decode.

add12  -4(fp),r0
   ; Compute -4(fp); alu=add
   ; Read mem; read r0
   ; Put sum into r0; NA = instruction decode.

Figure 33. Microcode routines for different add instructions.

can be made to carry out a three operand instruction with register operands in a single cycle. For sequential interpretation of instructions, such as those in Figure 33, this is a reasonable structure. To more directly support these multiple operand instructions, we will maintain more context, in terms of both control and data, in the data path of the machine. This adaptive machine is described now. The structure of the I-cache machine is as indicated in Figure 35. The process model for the execution of this machine is shown in Figure 36. There are two fundamental modes of operation for this system, dependent upon the existence of a hit in the I-cache, and whether node three or node five follows node one (the instruction invocation node). The limiting cases are those in which the I-cache always hits and when it always misses. This situation is similar to the analysis in the last chapter where the cache behavior of the system was of interest, except that here the time taken for execution is essentially fixed given that there is a hit in the I-cache. This is the result of the hardware (the primitive level) having been optimized for essentially direct execution of the translated instructions.
Figure 34. Block diagram of a typical microcode interpreter.
Figure 35. I-cache machine.
There is a further optimization that we propose for this graph, which is shown in Figure 37. In the original diagram of Figure 35, node six, which is the execution phase, can be carried out only after it has been decided that there is a hit. An alternative is to predict and locate the next sequential cache location and perform that operation while node one is deciding whether the next node is a hit or a miss. In the hit case, the computation is complete and the computed value only needs to be stored.

Figure 36. Process network model of I-cache.
Figure 37. I-cache with anticipatory execution.
When an instruction is first encountered, it is translated in the usual sequential fashion into instructions of the form indicated in the I-cache format in Figure 35. These instructions have three operands, with the operand addresses resolved. Hence, the field width is the same as the virtual address length. Most of them can execute in two cycles, since in practice most arithmetic operations have both operands in registers, or one in the cache and one in a register. In either case both operands can be accessed simultaneously. When the microcode enters the instruction decode state, the current next instruction, if in the cache, begins anticipatory operand generation as described. Since we can simultaneously access two registers and the cache, this is a much more powerful, and more expensive, implementation of the technique. For an I-cache hit in the case of the

\[ \text{addl2 -4(fp),-8(fp)} \]

instruction of Figure 5, we require only three cycles versus the six required by the VAX-11/780. As we have stated above, a potential efficiency of the I-cache is reuse of addresses of local variables. When an instruction is translated into the I-cache, any reference to a local variable is resolved by adding the offset to the frame pointer and storing this virtual address in the I-cache. Subsequent hits on that instruction save (at least) a cycle for adding the offset, if we assume a completely sequential interpretation of the instruction stream. Alternatively, one could resolve these references per access to allow us to use a more compact representation in the I-cache, but that does not significantly reduce the complexity in the case of C code or in other situations where it is known that the frame pointer is fixed during execution of a routine. If this is not the case, then detecting modification of the frame pointer and changing all I-cache locations that contain a dependence on the frame value become an issue. Similar statements apply to the references from other registers. Each of these cases is considered in our simulations.

Next we describe the techniques used to evaluate the performance of these ideas. One aspect we have not given exhaustive consideration to is the cost increase involved in such an implementation. This is certainly an important aspect of any design and we will return to it in the conclusion.
6.4. Performance Evaluation Tools

Measuring the benefit of these proposed techniques is quite difficult without essentially realizing the system either through simulation or by building a prototype. By using the VAX-11/780 as a vehicle for these enhancements, we have been able to avoid a major portion of the implementation task. To determine general behavior of high-level language code and to simulate the simplified instruction set machine, we use a general purpose event-driven simulation system. To evaluate the performance of the VAX instruction set with and without the I-cache, we built an instruction-level instrumentation system based on the \texttt{ptrace} facility of the VAX-11/780. The 11/780 implementation was also instrumented at the microcode level using a separate controlling computer.

6.5. Event-Driven Simulation Facility

The event-driven simulation system is based on the C language, and was described in Chapter 2. To extend the language for the simulation of highly concurrent systems, we have constructed a simulation environment which provides for process creation, termination, synchronization, and communication. This allows a very natural expression of the semantics of a digital system, since typical hardware systems can be accurately viewed as a collection of processes. The kernel portion is written in assembly language since it needs to be able to maintain multiple data segments for the processes.

