MIRAGE: A MODEL FOR LATENCY IN COMMUNICATION

Joseph Dean Touch

A DISSERTATION

IN

COMPUTER AND INFORMATION SCIENCE

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.

1992

_____

David J. Farber
Supervisor of Dissertation

_____

Mitchell P. Marcus
Graduate Group Chair

DEDICATION

To my parents, Ralph B. and Filomena Touch, for whom education is always first.

ACKNOWLEDGMENTS

This work is supported by the Information Science and Technology Office of the Defense Advanced Research Projects Agency, under contract NAG-2-639, and by an AT&T Graduate Research Fellowship, grant #111349.

There are also many people who contributed to the development of this dissertation, and without whose help it would not have been possible.

My classmates at the University of Pennsylvania Department of Computer and Information Science provided active debates encouraged these ideas. Members of the Distributed Systems Lab, including John Shaffer, Anand Iyengar, Ming-Chit (Ivan) Tam, Brendan Traw, Jonathan Smith, and Amarnath Mukherjee discussed and debated these ideas. Daniel Kulp (Univ. Penn. Materials Science) provided fertile discussions on the physics analogs which provided the basis of Mirage (Appendix B). Other sections were directly helped by Jon Freeman (Appendix C), Christine Nakatani (Appendix G), Amarnath Mukherjee (branching-stream model equations, Chapter 2), Richard Gerber (communicability and Smyth's PowerDomains, Chapter 3), Phil Christie (Univ. Delaware) (violation of the Liouville theorem, Appendix D), and Stuart Frieberg (Univ. Wisconsin - Madison) (*isopotency* of Chapter 2).

This text was edited through the advice of many students and colleagues, including Diana D'Angelo, Ted Faber (Univ. Wisconsin - Madison), Brendan Traw, John Shaffer, and Nick Short, Jr. who provided extensive feedback. I owe Nick more than a few beers.

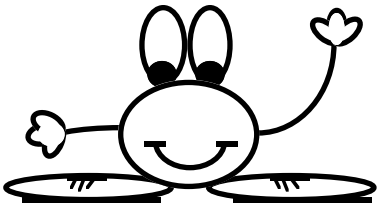
I also want to acknowledge a few of my mentors, most notably my first computer science teacher, John Mero, who was the first to introduce me to computers, and who was the first to give me the opportunity to study them at my own pace. I also would thank my undergraduate advisor Dr. Jack Beidler, who gave me the resources and opportunities to discover this science. Also, my first employer T. Russell Hsing, who is responsible for introducing me to this research, and my Ph.D. advisor, indirectly.

I also thank my dissertation committee members, for their participation, advice, and comments: David Sincoskie (Bell Communications Research), Ravi Sethi (AT&T Bell Labs), Jonathan Smith (Univ. Penn.), and the chair, Mitch Marcus (Univ. Penn.).

I want to thank my Ph.D. advisor, David Farber, for introducing me to the questions this dissertation investigates, for giving me the unique opportunity to investigate a new area of research on my own, and who supported this research to DARPA for a grant.

I also want to thank my friends for 'moral' support, including the Doctor's Club members Samuel Kulp D.V.M. (& Ph.D. 1993), Daniel Kulp (Ph.D. 1992), Scott Miller M.D., David Bradin (Esq., 1994), and Stuart Miller. I thank my friends Janice Luteran, Paula Luteran, Susan Alexander, Fran Weiss, and Nick Short, Jr., especially for contributing to the financial development of telephone companies everywhere. I also thank Kenneth M. Rubinstein, my first professional colleague (cartooning, 1974), and my oldest friend.

Finally, I want to thank my family, whose help and support made this work, and all other I may do, possible. My parents Ralph B. and Filomena Touch (yeah, yeah, 10%, I know), my brother Ralph F. Touch, Esq., my sister Suzanne M. Touch (M.D. 1993), and my grandparents 'Nonni' and 'Pop' Cianfrani, and 'Nonni' and 'Nonno' Touch. My father helped edit this dissertation, which was prepared on a Mac IIsi provided by Nonni Cianfrani (more jimmies, Nonni!).



ABSTRACT

MIRAGE: A Model for Latency in Communication

Author: Joseph Dean Touch

Supervisor: David J. Farber

Mirage is an abstract model for the design and analysis of high speed wide area network (WAN) protocols. It examines the effects of latency on communication, and indicates that *information separation* is the distinguishing characteristic of gigabit WANs. Existing protocols will exhibit performance failures due to an inability to accommodate imprecision in the remote state. The name *Mirage* denotes the difficulty with latent communication, namely nodes never really "see" each other precisely; rather, they work with (and around) the *mirages* which high speed and fixed latency conjure before them.

This dissertation describes the Mirage abstract model as an extended finite state machine that accommodates imprecision through the use of multiple simultaneous states and state space volume transformations. We introduce *guarded messages*, to accommodate nondeterministic data streams, and *communicability*, the upper bound on communication, given fixed latency and state predictability. Mirage demonstrates how excess bandwidth can be used to accommodate latency, and shows the bounds on latency constrained communication. Supplemental discussions includes consider Mirage as an extension of Shannon's communication theory, and compare it to physics analogs.

Mirage was applied to the Network Time Protocol (NTP), to demonstrate its use and exemplify its abstract components. We show the equivalence between variation in state space imprecision and variation in transmission latency. Several 'optional' components of the NTP specification are shown to be required, and layering is violated in permitting sender anticipation.

To show the model's advantages, Mirage was applied to processor - memory interaction as a protocol, calling the result *μ–Net* (MicroNet). Using anticipation, we develop a novel interface which achieves a hit rate equivalent to that of a 50K byte cache, using 400 bytes of storage. μ–Net complements conventional cache techniques, especially where communication latency is the limiting factor in code execution, and where excess communication bandwidth is available. Dynamic traces measured the latency accommodation possible by the various implementation versions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

Chapter 4 – A Mirage of NTP 72

PREFACE

The following is a section description of this dissertation. It includes the chapters, as well as the appendices. The appendices are auxilliary discussions or digressions, which are intended to indicate all discussions applicable to this work, regardless of development status.

Chapter 1 — Introduction

The introduction defines the problems that Mirage addresses. The original goal was to address anticipated protocol degradation in gigabit networks. We conclude that existing protocols would fail only in gigabit wide area networks, and that fixed latency is the real problem to be addressed. The protocol model developed addresses issues of latency in communication, where latency is fixed and known, and remote state evolves according to known state expansion functions.

Chapter 2 — The Mirage Model

The formal model based on state space subset transformations is presented. We introduce *guarded messages* and *communicability.* The model defines communicability in the presence of latency, and relates stability to communicability and the variability of a remote state. A notation of these transformations based on set notation is presented in Appendix E.

Chapter 3 — Prior Work

The discussion of prior work focuses on cybernetics and control theory, and abstract models of communication, most notably Shannon’s theory of communication. Petri Nets, finite state transition models, and temporal logic are also discussed. Other prior work includes distributed systems and databases, especially common knowledge, quorum

consensus, and client/server models. Similar protocol methods include VMTP, XTP, NetBlt, virtual clocks, flow protocols, Delta-t, URP, and TP++.

Chapter 4 — A Mirage of the Network Time Protocol

We apply the Mirage model to an existing protocol, the Network Time Protocol, to demonstrate the modeling method and exemplify Mirage's abstract components. This includes demonstrating the isomorphism between variability in state precision and variability in transmission latency, thus extending the domain where Mirage applies. We conclude that several 'optional' components of NTP are required for modeling, and thus should be required in the protocol, e.g., the logical clock and peer dispersion and data filter algorithms.

Chapter 5 — μ–Net

NTP was insufficient to model the unconventional aspects of Mirage, notably those which address latency compensation. We use the Mirage model to develop a processor-memory interface which anticipates opcode memory requests. This interface, called μ–Net (MicroNet), extends the conventional memory interface and is compatible with (and complementary to) a processor opcode cache. There are various degrees of implementation of μ–Net, whose complexity increases as anticipation handles larger subsets of opcodes. μ–Net reduces access latency across the interface, through the use of local storage and (in some cases) higher bandwidth requirements on memory.

Chapter 6 — μ-Net under a μ-Scope

Detailed measurements of opcode executions on a SPARC CPU indicate the effectiveness of the μ–Net designs. These measurements also specify the design parameters and describe the feasibility of the various implementations. For example, anticipating only fixed-jump and recursion opcodes achieves a predictive success rate equivalent to that of a 50K byte cache, with only 100 addresses (i.e., 400 bytes) of storage, with similar memory access load. The measurement tool, μ–Scope, is described in Appendix H.

Chapter 7 — Conclusions

Our goal was to design a model for latency in communication. One initial conclusion was that communication implies latency, and is defined as the sharing of state between temporally separated entities. Mirage has interesting properties in itself and elicits a novel view of existing protocols (NTP) and domains where telecommunication protocols are not normally applied (μ−Net). Future work includes a more formal comparison to Shannon's work (Appendix A), elaboration of the TreeStack data structure (Appendix G), and a complete implementation of μ−Net.

Chapter 8 — Bibliography

List of references.

Appendix A — Mirage & Shannon

Mirage can also be considered a temporal extension to Shannon’s communication theory. In Shannon’s work, encoding trades error for latency. Mirage demonstrates a complement of this theory, trading latency for imprecision in state (error). This is a discussion of the ways in which Mirage is an extension to Shannon’s theory, and the ways in which it is a complement to it.

Appendix B — Mirage & Physics

Mirage is based on principles from thermodynamics, statistical physics, General Relativity, and quantum physics. This is a discussion of some similarities noticed while designing the model.

Appendix C — Upper bound

In the Mirage model, both discrete and continuous equations for communicability were presented. This is a proof that the continuous equation is indeed an upper bound on the continuous equation, as claimed in Chapter 2.

Appendix D — The Liouville Theorem

The thermodynamic analogs in Mirage indicate an apparent violation of the Liouville Theorem, which restricts the extent to which state can vary over time. Here we explain how information and the open system of Mirage permit such an apparent contradiction.

Appendix E — Mirage in Set Notation

Mirage is described as an extension to state transition models, based on a notation of finite state subset transformations. Bounds are described, based on these equations and existing properties of communication.

Appendix F — Mirage & Petri Nets

Mirage is described as an extension to state transition models, but can be considered in terms of extensions to other models as well. We applied the Mirage principles to timed Petri Nets, and show the Petri Net transformations and equivalences of interesting components of our model.

Appendix G — The TreeStack

The TreeStack data structure is described in detail, as are the mechanisms for transformations required in the implementation of the Converger and Diverger components of the Code Pump of μ–Net (Chapter 5). The TreeStack manages a state space that permits multiple alternates and state space recursion simultaneously. It also reduces to a simple stack if recursion is prohibited, and to a simple tree if multiple alternates are prohibited.

Appendix H — μ–Scope Methods

Examining the feasibility of implementations and specification of the expected speedup required detailed dynamic opcode execution measurements, which were not possible using existing tools, such as PIXIE or SPIXTOOLS. μ–Scope (MicroScope) was

developed to make the required measurements on existing compiled code, with an execution speed between 3x and 7x slower than unmeasured code.

In Retrospect

Part of the evaluation of the work should include a description of the changes that would have been considered, had the conclusions been known from the beginning. As a note of comparison, this dissertation does not substantially differ from the dissertation proposal.

We initially intended to examine flow protocols in addition to NTP and μ-Net, but decided that such an analysis would not assist in the description of the components of Mirage. Future research may focus on analyses of these protocols.

The μ-Net research was originally intended to be an investigation into the Mirage model implications on distributed shared memory. We chose processor-memory interaction instead, noting that processor I/O requirements are large enough to require gigabit bandwidths. Conversion of processors to optical pin-outs increases the available bandwidth, but also increases the access latency across the inter-chip boundary, due to the cascaded parallel-serial / serial-parallel conversions. Conventional caching collapses in high latency situations because the cache may be off chip and would incur the same conversion latencies as regular memory. Focusing on distributed shared code memory also permitted the description of temporal transformations sufficient to facilitate communicability. This focus indicates that Harvard architectures may be better suited to anticipation than arbitrary architectures, especially those that permit self-modifying code.

Finally, we note that the μ-Net research results have implications on distributed shared memory, in suggesting similar mechanisms for proactive distributed shared memory. Future investigations may also examine these implications.

CHAPTER 1

Introduction

Mirage is an abstract model for the design and analysis of protocols, for application in high latency domains. The model is an extension of a finite state machine that represents states as sets, rather than as single values. Each node of the network expresses local state by a single point in state space and remote state (the state of a remote node) as a set of possible states.

The model constrains the state space imprecision (i.e., set size), based on the amount of information in transit. It also specifies conditions of stability of a system (i.e., permitting controlled interaction) given the latency and communication bandwidth available.

This is a description of the abstract Mirage model using streams of information and an extended state transition model. The model is applied to the Network Time Protocol [Mi89a], to illustrate the abstract concepts of the model, and as an example of protocol analysis using the model. Mirage is also applied to protocol design, using the domain of processor-memory interaction as a protocol paradigm.

1.1. Background

This dissertation represents a departure from the conventional studies of high speed protocols. An explanation of its origins and evolution into an abstract model may thus be useful.

Various documents and individuals claim that [Gr87], [Ra87], [Po88], [Mi90a], [Pa90a], [Pa90b]:

> Protocols will fail at gigabit speeds, requiring clean-sheet approach.

This statement implies that existing protocols will 'fail' due to improper design, and that a revolutionary approach will succeed, whereas evolutionary ones will not. Several questions arise from this pronouncement:

> Why will they fail? What is so special about these speeds?
>
> How will they fail? What does failure mean?

Failure here is a performance issue. Failure is usually considered a correctness issue, but if a protocol must achieve a certain level of throughput to be correct, then performance is a correctness criterion [St88]. The protocol *fails* to achieve a required level of performance, given the capability of the channel. There are other questions to consider:

> When do protocols fail? At what speeds will this become a problem?
>
> Assuming a failure exists, can it be avoided?

Latency, rather than speed, is the real issue. Latency remains constant as communication speeds increase. The result is an increase in the number of bits in transit, i.e., bit latency, between communicating entities; bit latency is information separation.

> Nodes are not separated in space; they are ***separated in information***.

Two networks that have the same information separation, yet different operating speeds and spatial distances, can be considered equivalent. The following question remains:

> What makes high speed protocols different from low speed protocols?

Nothing has changed except technology. An isomorphism exists between high speed LANs and low speed WANs; thus *newer* (faster LAN) networks can use *older* (slower WAN) protocols. Using this isomorphism, experimental networks without high bit latency can yield methodologies that will not apply to WANs. This has not yet been a problem, because the bit latency in WANs was not large if compared to node buffer size (we will address this later).

A protocol cannot differentiate the relative spatial scales of LANs and WANs, provided the corresponding transmission rates yield identical bit latencies. Alternately:

> Is a protocol affected by the speed at which it runs?

No, except for implementation considerations[1]. A protocol is affected only by the amount of data in transit; absolute speed has no other affect. Latency is the real issue, that is as bits in transit (bit-latency), rather than in units of time or space alone. Bit-latency is the unit measure of *information separation.*

This discourse leads to one final question:

> How can a protocol be characterized to address these questions?

Protocols are often viewed as Petri Nets or finite state machines (FSMs). These models are awkward and inadequate if bit-latency is the determining characteristic. Mirage extends the FSM model to incorporate imprecision of state, thus modeling the effects of latency.

Furthermore, current paradigms (i.e., the ISO stack) model the design structure rather than the design space of protocols. Many protocols are designed by implementation alone. Considering the space of all protocols, only arbitrary points in the space (i.e., instances) are being examined. Each axis of this space is a continuum with tradeoffs.

This space needs to be characterized, but not just for testing existing protocols; the space could suggest new protocols, as new combinations of characteristics of existing protocols. The characteristics of this space of all protocols is inferred by existing instances, but the space has not been well defined. Before examining the characteristics of such a space, there are remaining questions:

---

[1]Such implementation issues can be considerable, but are not substantial. In other words, such issues require attention, but not necessarily new methodologies.

Is there a space of all possible protocols?

Is there a way of examining part of this space in a useful way?

This dissertation provides a way to look at protocols, a model with which to test, design, and measure protocols, i.e., to examine this space in a more general fashion. We will show the following:

1) That existing protocols can exhibit low channel utilization in high bit-latency domains.

2) The advantages to modeling the endpoints of the link, rather than the channel itself.

3) Why the sender should anticipate the receiver.

4) How this results in a tradeoff between error and bit-latency.

5) Why achieving increased channel utilization necessitates avoiding layered protocols, i.e., why we need to look inside packets.

6) There is a limit to how well we can get around things, which is a function of:

   a) variability in the receiver state
   b) bit-latency
   c) power of the sender to accommodate this variability
   d) ability of the channel to accommodate this variability

We have made two other assumptions here[1], that all protocols exhibit performance failures with high information separation because they are similar, and new protocols can exist which do not fail. To understand the reasons for this claim, and to begin to answer the questions above, we need to start at the beginning:

What *is* a protocol?

[1]The other assumptions to this point are that bit-latency is the central issue, and that LAN, MAN, and WAN networks are distinguished primarily by physical scale (i.e., not topology).

1.2. What is a protocol?

The term *communication* is not very easy to define with existing texts. Canonical course textbooks are not helpful; [Ta88] doesn't define it, nor does [Sh63] or [Be87]. A particularly bad example is:

> "data transmission" [Ha88a]

This implies that communication is 'sending bits across a distance.' Although accurate, this doesn't explain much. Reference texts define communication as:

> "exchange of information for the purpose of cooperative action" [St87]

This definition implies that computers have no internal communication. Furthermore, cooperative action is not strictly required; 'Byzantine generals' exhibit uncooperative communication [Pe80]. Another reference lists:

> "transmission of information from one point to another" [Ja84]

This definition doesn't clarify communication either. English lexicography defines communication as:

> "a process by which meanings are exchanged between individuals through a common system of symbols" [Go86]

> "the imparting, conveying, or exchange of ideas, knowledge, information" [Si89]

These definitions are more descriptive, but not as general as we desire. Original texts in communication theory define communication as:

> "all procedures by which one (entity) can affect another" [Sh63]

This is general, but the term 'procedures' is undefined.

We define communication here as:

> **COMMUNICATION:** logically shared state among entities which do not physically share state [To90a]

Sharing of state provides all the earlier definition characteristics as consequences. The separation of the parties is specified by the distinction between *logical* and *physical* sharing of state. Logical separation is an abstraction, used in programming languages

(environment scope), whereas physical separation requires temporal separation, which cannot be abstracted away.

Now that we have arrived at our definition of communication, we are prepared to define a protocol. Again, a particularly bad example is:

> "rules and conventions used in a layer-N to layer-N communication, or a set of rules governing the format and meaning of the frames, packets, or messages that are exchanged by the peer entities within a layer" [Ta88]

A better definition incorporates the notion of a protocol as a mechanism that facilitates communication:

> "a set of rules formulated to control the exchange of data between two communicating parties" [Ha88a]

The more general definition is:

> "agreement between two peer entities on the means of communication" [Sh63]

We prefer the following definition:

> **PROTOCOL:** a method for maintaining shared state among *information separated* entities[To90a]

We define communication as shared state, so that two parties communicate only if they agree on some shared information. Conventionally this state is considered external to the entities (i.e., it is referred to as the state of the channel) , as in Shannon's theory of communication [Sh63]. The shared state is a portion of the total state of each entity, so the receiver shares the communicated component of the sender's state. A protocol is a mechanism for maintaining shared state. Because shared state is the basis for communication as we define it, a protocol is thus a method for providing communication.

We make no assumptions about the communicating parties; communication is the abstract process of two entities sharing state. We assume only that communicating entities are necessarily separated in time or space; actually, separation in time is our only criterion, because separation in space implies separation in time[1]. If two entities are not separated in time, they are not distinct to the point of requiring communication.

Communication is required if a separation of time exists. Bandwidth is the

---

[1]Given a separation in space, we define the separation in time as the minimum time required for interaction, which is the time it takes information to traverse the spatial separation.

capacity to transfer encodings across this separation [Sh63]. Latency measures the separation; if the latency is zero, there is no separation. Also, because physical separation implies logical separation, we can more generally define communication as:

COMMUNICATION: logically shared state among *information separated* entities

We make no assumptions about the state of an entity (i.e., a node), nor about the state that is shared. A state can be as little as the shared status of a protocol (i.e., current window number, current connection information, etc.), or as much as the contents of an entire file transferred. With as little as one bit of shared state maintained by a protocol, communication is possible (e.g., by an alternating bit protocol).

1.2.1. Communication as state concurrence

Our definition of 'communication' is referred to elsewhere in the literature as 'connection management', or the shared state which governs the transfer of data. We treat transferred data as the important component of the shared state, rather than "that which shared state facilitates." A protocol is a mechanism or method for maintaining communication, so what we call a 'protocol' others call 'communication'.

Algorithms and protocols are distinguished by information separation; an algorithm does not include interaction between agents separated by information distance[1]. An algorithm that spans an information partition is conventionally called 'distributed,' and requires an underlying mechanism for communication among its components, thus distinguishing between the algorithm part and the communication part [Be87]. On an individual node, an protocol is implemented by an algorithm.

1.3. The need for a new model

One characteristic of gigabit wide-area networks that differentiates them from their slower or more proximal counterparts is the unprecedented amount of latent data ('in the pipe'). We believe that fixed latency (forced by physics), combined with increasing data

[1]An algorithm using procedure calls is not a protocol, unless the procedure calls are remote (separated in time from the algorithm). The global variables and passed arguments restrict the scope visible to a procedure, but do not imply a time lag.

transmission rates, will result in network inefficiencies as bandwidth and network sizes scale, due to the performance failure of existing protocols [Mi90a], [Pa90b], [Pa90a]. The Mirage model describes the conditions of this performance failure and helps determine the relevant issues in designing protocols for these new domains.

The name **Mirage** denotes the difficulty with high-speed, wide-area network protocols, in that by the time requested information arrives, it may no longer be accurate. Nodes in a high-speed network never really "see" each other precisely; rather, they work with (and around) the *mirages* which high speed and fixed latency conjure before them.

The predicted performance failure of current protocols in the gigabit wide-area domain has been used to justify the search for new protocol implementations. Most of these efforts focus on the complexity of existing implementations and executing these protocols at gigabit rates, seeking simpler protocols or more efficient implementations (e.g., XTP [Ch88a], NetBlt [Cl87], VMTP [Ch88b]). Instead, we seek to understand the distinguishing characteristic(s) of gigabit, wide-area networks, so protocols developed with this model will work in these domains by design, rather than accommodation.

This research is based on some analogies from physics. Communication theory already incorporates physics analogs, most notably that between information and negative entropy; here we investigate other analogies as well. The Mirage model, of state space volume transformations and guarded messages, is an attempt to incorporate the concept of imprecision evident in quantum models into communication protocol analysis.

We need to determine the salient feature of gigabit, wide-area networks that may prevent existing protocols from operating efficiently. The primary problem is that latency does not scale with speed increases, causing conventional protocols to decrease effective channel utilization, because many of these protocols were designed for file transfer based on sliding-window flow control (e.g., TCP [Po81a], NetBlt [Cl87]). We expect channel utilization to drop as latency increases because existing protocols will not be able to predict the amount of data sufficient to fill the round trip delay at these rates. We will suggest a model where data prediction permits indeterminism, where the round trip time is used to send sets of potentially useful data, rather than only data that was explicitly requested.

The Mirage model describes the effects of latency on communication, permitting the analysis of gigabit wide area network protocols, and showing how increased performance can be achieved. Mirage accounts for latency, but also includes conventional domains as degenerate cases where latency is assumed to be insignificant.

One cause for the deterioration of efficiency in existing protocols is that they use a point model of communication, based on Shannon's communication theory [Sh63]. This theory accounts for channel error by sequence encoding; higher channel errors requiring encoding over longer sequences. The result is a tradeoff between error and the latency of encoding. Mirage proposes a view where latency can be tolerated by accepting information imprecision (a measured form of error). Information about remote nodes, formerly precise points in state space, become imprecise volumes in state space. Mirage defines communication operations as transformations of these state space volumes, and incorporates the effects of time in these transformations.