The simulation system implements the target as a collection of processes that run in the single address space of a UNIX process. This feature is critical to good performance. We incur a penalty of only about thirty instructions for signal and wait, since there is no need to call the operating system to communicate with other processes. This simulation system was used in this study to determine how much time programs spent in executing expressions, as in Figure 29, and to simulate the simplified instruction set machine.

6.6. Instruction-Level Instrumentation

In order to accurately evaluate the performance of the VAX with our proposed enhancements, we instrumented the system at the instruction execution level. This was done via the \texttt{ptrace} facility pro-
vided in the 11/780 implementation. *Ptrace* allows a process on the VAX to single step a child process, gaining control of the traced process after each instruction execution. The parent process then examines the traced process's address space and registers. The trace facility we developed is aware of all of the addressing modes on the VAX and simulates the memory and register reference behavior of the machine to determine which locations were accessed, whether that access was a read, write, or instruction fetch. This level of detail, i.e., knowledge of exactly which virtual addresses were asserted and in what order, can then be used to simulate the behavior of various implementations of the VAX instruction set. Since it is possible to trace any program without its prior consent, it is quite simple to generate instruction traces, and in our case, to evaluate the performance of an I-cache applied to the VAX, executing compiled code. All instrumentation at this level can be carried out during normal time sharing, unlike the microcode level instrumentation which requires that the system be operated stand alone.

6.7. Microcode-Level Instrumentation

A second instrumentation system was implemented at the microcode level, in order to relate these improvements to the 11/780 implementation and to understand the true character of this implementa-
tion. Information from this level is used to calibrate and validate the instruction-level and simulation-level results.

The instrumentation carried out at this level is done using a separate computer which is connected to the VAX-11/780 via the console interface, which is a serial line. This allows the control computer, which in our case was a Hewlett Packard desk top computer that supports Pascal, to act as the console. This allows programs on the control computer complete control over the VAX. To evaluate target VAX code, programs are compiled using a locally modified version of the stand-alone run-time system which comes with the Berkeley version of the UNIX operating system [36]. The machine is then halted and a program on the control processor invokes the test program on the VAX in such a fashion that the micro trap is taken. The control computer proceeds to single step the VAX in microcode recording each microinstruction address as it is executed. This microinstruction address trace is kept in
a file on the control computer. When the target program exits, the test program reloads the operating system and returns the VAX to multi-user use. The file of the microinstruction addresses is then transferred back to the VAX where the microinstruction-level behavior of the program can be analyzed. In this mode of operation, we can execute about 1000 VAX instructions per hour, limited mainly by the low speed (9600 baud) serial line between the VAX and the control computer.

In all the experiments reported here, the memory management in the VAX was disabled and the microcycles expended awaiting cache fills and SBI write stalls were filtered out in order to avoid instrumenting waits for the instruction prefetch buffer or interaction with the cache. This is done to ensure that all cycles counted were used in interpreting the instruction set.


The three systems whose performance we consider are the VAX-11/780, a somewhat hypothetical simplified instruction set machine which is similar to the RISC [23] machine, and the VAX instruction set with the I-cache enhancements. The performance of these systems was determined using the event-driven simulation for the simple instruction set, the ptrace facility to determine the performance of the enhanced 11/780, and instrumentation at the microcode level for the 11/780 implementation. Figure 38 is a graph of the performance improvement of the I-cache machine as a function of the size of the I-cache. The I-cache enhanced VAX instruction set is compared with both the simplified instruction set and the unenhanced VAX. The data used in the example was gathered from running diff, a file comparison program. The diff program is more than a factor of ten larger than the largest size I-cache that we considered indicating that the locality of the program is being effectively exploited. It is evident from the graph that increasing the size of the I-cache beyond 128 instructions is not justified in terms of increased performance.

In the experiments we carried out, performance is measured as the ratio of microinstructions per complex instruction for the considered program and the graphs are given in terms of the factor of improvement attained. Since each microinstruction requires a minor cycle time to complete, the
number of microinstructions determines the execution time. Each minor cycle takes two hundred nanoseconds to complete so, for example, a one thousand instruction program averaging four microoperations per instruction will take four thousand minor cycles to complete. Considering minor cycles allows us to compare microcoded and hard wired machines. The unenhanced 11/780 has a ratio of 5.12 micro-instructions for each assembly-level instruction and the simplified instruction set machine has an equivalent ratio of 3.88 to 1 for the diff example.
6.9. Matrix Multiplication

Figure 39 shows a more dramatic increase in performance for data generated from running a different program, an eight by eight matrix multiply. This program is free of subroutine calls and can behave significantly differently than the diff program. In this case, the size of the I-cache is not as significant as before in that the routine is much smaller and can fit in an I-cache of 128 instructions. A more significant factor here is the poorer performance of the 11/780 which averages more micro-operations per instruction in this type of computation. For similar matrix multiply routines, the ratio for the 11/780 has reached 6.44 microinstructions/instruction. These programs were coded with integer arithmetic so that the assumption of from two to three cycles for the execution phase of the instructions was realistic, and so that more direct comparisons with simplified instruction sets could be made. In any event, we assume that the multiplication on both machines is carried out by an instruction and that it is of equal speed on both machines. The improvement with the diff program ranged from a factor of 1.76 with an I-cache size of 64 instructions to a factor of 2.11 with an I-cache of 256 instructions. For the matrix multiply program the improvement ranged from a factor of 2.65 with an I-cache of 64 instructions to a factor of 2.75 with an I-cache of 256. The corresponding improvement over the simplified machine was from 1.34 to 1.59 for diff and from 1.27 to 1.31 for the matrix multiply. For both programs, it is assumed that addresses based on the frame pointer are reused, and that all other indexed addresses are recomputed only when they changed. We now consider the effect of altering these policies.

6.10. Reuse of Addresses

Figure 40 illustrates the relative benefit derived from decode amortization and the reuse of address computations in the I-cache. Three policies were simulated using the diff program. In the first, we assume that, whenever an address that is dependent on a register is needed, it is computed using the relevant register. Further, we assume that such a computation takes two microcycles. At an I-cache size of 128 instructions this yields an improvement of a factor of 1.57. The second policy in Figure 10 also assumes two cycle recomputation, but in this case we exploit the fact that the frame pointer is
invariant within routines so that we can reuse frame pointer addresses. This increases the improvement factor to 1.77 at an I-cache size of 128 instructions. The best performance is attained when we assume that we can recompute addresses that have changed in a single cycle and only those addresses which depend on registers that have changed are recomputed. In this case, we get an improvement factor of 2.001 at the same cache size.
6.11. Set Size

The last graph, Figure 41, compares the set size two and set size four I-cache running the \texttt{diff} program. The maximum percentage performance improvement due to the set size increase was less than four percent, even though the miss rate dropped significantly. This indicates that a set size of two is sufficient. Finally, we can consider the way the program is using the addressing modes of the machine. In Figure 42, the dynamic frequencies of each of the addressing modes used in the \texttt{diff} program are presented. The nomenclature used is that taken from the VAX architecture manual; hence, we should
Figure 41. Comparison of I-cache machine with set sizes 2 and 4.

explain that indexed as an addressing mode is different than the mode which we have been calling indexed. In VAX terminology, indexed refers to a powerful addressing mode that can be used to achieve double indexing for byte, word, long or quad values. What we have been referring to as indexed is represented here by the byte, word, and long displacement modes. The modes that use displacements of greater than 13 bits (the number available for offsets on the RISC chip) account for about 13% of the addresses computed. Byte displacements, on the other hand, account for 27% of dynamic frequency of addresses. Other programs that we have instrumented have very different addressing mode frequencies.
6.12. Summary of I-Cache Results

From the empirical results presented above, we see that it is possible to exploit several features of complex instruction sets at the architectural level of a machine. These improvements are attained without modification to the compilers used, even though we strongly suspect that a very high-level machine with compilers that were aware of the existence of high efficient complex instructions would offer even more performance.