1.4. What has changed?

In assessing the requirements of our new model, we first examine the distinguishing characteristics of the model's domain. Some suggest that existing protocols and protocol models will become inefficient as communication rates increase to gigabit rates [Mi90a], [Pa90a]. There are implementation challenges in scaling protocol processing rate and host bandwidth to accommodate the increased network speeds, but transmission latency does not scale, and cannot as directly be compensated.

In high speed protocols, an 'increase in latency' is usually named as the problem, although latency is a constant. For protocol operation, the important characteristic is information distance, or how many bits separate two communicating entities (bandwidth * delay product). We first show how a gigabit LAN is equivalent to a 400 Kilobit WAN, provided that we treat time as relative rather than absolute.

Changes in communication rates are equivalent to certain changes in scale. Increasing the bit rate foreshortens the bit length as it travels the wire (Figures 1.1, 1.2). Because bits travel at a constant speed, this allows more bits to be in transit at a given time.

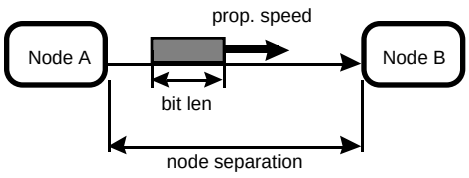


FIGURE 1.1
Network rate

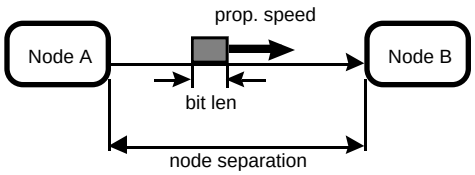


FIGURE 1.2
Increased rate as bit foreshortening

The foreshortening of bits is isomorphic to the original bits traveling faster, over a correspondingly longer distance (Figure 1.3). In this latter view, *"stepping on the gas" moves the destination further away*. Thus latency becomes a problem at high speeds; latency remains constant to external viewers, but distances (and latency) grow in the reference frame of the bit (i.e., 'bit times' to the destination)[1].

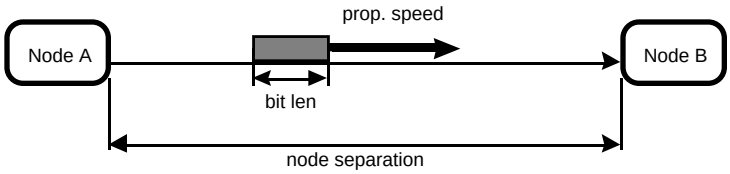


FIGURE 1.3
Figure 1.1, as it appears to the bits in transit

Finally, there is no absolute time reference in these networks, because existing protocols are independent of absolute time. There is no way to distinguish a protocol running on a network from that same protocol running on a network that is 10x larger, and whose data rate is 10x slower. A WAN (2500 miles) can be converted to a LAN (1 mile) by shrinking the network by 1/2500 and increasing the transmission rate equivalently, resulting in the same scale in information distance (bits in the pipe). We can treat a WAN running at 400 Kilobits as a LAN operating at a Gigabit, if we scale it appropriately.

The processing in gigabit LANs is accommodated by a combination of faster technology, increased parallelism, and more efficient algorithms[2]. A 400 Kilobit WAN

---

[1]This is a relativistic analogy. Information separation space dilation as transmission speed increases is analogous to time dilation as acceleration increases in General Relativity.

[2]These are order of magnitude arguments only.

was supported by TCP by the early 1980's (1.5 Megabits by 1985), when processor rates were 5 MHz, 16 bits wide. By porting TCP to a very small LAN (two back-to-back processors), on a CRAY Y-MP (167 MHz, 64 bits), technology provided an 130x processor speedup (increased word size and clock rates), and improved processing and limited implementations can support an additional factor of 15x speedup. The result is a nearly 2,000x speedup using these technology advances and optimizations alone, so that the previous 400 Kilobit WAN protocols can be used in LANs at 800 Megabits (800 Megabit TCP has been implemented [Ni91])[1].

Building gigabit LANs therefore is an issue of scaling processing speed (if possible), not of changing protocol design, disregarding the difficult electronics problems. Nothing has changed in that case, except the time scale relative to the distance scale (propagation latency). Absolute transmission rates indicate little of the characteristic of a network; information separation is a better measure of differences that are not merely technological.

Assuming existing protocols work at current network speeds, MANs need to exceed 133 Gigabits per second, and LANs 2 Terabits per second, before either exhibits the behavior of WANs operating at 1 Gigabit per second (Figure 1.4). A gigabit LAN is equivalent in this sense to a 400 Kbps WAN, where NCP operated -- it is no surprise that, in this environment, very lightweight protocols based on single packet transfer suffice, as they are the modern analog of NCP [Ca70]. Similarly, a gigabit MAN is equivalent to a 10 Mbps WAN, where TCP operates. This assumes that, in each case, we increase the clock rate or processor width of the MAN or LAN to accommodate the change in scale. In these cases, nothing has changed.

Figure 1.4 compares the characteristics of LANs, MANs, and WANs. We assume that scale does not also imply topology. Network type is denoted by vertical gray areas, indicating approximate network scale. The dashed horizontal denotes the gigabit threshold. Actual rates are plotted as points (■), and equivalent rates are connected by gray sloped lines. Latency determines rate equivalence; the slope indicates the contour lines of equal latency, in information separation units (bit-latency, i.e., bandwidth * delay product).

---

[1]Test cases were limited to 64K byte 'network' packets, software loopback mode (testing the TCP implementation only, with zero latency), 1.5M byte user packets.

WAN speeds can be compared to those of equivalently bit-latent MANs and LANs (Table 1.1).

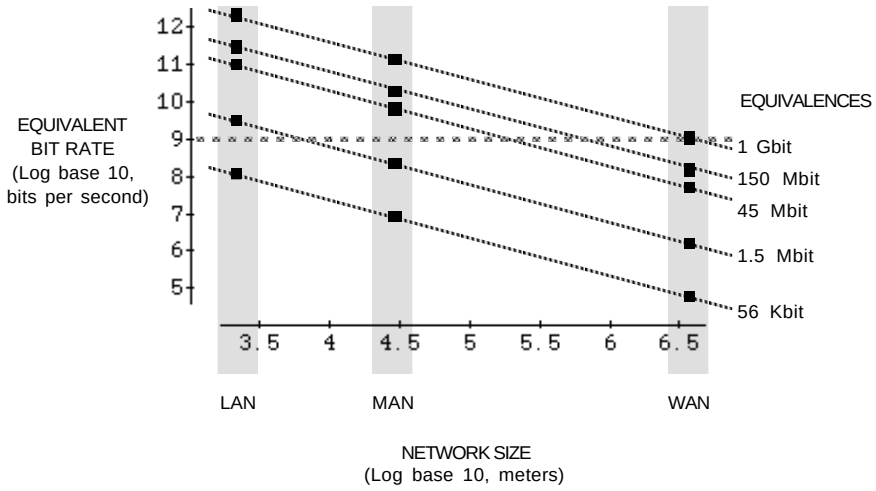


FIGURE 1.4
Network size and speed equivalences[1]

Year	WAN (4,000 Km)	Equiv. MAN (30 Km)	Equiv. LAN (2 Km)
1970	56 Kbps	7.5 Mbps	112 Mbps
1986	1.5 Mbps	200 Mbps	3 Gbps
1990	45 Mbps	6 Gbps	90 Gbps
1995	~150 Mbps[2]	~20 Gbps	~300 Gbps
2000	~1 Gbps	~133 Gbps	~2 Tbps

TABLE 1.1
Network size and speed equivalences

[1]This assumes that LAN, MAN, and WAN networks differ mainly by internodal distance. There may be additional topological implications to these classes; they are not considered here.

[2]'~' indicates projected estimate.

So, what has changed by going to 1 Gigabit per second? In the LAN and MAN cases, relatively nothing has changed. Only in the WAN case does speed cause a relative latency problem; there the bit latency exceeds that of any existing protocol domain, with the possible exception of satellite networks. Unfortunately, satellite protocol models may not be useful as WAN paradigms, because these assume topological constraints (central routing and control) which do not apply in WANs. The real change is the amount of information separation, and it is by this measure that protocols can be characterized, as in Figure 1.4.

The round trip data can be compared to the average size of the node computers in a network[1]. Bit-latency had previously been 1-2 orders of magnitude smaller than the node memory, whereas proposed wide-area gigabit networks will cause this gap to narrow considerably (Figure 1.5, from estimates in Table 1.2).

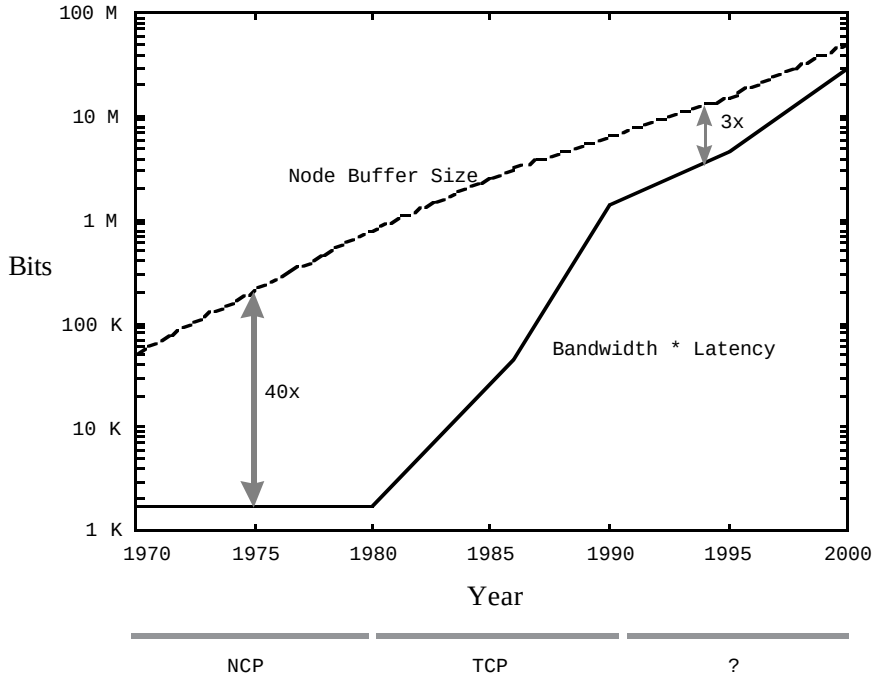


FIGURE 1.5
Node buffer size (memory) vs. information separation (bit -latency)

[1]Again, these are order of magnitude arguments.

Year	Bandwidth	BW * Delay	Typical Node	Node Buffer Size	Required Buffer Size
1970	56 Kbit	1.7 Kbit	PDP 11	64 Kbyte	51 Kbit
1980	-[1]	-	VAX	1 Mbyte	800 Kbit
1986	1.5 Mbit	45 Kbit	-	-	-
1990	45 Mbit	1.4 Mbit	Sun 3/4	8 Mbyte	6.4 Mbit
1995	~150 Mbit	~4.5 Mbit	-	~20 Mbyte	~15 Mbit
2000	~1 Gbit	~30 Mbit	-	~64 Mbyte	~50 Mbit

TABLE 1.2
Network and node characteristics[2]

1.4.1. Change in the use of memory

The use of buffer memory has also changed with the advent of high speed protocols. Buffer memory permits restoring altered message sequence within the network, and temporarily archives data for lost message retransmission. In high speed protocols, where latency is large compared to the transmitted information, buffers permit the protocol to both ‘run ahead’ and to amortize control effort over large chunks of data.

These uses of buffers by protocols assume that communication exists to facilitate file transfers. We are expecting a change in the use of the channel, where file transfer is superseded by interactive communication. In this realm, buffers cannot be used by the protocol to accommodate latency, because the protocol can no longer ‘run ahead’ in sending data. The data will no longer be a linear stream of packets, so the protocol cannot anticipate the data to fill the buffers, and the channel utilization plummets.

Large buffers need to be used for more than just linear sequences of data. We extend their use to accommodate sets of data, where only one item of the set is used. In

---

[1]‘-’ indicates ‘no value’.

[2]Bandwidth*delay values are based on a 30ms speed-of-light propagation across the continental United States; buffer sizes are based on 1/10 * node buffer size, converted to bits.

this way, branching in the data stream, i.e., imprecision in the protocol, can be used to increase the utilization of the channel.

The notion of a ‘channel with imprecision’ will be further elaborated when the Mirage model is described. At this point, it is sufficient to have shown the need for a new model, one that accommodates latency in a natural way, rather than as an extension of an existing model.

1.5. Model goals

We have several goals for this protocol model. Before discussing them, we should first outline our assumptions. We assume a network with no topological or routing restrictions, and where the bit-latency is high compared to the streams of data which are communicated. Current gigabit WANs (were one to exist) satisfy these conditions, where communication is dominated by tightly coupled interaction of the components of the network.

We introduce a model where the tradeoff between error and bit-latency can be expressed. We will trade imprecision for latency, so that constraints on the communication can be used to increase the utilization of the channel and more tightly couple the communicating parties. We will also show the uses of the increased bandwidth to compensate for the effects of latency, so that the detrimental effects of bit-latency which are increased by high-speed channels can be reduced by that same high-speed. We will also show the limitations of this compensation.

1.6. Preview

Mirage is an abstract model, which can be used to understand the issues in protocol design, or can measure the implications of a specific design decision. We will investigate these two roles of the model in two studies presented as part of this dissertation.

First, we look at the Network Time Protocol, and examine its operation using the Mirage model. We show how Mirage can indicate important aspects of the protocol, and new methods that are being proposed to augment the protocol.

We also apply the Mirage model to the domain of processor/memory interaction as a communication protocol. We show how Mirage provides a fresh view of this

interaction, and indicates new solutions, to which the literature has only recently alluded. Further, Mirage shows ways to measure the various implementations indicated, from the perspective of both channel utilization and design complexity.

We also include a description of the model as channel stream interactions, state space transformations, and Petri Net equivalences (for those who find this more helpful). The section on prior work has been restricted to discussions of predecessors to the abstract model. Similar discussions on prior work in time protocols and processor/memory interaction are included in their respective sections.

CHAPTER 2

The Mirage model

Mirage is an abstract model that describes latency in communication [To89], [To90a], [To91a]. The model consists of three components: a model of node state, a model of time and communication as state transformations, and a set of constraints on these transformations.

The model is designed to serve as a viewpoint for understanding existing protocols and for the development of new protocols. It is based on providing an abstract model against which protocol instances can be measured and compared. Prior work in protocol design and analysis is based on an alchemy of examining particular protocols; this work is designed to provide an initial framework for treating protocol research as a science.

Current research in gigabit wide area network protocols suggests a “clean sheet” approach is indicated. The transition from evolutionary design to needed new (revolutionary) protocols will provide a fresh opportunity for introspection into the mechanism of protocol models. Before suggesting new protocol designs, we consider how the problem of communication has changed, and develop a new model that incorporates these changes from the start. Mirage is an attempt to characterize the tradeoffs that emerge in the new domain of gigabit WANs.

2.1. Assumptions

As communication rate increases, fixed latency increases the amount of information in transit between interacting nodes. Increases in information separation alter the assumptions of the limiting factor in communication. We have already presented, in Chapter 1, our definition of a protocol and communication. We also assume a domain where latency is large, relative to the bandwidth of communication.

2.1.1. Initial description of channel utilization

Initially, Mirage can be described in terms of channel utilization. This describes the indeterminism of the model and provides a real performance measure that Mirage is intended to enhance.

Information in the channel can be modeled as a stream of data. In a transaction protocol, this stream consists of a single message, whereas, in a file transfer protocol, it consists of a linear sequence of characters. A linear stream can be extended to accommodate alternate possible streams; the stream of composite linear components is a ***branching stream***.

Branching streams are common in interactive systems, where initial data messages indicate subsequent choices to be made by the receiver, which guides subsequent messages. The stream can be represented by a tree structure, where the ***limbs*** represent linear streams of data (similar to file transfer), and the ***bifurcations*** indicate choices among alternate streams (to be made by the receiver).

This work assumes that the latency is large compared to the bandwidth, such that the bandwidth delay product is large relative to the local storage available at nodes in the network. For current networks, this means a bandwidth-delay product in the order of tens of megabytes.

A minimum delay between participants in the protocol is also assumed, which in most cases is the propagation delay. For most equations, a fixed delay is assumed, although we show later (Chapter 4) how a variable delay can be accommodated.

To some extent, messages are assumed not to be lost, or, if they are, retransmission is required as in any existing protocol. We assume, as concluded earlier, that the limitations of existing protocols in this domain will be their inability to predict sufficiently enough information to occupy the round trip latency. Finally, Mirage shows

how these limitations can be circumnavigated, given enough information about the branching stream of data. This has important implications on the limitation of layering in protocols and abstraction in the layers of a protocol.

2.1.1.1. Phases

There have been three phases of network protocol design (Figure 2.1). In the first phase, characterized by the Network Control Protocol (NCP) [Ca70] (used in the ARPAnet at 56 Kbps), the information sent is treated as an unstructured block of data. This method sufficed where the partitioning of the data served no purpose; because NCP assumes lossless, ordered transmission, this was a reasonable design. NCP is a sufficient model for request/response protocols.

Sliding window protocols, e.g., TCP (used in the Internet at rates of 56 Kbps-45 Mbps), extended this view by partitioning the message stream into multiple blocks. The partitioning can be thought of as a *lookahead* into a time-ordered list of NCP block entities – individual bits in a block are assumed to be lossless and ordered, but the block unit may be lost or resequenced as a whole. In NCP only one block is sent at a time, whereas TCP looks multiple blocks ahead into the time-ordered list of NCP-like blocks. This ***linear lookahead*** permits TCP to accommodate latency (and introduces resequencing problems due to multiple outstanding packets), provided the delay associated with either is limited to that which can be accommodated by the lookahead.

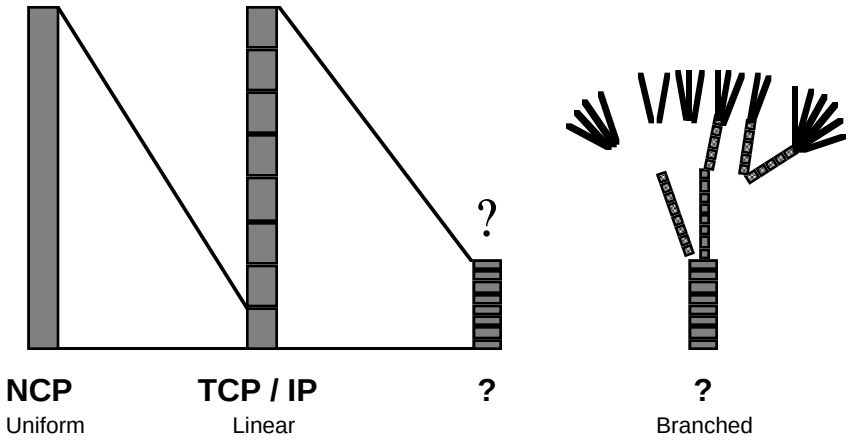


FIGURE 2.1
Kinds of protocol lookahead / branching

In the case of high bandwidth-delay product, linear lookahead will be insufficient to 'fill the pipe' with information, and thus utilize the channel efficiently. As the network bit rate increases, *there is a point at which the data required by the receiving end cannot deterministically be predicted and thus a TCP-like protocol will exhibit performance failure*. The time-ordered list of data blocks is limited to the amount of data that can be deterministically predicted to be required at the receiving end. This is where Mirage helps, by modeling the portion beyond the linear lookahead as branching (the tree in Figure 2.1). Mirage suggests transmission not only what is *known* to be needed, but also what *might be* needed.

2.1.1.2. Channel utilization: linear case

We define the efficiency of a protocol to be the ratio of communicated information to the channel bandwidth (percent utilization of the channel). Protocols are maximally efficient when the node buffer size[1] is less than the bandwidth-delay product (Equation 2.1). The formula is linear in the number of buffers, but inverse in the round trip time, so increases in latency may have severe effects on channel efficiency [Ta88]. There is a point at which the linear lookahead fails (i.e., cannot further anticipate the data stream required by the receiver), and utilization diminishes (Figure 2.2).

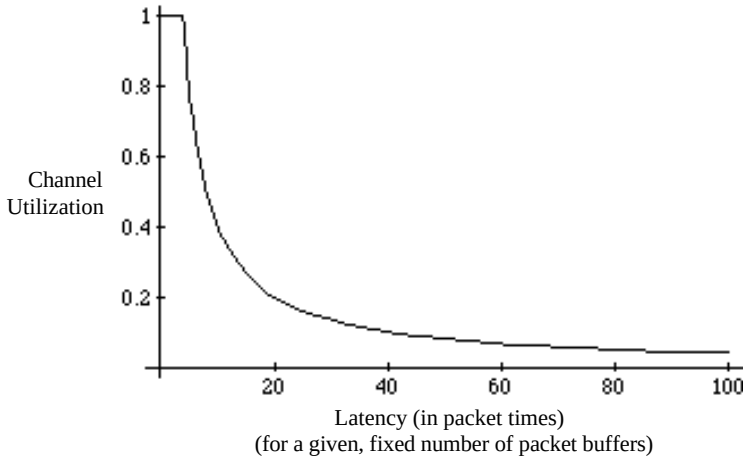


FIGURE 2.2
Utilization as latency increases (linear lookahead)

[1]Node buffer size is defined as the total size of the state space of the communicating entity.

Equation 2.1: $$\%util = \begin{cases} \frac{B}{R} & \text{where } R > B \\ 1 & \text{where } R \leq B \end{cases}$$

where B = number of buffers in sliding window protocol
R = $bandwidth * round\_trip\_time$ = buffers used in one round trip

2.1.1.3. Channel utilization: sender-based anticipation

We will now demonstrate how a branching stream can increase the utilization of a channel in the presence of latency. Assuming that such branching is characteristic of the data stream being communicated, Mirage suggests a protocol that is capable of supporting labeling in the data stream. We call this labeling *guarded messages*, which will be described in greater detail later in this chapter.

Guarded messages are labels on communication stream components, so that the sender can indicate the desired state of the receiver for a given message to be accepted. A set of messages with suitable guards can permit branching in communication stream. Branching of the stream permits the sender to 'run ahead' of a known (single) receiver state, into a set of receiver states, accommodating latency in the resolution of that state.

2.1.1.3.1. An example for illustration- the turtle

Consider a turtle moving on a two-dimensional map. The turtle moves at 5 spaces per minute. A message that directs the turtle takes 1 second to send (bandwidth), and the messages take 1 second for delivery (latency). In this case, traditional request/response communication suffices to direct the turtle and confine it to within 1 space.

If the messages are delivered with higher latency (e.g., 50 seconds, ~1 minute), or take less time to send (20 milliseconds), the turtle cannot be confined so precisely. The latency implies an error of at least 5 spaces in the turtle's location.

Guarded messages permit use of the 'possible messages in transit' before knowledge of the receiver's state is known. At most 50 messages can be sent in the latency. We send 1 message labeled with each of 50 points around the last known turtle location. Each message redirects the turtle to the desired goal of the communication, given each possible current turtle location.

Note that in this case, guarded messages accommodate the imprecision induced by latency, since an error of spaces of movement indicates an area of 50 spaces in a two-

dimensional grid[1]. Unfortunately, guards consume communication bandwidth, so that less than 50 messages are possible in the given latency. Further constraint of the turtle's movement would be required to guarantee that a message would be received at each possible location. These constraints necessitate grouping locations (coarse partitioning of the turtle's location space), or restricting the turtle's movement so that less than 50 spaces are possible. These will be discussed later in this chapter as well.

2.1.1.3.2. Branching streams utilization

In branching streams, only part of the anticipated stream is actually used at the receiver. The revised channel utilization formula must account for the differences between sent data (multiple streams) and utilized data (a single stream). For channel utilization to increase, two forms of prediction are necessary – *predict enough* data to send, and *ensure the utilization of enough of the predicted data*.