Based on these results, it is tempting to claim that complex instruction sets are better than simple ones. A more accurate statement is that complex instruction sets are less architecturally limited in performance than simplified ones. One can achieve better performance using a complex instruction set, but it is unlikely that a system such as that suggested here could be put on a single chip today, although it is useful to have an idea of what kind of architectural improvements could be made given the capability that true VLSI promises. The fact that the performance of the simplified instruction set machine is
greater than the 11/780 implementation says something about the cost effectiveness of the simplified instruction set approach for supporting languages similar to those considered here. This is not to say that the VAX-11/780 implementation is naive. We have conveniently not been bothered with considerations of cost or restartability of instructions.

When considering support for languages such as LISP and PROLOG, the situation is somewhat different. These languages are very demanding in terms of run-time environment which leads us to suspect that more direct support at the machine level would offer a substantial improvement over the usual implementations on conventional machines. Studies we have done have shown that the dynamic incidence of so-called overhead instructions in LISP is more than twice that of C code on the VAX, and nearly half of the activity of the program. By overhead instructions, we mean those which move data to set up for function calls and which move computation results to locations where they can be used. All of this indicates that efficient interpretation of these languages will probably derive great benefit from hardware support different from what is provided on the VAX. Further, the simplified approach to this problem as implemented by the SCHEME chip [45] supports a variant of LISP that is lexically rather than dynamically scoped for reasons of simplification of the chip. This indicates that support for LISP and PROLOG in their current form will be best carried out with an even more complex instruction set than those available now.
7. EXPLOITING HIGHER-LEVEL SEMANTICS

In the previous chapter we improved program performance by increasing the level of complexity of the instruction set in a bottom-up fashion, by combining low-level operations into larger sets of operations that comprise the instruction in the I-cache. Here we will consider increasing the level of the machine via a top-down approach. We consider the specification of what the program must do, and then choose an algorithm that is suitable for that task. This algorithm, if put into hardware, results in a fairly high-level machine, which performs better than merely implementing the same algorithm from the primitives (instructions) found on a conventional machine. The algorithms we will implement are used in support of a high-level language rather than being the end computation desired. Hence, this technique is legitimately applicable to the design of high performance machines supporting high-level languages.

The approach of isolating a frequent task and providing specialized support for it can improve performance in two ways. First, the task can be implemented with a faster algorithm than is feasible in a serial machine. Second, with lower-level support for a task, the overhead of interpreting a large number of very small tokens (instructions) is eliminated, even if the same algorithm is used. As an example of the first kind of improvement, consider the implementation of floating-point arithmetic and, in particular, the normalization operation. If conventional machine instructions are used, it can require a loop with a conditional to shift the characteristic to the normalized position. If a normalize instruction is implemented in microcode, then it becomes feasible to use a different algorithm for the normalization. Namely, we can just count zeros, and shift by that amount. With an encoder and a bit resolution shifter, this operation can be carried out in a single minor cycle. This algorithm differs from the previous looping method in that the test for leading zeros is done in parallel (by the encoder).

7.1. Algorithm Issues

As an example of how the same algorithm implemented as a primitive can be faster due to decode amortization, consider a pattern matching instruction. In [46], a comparison of several
implementations of a pattern matching instruction is made which indicates that microcoding the instruction can result in a savings of more than a factor of four over the same algorithm coded in machine instructions. This is a result of better utilization of the hardware since the decode for the individual instructions is eliminated. In fact, this microcoded implementation on a machine with cycle time of 320 ns is still faster by a factor of two than a machine with cycle time of 200 ns using the same algorithm implemented in the native instruction set of the machine. As expected, the variation of the algorithm is the more critical optimization, and we will exploit this technique in the following.

One additional enhancement to this choice of algorithm approach is to vary the algorithm over time, depending on the data. This is intuitively what is meant by dynamic interpretation. Suppose that we have a program graph $P$ which at some point can take one of two different branches. If it is the case that the results of the computation are identical in both cases, and the only difference in state of the system is that the time required for the branches is different, then varying which branch is taken is a dynamic interpretation of the program. This represents a performance enhancement if the two branches have the property that the optimal branch depends on the character of the data the program is manipulating. This is identical to the procedure for generating machine structures that was proposed in Chapter 3, except that the parameters for generation were assigned statically. If instead one allows for a dynamic variation of the lifetimes (for example) of the basic processes which are dynamically instantiated, and if the instantiation policy is determined at run time by the hardware, we have a dynamic interpreter which can, in principle, track the behavior of an algorithm.