If communication is limited to the linear portion of the predicted data, channel utilization will decrease as the latency increases (as before, in Figure 2.2). As latency increases, there is a point at which the amount of linear lookahead is insufficient to occupy the channel during the round trip time. In current protocols, as bandwidth increases there is often insufficient buffer space to accommodate further linear lookahead. As buffer space increases (i.e., as memory becomes cheaper and larger buffers can be accommodated), this will cease to be a primary issue; the linear portion will, at some point, no longer suffice to fill the buffers. The problem is a failure to predict which data stream to send, after an initial period of success. *The lack of buffer space is a short term issue; in the longer term, we expect the indeterminism of the data stream to dominate the problem.*

A simple model of the indeterminism of the data stream following some linear prefix assumes that the branching stream has finite *branch degree* (branching factor) and finite linearity before branching recurs (*limb length*, to extend the 'tree' analogy). Members of a branch are assumed to be **isopotent**, which we define to mean "information redundant." Redundancy usually refers to duplicate data used to protect against corruption. Isopotency indicates that members of a set affect the node similarly, and that only one member of the set has any effect.

---

[1]If the turtle can move 5 spaces, then it can move 3 spaces left and 2 spaces up on the grid, for example. The total area is 10 by 10 on the diagonal, or 50 possible spaces.

For an example of isopotency, consider the set of messages that directs a turtle to the center of a grid without its knowing that goal (this is a relative of the earlier turtle of Section 2.1.1.3.1.). Assume that the turtle can determine its own grid position, it goes 5 grid units in a unit time, and messages are delayed by 2 units of time. The turtle enters the grid from the north-west, at some distance.

The message “go south-east” can be sent for some time, but only until the turtle is within 10 units of the center. Any fixed messages sent can be incorrect in their assumption of the turtle’s current position. Instead, sets of messages are sent, guarded (in Dijkstra’s sense of guarded commands [Di76]) for each quadrant of the area around the center. These messages constrain the turtle to smaller and smaller areas, where it finally rests on the center. The messages are distinct, but the set of messages together has the same net effect, of directing the turtle towards the center of the grid. This is isopotency.

Stated another way, the remote node is in some known set of states. The guards partition this set, and a message affects only the states within its indicated partition. After the set of messages has been received, the remote node is in some other set of states (the union of the original partitions, transformed by the messages), regardless of the original state. The guards all have the same effect (i.e., are isopotent) – ensuring subsequent membership in this set, even though the messages are distinct.

Channel utilization is defined here as the percent usable messages per unit time, where all messages are of uniform length, time units are normalized at one message per unit time, both ***branch degree*** (D) and ***limb length*** (L) are fixed and finite, and that the branch alternates are equiprobable. The model for the exact formula of channel utilization accommodating branching is described here, under these assumptions.

The branching stream forms a tree, where the trunk represents the linear lookahead, the branch degree is the tree degree, and the limb length is the distance in messages between levels (Figure 2.3). The set of messages used by the receiver are the sum of the linear lookahead (all of which are used by the receiver), the number of full tree levels (because one path through these levels must be useful), and the probability of utilizing the number of leaves at the last, partially unfilled level (Equation 2.2).

Equation 2.2: $$\%util = \frac{P + L * full\_tree\_depth + extra\_leaves * prob\_leaves}{rtt}$$

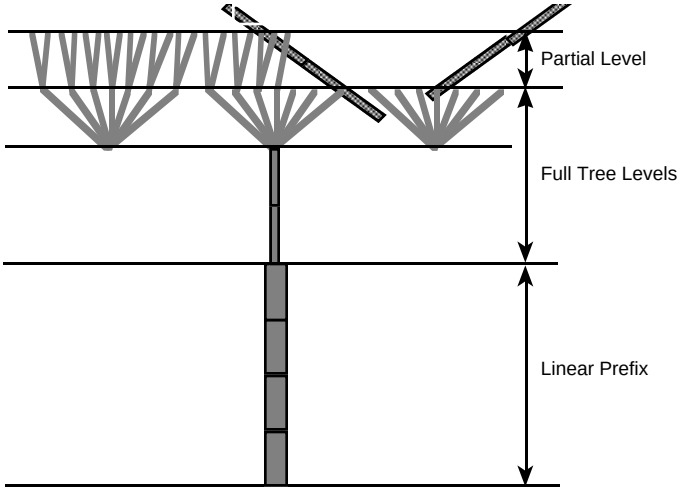


FIGURE 2.3
Tree levels

In a given round trip time (*rtt*), the sent messages include the linear prefix (P), some number of messages corresponding to the filled tree levels (*full_tree_depth*), and some remainder of leaf messages at the last level (*extra_leaves*) (Eq. 2.3). The number of messages in the filled tree levels requires knowledge of the branch degree of the tree (*D*) and the limb length (*L*). Using this and the identity of a summation of a power (Eq. 2.4), Eq. 2.3 can be rewritten as Eq. 2.5. Eq. 2.5) can be solved for *full_tree_depth* (Eq. 2.6).

Equation 2.3: $$number\_sent = rtt = P + \sum_{i=1}^{full\_tree\_depth} D^i * L + extra\_leaves$$

where P = linear lookahead length
L = limb length (number of messages between branchings)
D = branch degree (used in later equations for *full_tree_depth*, etc.)
rtt = round trip time

Equation 2.4: $$\sum_{i=1}^{n} x^i = \frac{x * \left(x^n - 1\right)}{x - 1}$$

Equation 2.5: $$number\_sent = P + \frac{D * \left(D^{full\_tree\_depth} - 1\right)}{D - 1} * L + extra\_leaves$$

Equation 2.6: $$full\_tree\_depth = \lfloor tree\_depth \rfloor = \left\lfloor \log_D\left(\frac{(rtt - P)*(D-1)}{L*D} + 1\right)\right\rfloor$$

where $$tree\_depth = \log_D\left(\frac{(rtt - P)*(D-1)}{L*D} + 1\right)$$

Similarly, the number of leaves remaining in the partially filled level can be represented (Eq. 2.7), and using the identities in Eqs. 2.8, 2.9 and the notation of 'fractional part' in Eq. 2.10) a more simplified overall utilization formula is derived (Eq. 2.11).

Equation 2.7: $$extra\_leaves * prob\_leaves = \frac{(rtt - P) - \frac{D*\left(D^{full\_tree\_depth} - 1\right)}{D-1} * L}{D^{full\_tree\_depth} + 1}$$

Equation 2.8: $$rtt - L = \frac{D*\left(D^{full\_tree-depth} * D^{\mathbf{F}[tree\_depth]} - 1\right)*L}{D-1}$$

Equation 2.9: $$extra\_leaves * prob\_leaves = \frac{D^{\mathbf{F}[tree\_depth]} - 1}{D-1} * D$$

Equation 2.10: $$\mathbf{F}[x] = x - |x|$$

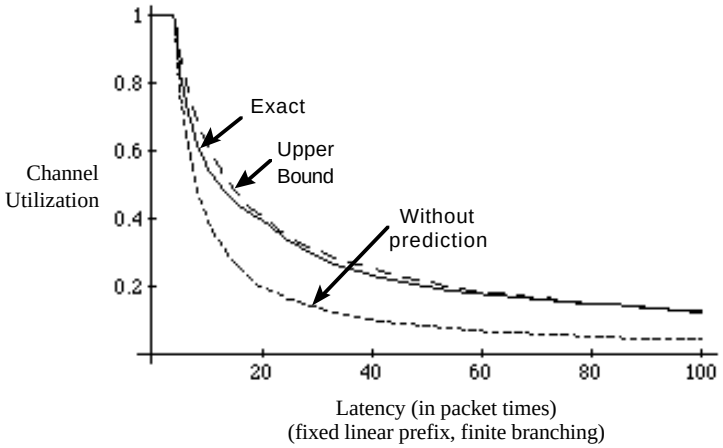


FIGURE 2.4
Utilization of branching lookahead (P=5, D=2, L=4)

Equation 2.11: $$\%util = \frac{L + L * \lfloor tree\_depth \rfloor + \frac{D^{\mathbf{F}[tree\_depth]} - 1}{D - 1} * L}{rtt}$$

Equation 2.12: $$\%util = \frac{P + tree\_depth * L}{rtt}$$

Equation 2.11 is a discontinuous curve (Figure 2.4, Exact). The upper bound of this curve uses *tree-level* in its original (continuous) form, rather than its discontinuous floor function (Eq. 2.12, Figure 2.4, Upper Bound) (see also Appendix C). Comparing the linear stream curve to the branching stream exact curve and branching stream upper bound, a utilization increase is shown, due to anticipation of receiver requests which branching stream alternates permit. The utilization of the channel, relative to using linear lookahead only, increases without bound (logarithmically) (Figure 2.5). The decrease in channel utilization is due to indeterminism in the data; the channel is full of data, but not all the data sent is actually useful. For a given limb length and branch degree, this is an upper bound on channel utilization, given limited prediction capability (i.e., limited to the linear lookahead).

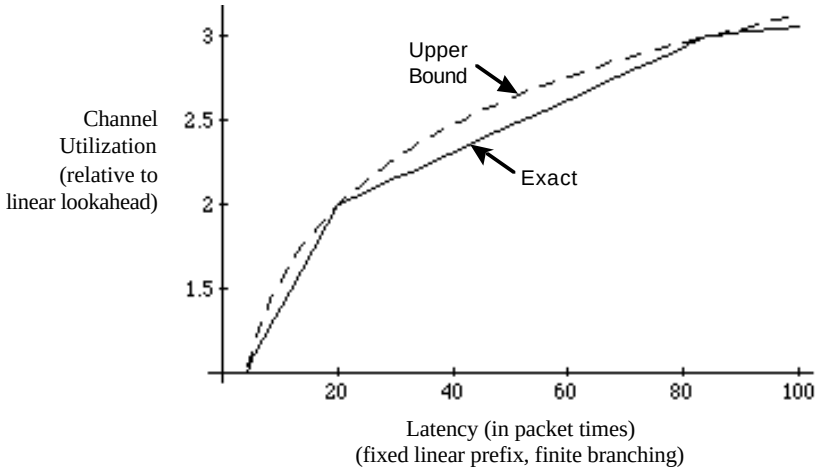


FIGURE 2.5
Channel utilization (relative to lnear) (P=5,D=2,L=4)

2.1.2. Conclusions of the branching model

The previous discussion describes how a branching stream model of communication can permit increased utilization of the channel in the presence of latency. Branching streams presented thus far are a simple model in which a state space evolves according to static, context independent rules. A more complete model includes context sensitivity, so that the branching and limb lengths are irregular; this model is provided by Mirage in Section 2.2. of this chapter.

In the case where branching is regular and context independent, the channel utilization increases logarithmically with an increase in its bit-latency. These results are verified in a real example in Chapter 5, in which this model is applied to processor-memory interaction as a communication protocol. Measurements indicating the real values of the branching and limb lengths are presented in Chapter 6, in the analysis of the feasibility of the implementation of the designs suggested by this application of Mirage's principles.

Before proceeding, some tests can be done to verify the reality of the branching stream model. These include tests of the limiting cases, i.e., considering the boundary conditions of the equations.

2.1.3. Some reality checks

Given the above model for a channel with imprecision, consider the characteristics of these formulae as various parameters are taken to their limits. These limiting characterizations should match their real-world counterparts.

As limb lengths increase, the indeterminism of the receiver is postponed at each decision. There is less indeterminism in the stream (Figure 2.6), and thus communication holds to the limb paths longer. For example, on the first branch, as the arm length increases, the utilization approaches 1/*branch-degree*, because utilization is dominated by the first choice in the branching stream. In the case where branching is binary, the limit of the utilization as the branch lengths approach infinity is 50%.

A linear stream exhibits determinism during the linearity, and complete indeterminism thereafter; an infinite limb length stream has a linear portion followed by a single branch point. The communication after the branch point dominates the channel

utilization, and the branching never recurs, both due to the extreme length of the limb. In other words, a linear stream exhibits no choices (complete determinism), whereas a branching stream with an infinite limb length exhibits one choice among *branch-degree* alternates.

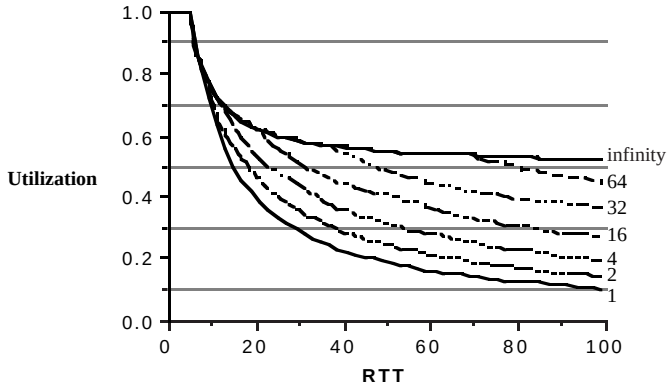


FIGURE 2.6
Utilization vs. branch arm length (geometric)

As the branch degree increases, utilization approaches the linear stream case (Figure 2.7). Sender anticipation is increased when behavior of the receiver is predictable, i.e., when the branch degree is minimal. A larger branch degree represents mode indeterminism in the branching stream. As the branch degree approaches infinity, the linear stream case results, because a linear stream consists of a fixed linear lookahead followed by infinite indeterminism, i.e., no prediction of data subsequent to the known lookahead.

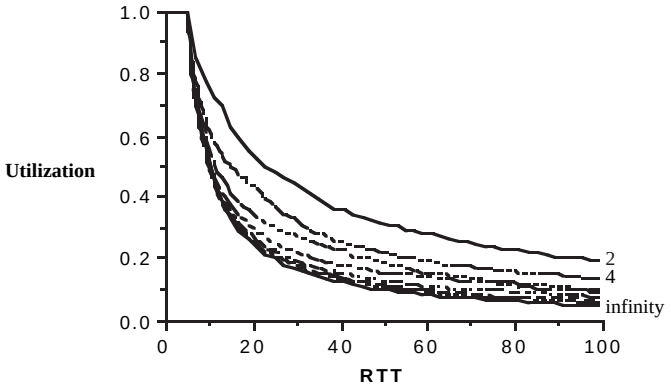


FIGURE 2.7
Utilization vs. branch degree (geometric)

A conventional TCP-like stream is the result of a limb length or branch degree of zero, (i.e., no tree beyond the linear lookahead), where the utilization is 1 whenever the buffers can accommodate the round trip latency via linear lookahead.

These examinations are the effect of varying limb length and branch degree, and consider the consequences in the resulting channel utilization. Limb length and branch degree are characteristics of the communication stream, determined by the nature of the communication and the protocol facilitating that communication. They represent the indeterminism in the communication, and are modeled by discrete finite indeterminism increases (branching degree) recurring at known intervals (limb length).

2.1.4. Some common sense

Channel utilization has been discussed, along with a method that accommodates data stream imprecision (i.e., multiple possible data streams) allowing a higher channel utilization than conventional linear streams. We believe that existing protocols will exhibit performance failures in gigabit WANs because the linear lookahead will be insufficient to occupy the channel during the round trip time.

Before discussing the particulars of the Mirage model, and how these bifurcating streams are accommodated within it, some common sense rules of protocols should be mentioned. These are commonly known constraints, but which are rarely included in protocol models.

A remote node with a highly fluctuating state requires a higher bandwidth or lower bit-latency to communicate effectively, because its requests are more unpredictable. Making pre-existing rules that restrict the fluctuation is the only way to overcome this limitation. We live with these rules daily. A human parent requires a high bandwidth and low bit-latency channel to his infant child, because there are very few assumptions that the parent can make about the safety of the child. Low latency is provided by proximity to the child. The child represents the highly fluctuating remote node, with respect to the parent.

As the child is moved away from the parent, either the constraints increase (via the addition of a sitter to the infant) or the bandwidth increases (e.g., via a nursery monitor). A nearby infant is not attended to as intensely as a monitor, because the increase in latency (time before parental intervention is possible) necessitates an increased anticipation of imprecision. A nearby crying baby is often ignored in the short term,

because the low latency (proximity to the parent and low time until the parent can attend the child) permits the parent to postpone action until absolutely required. A crying baby in the next room causes the parent to act on each cry, in anticipation of a more serious event, which would require more time to act upon.

The same parent may talk to his child once a week when the child is in college, because by that time there is sufficient knowledge of constraints in the child. Known fluctuation constraints permit a relaxation of the latency limitations[1].

A gigabit WAN has too much information in transit to manage. Additional constraints are required which describe how the stream branches (bifurcates) and which denote redundancy in the branches, in order to permit effective communication at high bandwidths. The system needs to be sufficiently predictable to utilize the bandwidth, but not so predictable that communication is obviated. The Mirage Model provides measures for being *"predictable enough"*.

2.2. The Mirage model

The abstract Mirage model is based on representing remote nodes as volumes in state space where data transmission and reception, as well as time evolution, are modeled as transformations on those subspace volumes. State space volumes represent imprecision in state, i.e., the volume is the subspace that contains the set of possible states.

Inherent in the Mirage model is the notion of measurable latency. Shannon's model of a communication channel [Sh63] can be extended by including latency measurements (Figure 2.8). Latency is assumed to be either constant or predictable (effectively computable). Flow in the Shannon model is described as the motion of a volume along a communication pipe, and latency is the length of that pipe. As such, incorporation of latency into the model reveals a spatial aspect to the formerly topographic Shannon model[2].

---

[1]Experience of my advisor, David J. Farber, suggests that these constraints may require failure compensation mechanisms, or at least a meta-communication that negotiates and monitors such constraints.

[2]Further discussion of the relationship between Mirage and Shannon's model, along with a brief discussion of the latter, appears in Appendix A

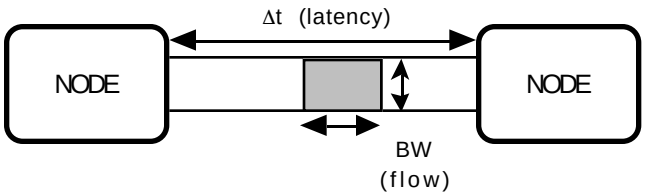


FIGURE 2.8
Mirage communication channel

The Mirage model describes communication among similar components of a network, called nodes. These nodes are separated by some minimum time lag, and through some maximum bandwidth communication channel; this separation characterizes the network for our purposes. The minimum time lag is representative of speed-of-light signal propagation delay, and the maximum bandwidth is representative of physical limitations on signal power per unit time. The nodes consist of some finite information storage; here the connectivity among the nodes and algorithmic power of each node is considered inconsequential to this preliminary analysis.

2.2.1. More definitions

The basis for this protocol model begins with state space transformations, extended to account for latency. This can be considered an extension of the FSM model, but with some exceptions.

First, FSMs usually describe a system in state space, whereas Mirage uses the powerset of this space. State spaces permit only a single value for each dimension in space, i.e., they accommodate only an individual point in the space. Mirage uses sets of these points, or volumes (ensembles, in either case) to describe the indeterminism of the knowledge of data in a remote node.

Mirage models the nondeterministic operation of a remote FSM, just as a deterministic finite automaton (DFA) models a nondeterministic finite automaton (NFA). A DFA state represents a set of states in the NFA, just as a Mirage state volume represents a set of states of a remote node. Mirage permits these volumes to vary, whereas the DFA model of an NFA fixes the state sets when the model is computed.

Mirage provides a way to describe a Turing machine system with explicit indeterminism and information delays. Traditional temporal extensions to FSMs describe the time bounds between transition transformations by describing the length of the time

arcs in the transition sequence. Mirage is concerned with the number of simultaneous arcs in the transition (i.e., the number of possible FSMs of the remote node), and so describes the transition as a function of time, not time as a function of the transition.

Mirage uses state space volumes as they are used to describe error correcting codes, because error induced by latency is similar to that caused by true corruption of the data. These analogs, to NFA-to-DFA transformations, to error correcting codes, and to temporal FSMs are all described in further detail as prior work in Chapter 3.

Because Mirage uses state space volumes to describe the possible states of a remote node, a set-based description of Mirage is most direct. This description is elaborated in Appendix E. Another example of its description in terms of Petri Nets is provided in Appendix F. The Mirage model is more general than either of these examples, i.e., it represents a model of which set notation and Petri Nets are instances.

Consider a set of nodes in the network. These nodes are considered completely connected, each pair connected with a finite maximum communication bandwidth and a finite minimum communication delay. Nodes possess finite **storage**. This storage is used both to denote the node's dedicated local storage and perceptions of the storage of remote nodes. The local **state** of a node is the local component of its storage.

One node has a **perception** of another (remote) node, which represents a subset of the possible states of the remote node. A node's **view** of the network consists of its own local state and the set of perceptions of the other nodes in the network. Mutually recursive knowledge is permitted, provided that the recursion is bounded and finite, as required by the fixed size of local storage at each node.

2.2.2. A description of the model

Like every good model, Mirage has some governing principles. Some of these can be considered axiomatic, i.e., self-evident truths used as the basis of the model Some are tenets, i.e., beliefs common to a group (the network research community), but as yet unproven. Some are thesis statements, to be proven by this discussion. The common thread among these statements is that they are generally believed, but to date no model existed in which their truth could be debated. Mirage presents such a model. As such, we believe the most appropriate label is 'tenet'. The following are a few we think are important.

TENET 1: Communication is logical information synchrony among information separated entities

TENET 2: A protocol is a mechanism for maintaining communication

TENET 3: Information separated entities are separated in time*space, in units of *pending-information*

TENET 4: Bandwidth-delay product is a measure of *information separation*

When the system begins, each node is modeled as a point in state space, a particular individual state. As the system progresses, this single state evolves into a set of states. Which of these states exactly describes the remote node is not known; it is known that the state of the remote node is in this set. Thus from the initial individual point, volumes in state space result corresponding to the set of possible states of a node.

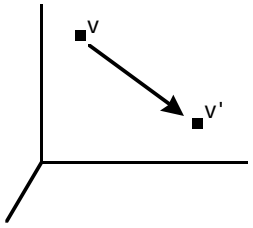


FIGURE 2.9
Shannon's model: states are points, transformations move points.

In Shannon's model [Sh63], information about remote nodes is modeled as a point in state space, and any operations that affect this information translate the point in space (Figure 2.9)[1]. In Mirage, a remote node is modeled as a volume in state space, where operations become transformations of that space. Volume before the transform is denoted by a thin outline, volume after the transform is denoted by gray shading, and the action of the message or time is denoted by the thick outline. Time expands the volume of a space, transmission yields the union of the original volume (thin outline) with the transformed copy (thick outline), and reception collapses the volume to a sub-volume (Figure 2.10).

[1]In Shannon's model, the receiver gets a single message, and back-calculates the state of the sender (i.e., what was sent) from this. The goal is to determine, from a sequence of messages, the exact states of the sender. The sender does not model the receiver.

These transformations are derived from the progression of time and the action of messages, as they leave the originating node and as they arrive at the destination. These are further discussed below.

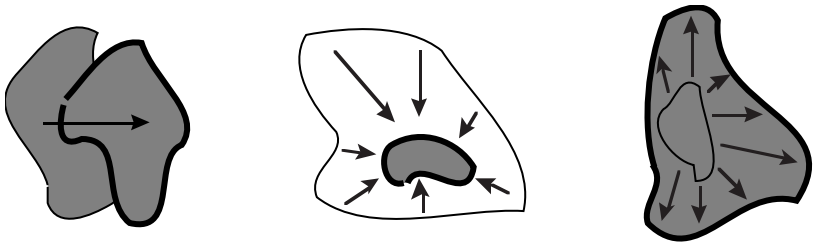


FIGURE 2.10
Visualization of state space volume transformations

2.2.2.1. Time

Time is modeled by the expansion of the state space volume. A node's local state does not grow in volume over time, because there is no imprecision in local state information, i.e., a node knows itself. The perceptions of remote nodes become less precise over time; it is this imprecision which the time transformation models.

Local state volumes cannot expand over time, because this would imply a violation of the Liouville thermodynamic theorem. The theorem indicates that local state spaces in physical systems cannot expand. This is not an issue here, because local state remains a point model; only remote state expands, and the theorem does not apply to perceptions (see Appendix D).