In principle then, we want to dynamically change parameters in the process instantiation policy so that the overall performance of the program is improved compared to statically setting this policy. In practice, there are three major problems that must be overcome for this approach to be of any value at all.

data dependence
The operations that we want to accelerate must exhibit some known data dependence so that alternatives can be chosen.

prediction
We must be able to predict which of the methods will best fit the current state of the data.
algorithm switch

It must be inexpensive to change, at run time, which algorithm is being used.

Each of these criteria are more easily satisfied if they are considered with respect to relatively high order operations, since the further from primitives we are, the easier it is to consider alternative algorithms for a particular task. As an example of how this technique becomes more difficult when we consider improving low-level operations, consider a simple integer add. The data dependence of integer addition is such that for some arguments, say zero and one, there is no carry generated, and the computation can be carried out in a single gate delay. Other operands, such as 0111...1 and 0011...1 require that a carry be propagated along the entire length of the operands. If no carry lookahead is used, this can take many times longer than the single gate delay required in the first case. Hence our first point, that of data dependence is satisfied. The second and third points are not usefully satisfied, however, since choosing the best algorithm between a no carry exclusive or and a full carry lookahead is as difficult as forming the sum using the worst case carry lookahead method. Hence, in practice the worst case time is assumed, and all adds are carried out in the same amount of time, even though some would finish sooner if it were known that carries would not be generated.

The addition example is a case in which the method fails because the operation being optimized is at too low a level relative to what is feasible to implement directly in hardware. There are actually three cases in which the automatic run-time variation of the algorithm is inappropriate.

(1) The level of the algorithm that is being improved is such that any evaluation of its performance is as expensive as the computation itself.

(2) The performance of the algorithm does not vary for the data encountered.

(3) The performance does vary, but the variation is known at compile time and the appropriate algorithm can be explicitly predicted at compile time.

The first point leads us to believe that any improvement using these techniques will have to rely on a high-level computation or instruction. The second and third concern increasingly interesting performance dependence on the data and, in each case, static compile time choice of the algorithm is
appropriate. Hence the only time that run-time dynamic interpretation is useful is when it is not known at compile time what the character of the data is going to be, and when given a set of data, the optimal algorithm for that data can be inferred.

7.2. Probabilistic Considerations

We can take a more abstract view of this situation by considering the process graph of the computation and how the path through the graph is affected by the data. The graph in Figure 43 is a computation that is amenable to dynamic interpretation in the sense indicated above if we assume that whether branch A or B is taken, the results of the execution are the same. The performance of this algorithm is determined by the branches at A and B, and at A1 and A2. Given that each state is equal time, the possible paths are (B), (A,A1), and (A,A2), with respective execution times (for the one trip through the graph) of 7, 8 and 6. Given the probabilities of these branches, we can compute the expected performance of the system as

$$3 + 4P1 + (1-P1) \left[ 5P2 + 3(1-P2) \right]$$

where P1 is the probability of branch A and P2 is the probability of branch A1. Note that these probabilities need not be independent either of each other or of the data. If they are, then the system is very well-known and the performance impact of various improvements (shortening of path lengths) is known.

The interesting case occurs when the probabilities depend on the data, or equivalently, on the past history of the program. This kind of behavior is present to some degree in numerical analysis programs that attempt to adaptively alter their step size, or other parameters. The adaptation, however, is well-known at compile time and is based on the mathematical properties of the program. Of more interest here is nonnumeric processing where it is common that such dependencies occur, and in such a fashion that it is impractical for the programmer to be concerned with the dynamic performance of the program. The programming environment we are alluding to is LISP, and an example we will soon
analyze is a program whose progress depends on the state of a database that is accessed via pattern matching operations. To better understand our claim that the performance of this program depends on its data in a fundamentally different way than do most numerical analysis programs, we will now compare a linear system solver program and a program that makes inferences based on a database that
is accessed via pattern matching. We begin with an informal discussion of the relative difficulty of the tasks and then present simulation and graph analysis that quantifies these differences.

7.3. A Numerical Analysis Example

Our example of a linear system solver is the same program that was used for the memory reference study reported in Chapter 3, and will be referred to as GAUSS. For the purpose of this example, we assume a system of order thirty. We will compare this program with MCPAM, a program written in LISP which makes inferences based on incomplete specification of a story. These inferences are made by searching a data base for patterns that to some degree match input text so that the program can infer, for example, that someone who has a car and who goes to a remote location may have used the car to travel. The solution in the case of GAUSS consists essentially of the inverse of a matrix. The MCPAM solution is a data base which has been modified to reflect the program's current understanding of the world and a list of inferences from the data base that explain the story the program was given. We want to examine two aspects of these programs.

(1) The amount of computation that must be done for state change toward solution, and,

(2) The degree to which a given quantum of state change (matrix element update or new data base entry, respectively) varies in its performance during the execution of the program.

In the case of GAUSS, we can assume that for a typical row operation we have to form the pivot element, and then operate on the row of an augmented matrix. This requires $2 \times 30$ operations for each row to effect a multiply and another 60 operations for the addition. Hence, we have a minimum of $(120 \text{ operations per row}) \times (1 \text{ row per } 30 \text{ entries}) = 4 \text{ operations per entry}$. Assuming overhead of a factor of two, we get two operations per byte for the inner loop of the computation.

7.4. Comparison with MCPAM

We can compare this to the MCPAM program. The system in GAUSS requires 3.6 kilobytes assuming 4 byte words to represent a system of order thirty. The data base used in MCPAM is represented as a list structure in human readable form. In 3.6 kilobytes we can store 45 lines of 80
characters each, which is small for a data base. MCPAM accesses this data base in a fashion that is not nearly as constant as the matrix manipulations mentioned above, but an average we have seen is approximately 20,000 operations to effect a 20-byte update of the data base. Hence, the progress of MCPAM is about 1000 operations per byte. The main point, which we will presently illustrate with simulation data and graphical analysis, is that while GAUSS makes consistent known progress toward a solution, MCPAM does not, and this fact makes analysis of MCPAM much more difficult. It is useful to relate this example to our previous comments on the floating-point normalization problem. On most machines and, hence, in GAUSS, the floating operations are carried out in essentially the same amount of time, regardless of the actual operands of the operation. This results in the timing of the basic operations being fixed. At a slightly higher level, the algorithm itself is essentially free from run-time data-dependent branches, as can be seen in the control flow graph of Figure 44. All of this contributes to the situation of being able to decompose the performance into a number of known cost loops. This is not the case for MCPAM.

The program control graph for a subset of MCPAM is shown in Figure 45, and the block titled *Predicted* is expanded in Figure 46. Here we can see that the program goes through a list of lists (the data base) and checks for a match of the predicted explanation with an element of the data base. This operation is complicated by the fact that the data base is constantly growing, by the need for an exhaustive search each time through the loop, and by the fact that the routine that implements the check for a data base hit is complicated. This search for a hit depends on a pattern matching routine that cannot be accurately modeled (performance wise) by its average execution time, since this time depends so unpredictably on its data. There is further complication due to the recursive nature of the routine. The first question then is whether or not we can predict, or track the behavior of, this program. If so, then we should be able to achieve better performance. We will see shortly how an observer system can be used to detect certain modes of operation of the target system and make policy based on that information. First, however, we turn to the behavior of GAUSS.
Figure 44. Control flow for GAUSS.
To illustrate the behavior of GAUSS, and to illustrate a graphical analysis technique, consider the graph model of GAUSS in Figure 44. This algorithm is cyclic, as defined in Chapter 2, so that we can apply the moving average for run time to the graph. From the graph, we can state that if the branch probabilities are known, the expected execution time is of the form

\[ T = kL \]
where $L$ is the time required to execute paths in the minor loop pertaining to finding the maximum in the column, and interchanging rows, and $k$ is the number of times the major loop is traversed. We can further analyze the graph of Figure 44 by considering the intervals between visits of the algorithm to node A in the figure. We form the instantaneous run time of the algorithm by carrying out the following steps.

(1) Define the set $P$ of all possible paths through the graph.

(2) Form the trace $I$ of states that are attained during the execution of the program.