The transformation of a remote node's representation over a time interval is represented by a function that describes the known bounds on the variation of state space evolution over time. The extent to which the remote node is correctly modeled depends on the precision of this function, which is characterized by the amount of state space expansion per unit time, a form of induced entropy[1]. The imprecision describes the difference between the node actual state and the perception of that state. The entropy change per unit time is a measure of the minimum bandwidth required to compensate for

[1]The notion that state reduction and volume ratios are related to entropy is not new; it has been discussed before, in [Sh63] and [Ha28].

the entropy change, and can be bounded by the ratio of the volumes of state imprecision at the beginning and end of the time interval. Formulae expressing these relationships are described in Appendix E.

The computation function, which describes the evolution of the space over time as viewed at a distance, is a combination of the remote space evolving over time and the messages that it can receive over that time. Analysis of this function can be complex, because all possible permutations of messages and computing intervals must be accounted for. The computation function encodes known internal computation in the remote node, known bounds on the information received by that node, and known message emissions from that node (i.e., all known constraints on the remote node).

In the case where the time transformation is expressed by a probability density function (pdf), the computation function reduces to a convolution of the entire set of remote nodes over the set of probability density functions (pdfs) of the transformations of individual messages that can be received and a time transformation pdf. This reduction to convolutions requires that the time transformation is time invariant, i.e., it depends on the interval of elapsed time, but not the absolute time at which the interval occurs.

2.2.2.2. Receive

Receiving information collapses the perception volume of a remote node to a subspace of its former volume. Consider the case where a node receives data from a remote node. The received message affects part of the view that models the source of the message (the perception of the sender). There is a limit to the amount to which the message can reduce the volume of the perception (on average), because the volume reduction caused by the incoming information is bounded by the information content of that message, and because volume reduction is equivalent to reduction in entropy.

2.2.2.3. Transmit

Transmitting messages expands the perception of a remote node’s state, similarly to the expansion induced by time. Rather than accounting for the temporal transformation of the remote space, the message itself causes the transformation of the space. The expanded space is logically OR’d with the entire original space, because the message may be received later, or lost, and both cases must be accounted for. The message affects a node’s view by transforming its perception of the remote node to which the message is sent. Again, the information contained in the sent message is limited by the

transformation it effects, in terms of relative volumes of state spaces indicated (i.e., entropy).

Lost messages increase the state space of the perception, which is collapsed either when the state of the remote node indicates, or when the sender decides that the message loss can be ignored (i.e., a time-out that forces perception collapse). The use of time-outs to force collapse denotes the potential conflict with message loss assumption (i.e., that the reappearance of the message can cause an inconsistency in the perception).

2.2.3. Implications of the model

Several constraint conditions have already been presented, relating message effects on state space volume transformations and entropy limitations. Other correctness criteria have been presented relating received state to a subset of prior state. There are other constraints that are implied by the model, when further considered.

2.2.3.1. Lag and stability

The Mirage transformations can be considered with respect to stability. There are two variations on the definition of control stability. The first assumes that the state space reaches some fixed-point value, i.e., that it focuses on a specific point, within some variation, and remains there. The second maintains that the state space *entropy* is the value that becomes stable, i.e., that the imprecision of remote information reaches some fixed point, rather than the value of the state itself.

Consider the system whose state space is *A*. Over time, the space evolves to *A'*, whereas given communication (traditionally feedback/feedforward), it would become *A''* (Figure 2.11). ***Stability*** is defined as *A''* being a subset of A (after some minimum time[1]) (Eq. 2.13), whereas ***entropic stability*** is defined as the volume of *A''* being smaller or the same as the volume of *A* (Eq. 2.14). Neither criterion applies to the uncontrolled state *A'*.

Equation 2.13: $\forall_{\Delta t > t_{\min}} \left(A'' \subseteq A \right)$

Equation 2.14: $\forall_{\Delta t > t_{\min}} \left(|A''| \leq |A| \right)$

---

[1]Feedback stability commonly requires a minimum time lag, in order that the requisite circularity of information and action exists.

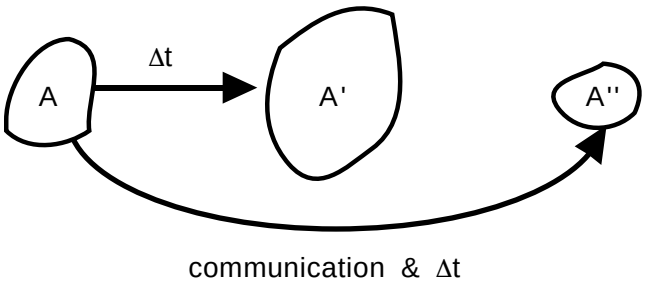


FIGURE 2.11
Control space evolution

2.2.3.2. Communicability

We can now express the most important formulae in the description of Mirage that defines the goal of the model. The space of a node consists of a finite number points, so a message could be sent, suitably guarded, for each point in this space. Assuming each message is of arbitrary length, destination state space can be transformed as precisely as desired, guaranteeing either stability criterion.

The trick is to send messages that are short enough, and to partition the space coarsely, to send as few of these messages as possible (each with a similarly brief guard), otherwise the required bandwidth would be unmanageable. The ultimate goal is a suitably efficient partition *K* (i.e., smallest number of component partitions in *K*) that satisfies these bandwidth criteria, and that ensures stability over all time frames beyond some minimum (Eq. 2.15)[1].

In Equation 2.15, *A* denotes the state of the remote node (i.e., the perception to be stably modeled), *K* denotes the partition, *A''* denotes the perception thus stabilized, and *A:M* denotes the state A as transformed by the set of delivered messages *M*. The goal is that for all intervals larger than some minimum, the system is entropically stable. The goal also includes ensuring that the smallest message set *M* be chosen, and that the set of messages can be communicated in the time given. One message m_i is sent to each component of the partition *K*.

---

[1]The number of components in the partition is the branch degree which alters the graphs (Figure 2.7). The message length is related to the limb length mentioned before and reflects the directed state path between branchings.

Here we denote this condition as a predicate, called ***communicability***. The predicate holds where entropic stability is permitted by a given partition under the given communication bandwidth and latency parameters. This predicate can be used to specify the bandwidth and latency for a given partition, or to govern the search for a minimal partition, using bandwidth or latency as a measure.

Equation 2.15: Given: $\begin{cases} K \text{ - a partition of A's perception at B,} \\ \quad \text{i.e., in a set of guarded messages } M \\ t_{\min} \text{ - latency} \\ BW \text{ - bandwidth} \end{cases}$

Node A is COMMUNICABLE from node B if and only if:

$$\left\{ \underset{\Delta t > t_{\min}}{\forall} \left(|A''| \le |A| \right) \text{ where } A'' = A{:}M \right\} \text{ and } \left\{ |M| \le BW * t_{\min} \right\}$$

The condition under which such a partition exists, called ***communicability***, represents the ability to communicate sufficiently with a remote node so as to ensure its stability, within the given bandwidth and latency criteria.

Stability need not be strictly guaranteed; in many cases, it is sufficient that the stability be highly probable. By assigning probabilities to each path in the state space evolution, the expected entropy change, rather than worst-case, can be considered. In this way statistical methods can be incorporated into the realization of the model. The volumes of state space become probability density functions (pdfs) in that space. Set operations on these volumes then become compositions of the pdfs. One case where similar statistical methods have already proven useful is clock synchronization [Cr89]; this is discussed in Chapter 4.

One important result of this description is that protocols which operate in high-latency environments require sufficient constraints on control space evolution. The stability of the system relies not only on the messages sent, but on the existing constraints of the computation function as well, because the state space is constrained by the interaction between the two.

The communicability formula is easiest to compute as a test. Given a fixed latency, does there exist a partitioning that results in a set of messages that can be sent during the round trip time and that result in stability in the perception of all remote spaces? If the

time transformations of the remote nodes are sufficiently constrained, and if sufficient bandwidth exists to overcome any remaining imprecision via controlling messages, then the answer is “yes”.

Another way to view the situation is that error and lag are conjugate variables. A communication system that requires zero error thus requires infinite lag, to collect arbitrarily precise information about a remote node before making a decision. A system that tolerates infinite error also tolerates zero lag – the instant a query is asked, a reply is given. The lag can be zero, because the answer is allowed to be arbitrarily wrong.

2.2.3.3. Guarded messages

Thus far, the description of a protocol as a set of state space transformations is very similar to conventional statistical communication theory. Rather than using the state space volumes merely to describe or analyze the protocol, Mirage uses guarded messages to manipulate portions of these volumes, as part of the control mechanism [To89].

Guarded messages are similar to guarded commands as used in programming languages [Di76]. Prior to executing a set of guarded commands, the state of the machine is within some set of states (the union of the states specified by the guards); afterwards the state of the machine is within another set of states (the union of the states resulting from each guarded command)[1].

Guarded commands are used during programming to counter the uncertainty in the machine state during execution of the program; guarded messages do the same for communicating nodes. Guarded commands account for the latency between coding and execution (and uncertainties that arise in that interval), whereas guarded messages do the same for transmission latency.

Guarded messages permit the transmission of multiple sets of information to a remote node (Figures 2.12, 2.13). The perception of the remote node can be a volume in state space, so messages can be sent that are labelled with various regions of that volume. The remote node compares its current local state[2] to the label of the incoming message, and acts on the received information only if the two match.

---

[1] See the discussion on isopotency.

[2] A node’s local state is known as a point in state space. The imprecision in the perception of the state of a remote node causes the volume to be introduced.

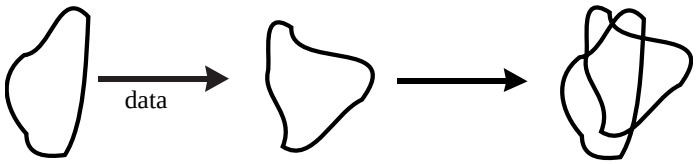


FIGURE 2.12
Entire space affected by unguarded message

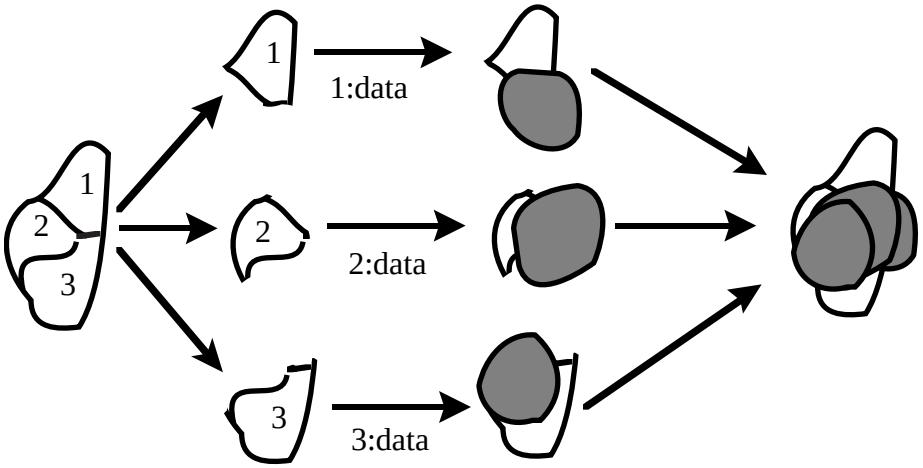


FIGURE 2.13
Guarded messages affecting partitions only

2.2.3.4. Isopotent sets

Isopotency describes a set of messages whose actions are equivalent, albeit by different actions on separate partitions of the state space. Guards differentiate the component messages of an isopotent set, to partition the space as desired.

The notion of isopotency leads to the distinction between a ***physical message*** and a ***logical message***. A physical message is conveyed by the unguarded data, whereas a logical message is the message indicated by an isopotent set. ***Isopotency*** denoted the *single effect* on the whole state space, as indicated by its component messages and their corresponding guards. The union of these actions is the logical message.

2.3. Discussion

There are several results to this abstract model, which are the consequence of this view of protocol models. Mirage is a model for a channel that accounts for imprecision in the communication, as introduced by latency.

We also have shown some formulae for the limitations of the ways in which latency can be accommodated (communicability). Most importantly, these formulae depend on the ways in which the state space can be partitioned, which in turn depends on semantic information about the state space. *The result is that protocol layering prohibits this partitioning, by hiding the semantic structure of the space.* Layering prevents the effective partitioning of the state space, and thus prevents any accommodation that could have occurred by sender-based anticipation using logical messages.

2.3.1. A channel with imprecision

The concept of a channel with imprecision can be elaborated. The limitation of existing protocols in gigabit WANs is due to an increase in the bit-latency. The increased amount of pending communication, i.e., information in the channel, requires modeling to permit channel utilization to increase. Further, this modeling can be performed only where prediction is possible, where the layering does not completely obscure some structure of the time transformation.

In Mirage, the characteristics of the data stream that are required in order to permit sender-based anticipation can be specified. The linearities in the stream express sender determinism, so that, regardless of the information communicated in the data of the stream, the sender knows which data to emit. Branching allows indeterminism in the sender, where the data sent depends on some unknown state of the receiver, permitting context sensitivity of the data stream.

The conclusion is that there are limitations to the utilization of the channel, and that these limitations can be overcome only if the internal structure of the data stream is examined. The sender can predict the next required information only if it knows what to expect. If these expectations are not fulfilled, round-trip delay penalties are incurred, in order to resynchronize the sender to the receiver’s state.

> We note the imperfection of the simulation. Although the Holodeck was successful in most details of the simulation, it lags when we strayed from the expected.
>
> - Star Trek, the Next Generation
>
> "Future Imperfect"

Some observations include the equivalence between infinite linearity and TCP-like existing protocols, and between infinite branching and NCP-like request/response protocols.

2.3.2. Looking into the structure of the stream

The structure of the stream can be described more completely in diagrams (Figures 2.14, 2.15). In the first case, bit-foreshortening (via increasing the channel transmission rate) causes the channel to be utilized less effectively. The result of the bit-foreshortening is an increase in the amount of the data stream that is "looked into" (fetched for transmission) during a round-trip time. So long as this stream continues to be linear, current protocols accommodate the lookahead (given sufficient buffer space).

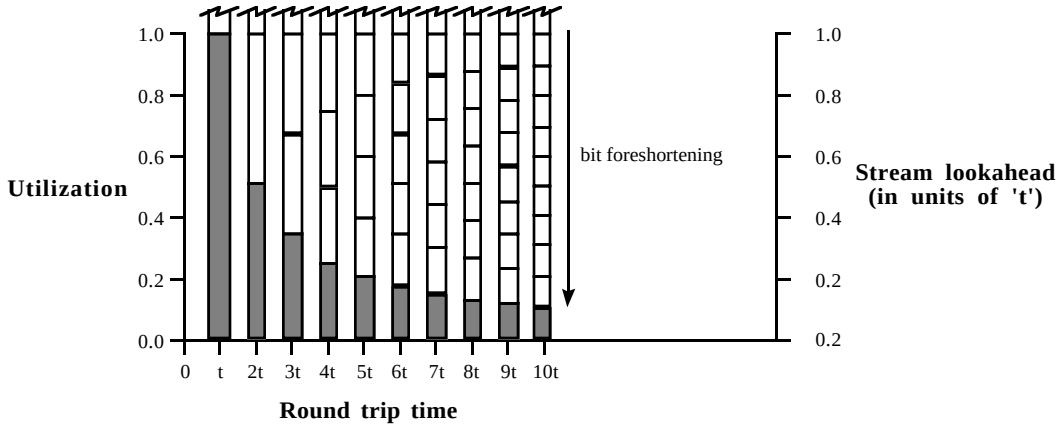


FIGURE 2.14
Bit foreshortening and its effect on lookahead / utilization

In the second case, bifurcations in the data stream cause the channel utilization to drop, because lookahead is permitted only until the first branching. By permitting the protocol to accommodate the branching, the remainder of the stream can be anticipated, albeit less effectively than the initial linear portion. This is presented in more detail in Chapter 5, in the discussion of the processor architectural implications of the protocol analysis of a processor-memory interaction.

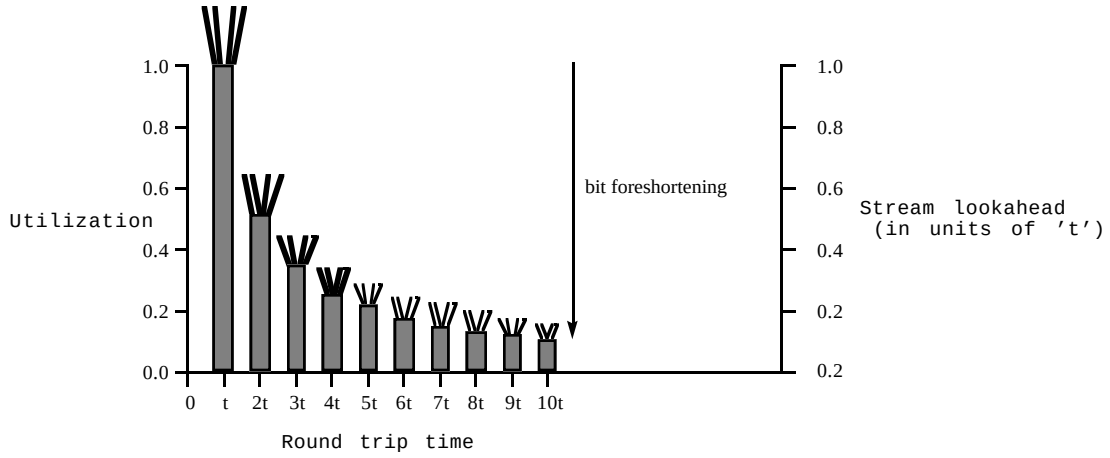


FIGURE 2.15
Bit foreshortening and branching effecting utilization

2.3.3. Implementations

Mirage is an abstract model, using state space set transformations to describe the stream with imprecision. There are various ways to implement the model, such that the implementations are equivalent to the abstract form. Some of these implementations are direct analogs of the abstract model, suitably collapsed or condensed to permit their realization. A more specific example is investigated in a later chapter (Chapter 5).

2.3.3.1. Projections

The Mirage model is based on state space transformations, so one obvious implementation is the realization of a projection of the model, where some dimensions of the model are ignored, or groups of dimensions are collapsed into one.

The complete form of the model incorporates not only sets of points in state space, but also probabilities for each state. Where probabilities are not known, the worst case is assumed, which in information theory is the case where each possible state is equiprobable. This results in a uniform distribution among members of the set.

Each point in the state space is assigned a probability, where omitted points (points where the receiver's state cannot lie) have zero probability, so probability density functions can be used to express the distributions as a function of state value. Transformations of the state space in Mirage are then most completely expressed as pdf transformations, which are the convolutions of the individual pdfs.

These pdfs can be restricted to ease their implementation. For example, the state space volumes can be limited to be uniform and orthogonal, such that the value of the pdf (i.e., probability of the state space value being accurate) is independent in the dimensions of the variables of the state space. This is equivalent to a range-value system, where we can express the pdf as high/low values, between which the probability is uniform, and outside of which the state cannot lie. The result is an implementation that tests only bounds of the state space, rather than true likelihood. Such a system would be useful in real-time systems, or fault-tolerant systems.

The pdf can also be replaced with an average/standard deviation pair, but only where the pdf is orthogonal and has an internal structure that is adequately approximated by these first order statistics. This is useful in 'aiming' protocols, where boundaries are not an issue, but localization of a shared value is; such is the case in clock synchronization protocols.

2.3.3.2. Granularity

Another consequence of the Mirage model is an acknowledgment of the desired dichotomy between the state space of the receiver and that same space as modeled in the sender, for the purposes of controlling data anticipation.

The receiver has a state space that is a fine partitioning of the state space, fine enough to express the limit of the granularity of the space. The fineness of the granularity of this space is defined by the degree to which the receiver partitions (or does not partition) it.

The sender's view of the receiver is a more coarse partitioning of this space. The coarseness of the granularity reflects the 'need to know' principle of this model – the sender models the state of the receiver only so explicitly as it needs to, in order to permit effective use of the channel. From the equations of stability and control, a larger granularity means that fewer messages need to be sent to anticipate the partitions of the receiver's state, which in turn allows the individual messages (to each component of the partition) to be longer.

The result is a system in which the sender models the receiver only to the extent that it must in order to send data in anticipation, and the receiver completes the structure of the partition down to the level of each state value. The sender needs to model only so far as to anticipate, but the receiver will use the sent data along with local information to achieve the desired computation.

2.4. Insights

There are a few useful insights from the investigation of this abstract model. There are several types of information in a system: direct, indirect, and a new kind, virtual. Error and latency are related, as conjugates. Finally, entropy and communication have been discussed in more abstract terms.

2.4.1. Kinds of information

In distributed systems, two kinds of information are usually described: direct, and indirect. ***Direct*** information is data about another node which that node explicitly sent. ***Indirect*** information is inferred data, sometimes called ‘common knowledge’[Ha84], [Go88], which is information about another node that is inferred from global constraints and direct information from the rest of the system. Indirect communication occurs when we know a-priori that 3 of 5 nodes hold a copy of a single datum, and we have received 2 replies where the datum is absent; we can immediately conclude from the global constraint and the received information that the remaining 3 nodes contain the datum.

Mirage suggests another kind of communication, that of virtual data. ***Virtual*** communication is not the result of any direct communication; it is the consequence of the lack of communication over time, and some specific constraints about the node ‘not heard from’. Virtual communication results from the time transformation, i.e., how the state space of the remote node behaves over time, unless otherwise heard from.

2.4.2. Error and latency as conjugates

Mirage indicates ways in which error and latency are conjugates. C. Shannon noted that error could be reduced as small as desired by encoding information over a long enough sequence [Sh63]; the process of encoding induces a latency of the length of the encoding sequence. Mirage shows how to reduce latency by anticipation, with a corresponding increase in the error of the perceived state of the remote node.

2.4.3. Entropy

When the state space volumes were described as corresponding to entropy, a set of constraints was introduced, by analogy. If the log of the state volume is entropy, then all the conventional physical constraints on entropy should apply.

For example, in physical systems, entropy always increases. In this system, entropy increases with time and with emitted messages (i.e., entropy increases in the sender when it sends data to the receiver). Mirage permits the collapse of these volumes, when information is received, contrary to traditional physical laws (i.e., entropy decreases). However, because data is created in the nodes of a network, as has been claimed in biological systems [Ja55], the state space volume may reduce.

2.4.4. Constraints

There is an interaction between error, latency, communication, and the need for constraints about the ways in which the receiver traverses the state space.

> Time and space and thought aren't the separate things they appear to be. These things are dangerous to say.
> - Star Trek, the Next Generation
> "Where no one has gone before"

2.4.5. Contrasts & comparisons

There are comparisons between the abstract Mirage model and aspects of forward error correction (FEC) FEC uses the same kinds of state space volumes, where the state space is divided into equivalence classes, so that when any point in the class is received, the canonical member of the class is presumed to have been sent. Mirage uses what can be considered a dynamically reconfiguring FEC scheme to communicate the remote state.

Mirage also uses multiple possible messages, in the isopotent set. It explores the state space in a breadth-first sequence (BFS), rather than depth first, as in conventional receiver-based anticipation. The use of BFS techniques removes the need for sender rollback, because all possible states are covered with the traversal of each level of the tree of possibilities of communication.

Other anticipatory schemes sometimes use replication of multiple independent DFS explorations, in which the results are collected upon termination of each probe [Sm89]. Although there are analogies to our method, Mirage relies specifically on the differences which BFS affords; specifically those removing the need for rollback. Further, there is a difference in the characterization of the state space in these two methods. [Sm89] assumes one of the DFS paths will terminate before others, and that which path is shortest is not computable before the actual probing of the space. In Mirage, the space is computable, with sufficient delay. Mirage tries to get around the delay of communicating exact state by permitting the states to be mere approximations.

There are direct correspondences between the way in which the receiver filters messages according to guards, specifically related to the Universal Receiver Protocol (URP) [Fr89] and the Knockout switch [Ye87]. These similarities, as well as other prior work, are discussed in Chapter 3.

Finally, there is an interesting comparison between the implications of Mirage and the selection of optimal buffer sizes in sliding-window protocols. The optimum buffer size for communication is the size of the round trip bandwidth-delay product, and inefficiencies result if the buffer size available is less than this product, addressing the TCP/sliding-window protocol situation. Mirage asks the question ‘what happens if the buffer size indicated is negative’, i.e., if the round-trip time is much larger than the maximum possible window (i.e., linear lookahead in the data stream). In this case, Mirage suggests an advantage.