(3) Form the moving average

Figure 46. Control flow for the routine Predicted.
\[ T_j \text{ where } T_j = \frac{1}{n} \sum_{i=j-n}^{j} X_i \]

where \( j \geq |P| \). The sequence \( T_j \) is useful since it gives a measure of the data dependence of the program performance. Letting \( n \) become large makes the values of \( T_j \) approach the average loop time, and letting \( n \) take on the value \( |P| \) results in the values \( T_j \) changing only when there is a data dependent change in the performance of the algorithm. We can now derive this sequence for GAUSS. This is the same moving average of abstract state that we defined in Chapter 2.

7.5. Comparison with a Pattern Matching Program

Rather than comparing GAUSS with the entire MCPAM program, we consider a simplified subset of the program. The example used is a pattern matching program that attempts to find a pattern in a text file. The pattern is specified in the following simple language:

1. dot, ".", matches any character.

2. star, "*", is read as zero or more occurrences of the previous character. For example, "b*" matches zero or more occurrences of the character "b" and ".*" matches zero or more characters.

3. Any other character matches itself.

Patterns are matched successively within a line. For example, the pattern ".*and.*" would match any line containing the word "and". This program, which is similar to, but has a far less complex pattern matcher than MCPAM, was instrumented and the average state, as defined in Chapter 2, was plotted. The behavior of this kind of computation can be contrasted with the much simpler case of GAUSS.

7.6. Semantic Locality

Figure 47 is an average state graph for the GAUSS run on a 30 by 30 system. In this graph, we can see that the GAUSS program is very predictable and regular. In fact, the behavior can be predicted at compile time, and is essentially independent of inputs. In contrast, the graphs of Figures 48 and 49 show greatly differing behavior even though they are average state graphs for the same program. The
difference is that the graph in Figure 48 considers environment binding for subroutine calls to be free. In Figure 49, the environment binding is included in the average state, making the current environment explicit. This behavior does not occur in GAUSS because there is no change in environment as the program runs. Note that the graph in Figure 49 is in some sense an integral of the graph in Figure 48. It is an integral in the sense that the regions of constant, or nearly constant slope of the environment binding graph, are regions of constant abstract state for the case where environment is ignored. Since this behavior is so data dependent, we can’t predict it at compile time, but we can detect it at run time, and exploit it in two ways.
Figure 48. Abstract state for Match without environment.

To take advantage of this behavior we have to detect it at run time. To do this we can form the moving average using a circuit similar to that in Figure 50. This circuit computes the moving average of the microaddress stream to approximate the behavior of the system as shown in Figure 48. The circuit relies on the fact that at time \( n \), the current moving average can be written

\[
\alpha_n = \frac{1}{N} \sum_{i=-N}^{n} \beta_i
\]

where \( \beta_i \) is the microaddress at step \( i \). Now the \( n+1 \) term can be formed as

\[
\alpha_{n+1} = ((\alpha_n \times N) - \alpha_{n-N} + \alpha_{n+1}) \times \frac{1}{N}.
\]
If $N$ is a power of two, then the multiplication and division are easy and one only has to subtract off the $n-N$ term from the sum and perform the appropriate shifts and adds. Since this algorithm requires a subtraction and an addition, the observer in Figure 50 is pipelined, allowing the average to be computed each minor cycle. This results in the average being computed three cycles after the current microcycle. The alternative is to conjecture an addition time that is half of a minor cycle time, but this isn't possible in a high performance machine. The current and last averages are then subtracted and the value derived gives a measure of the degree of locality that is being exhibited by the machine. To include the environment binding, we can include the occurrence of an environment binding (a kind of procedure call) as a counter that participates in the computation. This results in the graph of Figure 49 with the value trailing the machine by three cycles. The value in the latch can be used directly as a
Figure 50. Pipelined abstract state observer.

microcode address branch condition so the microcode can adapt to the behavior of the program. One could apply this technique at the macro instruction address level as well.

Having this degree of observability into the run-time behavior of the system allows us to improve the performance in several ways. We can detect the change of environment indicated in Figure 49 and inhibit write-through in systems with a write-through cache. We can also begin prefetch of the environment bindings for the program, if we are executing LISP. Other alternatives include prefetching of data pages. This last technique is useful primarily in the context of a database system, where a buffer cache would be ineffective due to its small size, relative to the amount of data in the database.

From this discussion it should be clear that these techniques are more easily applicable to algorithms that are specified at a fairly high-level, since at the lower levels a great deal of information is lost. This situation is similar to that of compiler optimizers. A great deal more can be done to optimize
a parse tree, or DAG representation of a program, than can be done with a peephole optimizer, for analogous reasons.
8. CONCLUSIONS

In this thesis we have dealt with many aspects of computer architecture. We have analyzed static and dynamic interpreters for instruction sets. We have made studies of realistic systems which have validated the methodology of abstract consideration coupled with detailed simulation, and which have provided insight as to the behavior and performance of these systems that would be unavailable without this detailed evaluation. We have shown that a concurrent object-based programming language is an important tool for such studies as well.

The most important statement of this research is that the abstract consideration of systems, using the process model proposed here, is useful in describing not only the structural characteristics of systems, but their performances as well. This theoretical view of systems has potential beyond what has been done here, however.

The stance we have taken here is that the general model of computer activity is that of the algorithm and, based on that, one could argue that a machine that is capable of very efficiently executing the primitive logical and arithmetical operations that are the basis of all computations can be made to execute any algorithm at high speed relative to a slower adaptive or dynamic machine. There are two fallacies in this argument. First, the algorithm carried out in a dynamic machine is by definition a parallel cooperation of an observer machine that detects how a computation is going, and which feeds back information that allows a more nearly optimal choice of method for subsequent operation. This kind of structuring (as concurrent processes) is not at all well supported by the hardware of typical highly pipelined numerical oriented machines and, hence, it is not practical to implement from user code. Second, and more important, the coding of a complex expert system in the assembly language of a conventional machine is far too complex. While it is feasible to expect programmers to make the abstraction from index registers and pointers to the implementation of multidimensional arrays, it is quite a different matter to expect them to be constantly aware of all aspects of manipulation of a programming environment that may at any time have hundreds of instantiations. It is quite unlikely that
progress in expert systems (for example) could have come as far as it has today if the only tool programmers had was FORTRAN. The mental gap is just too great. For these reasons, the support of non-numeric processing will be far better advanced by more direct implementation of systems supporting sophisticated environments and programming styles, such as LISP and PROLOG, than the alternative of attempting to map sophisticated concepts onto very simple hardware by hand. What we have shown here is that adaptive dynamic interpreters are a viable approach to providing this kind of support.

8.1. Future Work

The theory presented here can be extended in many directions. In terms of very pragmatic use, it would be quite useful to relate the mathematical model to the automatic generation of optimization problems. The dynamic nature of object creation in L1 is well-suited to this task and some relatively minor additions to that language would allow the run-time characteristics of processes to be evaluated. This allows one to consider a truly dynamic process interconnection structure that is driven primarily by the utilization of the constituent processes. This would allow one to formally and automatically deal with those problems that can be stated in terms of known tradeoffs.

In the area of abstract modeling, there is great potential for developing a notation that is tractable for considering architectures as related to the degree of data dependence they exhibit when considered as instruction set interpreters. The method of computing the moving average of the abstract state of a computation was introduced as a tool by which the balance and degree of performance data dependence of a system could be quantified, and the application of this technique, along with the automatic analysis alluded to above, would be quite interesting in the area of performance characterization of complicated algorithms such as those used in expert systems.

There is also a great potential in extending the abstract process model work. One thing that this research has shown is that taxonomies such as stack machine versus register machine are not very useful for determining which is better for a given task. The abstract process model is useful in evaluating a given system, but it is not yet sufficient to completely characterize architecture in a succinct fashion. Generality is not lacking, and utility will come with use, as in any scientific formalism.
REFERENCES


VITA

Richard Norton was born in the U. S. Canal Zone on September 22, 1951. He attended New Mexico State University, attaining the Bachelor of Individualized Studies, and a Master of Science degree in mathematics. He received a Master of Science degree in electrical engineering at the University of Illinois. Mr. Norton has worked at several laboratories and software houses, primarily in the area of system programming. He has presented papers at the International Conference on Parallel Processing in 1982 and at the International Conference on Computer Architecture in 1983. These were entitled, "Using Write Back Cache to Improve Multiprocessor Performance," and "Exploiting Higher Level Language Semantics," respectively.