CHAPTER 3

Prior Work

The Mirage model evolved from concepts from a variety of disciplines, and is also a paradigm for latency in communication. As such, the relevant prior work spans a breadth of disciplines in traditional protocol research and communication models, as well as in distributed systems, cybernetics, and physics. This is a summary of that prior work and a discussion of Mirage in comparison.

This chapter discusses prior work in general protocols and communication, rather than as it relates to specific applications of the model. Discussion of prior work relevant to these particular applications is presented in the chapter where the Mirage model is applied to specific protocols (the Network Time Protocol, Chapter 4) or systems (processor-memory interaction in the μ-Net evaluation, Chapter 6). Elaborations of particular similarities are presented in appendices, e.g., relevance to Shannon's model of communication (Appendix A), equivalences in timed Petri Nets (Appendix F), and original conceptions from analogies in physics (Appendix B).

Mirage is also related to some recent high speed optimizations of protocol implementations. Most of these optimizations are designed to reduce computational requirements of high speed protocols, rather than to compensate for increased effects of bit latency, as Mirage is intended. In hindsight, these optimizations can be viewed as

versions of some aspects of communicability in Mirage, although none has been derived or presented as such.

Mirage is a system comprised of sender modeling of a receiver, sender anticipation, entropic stability, and guarded messages. All of these components are intended to describe latency compensation if the information in transit exceeds the determinism of the system. Whereas some of these components appear in prior research, their combination in Mirage provides a novel system for understanding the effects of latency in communication.

Mirage originally evolved from analogs to principles in physics, most notably those of imprecision of state, quantum interaction by the exchange of field particles, and Feynman path integrals. These analogies are addressed further in Appendix B, so as not to confuse the conceptual basis for Mirage with its development and current design as a model.

3.1. Prior models of communication

There are a few canonical finite state machine (FSM) models used in protocol analysis; these include Shannon's model of communication, communicating FSMs, Petri Nets (and their variants), and language-based descriptions such as LOTOS and Estelle [Si91], [Ve86], [Sc82a]. Mirage differs from these models primarily in its use of partitions for communicability, although individual models individually exhibit similarities to components of Mirage.

3.1.1. The models

The Mirage model is most similar to the anticipatory feedback mechanisms of a compensator in the presence of effector delays, from cybernetics theory [Wi48]. This work describes the notion of sender (compensator) modeling of receiver (effector) state to maintain stability. This modeling is the converse of that of Shannon's model of communication. In that model, the receiver models the sender to back-calculate the sender's state from received messages with transmission errors.

The abstract model, as presented in Chapter 2, is an extension of existing finite state machine methods. Mirage is also presented as an extension to timed Petri Net models in

Appendix F, although it may be applicable to any general instance of a compensator in a communication or control system.

Most of the relevant prior work on protocol models involves temporal extensions of finite state machines, such as timed Petri Nets and extended finite state models, and temporal logic. These all incorporate time as a boundary condition (event), record age as a static property of a variable, or denote only discrete time intervals, whereas Mirage considers the continuous effect of time on communication. These models also consider state as a point in state space and thus do not accommodate Mirage's extension to the use of probability density functions[1].

Mirage differs from FSM models in its use of time as a continuous parameter, in treating the remote space as indeterminate (as sets of possible states), in the dynamic use of partitionings of the possible remote state spaces, and in the use of such partitions to determine communicability and stability constraints.

3.1.1.1. Shannon's model of communication

Mirage can be considered to be an extension of Shannon's communication model [Sh63]. This is elaborated in Appendix A, but will also be briefly described here.

Shannon's model of communication is based on a point model of interaction. The state of the sender is a known point in state space, and the receiver interprets the arriving message as it refers to that point. This model describes error within the communication channel, of which a fundamental theorem is that the probability of error can be reduced to an arbitrarily small value, provided that the messages are encoded over a correspondingly arbitrarily large sequence.

Mirage can be considered an extension of this model, where Shannon's point value is a set in Mirage, and where point motion becomes set expansion, contraction, or translation due to various communication operations (Chapter 2). In Mirage, the conventional notion of connectivity and bandwidth is extended to account for latency, so that the connectivity of the communication graph is extended to denote topology, using link 'lengths'. Our model also extends the notions of effectiveness, correctness, and precision as described in Shannon's model, to temporal equivalents of timeliness, controllability, and communicability (see Appendix A).

---

[1]Our use of PDFs is analogous to the analyses of state evolution of physical systems in thermodynamics.

Mirage is also a complement to Shannon's model, in several ways. Shannon's model considers a receiver's model of the sender, whereas Mirage considers the sender's model of the receiver. Shannon's error theorem trades error for latency, through sequence encoding, whereas Mirage trades latency for error (state imprecision) or restriction of operation (constraints).

3.1.1.2. Communicating finite state machines

Communicating finite state machines are a class of models for protocol analysis. These include communicating FSMs [Bo78], [Ok86], and FSMs extended with temporal constraints [Si82], [Ag83]. Neither case considers communicability. Extensions of these models are the basis for the Mirage protocol.

3.1.1.3. Petri Nets

Petri Nets [Pe62] are a variant of FSM models. The timed Petri Net model [Me76] can be extended to exhibit the characteristics of Mirage, just as the FSM model was; this extension is described in Appendix F. This demonstrates that the Mirage model is not restricted to a single paradigm.

3.1.1.4. Estelle / LOTOS

LOTOS [IS88a] represents a class of programming language models of protocols, which strictly includes Hoare's Communicating Sequential Processes (CSP) [Ho78], ISO's LOTOS, Milner's Calculus for Communicating Systems (CCS) [Mi80], and Real Time Attribute Grammars (RTAG) [An88], and which also includes hybrid programming language / state machine models such as Estelle [IS88b]. These models similarly lack expressions of communicability, although it may be possible to extend these models similarly. The most similar is RTAG, because the real-time scheduling constraints are a version of communicability, albeit on an individual message basis.

3.1.2. Partitioning the state space

Each of these traditional protocol analysis models is plagued by an explosion in the state space of the model [Bo78]. As the state of the analysis evolves, indeterminism of state requires modeling the powerset of possible states (i.e., all possible subsets). This

state space explosion is accommodated using imaging projections, factoring, and Powerdomains.

Part of the state space explosion is due to a set of messages in transit. FSMs cannot model a protocol with more than only a few messages in transit, and as such they cannot effectively model high information latency systems [Bo78]. The state space explosion can be managed by introducing state variables to factor some dimensions of the original FSM into variable-managing components.

By additionally partitioning the state variables, latency can be incorporated into the extended FSM [Si82]. This partitioning allows the separation of the effects of messages being emitted from a sender and collected by a receiver.

The state space can also be managed by partitioning, factoring, or imaging. Such partitions and projections may aid real implementations of the abstract Mirage model. The abstract Mirage model is intended for direct computation, but equivalent substructures of the model may be suitable. A more coarse-grained partitioning of the state space, using equivalence relations, accomplishes this. The subspaces become partitioned as a result of the imposed homomorphism of the relation, and particular location in the subspaces becomes less important than the traversal of these boundaries [Ch81]. This work was examined as ‘protocol conversion’, in both projections [La86] and general FSMs [Ok86].

3.1.2.1. Partitions

Partitioning a FSM is similar to the technique of transforming a nondeterministic finite automaton (NFA) into a deterministic one (DFA) [Sc82b], and to the partitioning principles of error detecting and correcting code analysis. Partitioning segments a FSM into independently verifiable components. Various methods determine an appropriate partition, including the use of artificial intelligence (AI) to differentiate between the explored and ignored components of state [Li87]. Mirage uses a similar partitioning in the communicability function, but ours is a dynamic process balancing partition and message sizes to ensure stable interaction among communicating entities.

3.1.2.2. Factoring

A FSM can be factored into its components. The reverse is more common, in which the product of component FSMs of a protocol generate the complete state space, factoring out statistically favored components. The remaining protocol is factored back down,

where possible. This results in a very efficient implementation, because a separate partition handles the most common FSM states, and a larger, more complex FSM is used only if necessary [Wo90].

3.1.2.3. Projections / imaging

Image protocols are formed by projecting functionality from the FSM. Given a specific function, an image of the FSM includes all states relevant to the provision of that function [La82]. These projections are similar to Mirage's modeling of remote state, which hides invisible state. They too can be extended to include latency and time [Sh82]. Projections also describe the partitioning of the state space in communicability, although in Mirage such partitioning is a dynamic process, whereas image protocols compute static partitions for analysis only.

Another version of imaging is found in entity models of protocols [Mo82]. These models consider the visibility of local state variables, in the same way as Mirage defines a perception.

3.1.2.4. Powerdomains

Mirage's analysis of state space subsets is similar to that of Powerdomains [Pl80]. Both Mirage and Powerdomains describe properties of the partitioning of state space into sets. Mirage bounds the size of the set of states as the system evolves; Powerdomains focus on the computability of the partition itself. A partition describing communicability therefore satisfies computability, because communicability requires a computable function to describe the evolution of the remote state space. Powerdomains consider set orderings as subset inclusion, whereas we consider a set ordering on size alone. These similarities are the result of the similarity between Dijkstra's guarded commands [Di76] (as analyzed by Powerdomains) and Mirage's guarded messages.

3.2. Protocol optimizations

The Mirage model can also be compared to model implications of specific protocol instances. Because Mirage addresses latency, especially in high speed wide area networks, comparisons focus on the model implications of light weight protocols

developed for similar domains [Do90], or on the model implications of flow control protocols for high speed networks.

Light weight protocols are developed by omission in functionality, hand-coded optimizations, or by limiting the protocol to a very specific domain. In contrast, Mirage is a general model for latency in communication, and it describes the optimizations used in light weight protocols in more general terms.

These light weight protocols include URP, VMTP, XTP, and versions of existing protocols (TCP) or variations that accommodate high bit latency (NetBlt). Many of these light weight protocols are instances of a general method we call ‘Cross Product Protocols’. New protocol designs have also focused on timers rather than packet exchanges, such as Delta-t, Virtual Clock, SNR (leaky bucket) and TP++. These protocols raise layering issues which Mirage avoids by its requirement of the predictability constraints of communicability.

3.2.1. Universal Receiver Protocol

The Universal Receiver Protocol (URP) [Fr89] is a data transport protocol that accommodates high bandwidth by a combination of omission and reorganizing the protocol components. URP assumes that the network will not reorder packets, and as such omits the reordering mechanism from the protocol specification. It also achieves high packet throughput by balancing the protocol load between receiver and transmitter.

One of the observations in URP is that conventional protocol code is unbalanced because most of the work is on the sender end of the protocol. Recent protocols mirror this imbalance, shifting the load onto the receiver (e.g., PROMPT) [Ba90a]. URP claims that a protocol is most efficient if the protocol work is evenly distributed between sender and receiver.

The Mirage model balances the actions of sender and receiver when the two interact evenly. The sender models the receiver, and anticipates its needs, whereas the receiver accepts and reacts to messages that correspond to its current state. In cases where the receiver can anticipate the reception of certain types of packets, Mirage would consider the data receiver to be modeling the state of the data sender. In Mirage the data flow direction doesn’t matter; state is modeled in either direction as desired, to the extent desired. The balance of the protocol depends only on the balance of the state modeling required.

Mirage's expansion of state space upon data transmission, and collapsing upon message receipt, has been examined in the design of buffer 'barriers' of URP [Fr89]. 'Barriers' is a modified flow control mechanism that attempts to equalize the uncertainty of communication among transmitter and receiver. Mirage's expansion and collapse of state modeling are the same as the expansion and collapse of buffer space allocation in barriers. Barriers apply to only buffer space state, and Mirage applies to the entire state of the remote node.

3.2.2. VMTP

The Versatile Message Transaction Protocol is a lightweight transfer protocol based on message transactions [Ch88b], [Ch89], [Ch86]. It provides a request-response capability, as in RPC (remote procedure call [Su88]), in addition to stream transmission, as in TCP. VMTP was "rethought from first principles," [Ch89], but these principles do not include domains of high latency, therefore much of VMTP is not applicable to Mirage's domain. VMTP reduces packet processing overhead and response latency, but does not mitigate the effects of transmission latency.

What is more important for future work in Mirage, VMTP combines concepts of RPC with NetBlt [Cl87] and flow protocols [Zh90]. Mirage may provide a model paradigm from which to unify these two datagram and data stream domains.

3.2.3. XTP

The Xpress Transfer Protocol (XTP) is another high speed transfer protocol [Ch88a], [Sa90]. XTP is designed for hardware implementation, and facilitates high speed communications, but was not designed for high latency.

XTP uses the exchange of state to manage rate control monitors, controlling the size and spacing of packets. The integral flow control mechanism manages the state exchange, and uses timers to force the exchange of state to reconnect after some types of failures. In the Mirage model, state is managed explicitly and the exchange of state is specified by imprecision, rather than on certain (error) events.

3.2.4. TCP

In Chapter 2, Mirage was analyzed with an extension of the conventional stream-based model of communication that the Transmission Control Protocol (TCP) provides [Po81a]. TCP supports only linear streams of data, whereas Mirage models branching streams as well. Protocols designed using Mirage reduce to TCP where the remote state is precise, i.e., where the stream does not branch.

TCP's sliding window flow control expects a linearity of the data stream at least as long as the window size requires, i.e., the round trip bit latency. Due to connection management overhead, TCP works efficiently where the data stream has a linearity that is at least an order of magnitude greater than the round trip time.

TCP has been extended for high latency domains by extending the maximum possible window size [Ja88a], and by modification of the windowed flow control feedback algorithms (also discussed later) [Ja88b]. These modifications assume that the round trip bit-latency can be filled with deterministically predicted data; Mirage assumes this is not the case, and that state imprecision will prevent effective utilization of the channel as a result.

Optimized TCP implementations have resulted in connections of 720 Megabits/second, demonstrating that TCP can operate in high speed networks [Ni91]. This verifies our original argument (Chapter 1), however, because these rates were achieved over very short (1-2 meters) distances (500 Megabits/sec), and in loopback mode (i.e., zero distance) (720 Megabits/sec). High speeds increase bit-latency, so because TCP works on a 45 Megabit WAN (the Internet), it should work equally well (modulo technological advances) at rates up to 10 Gigabits/sec on a MAN and up to a 100 Gigabits /sec on a LAN.

3.2.5. NetBlt

NetBlt is a variation of data transfer protocols optimized for bulk data transfer [Cl87]. NetBlt assumes that the data stream has a linearity that is an order of magnitude longer than TCP does, and so is generally less applicable to Mirage's domain than TCP is. NetBlt optimizes bulk transfers by amortizing packet and acknowledgment overhead over large blocks of data, and adds a pacing (rate control) mechanism. This pacing

mechanism is contained in the Mirage model description, i.e., the sender emits packets when its model of the receiver determines that a receive buffer is available, according to known state of the receiver and elapsed time.

3.2.6. 'Cross Product Protocols'

Another means to adapt existing protocols to high speed networks is a method of optimization we call 'Cross Product Protocols' (CPPs, for short). This includes methods of header prediction as in URP [Fr89] and TCP [Cl89], and methods of protocol layer merging and reduction as in XTP [Ch88a] and VMTP [Ch88b]. Header prediction and layer merging are both in the 'protocol bypass' [Wo90]. All these methods are included in the general method we call CPP.

The basis of a CPP is that the entire functionality of a protocol, either in implementation or layers of specification, is derived by a cross product of its components. Transfer protocols that implement a combination of the OSI transport and network layers are one example; protocols that combine features of RPC and streaming data transfer are another.

Indicated states are removed from the resulting protocol and treated as special cases, and the remainder of the protocol is factored down where possible. These states can be statistically favored (by direct measurement) [Kr85], selected to favor particular protocol operations, as in the 'protocol bypass', or determined by the current protocol state, as in the header prediction mechanisms of URP and an optimized TCP. The remaining states can be included in the resulting protocol, or removed to favor a 'lightweight' implementation (i.e., XTP, VMTP).

CPPs optimize protocol processing speed, but do not compensate for high bit latency. They are similar in focus to the statistical aspects of Mirage, in which pdfs determine expected values of the remote state. In addition, a CPP also collapses the *implementation* of the ISO protocol layer model, whereas the Mirage model considers the entire layer stack as the protocol. The result is that a CPP instance demonstrates how layering inhibits efficiency by restricting interlayer interaction, just as the Mirage model describes that layering prohibits latency compensation by hiding information required for the communicability function (i.e., regarding the state evolution function). Mirage can include layering, but only where it does not inhibit communicability (i.e., doesn't hide

information required for prediction), in a manner similar to that of ‘information flow’ protocol analysis [II87], in which information is hierarchically maintained.

3.2.7. Delta-t

The Delta-t protocol optimizes low latency response and high throughput stream transfer [Wa89], [Wa81]. The shared state of a connection is assumed from a default, and updated only when a connection is in use, thus avoiding explicit connection management. Implicit connections support datagram and RPC messages without connection overhead, and without unnecessary agreement of stream-oriented buffer, window, and rate values.

A connection is assumed to have a predefined state, unless a current connection record exists that supersedes that information. The role of timers in state management is also shown, i.e., that timers are required in all protocols in lossy environments. Mirage uses time information to determine perception evolution, and so timers are implicit in the model. A protocol designed using Mirage implicitly includes bounds on the state evolution, whereas Delta-t enforces bounds by functions expressing viable states, as in the functions that describe the retransmission strategy. The implicit state of an unused connection expresses the initial conditions for communication, i.e., there must *be* shared state to *maintain* shared state; Delta-t permits these initial values to be nontrivial, whereas explicit connection protocols consider unopened connections to express an ‘undefined’ state. Further, Delta-t describes the initiation and evolution of the connection management information, which Mirage considers the ‘protocol’ itself.

3.2.8. Virtual Clock

The Virtual Clock (VC) protocol emulates statistical multiplexing using temporal windows to control packet flow [Zh89], [Zh90]. It uses time-based state information of the receiver’s ability to process packets to regulate the sender’s scheduling of packet emission. VC’s control of receiver buffer usage via sender constraint equations thus exhibits aspects of Mirage’s temporal evolution function.

There are aspects of the VC protocol to which Mirage will be applied in the future, notably the resynchronization of the clocks. As time progresses, the sender accumulates rights to send packets, assuming that it will use those rights and that the receiver will be processing them. If the sender does not use these rights, they expire. VC removes transmission access rights by the periodic resynchronization of the clocks. This is similar

to expansion of the state in the Mirage model, and a constraint that the space will not expand beyond a determined bound.

3.2.9. SNR (leaky bucket)

The SNR ‘leaky bucket’ protocol is a transfer protocol based on periodic exchange of state [Ne90], [Sa89], whereas comparable protocols (NetBlt, VMTP, XTP, Delta-t, etc.) exchange state primarily at the occurrence of an event. Because the complete state is communicated asynchronously to such events, SNR is self-correcting, i.e., it recovers gracefully from corruption and loss errors without additional mechanisms. Whereas SNR claims to be self-stabilizing (with proofs forthcoming, [Go]), Mirage sets stability as the criterion for state information emission, so protocols designed using Mirage are stable by construction.

SNR is a linear data stream oriented protocol that incorporates amortization methods as in NetBlt, and minimizes packet processing overhead as in VMTP. Amortization assumes an increase in the expected linear data stream length, and prevents branched stream utilization. The protocol is simplified by the use of a single timer mechanism to maintain state, resulting in reduced overhead. Further, both rate and window based flow control are consequent to maintaining shared state, i.e., by sharing buffer availability status.

The exchange of complete state allegedly prevents the sender and receiver from being ‘out of sync’ due to high transmission latency. State information packets are sent in anticipation of their request to ensure management of the perception imprecision. This provides greater state synchronization than explicit request/response mechanisms and accommodates known latency, given deterministic remote state evolution. If the remote state evolves nondeterministically or if the latency is variable, the mechanism in SNR does not ensure synchronization of state. The reasons for this are described further in Chapter 4, in the analysis of the Network Time Protocol.

SNR causes the receiver to send state updates to the sender in a pipeline to keep the sender’s perception of the receiver as small as desired, whereas protocols designed using Mirage cause the sender to emit messages in anticipation of the receiver’s request. SNR can be viewed as a restricted implementation of Mirage’s anticipation mechanism, where the receiver in turn models the sender’s model of itself, and endeavors to constrain that model sufficiently by anticipating its temporal expansion. For example, the sender

models the receiver, but that model also includes expansion in its perception of the receiver. The receiver is in a stable state, and knows that the sender is not aware of that stability, so it pipelines state information, recollapsing the perception at the sender each time a message arrives there.

SNR uses periodic exchange of state as a tradeoff between bandwidth (assumed in excess) and reduced processing overhead, and yet uses selective retransmission for packet loss. As little as a single bit can govern flow control [Ra88], but SNR exchanges complete state both for redundancy (thus error resilience) and because excess bandwidth is available, preventing computationally intensive packet processing. In addition, if bandwidth is excessive, forward error correction should dominate packet loss solutions, with simple bandwidth-wasteful retransmission as a backup (i.e., go-back-N). Selective retransmission reduces bandwidth at the cost of processing, contrary to the other design decisions of the protocol; this indicates that the protocol takes advantage of reverse path bandwidth excess only. Using the forward path in a lossy way is not considered, whereas in the Mirage model it is considered a viable complement to latency.

3.2.10. TP++

TP++ [Fe90a] is an extension of TP4, the OSI Transport Class 4 protocol which provides error detection and recovery [Ro90]. It provides a combination of latency constrained, transaction (RPC), and bulk data transfer (data streaming) communication. TP++ uses multiplexing to reduce state information among virtual connections. Multiplexing is claimed beneficial in reducing bandwidth requirements, but detrimental where particular state information must be retrieved.

The Mirage model considers multiplexing to reduce state information harmful. Such multiplexing inhibits the determination of the communicability constraint, and prevents latency reduction by anticipation with guarded messages. The Mirage model assumes that bandwidth is not a premium, and should not be reduced in ways that restrict latency compensation.

One statement associated with TP++ but not explicitly in the available literature is that of "running until you run out of state."[1] This (unstated) assumption of protocol design is the explicit basis for the communicability and stability constraints in Mirage.

3.3. Communicability

One of the fundamental characteristics of Mirage is the communicability constraint, which indicates the conditions under which fixed latency can be accommodated by a protocol. Communicability denotes the requisite partitioning of the remote state volume, the need for guarded messages, and the need to express the temporal evolution of the remote state. Communicability also specifies the conditions under which stability can be maintained.

The basis for communicability lies in traditional communication theory, of cybernetics and control theory. Its implementation is similar to constraint mechanisms in monitors and methods for real time systems.

3.3.1. Cybernetics and control theory

Mirage is most fundamentally based on notions of controllability from cybernetics and control theory [As56], [Wi48]. Cybernetics is the basis of stability, as extended in Mirage, and sender anticipation. Mirage's use of entropy as a measure of information dates back to Hartley [Ha28], although originally attributed to John von Neumann. This was described in Shannon's communication theory, whose relevance to Mirage is elaborated in Appendix A [Sh63].

Stability has three variations — equilibrium, cycle stability, and phase space stability [As56]. Equilibrium occurs when a system reaches a single unchanging state. Cycle stability occurs when a system enters a set of states that is subsequently never exited. Phase space stability occurs when a system enters a region of phase space that is subsequently never exited. All of these types of stability are traditional, as referred to by 'stability' in Chapter 2; Mirage adds another type, entropic stability, as a further

---

[1]This was mentioned by David Feldmeier in a presentation on TP++ at IFIP Protocols for High Speed Networks, 1990.

extension of these, in which stability occurs when the volume of a region of phase space is constant, rather than the particular region being fixed.

In addition, the TreeStack structure (Chapter 5 and Appendix G) in particular and the perception that it represents are encodings of persistent phase space trajectories (PSTs). PSTs are used in control and feedback theory to determine the stability of a system, usually defined as the confinement of trajectories to some bounded region of phase space.

In Mirage, perception encodes pending PST information. A perception is an afterimage of the traces of the possible futures of the remote state, held for the duration of the latency, until feedback resolves imprecision in the PST. This explains the need in Mirage to extend the notion of stability to include volume stability as well as the traditional state stability. In traditional control theory, PSTs converge to a finite region, whereas Mirage ensures stability by bounding the size of the afterimage, regardless of the path of the PST itself.

3.3.2. Time

There exist models of time in protocols that are more closely related to that of the Mirage model [Sc82b]. Incorporating time as a valid period for each state of a protocol machine is similar to denoting the interval over which the expansion of the subspace is well-defined [Ag83]. The extension of this method that presents hold times of protocol states as cumulative distribution functions is similar to a time extrapolation of this state space.

Time constraints have also been added to conventional protocol models, including timed Petri Nets [Me76], protocol projections [Sh82], and finite state machines [Sh85]. Time boundaries have also been used to denote limitations on state expansion explicitly [Fe90b], as in Mirage. These constraints are usually expressed as boundaries, but some describe state as a distribution function [Ag83], as does Mirage.

The incorporation of time into protocols has thus far been limited to two methods, where time is modeled either by boundaries or by finite time-steps. In the former, time is denoted by "$T_1 \leq T \leq T_2$", for some T_1, T_2. Actions occur upon violation of these boundaries, as in time Petri Nets, temporal logic, or time-out timers [Sc82b]. In finite time-step models actions occur at some time instant, where "$T = T_1$". In Mirage, time is a

fully parametric value. The values of all other entities may change over time, and no barriers or finite points exist. Time is a continuous entity, over which other entities vary.

3.3.2.1. Real time systems

Real time systems also exhibit constraint mechanisms, usually in either an analysis or runtime scheduling capability. Here ‘real time’ means that operations terminate within a predeclared deadline, as opposed to the less formal definition of speed, i.e., fast enough to support ‘interactive’ requests. Real time system description languages include Hoare’s Communicating Sequential Processes (CSP) [Ho78] and Milner’s Calculus of Communicating Systems (CCS) [Mi80]. Mirage includes a scheduling constraint notion in communicability, i.e., in the scheduling of packets during the latency.

3.3.2.2. Aging variables

The modeling of time as an aging variable [Sc82b] is not as general as that in Mirage. Time markers age, but other entities do not vary with time. Aging reduces the precision of an entity. The aging of other variables may also affect the aging of a variable, so the function of time on the variables is arbitrary and variables are interdependent. In the Mirage model, time transforms the subspaces arbitrarily; the transformation is known at the time it is invoked, but the model does not restrict the transformation a-priori.

3.3.2.3. Timers

Other systems model time not by a variable but by timers that govern error recovery as in TP++ [Fe90c] and Delta-t [Wa89], or state maintenance as in SNR [Sa89] or Delta-t [Wa89]. Mirage uses time to model the entire state space, not just flow parameters, error control, or recovery state.

Time can also be a boundary, as in Real Time Communication (RTC) [Fe90b], including statistical boundaries, as in Mirage. In RTC there are temporal constraint formulae as in Mirage, including those for connection management, buffer availability, delay bounds, and statistical properties of ‘saturation’ (real time deadline overflow, i.e., saturation of the scheduling mechanism). Rate based flow control is a side effect of this use of timers and constraint equations in RTC, and future research in Mirage may also show other aspects of existing flow control protocols to be side effects of maintaining the communicability and stability constraints.

3.3.3. Constraints

The notion of restricting a machine to operate only within the valid subspace is an extension of distributed/replicated database techniques, most notably read/write quorum strategies [Gi79]. In addition, there have existed designs for external monitors that maintain constraints on a system, one of which is called the Overseer [Fa76a]. The use of these environments or supplemental programs to warn of dead-ends, maintain locality, and restrict other programs to within some valid subspace is similar to the methods used here. Mirage differs in that it appears that these notions are central to the operation of the protocol, not external, supplemental constraining devices.

Common knowledge methods of constraint are also relevant to Mirage, including conventional common knowledge [Ha84], database knowledge as in quorum consensus [Gi79], and in protocol analysis [Go88]. Each of these uses individual inference based on group constraints, as applied to incomplete information about the group. Mirage uses individual constraints, but includes similar inference, although common knowledge does not include Mirage's anticipation mechanisms.

Common knowledge as applied to protocols [Go88] denotes knowledge as a superset of state, e.g., knowledge includes facts true in all possible states, whereas state may include particular facts which may be asserted or omitted (although not inhibited). Mirage's 'perception' models locally all possible states of a remote entity, so in Mirage knowledge is explicitly denoted by the modeling of state. For example, if some states in a perception require a message emission, and no other states inhibit it, the sender would emit the message guarded to be received by the indicated states only.

3.4. Anticipation

Prediction in cybernetics utilizes constraints with sender/receiver modeling [As56]. This permits regulation, defined as isolating an external observer from observing internal system effects. In Mirage, this corresponds to anticipatory latency accommodation as it isolates the communicating entities from observing the actual latency between them, i.e., from being adversely affected by the latency. Regulation, also referred to as homeostasis, reflecting cybernetics as focused on biological systems, also is described in the presence of error (i.e., latency), and in the presence of constraints on the anticipatory control capability (i.e., communicability).

Anticipatory feedback is defined as the action of a ‘compensator’ (sender) when the ‘effector’ (receiver) has a time lag in action. As noted therein, “the conditions of stability and effectiveness of anticipatory feedbacks need a more thorough discussion than they have yet received,” [Wi48] which we believe is just as true today.

Anticipation mechanisms that are similar to Mirage exist in operating systems and congestion control, as well as areas of client/server extensions and some recent discussions in protocol research.

3.4.1. Operating systems

Most versions of anticipation use a depth-first search (DFS) of the possible state space. Some extend this to a version of breadth-first search (BFS) that examines some paths of possible state, and either terminates all but the desired path [Sm89] or rolls back the state of the paths that fail (Time Warp) [Je85]. Mirage exhibits BFS anticipation without rollback, but the source of the indeterminism (of the BFS tree) is latency induced imprecision, rather than algorithmically explicit, e.g., Monte Carlo execution methods [Sm89].

3.4.1.1. Concurrent execution

Jonathan Smith’s dissertation addressed the speedup of BFS searches by distributed BFS execution, with successful paths preempting the execution of concurrent attempts [Sm89]. Mirage is similar in the use of excess resources to save time. Mirage differs from such a BFS search in that entire levels of the possible state ‘tree’ are accounted by messages, so no preempting of failed paths is required. ‘Sibling elimination’ collapses the ensemble of processors to a single valid member, whereas Mirage requires a more intricate TreeStack pruning to remove models of multiple possible remote states.

Smith’s method is similar to Mirage if the set of remote processors is considered a single remote entity; in that case Mirage’s state imprecision corresponds to Smith’s indeterminism of process execution time or termination. Smith’s method uses BFS to permit multiple processors to participate in latency reduction in ensemble fashion, whereas Mirage models temporal imprecision in the state of the remote node.

3.4.1.2. Time Warp

The Time Warp system uses multiple distributed processes, each looking ahead, with rollback providing resynchronization (i.e., Time Warping) [Je85]. Time Warp is a distributed system, but in one of our first discussions (Chapter 1) we describe that there is little difference between a distributed system mechanism and a protocol, for which Mirage is designed. Even though rollback is a common method of error recovery in distributed systems [Ra78], Time Warp uses rollback to permit the distributed processes to proceed into possible future paths. Rollback occurs when a particular future of a participant is inconsistent with that of some other participant.

Messages in Time Warp are modeled similarly in Mirage, with sent messages being modeled as being both received and not received, and resolution of state occurring at a later time. In Time Warp the local process examines a single DFS path into the future, whereas Mirage examines all paths in BFS order, avoiding rollback. Resolution in Time Warp occurs via rollback, whereas in Mirage it occurs when the modeled state collapses using the subtree selection mechanism of the TreeStack.

Time Warp differs also in that it deals with message loss as the source of state imprecision, rather than time lag, as in Mirage.

3.4.2. Congestion control

Anticipation is beginning to appear in congestion control methods as well. We distinguish general methods of anticipation from congestion control anticipation because the latter uses a restricted state space. Congestion control, also called flow control, is performed by feedback, timers, anticipation, or a combination of these. The relative merits of each system can be evaluated as compared to Mirage.

3.4.2.1. Anticipatory congestion control

Predictive congestion control (PCC) “predict(s) (the) behavior of the system at a specific time instant in the future” [Ko90]. Mirage assumes that propagation delays dominate the system, whereas PCC claims that switching delays dominate. PCC attempts to reduce the switching latencies caused by buffering. In either case, the systems are based on the notion that the “state of a node may change considerably between sending

messages (with control) and receiving them" [Ko90]. In Mirage, all messages control the state space, but the assumption that state varies during communication latency is the same.

Reactive congestion control fails due to the fixed latency, so proactive measures are indicated. PCC is similar in principle to Virtual Clock, albeit derived differently. PCC, like Delta-t and SNR, uses periodic exchange of data and rate control state to govern flow control. PCC is directed at individual hop flow control, but extends to end-to-end methods as well.

Mirage is also proactive, although as it regards the entire controlling state of a node, rather than just the flow of data. Mirage is thus a more general form of PCC, based on more general principles of state sharing and stability.

3.4.2.2. Timer-based congestion control

The Stop-and-Go flow control protocol reduces latency jitter in packet transmission [Go90]. This protocol is designed to be used where the state of the system (available buffers) is deterministic with respect to time, but where latency jitter induces the same imprecision in remote state perception as nondeterministic state evolution would have. This isomorphism is addressed in Chapter 4. Jitter is reduced at the expense of an increase in the minimum latency, which is advantageous only where the remote state imprecision is a function of jitter alone, i.e., where the remote state is otherwise deterministic. Such is the case in some time protocols, such as the Network Time Protocol [Mi90b], also addressed further in Chapter 4.

3.4.2.3. Feedback congestion control

Other mechanisms more indirectly infer feedback information for flow control [Ja89]. Jain's mechanism uses round trip time variability to determine network overload, and to adjust packet flow accordingly, but this is a reactive method of control, in which adjustments occur after a round trip latency. Reactive flow control works properly on the time scale of tens of round trip times, whereas proactive control methods, such as in Mirage, are required for more spontaneous activity [Ja90].

Jain also argues for multidimensional flow control, noting that window vs. rate control, open vs. closed feedback, router vs. source controls, and reservation vs. 'walk-in' bandwidth allocation mechanisms each affect the entire scheme of flow. We consider that these are each an endpoint in a continuum, and that the entire set of controls denotes a

space of possible flow protocols, because each requires additional state modeling in Mirage. Conventional mechanisms investigate endpoints of the flow protocol space, whereas further investigation using Mirage may demonstrate protocol versions that exhibit any location in that space.

3.4.2.4. Combination control

Other attempts have been made to reexamine round trip time estimation, as it pertains to flow control and state variability, including those by Jacobson [Ja88b]. In terms of Mirage, state is the send window size and the rate parameter of the flow control, as managed by predictions of round trip time and altered by feedback information. Initial state is assumed to have a unit window and a minimal (one-time emission) rate, which is a minimal initial state resulting in sustained communication, as compared to the non-minimal initial state of the Delta-t protocol. Methods to adjust this state are expressed by the so-called Slow Start and Exponential Backoff algorithms, and the inverse exponential 'round trip time' sequence averaging. In the past, these algorithms have been considered extraneous to the declaration of the TCP protocol [Po81a], but Mirage indicates that the TCP protocol cannot be modeled without them, because these equations govern the evolution of state and thus communicability and stability.

3.4.3. Other forms of anticipation

Other versions of anticipation include extensions to client/server models, recent conference presentations in protocols, and most notably computer architecture. The similarity between methods of protocol anticipation in Mirage and computer architecture is described in detail in Chapters 5 and 6, in which we describe a novel architecture for processor-memory interaction using anticipation mechanisms in active memory architectures.

3.4.3.1. Client / server extensions

Existing client/server protocols exhibit traditional request/response protocols, such as Remote Procedure Call (RPC) [Su88] and the Network File System (NFS) [Su89]. Extensions to RPC include Remote EValuation (REV), which sends the code to the data, rather than the conventional converse of RPC [St90]. A further extension of REV, called Late Binding RPC, provides for some kinds of anticipation [Pa91]. The difficulty with

RPC mechanisms is that the evolution of the perception of the remote space is governed by the function within the RPC. The RPC functions thus determine communicability, and their variation makes stability more difficult to ensure.

3.4.3.2. Recent protocol discussions

A recent conference in high speed protocols resulted in several current considerations which are similar to components of Mirage [Pa90b]. These include 'parallel RPC', network anticipation, and 'asynchronous RPC'.

'Parallel RPC' (of Craig Partridge) requests data in excess of expected utilization, in the hope that some will be of use. Excess bandwidth is wasted to keep the CPU fed, in the prediction that CPU starvation will result from increased bit latency. This method is similar to the parallelization in Jonathan Smith's method of distributed execution [Sm89], which is ensemble based, whereas Mirage is based on possible states of a single remote process.

'Network anticipation' (of Jonathan Smith) is a receiver version of sender anticipation in Mirage, and is similarly used to keep a starved CPU fed. This method is the network analog of conventional cache anticipation mechanisms, whereas Mirage suggests new versions of active memory anticipation in contrast. Sender anticipation in active memories is addressed in Chapters 5 and 6.

'Asynchronous RPC' is a new paradigm for interaction that accommodates latency (Thomas Joseph). He notes that asynchronous RPC may require larger messages, but that wasting bandwidth to permit asynchronous return frees the local processor during response latency. A-RPC is not anticipatory, as Mirage is, but it does provide a mechanism in which Mirage's analysis may also prove useful.

David Cheriton also presented a talk recently [Ch91] about 'dissemination oriented communication' (DOC). It describes an eager communication system, in which round trip latency is avoided for conditional behavior. This system is similar to a SIMD[1] conditionally executed instruction, i.e., where opcodes are executed or ignored depending upon local processor state; messages are sent and accepted only if the receiver is in the proscribed state. Such messages are the same as Mirage's guarded messages, although the guard performs different functions in each. DOC is geared towards conditional execution

---

[1]Single Instruction Multiple Data, denoting a multiprocessor system in which opcodes are broadcast to the processors, each of which operates on its own data stream.

in an ensemble of processes, as in the SIMD system, whereas Mirage models imprecision in knowledge of a particular state (i.e., temporal imprecision).

3.5. Physics analogs

Mirage was originally developed from analogies from physics, as described in Appendix B. The analogies are based on the multiple-worlds model of quantum interaction, and the Feynman path integral method for particle interaction analysis. The multiple-worlds model is also similar to those in Truth Maintenance Systems, which describe data structures for maintaining simultaneous logical states, similar to the TreeStack structure developed for the μ-Net architecture presented in Chapter 5, and discussed in Appendix G.

3.5.1. Truth Maintenance Systems

The term 'Truth Maintenance System' (TMS) was coined by Jon Doyle, and refers to a logic system that maintains possible logical states [Do77], [Do79]. This is similar to the sender's perception of the receiver as modeled as a set of possible states in Mirage. In a TMS, this set of states is modeled by a data structure, which is maintained by a combination of instantiating extensions and collapsing revisions; in Mirage, the set is expanded upon message emission, and collapsed upon message reception. The perception in Mirage has been extended to model recursion in the remote state (in μ-Net, in Chapter 5), but it is not clear whether a corresponding recursive TMS exists, although recursion is used in TMS methods.

3.5.2. Physics in protocols

The Computational Field Model (CFM) is recent research in the direct application of physics analogies to distributed systems issues [To90b]. CFM is an "object oriented open distributed environment" that addresses the issues of computing processes (i.e., programs) affecting the environment of the computation, permitting redistribution of the processes. CFM also notes that message passing and object oriented hidden messages are not capable of modeling communication latency; instead, a model of "assimilation / dissimilation" is developed.

CFM defines distributed computing as parallel computing with interprocess communication delay, similarly to our definitions of communication in the Introduction (Chapter 1). *Distance* is defined as geographic distance, under the presumption that this is proportional to communication latency. *Mass* is defined as the size of the object, and *inertia* as the migration overhead. These definitions are not otherwise justified, i.e., inertia is necessarily linearly related to mass, as in their physical analogs.

Gravitational force is defined as the bandwidth of communication (size and frequency of communication, therein). A counteracting *repulsive force* is defined as "the product of the size of the objects over the distance to the n-th power." No rationale for the definition of these forces is given. Gravity should be a benefit function that causes objects to migrate together. Repulsion is a cost function causing objects to migrate apart.

A message is considered a "special object," whose migration is handled through the addition of primitives of dissimilation, migration, and assimilation primitives. It is not clear why a more unified approach is not used, where messages are simply objects that are defined as having an infinite repulsion to the sender and an infinite gravity to the receiver. *Assimilation* is message creation and *dissimilation* is message absorption, which are required in either case to describe object 'fission' and 'fusion' (terms we prefer, because they imply messages and nodes are the same 'matter-energy'). In CFM, delivery service classes and timeouts can be specified in terms of message inertia, mass, and lifetime to decay, rather than introduced as separate characteristics of migration.

The CFM bears a resemblance to our original model basis, in which communicating entities are described as particles, and communication is described as particle exchange. Our original model, however, does not differentiate messages from endpoints, and presents a more unified approach that we feel more accurately adapts physics analogs to communications principles. This may be the effect of our choice of utilizing the particle exchange description of physical forces, rather than fields, described in detail in Appendix B. Particle exchange seems more accurate because messages represent particles. The field model of forces is likely to be more appropriate to repulsive cost description, because multiple objects should be prevented from sharing a single processor. This latter field is a binary function, because there should be no difference in the repulsion between objects on two adjacent vs. two separated processors; other costs of interaction (communication) should govern their attraction, or permit them to move about independently.

CHAPTER 4

A Mirage of NTP

This is an application of the Mirage model to an existing protocol, the Network Time Protocol (NTP). We chose this protocol for its simplicity and its prevalence. Here we apply the abstract principles of Mirage and perform an experiment that helps further refine the Mirage model.

This analysis demonstrates the equivalence between latency variability and imprecision in the local perception of the remote state, thus extending the domain of the Mirage model to variable latency regimes. We also examined how Mirage relies on time measurements, due to the seeming contradiction in modeling a clock protocol with a time-based model.

We concluded that several optional components of the NTP specification are integral to our model of NTP in Mirage, and that these components should therefore be required. These components include the logical clock, peer dispersion, and data filter algorithms.

The Mirage model of NTP reduces to a hardware clock signal where delay variability is zero. This implies that the variability of the latency limits clock synchronization, rather than the overall latency magnitude. This reduction resulted in a refinement of the variability of the measured offset.

NTP has proven useful in demonstrating the application of some basic components of Mirage. Unfortunately, many of the novel aspects of Mirage are not applicable to NTP, e.g., guarded messages, state space partitioning, and communicability tradeoffs. Later chapters of this dissertation (μ-Net in Chapter 5, and μ-Scope in Chapter 6) demonstrate those components that NTP cannot.

4.1. An overview of NTP

We begin with a summary of NTP. Whereas there have been 4 versions of NTP (numbered 0 through 3), this exercise focuses on core aspects of the protocol, which have changed little in recent versions. Thus Version 3 is referenced here [Mi90b].

The Network Time Protocol is a protocol for synchronizing system clocks. It consists of a request/response engine for exchanging timestamp messages, a logical clock providing an adjustable time reference, and a set of algorithms for integrating sets of message responses to determine an appropriate adjustment.

The logical clock provides a mechanism for a user-adjustable time reference, if not already explicit in the node's operating system. The NTP logical clock engine is named the *Fuzzball*. The clock consists of an absolute offset, called an *epoch*, and a frequency ratio scale. The logical clock is defined as the epoch plus the hardware elapsed time multiplied by the frequency ratio, so an otherwise fixed system clock can be adjusted in both absolute offset and frequency. *Drift*, the variability in frequency, is not compensated for in the logical clock provided in NTP, and is assumed to be zero. Furthermore, the logical clock is described as 'optional' in the NTP specification. Between two logical clocks, *offset* is the difference in epochs, *skew* is the difference in frequency. *Dispersion* is the known error in the local clock.

NTP also includes a mechanism for organizing a set of clock servers into a hierarchy based on clock *strata*, where stratum is a group of clocks at a specified precision. Descriptions of various strata are given in [Mi90b]. Lower strata (e.g., Stratum 1) describe more accurate clocks; strata range from 1 to 255, with 0 representing an unspecified stratum. Stratum 1 clocks are called primary servers; strata 2-255 are called secondary. Stratum 1 clocks are the root(s) of the timeserver hierarchy, and a stratum *k* server is defined as having a distance of *k* -1 to the nearest root.

The core of the NTP protocol is a client/server engine that stamps messages when sent and received. A message initiated at a host returns with 3 timestamps in its packet body (sender out 'originate', receiver in 'receive', receiver out 'transmit') and a fourth stamp retained upon receipt of the message, but separate from it (sender in 'input') (Figure 4.1). Received messages are automatically replied, and sent messages are periodically initiated according to local state information.

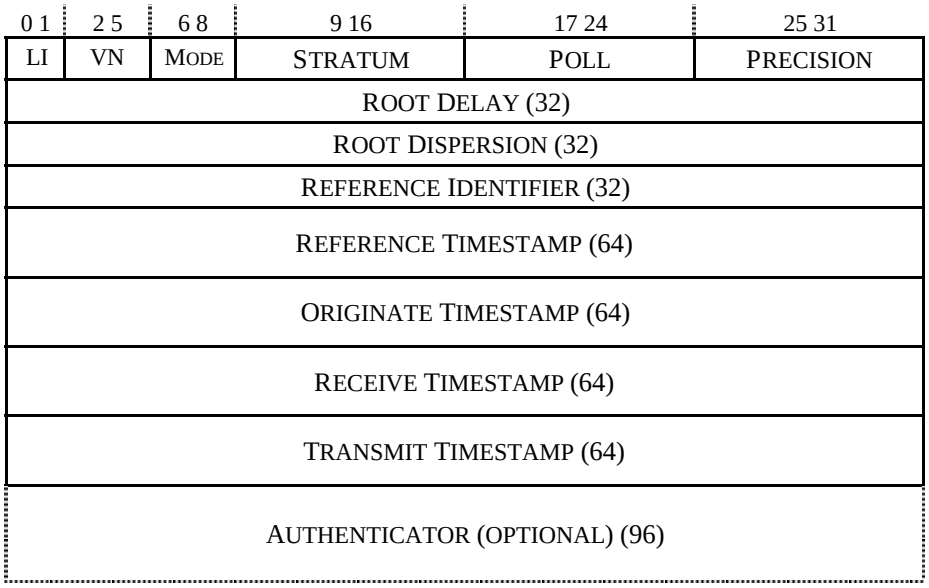


FIGURE 4.1
NTP message format

The local state information is combined with sets of replies. These replies are processed to prevent individual server clock errors from affecting clocks of their clients, where possible. Various combining and filtering algorithms provide this processing, and adjust the current local clock and maintain the clock server hierarchy. The hierarchy determines which servers will be consulted for future clock data.

There are two types of sets of message replies considered in NTP. Temporal sets of data are taken at different times from the same server; ensemble sets are taken at (nearly) the same time from a set of different servers.[1] Temporal sets are held in a shift register, and combined using a *filter algorithm*; ensemble sets are combined using the *peer selection algorithm* and the *combining algorithm*.

[1]The terms 'temporal set' and 'ensemble set' occur in both statistics and physics, as they are used here.

Other components of NTP include control messaging, authentication, and asymmetric modes of operation. The conventional symmetric mode of operation is outlined above; asymmetric modes include servers broadcasting to a set of workstation clients, and those clients, as well as root servers (read-only clocks). Control messaging provides user-level access to protocol state, for remote monitoring and manipulation. Authentication provides security and additional protection from Byzantine failures [Pe80]. These components are optional in the NTP specification. They are not considered in this analysis because they are supplemental to the basic operation of the protocol.

4.1.1. How NTP reads a clock

NTP reads a remote clock by sending a NTP message to the remote node (Figure 4.1), and waiting for a reply. Upon receipt of the reply message, the client/server engine pairs the incoming packet with the current local time (‘input time’, a.k.a. ‘peer receive’), and processes it. The message is stamped at each reception and emission (Figure 4.2).

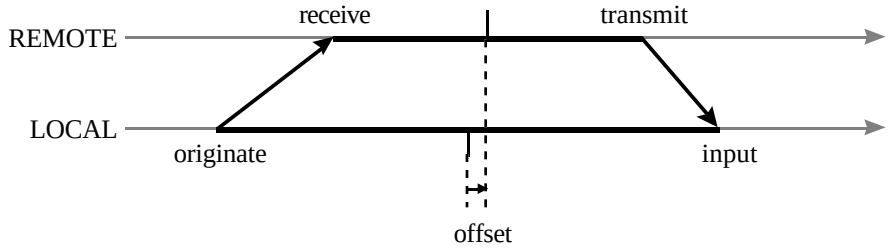


FIGURE 4.2
NTP message exchange

The four timestamps – originate, receive, transmit, and peer receive (input) – are used to compute round trip time and offset measurements. The round trip time (δ) can be computed from the difference between the sender interval and the receiver interval (Equation 4.1). This formula appears in the NTP specification.

‘Clock time’ is defined as the average over an interval, i.e., local time is the average of the originate and input values, and remote time is the average of the receive and transmit values (Equations 4.2, 4.3). ‘Offset’ (θ) is defined as the difference between the remote clock time and the local clock time, where a positive offset indicates that the remote clock is ahead of the local clock (Equation 4.4, Figure 4.2). This formula is

derived differently than in the NTP specification, but the result is the same (Equations 4.5, 4.6).

Equation 4.1: $$\delta = \left(T_{in} - T_{orig}\right) - \left(T_{xmit} - T_{recv}\right)$$

Equation 4.2: $$T_{local} = \frac{\left(T_{in} + T_{orig}\right)}{2}$$

Equation 4.3: $$T_{remote} = \frac{\left(T_{xmit} + T_{recv}\right)}{2}$$

Equation 4.4: $$\theta = T_{remote} - T_{local}$$

Equation 4.5: $$\theta = \frac{\left(T_{xmit} + T_{recv}\right)}{2} - \frac{\left(T_{in} + T_{orig}\right)}{2}$$

Equation 4.6: $$\theta = \frac{\left(T_{recv} - T_{orig}\right) + \left(T_{xmit} - T_{in}\right)}{2}$$

The calculated round trip delay is ‘exact’, i.e., the four timestamps exactly determine the total round trip time. The offset calculation, however is not similarly exact. The NTP formula for offset makes several assumptions: (1) that clock time varies only linearly during the exchange, and (2) that outgoing transit time and incoming transit times are identical . Assumption (1) is required – were it not, the protocol would not be able to determine a clock value. The clock value is thus defined to be the linear interpolation of the send and receive times of the message.

The calculated offset is affected by the difference between the outgoing and incoming transit times. Outgoing transit time adds to the measured offset, whereas incoming transit time subtracts from the measured offset. Under no other assumptions, the one-way delays are bounded between 0 and the total round trip time (δ) (Equation 4.7). This loose bound on delay results in a similarly loose constraint on calculated offsets (Equation 4.8). These bounds are shown visually (Figures 4.3, 4.4). Figure 4.3 shows that if outgoing delay is larger than incoming delay, the offset is incorrectly

measured as larger than intended; if incoming delay dominates the round trip time (Figure 4.4), the offset is incorrectly measured as smaller (falsely negative).

Other boundaries exist in the specification that require local time lines to be non-negative, a receive time between the originate and transmit times, and a transmit time between the receive and input times as well.

Equation 4.7: $0 \geq \delta_{unidirectional} \geq \delta_{bidirectional}$

Equation 4.8: $\theta + \frac{\delta}{2} \geq \hat{\theta} \geq \theta - \frac{\delta}{2}$

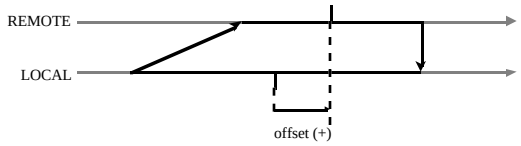


FIGURE 4.3
Exchange slid forward (maximum offset)

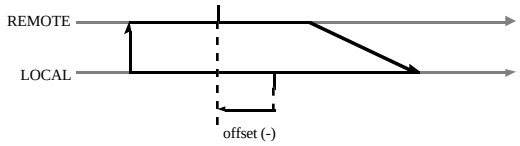


FIGURE 4.4
Exchange slid backward (minimum offset)

4.1.2. NTP background

NTP relies on the UDP protocol for connectionless transport-layer services [Po80], i.e., unreliable datagram services. UDP provides user-level access to the IP protocol [Po81b] and is the basis of the request/response transport mechanism, but the NTP protocol does not require either IP or UDP specifically. NTP evolved from earlier time protocol extensions of the Internet protocol suite, specifically the Time Protocol [Po83] and ICMP Timestamp message [Po81c].

Various versions of NTP exist, beginning with the original RFC [Mi85], which included only data and packet formats, and the specification of the client/server engine. Version 1 [Mi88] added a self-organizing clock hierarchy and a logical clock algorithm, neither of which is integral to the protocol. Version 2 added authentication, control message capability, and asymmetric modes of operation, also supplemental to the protocol [Mi89b]. The latest NTP, Version 3 [Mi90b], includes an “overhauled” local clock algorithm, and new algorithms for peer offset combination; it also refines the definitions of delay, offset, and dispersion.

We use Version 3 NTP for our discussion and analysis. A summary of the version enhancements of NTP appears in Table 4.1. “Required” indicates whether the NTP specification denotes the component as required or optional, and “first version” indicates the version number in which the component first appeared.

NTP component	Required?	First version
packet exchange engine	y	0
logical adjustable clock	n	1
self organizing hierarchy	y	1
overcome unreliable	y	1
peer select / filter	n	1
control message	y	2
asymmetric modes	y	2
authentication	n	2

TABLE 4.1
NTP versions and components.

4.2. Casting NTP into Mirage

Viewing NTP using the Mirage model (‘casting’) involves describing the state space of NTP and interpreting messages of NTP in terms of the state space transformations of Mirage (Chapter 2). We define the state space of a protocol by the variables that characterize the protocol operation, rather than by those of the protocol specification. We differentiate the state that the protocol manipulates from the state required to manage that manipulation, in the sense of first-order state and second-order state. First order state characterizes the state space of the protocol, whereas second order state helps define varying characteristics of the transformation functions.

The Mirage model contains numerous references to time as absolute, so it may seem confusing to apply it to a protocol that manages clocks. Prior work in clock synchronization has called a clock “a function that maps real time to clock time” [Ma85], so the ‘time’ variable that NTP manages is just another linear state space. Clock protocols manage the agreement of a single variable (clock value), as it changes over time. Although time is used to manage the clock (e.g., in NTP using the polling interval), the effect of clock manipulations on the time variable of the protocol is ignored. NTP (and Mirage) assume that such variations are small in comparison to the intervals measured.

NTP endeavors to replicate a remote clock value that is assumed to be precise and accurate in its local space. The precision and accuracy of this clock are denoted by user classification, not within the protocol. The correspondence between the clock value and real time (true accuracy) is a semantic issue that is arguably not provable and that neither NTP nor Mirage proposes to address.

4.3. Resolution of domain differences

The Mirage model is designed for the domain where messages are delayed by a fixed, known amount, and the remote process exhibits variability in its computation. Conversely, NTP operates where messages are delayed by a variable amount (and potentially lost), and the remote process (clock) has very little intrinsic variability (error). We first describe how variability in message latency is equivalent to variability in computation at the remote node.

In NTP a round trip message exchange specifies a delay and offset; delay is the round trip message latency, and offset is the difference between the clock values. Consider the case where two nodes are absolutely correct, but the communication latencies vary. The true offset is zero, but the measured offset is not.

Let the unidirectional message delay be a probability density function (pdf) approximated by a Poisson distribution (Figure 4.5). The round trip delay is the sum of the unidirectional delays, and can be computed by the convolution of the unidirectional delay with itself (Figure 4.6). The offset is affected by the difference of the unidirectional delays, because outgoing delay causes the remote clock to be measured as late, whereas incoming (return) delay causes the remote clock to be measured as early. The difference in these delays is the convolution of the unidirectional delay with a reverse of itself ($f(-x)$,

as in Figure 4.7). Both these formulae assume that the unidirectional delays are identical pdfs.

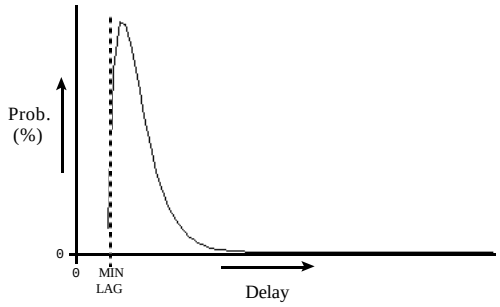


FIGURE 4.5
Unidirectional delay as a Poisson pdf[1]

FIGURE 4.6
Bidirectional delay is the convolution of unidirectional delays

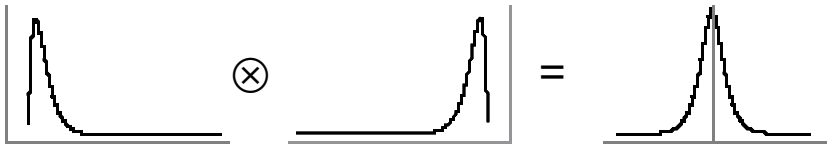


FIGURE 4.7
Measured offset is unidirectional delay convolved with its reverse

The resulting bidirectional delay has an Erlangian distribution (by definition[2]) (Figure 4.8). The measured offset exhibits the effects of the difference between incoming and outgoing delay, and that even perfect clocks are measured as statistically variable when communication latencies vary (Figure 4.9). The consequent offset pdf is symmetric

---

[1]probability density function.

[2]An Erlangian distribution is defined as the convolution of a fixed number of identical Poisson distributions. In a queuing network, if each stage is Poisson in processing delay, then a path through the network exhibits Erlangian delay.

about the zero offset, and appears Gaussian in form. An asymmetric pdf would result if the communication latency were asymmetric (forward vs. reverse paths).

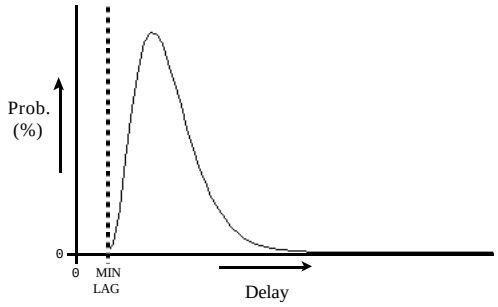


FIGURE 4.8
Bidirectional delay as Erlangian (N=2)

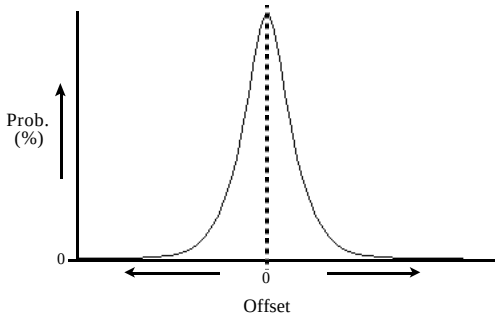


FIGURE 4.9
Measured offset as pseudo-Gaussian.

The bidirectional delay and measured offset of this system can be combined by an outer product, to show one possible structure of a delay/offset pair (Figure 4.10). The internal structure of such a graph cannot be determined from the delay and offset pdfs alone, because two one-dimensional pdfs underspecify the 2-dimensional delay/offset pdf. Even so, the outer product shows a structure similar to that of real NTP measurements, shown later. The result is that, even in the case of perfectly aligned clocks, a wide distribution of offset values occurs.

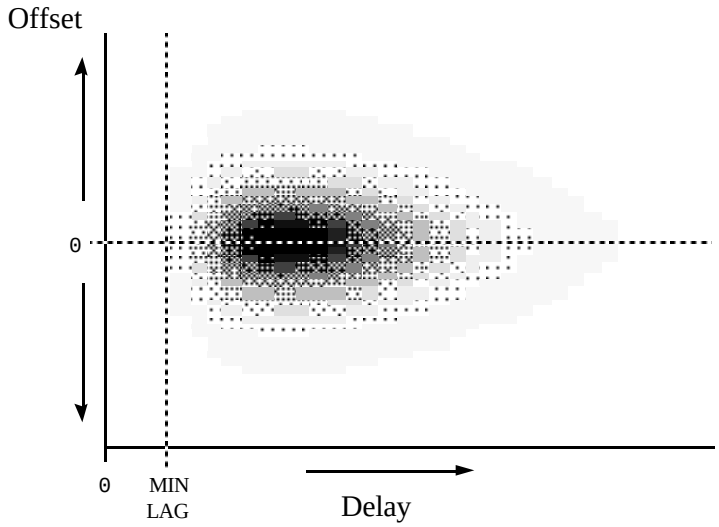


FIGURE 4.10
Probability vs. [delay, offset] pairs, density plot.

4.3.1. Analysis of delay and offset measurements

To verify these assumptions, we performed some measurements of NTP. We used a UNIX[1] shell script to exchange NTP packets (with timeout) between a local host (in Pennsylvania) with a remote node in California (a Stratum 2 NTP server). Each set of measurements reflects 2200 NTP requests (with a failure rate of approximately 1.5%, or 33 lost requests), at a rate of approximately 200 per hour, for a total of 11 hours, beginning at the time indicated. "Friday 8am" indicates a peak period (Friday 8am – 7pm EDT), and "Tuesday 8pm" indicates an off-peak period (Tuesday 8pm – 7am EDT).

The offset curves are Gaussian-like (Figures 4.11, 4.12), in which the peak measurements (Figure 4.12) have *slightly* higher variance than off peak (9.3% higher variance for peak).

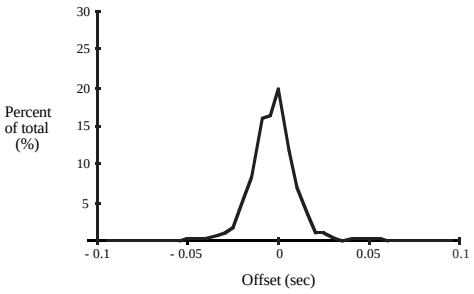


FIGURE 4.11
Tuesday offset values (off-peak)

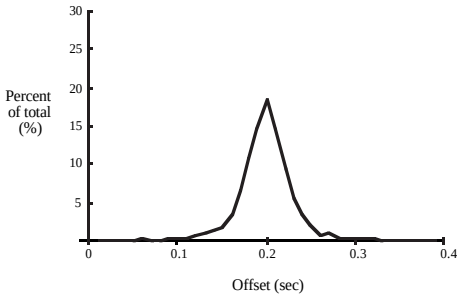


FIGURE 4.12
Friday offset values (peak)

The delay measurements are also as expected, approximating Erlangian distributions (Figures 4.13, 4.14). The off-peak measurement exhibits a smaller minimum delay (Figure 4.13). Both experiments show an unexpected quantization, especially because it appears in the delay measurements but not in the offset measurements, and both quantities are derived from the same timestamp sets (from NTP message headers).

The quantization may be the result of using a fixed set of IP paths, except that the quanta are nearly exact units of 0.01 seconds (10 ms). We assume that this quantization was due to a fixed service interval in the local NTP server response loop. This conclusion is further complicated by the fact that some replies have delay quantizations that are slightly smaller than the 0.01 second quanta. Similar quantizations are manifested in ensemble averages, so we conclude that the code of our local NTP implementation is the

[1]UNIX is a registered trademark of Unix Systems Laboratories.

most likely suspect. Further analysis of NTP may prove this conclusion, but is beyond the scope of this dissertation.

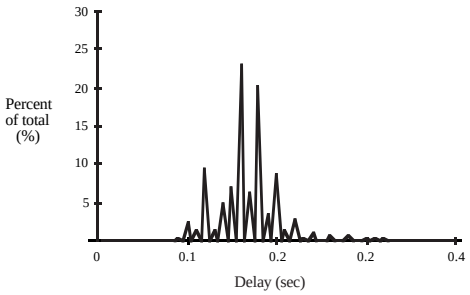


FIGURE 4.13
Tuesday delay values (off-peak)

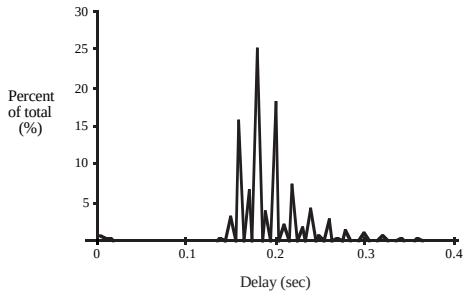


FIGURE 4.14
Friday delay values (peak)

Regardless of the quantization effects, the curves are bounded by Erlangian-shaped envelopes. Again the peak values have a slightly higher variance (15.7% larger variance for peak), although this is because the peak delay pdf has a longer ‘tail’ of values of higher latency. The peak pdf is has a larger minimum delay, and delay values are grouped more tightly about the mode than in the off-peak. The higher latency is the result of higher network loads during peak times. The more tightly grouped latency values are most likely the result of side effects of network load on routing information accuracy.

The NTP implementation we tested is supported by the IP datagram transport protocol. IP routing information is maintained as a side effect of IP datagram transmission [Co91a]. Higher network loads result in more accurate routing information, which in turn results in more precise (i.e., repeatable) latency. Lower network load can permit routing table inaccuracies to persist longer, so that datagrams experience larger variability in route paths, in turn increasing delay variability.

These measurements indicate that: (1) there is a minimum delay that is not affected by the latency variability, so there exist tighter bounds on the measured offset, and (2) forward and reverse path delays are not equal. The distributions of delay/offset pairs for both peak and off-peak data sets are shown in Figures 4.15, 4.16, respectively. The previous plots of offset and delay are the (respectively) horizontal and vertical projections of these plots (Figures 4.13, 4.14, 4.11, 4.12). For comparison, we also present a 3-dimensional plot of the off-peak values (Figure 4.17).

The round trip latency can be partitioned into two components – a fixed minimum, and a variable additional delay, where each component is non-negative (by definition).

The corresponding fixed minimum components of the two alternate bounds are plotted as vertical lines (gray and dashed).

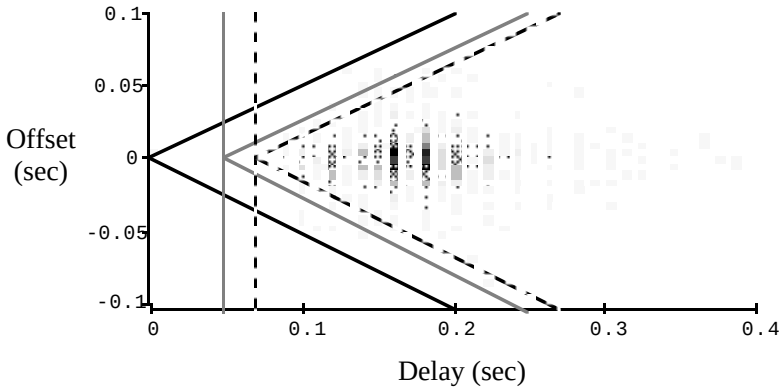


FIGURE 4.15
Delay v.s offset, time-repetition density (off-peak)

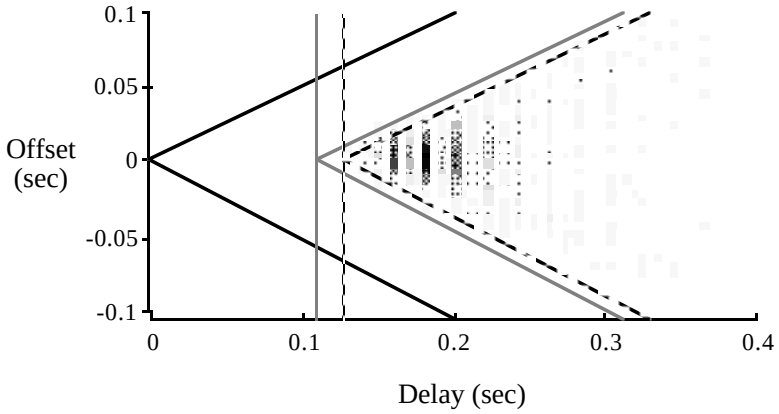


FIGURE 4.16
Delay vs. offset, time-repetition density (peak)

In these plots, the NTP bounds (Equation 4.8) are shown as solid diagonals. These bounds appear too loose, and are based on the assumption that one-way latency is bounded by the total round trip latency only. Shifting these bounds to the right yields more appropriate boundaries to the displayed data; two such alternate bounds are shown (loose in gray diagonal, and more restricted in dashed diagonal lines). These alternate boundaries suggest reinterpretation of the bound on the difference between outgoing and incoming latency, and their effects on the measured offset values.

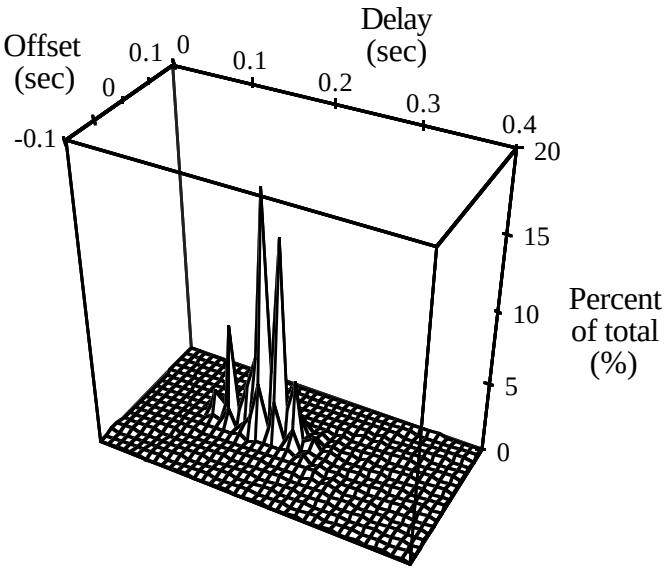


FIGURE 4.17
Delay vs. offset vs. probability, time-series (off-peak)

Off-peak measurements have larger minimum delay components, but there is less variability in our visual estimation of the minimum delay, compared to peak measurements. Again, this can be explained as an effect of the routing table maintenance, where light network load results in low delay, but in high variability in delay, because route information is updated less often. High load causes high delay, with low variability because heavy traffic helps maintain accurate route information.

The partitioning of round trip latency into non-negative fixed and variable components (Equation 4.9) can be incorporated into the previous constraint equations (Equations 4.7, 4.8), resulting in Equations 4.10, 4.11.

Equation 4.9: $\delta_{total} = \delta_{fixed} + \delta_{\text{var}}$

where $\delta_{fixed} \geq 0$ and $\delta_{\text{var}} \geq 0$

Equation 4.10: $0 \geq \delta_{uni\text{-}var} \geq \delta_{var}$

Equation 4.11: $\theta + \frac{\delta_{\text{var}}}{2} \geq \hat{\theta} \geq \theta - \frac{\delta_{\text{var}}}{2}$

Visually, the effects of this partitioning of the delay can be shown by modifying the NTP message exchange diagram (Figure 4.2) to Figure 4.18. Black portions indicate fixed components of the delay, and gray indicates variable components; the remote interval is bounded between the fixed components of latency. The bounds on the offset are more tightly restricted as a result (Figures 4.19, 4.20), compared to the earlier diagrams (Figures 4.3, 4.4).

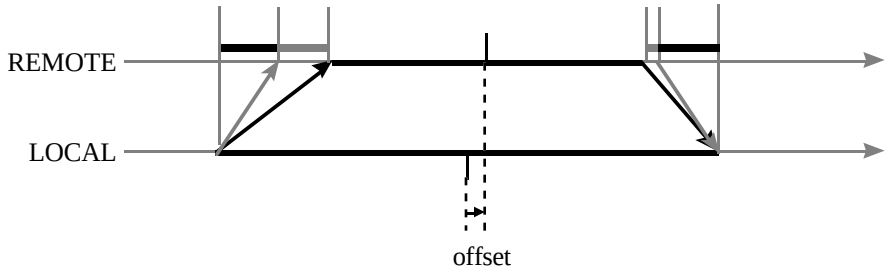


FIGURE 4.18
Fixed (black) and variable (gray) delay in message exchange

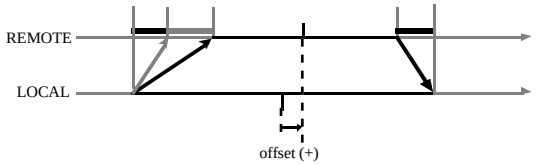


FIGURE 4.19
Exchange maximum offset, variable delay

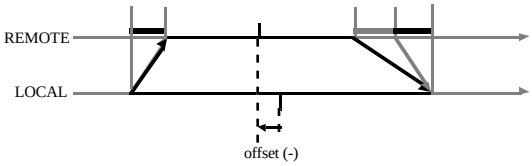


FIGURE 4.20
Exchange minimum offset, variable delay

Thus far the delays have been assumed identical on the forward and reverse path of a message. If this were the case, all measurements would exhibit a symmetric offset distribution. Temporal data sets do exhibit this behavior, but ensemble data sets are heavily skewed toward positive offsets (Figure 4.21). This skew is the result of a larger average forward latency, which is the result of routing effects in the underlying IP transport mechanism.

An ensemble data set consists of a set of unique message destinations. Each message is routed to the destination and back. The reverse path of a message occurs by an updated and presumably shorter path than the forward message, because the forward message updates routing information as it is delivered. Forward messages are routed by comparatively stale information, exhibiting longer routes as a result.

Lower strata (e.g., Stratum 1) servers exhibit large offset skews, which disappear as the stratum increases (Figures 4.22, 4.23, 4.24, 4.25)[1]. Offset skews should tighten as strata decrease due to higher precision in local clocks, whereas empirically, offsets tighten and become more symmetric as strata increase. Both the tightening and more symmetric nature of higher strata can be attributed to the locality (and thus known routes) of lower strata servers, whereas higher strata servers were typically more distant; this is a side effect of the NTP server hierarchy and its partitioning of siblings according to their parent and topological access criteria. Empirically, lower strata servers were more distant than higher strata servers.

Figure 4.21 also includes NTP bounds (black diagonal lines), violations of the NTP bounds (circles), and a possibly more constrictive bound (gray diagonal lines) resulting from an assumed latency minimum (gray vertical line).

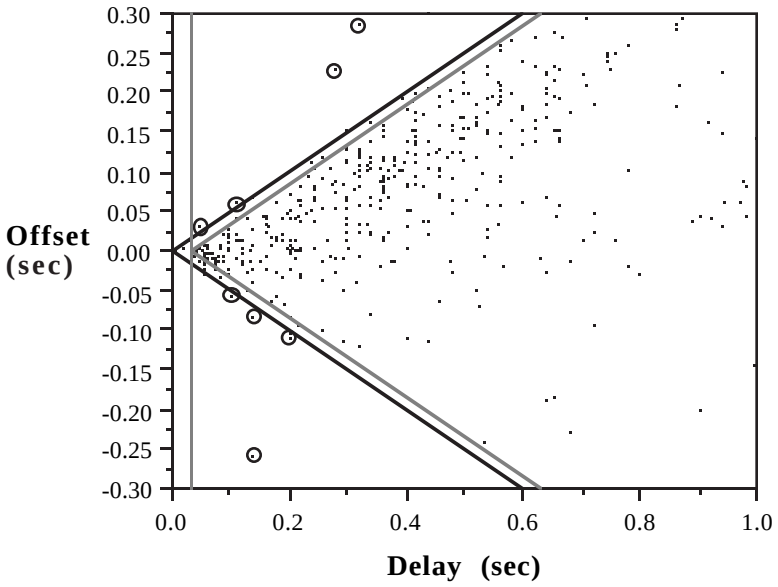


FIGURE 4.21
Delay vs. offset, entire ensemble (all strata)

[1]Stratum 0 server replies are not shown; only 5 were accessible, and ‘0’ indicates an unknown stratum.

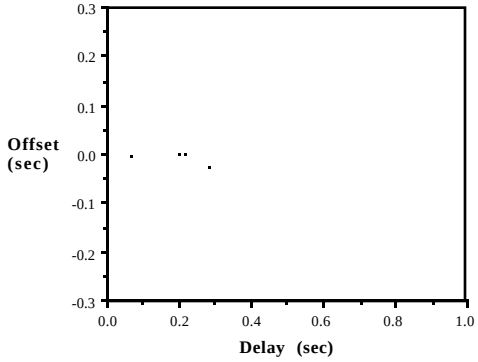


FIGURE 4.22
Stratum 1 ensemble

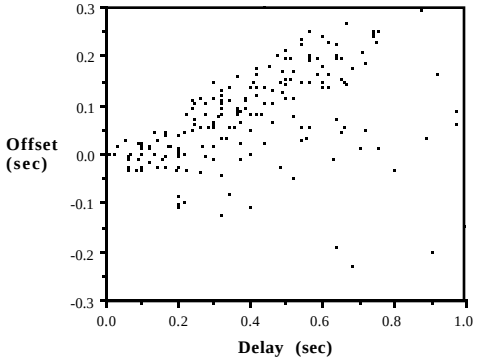


FIGURE 4.23
Stratum 2 ensemble

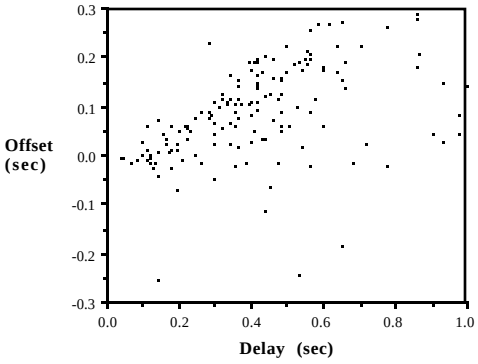


FIGURE 4.24
Stratum 3 ensemble

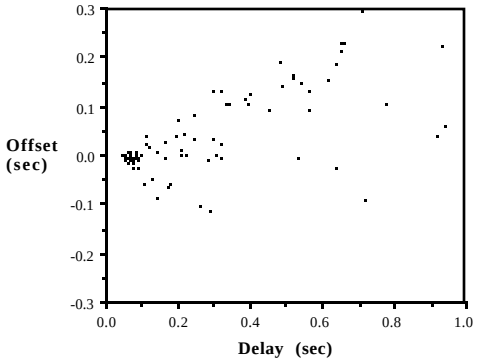


FIGURE 4.25
Stratum 4 ensemble

4.4. Description of NTP in Mirage

The following is a description of the variables of NTP, (some individual, some groups), and their description as Mirage indicates (Table 4.2). *Local state* denotes the protocol state space as Mirage considers it, and *remote perception* is the value obtained by communication with a remote node, i.e., part of the node's total view. *Computation function parameters* denote those components of NTP that are part of the Mirage time-

transformation function. *Protocol management* denotes components that affect the boundaries of the computation function, and that govern the send actions. These last management operations are part of communicability, and the optimization thereof.

NTP 'state'[1]	NTP description	Mirage description
(time)	current clock value	local state
(frequency scale)	clock frequency adj.	comp. func. params
drift	frequency variability	comp. func. params
wander	drift variability	comp. func. params
offset	remote clock shift	remote perception
delay	packet RTT	protocol mgt.
dispersion	variance in packet data	protocol mgt.
elapsed time	time since last request	protocol mgt.
poll interval	time between requests	protocol mgt.

TABLE 4.2
Mirage interpretation of the state variables of NTP

4.4.1. State space

The local state of NTP consists of the state of the local clock, i.e., the current time. The local perception of a remote clock is the offset, i.e., the difference between the remote clock and the local clock. The remainder of the state variables of the NTP specification refer to protocol management or expression of computability (to describe the time transformation). The only variable that is managed by NTP is time; all others serve to facilitate this management.

[1]Parentheses indicate variables which are not listed as explicit in the NTP protocol description, but are implicitly required.

4.4.2. Transformations

Transformations of the state space of NTP, as described by Mirage, map sets of time values onto other sets of time values. These sets affect the perceptions of remote clocks. In NTP, sets of time values are denoted by continuous intervals, so transformations map time intervals to other time intervals.

Some components of NTP resemble those required in Mirage to manage the state space and maintain stability. *Elapsed time* indicates the time since the last clock update, and indicates when stability will fail. This, together with the assumption of a fixed skew (0.01 sec/day), indicates the current imprecision of the clock (*dispersion*), and is used to determine the expansion of the time transformation. The *poll interval* indicates the time between messages, and is a precomputed expected time when the state becomes imprecise enough to require a message initiation, i.e., the maximum interval over which stability can be maintained.

4.4.2.1. Time

The local clock of a node changes as a function of time, both by tracing a path of sequential clock values (ticking), and by becoming less precise as time progresses (in the absence of received messages). The ticking of the clock is modeled by Equation 4.12 (depicted in Figure 4.26), from the definition of frequency as the first derivative of the clock transition function, drift as the second derivative, wander as the third, etc., and includes noise.

The imprecision of the clock is modeled by Equation 4.13. The imprecision describes the expansion of the state point (clock time) to a state interval (time +/- imprecision), where the distribution within this interval is assumed to be uniform (Equation 4.15, Figure 4.27).

Equation 4.12: $T(\Delta t) = noise(\Delta t) + epoch + F(\Delta t) + \frac{1}{2}D(\Delta t^2) + \frac{1}{6}W(\Delta t^3) + \cdots$

Equation 4.13: $imprecision(t) = \{bit\_precision\} + max\_skew\_rate * t$

Equation 4.14: $T(t) = noise(t) + T(t_0) + R(t_0)[\Delta t] + \frac{1}{2}D(t_0)[\Delta t]^2$

Equation 4.15: $T\_interval(t) = [T(t) - imprecision(t), T(t) + imprecision(t)]$

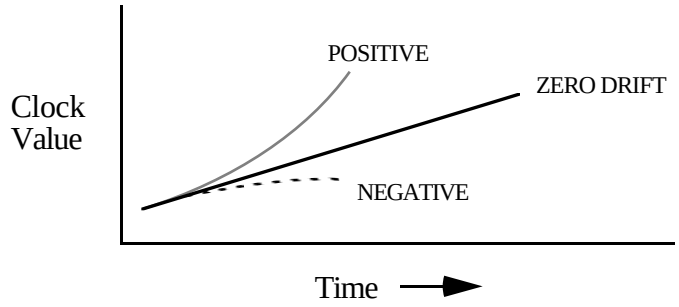


FIGURE 4.26
Clock value is a function of time

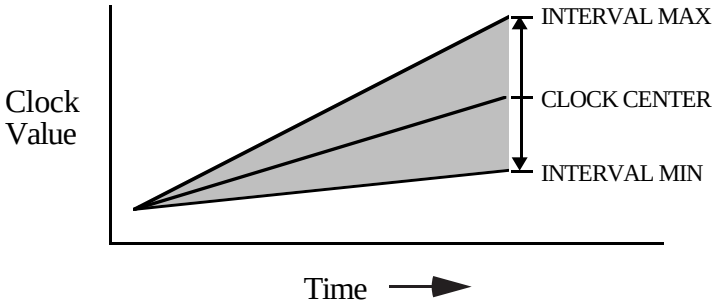


FIGURE 4.27
Clock interval as fixed expansion centered on time function

In NTP, wander and higher order variability effects are not considered. The combination of the equations for time transformation and imprecision growth result in the same formula as in NTP (Section F.3 of [Mi90b]), which describes the operation of the logical software clock (Equation 4.14). The imprecision of the clock is a fixed, linear function of elapsed time since the last clock adjustment, and is not determined by clock precision, strata, or previous adjustments.

The software clock model of NTP is called supplemental and external to the protocol in its specification. We disagree, because the clock equations are required to specify the time transformation of the Mirage model of NTP.

NTP makes the same assumptions as Mirage, that although ‘time’ can vary as the result of the protocol, variability in the progression in time is small enough that it does not affect the model substantially, i.e., it is not contradictory to model displayed time against a virtual absolute timescale (even if the latter is unknown).

NTP further describes the noise function as a pdf, in which a single initial clock value spreads into a zero-centered bell-shaped curve (Gaussian) of the pdf. This curve occurs as the vertical cross section of the time transformation graph (Figure 4.27), so that the figure has a bell-shaped surface in the third dimension (not shown).

There are also several constraints on the clock function. Clocks never run backwards, so the drift parameters can never combine to cause reversal of the clock. The clock distribution is similarly limited, so that the slope of the interval minimum line can never be negative.

4.4.2.2. Send

The effects of a sent message are not considered in NTP. The protocol is intended to create a hierarchical organization of clients and servers, where clients are served by their parents in the hierarchy. As a result, clocks are modeled unidirectionally, i.e., the child models the parent only. The parent does not model the child.

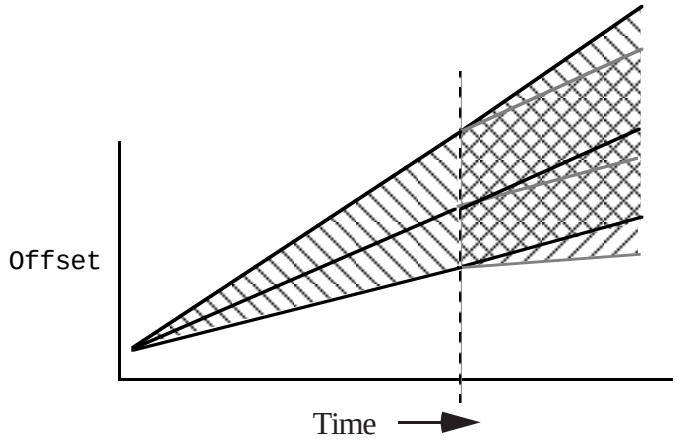


FIGURE 4.28
Transformation of a perception due to a sent NTP message

We can extrapolate our investigation to the case where the parent would model the child. Emitting a time message would cause an adjustment in the center-line of the expanding state (Figure 4.28). The resulting state space consists of the union of the

affected and unaffected expanding state spaces, indicating an increase in imprecision caused by the adjustment. The parent node doesn't know whether the sent time is accommodated into the child node, so it adjusts its perception based on the new state emanating from any existing possible clock. The result is an adjustment of the clock frequency only.

4.4.2.3. Receive

Received messages alter the local state in NTP, as proscribed by Mirage. A received message causes a node's perception of the remote state to be updated, which collapses the interval to a point, from which subsequent expansion will resume (Figure 4.29). Other information is also extracted from this collapse.

If the received time is in the upper region of the interval, then the local model of the remote clock is slow (the expanding triangle is angled too far downward), and needs to be compensated by shifting the triangle up. The shift in the triangle is reflected in an update of the frequency parameter of the computation function.

From this analysis, we conclude that the components of the computation function, i.e., frequency, drift, etc., are not constants, as initially perceived, but variables in the model which need to be incorporated. There is an interaction in these equations that indicates that the variables are not an orthogonal basis of the system space. The result is a revised set of time transforms (Equations 4.16, 4.17, 4.18), plus a set of receive transforms (Equations 4.19, 4.20, 4.21).

The receive transforms use the delay and offset information derived from the received message, and assimilate it into the local time view as specified by recombination algorithms in the protocol. NTP does not manage multiple perceptions within a single node; an ensemble data set is combined using the peer selection algorithm. Similarly, a temporal data set is collapsed to a single value by the data filter algorithm. This is required to reduce the computation overhead in resolving a set of clock values to an indicated correction, and to reduce the storage required over time to maintain data of previous state.

Note that both the peer selection and data filter algorithms are integral to the Mirage model of NTP, because they specify the receive transformation operations, whereas the NTP specifications refer to the equations as optional "suggested implementations".

The slope of the expansions is indicated by the error function (Equation 4.22). NTP uses fixed values for the skew, currently set to a constant of 0.01 second/day. In Mirage,

this corresponds to the computation function, and need not be linear in the elapsed time since last update. In NTP, time updates never alter the value of the expansion, only the center-line of it.

Equation 4.16: $T(\Delta t) = noise(\Delta t) + epoch + F(\Delta t) + \frac{1}{2}D(\Delta t^2) + \frac{1}{6}W(\Delta t^3) + \cdots$

Equation 4.17: $F(\Delta t) = F(t_0) + D(\Delta t) + \frac{1}{2}W(\Delta t^2) + \cdots$

Equation 4.18: $D(\Delta t) = D(t_0) + W(\Delta t) + \cdots$

Equation 4.19: $T(\Delta t) =$ {weighted avg. of clocks}

Equation 4.20: $F(\Delta t) =$ {exponential avg. of clock differences}

Equation 4.21: $D(\Delta t) =$ {current freq – avg. freq.}

Equation 4.22: $error(\Delta t) = \{bit\_precision\} + max\_skew\_rate(T_{in} - T_{orig})$

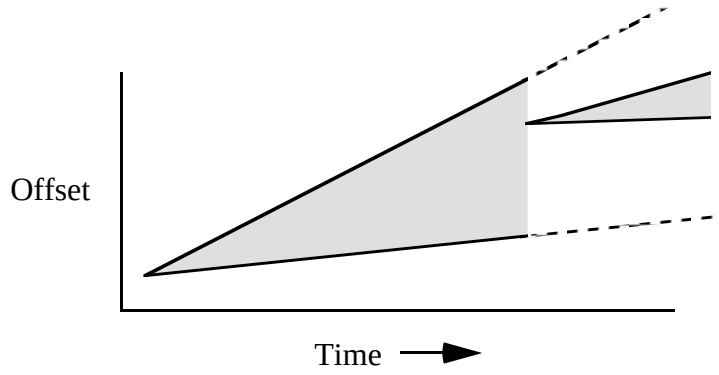


FIGURE 4.29
Transformation of a perception due to a received NTP message

As noted before in the Mirage model description (Chapter 2), the delay in a message's transit since its origination must be measured, and the current state must be back-calculated from it (Figure 4.30).

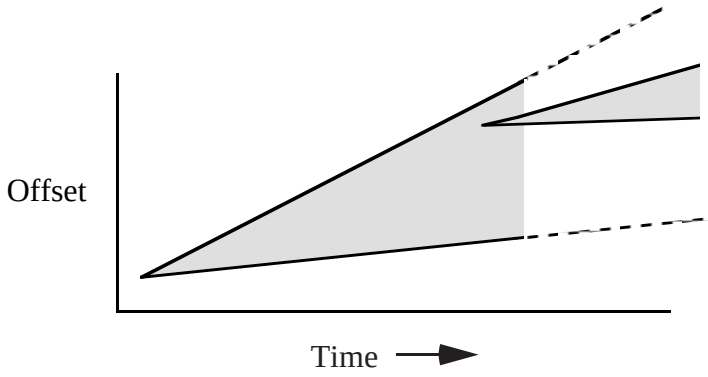


FIGURE 4.30
Receive transformation, accommodating transit time effects

As also noted in the Mirage model, there are restrictions on the collapse of the state space, as indicated by the received message. If the message indicates that the new state space value is *not* contained in the current model of the remote node, then an inconsistency exists. NTP calls this type of inconsistent received message a *Time Warp*, and it correlates to Mirage receive constraint criterion violation (Figure 4.31). Specifically the violation occurs in NTP because the received offset cannot be incorporated within a continuous valid interval, whereas in Mirage the violation occurs because the collapsed state is not a member of the set of current possible states.

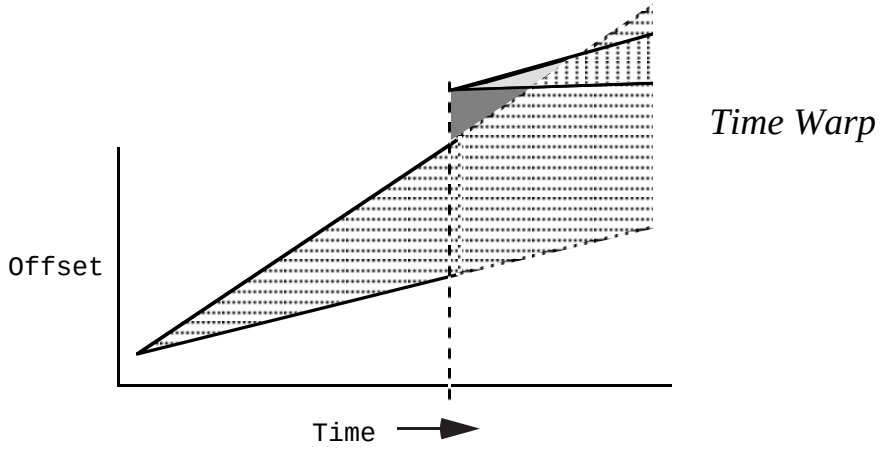


FIGURE 4.31
Representation of a Time Warp in the state space timeline

4.4.3. Partitioning of the state space

In NTP, the state space is modeled as a continuous interval of possible clock values. In Mirage, the bit size of the message guard indicates the extent of the partitioning of this interval (here we assume an equipartitioned interval).

NTP does not send guards on the messages; they are inferred from the relative position of the received clock adjustment to the computed valid interval. If the received message is in the upper portion of the interval, the current clock frequency is low (i.e., the clock is slow), and it needs to be speeded up.

If the interval is not partitioned, received clock adjustments result in new clock values, but frequencies, drifts, etc., are not compensated (Figure 4.32, 0 bits). If the interval is partitioned into two (i.e., as if a one bit 'virtual' guard is inferred), then the frequency is adjusted faster or slower, but only by a fixed amount in either case (Figure 4.32, 1 bit). If the interval is partitioned into progressively smaller components, more accurate adjustments of the frequency can be made (Figure 4.32, 0-4 bits shown). NTP never alters the drift value.

As a result, NTP uses 'virtual' guard information to adjust the clock by a value indicated by a partition inversely proportional to the number of unique frequency adjustment parameters. Guards are inferred from clock timestamp information in the NTP messages, which are much larger than the required guard size.

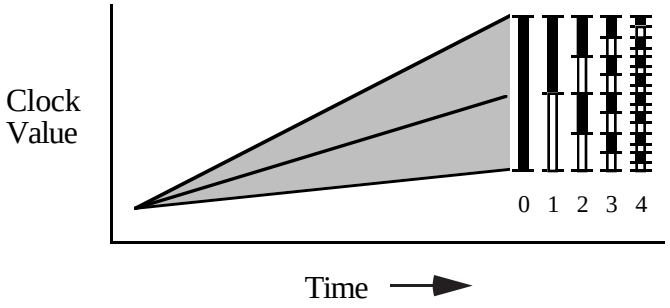


FIGURE 4.32
Partitioning of the state space results in variably 'tight' state collapse

Another interpretation of the partitioning is the way in which larger messages specify more precise clock update values. If no bits are sent, the clock is not updated. If one bit is send, then the clock is in one of two indicated subintervals of the currently valid

interval. The more bits that are sent, the more precisely the receiver can update the interval (again, Figure 4.32).

In either case, the guards are virtually specified by the sent clock value. The larger the guard, the more finely partitioned the state space (interval) becomes, and the more precisely the space can be maintained. The problem is that larger guards also increase time between sent messages, because larger guards require longer messages. Consider the case where messages specify clock values to within 10 ms, and take 1 minute to send, vs. messages that specify the clock value to within 1ms, and take 1 hour to send. If the clock skew rate (computation expansion function) is small (less than 1ms/hour), then the long, (infrequent) precise messages are more effective in synchronizing the clock than short, (frequent) imprecise messages. When the clock skew rate is large (more than 60 ms/hour), the short, (frequent) imprecise messages can maintain the clock to within 60ms/hour, whereas the long messages also let the clock skew out of adjustment by a larger amount, even though it may be more tightly synchronized for a short time immediately after being adjusted.

4.4.4. NTP degenerates to a clock pulse

NTP clock value messages degenerate to a clock pulse (hardware clock signal) under certain conditions. If the latency variability is zero, the clock is completely described by the temporal transition function. If drift and other higher order components are zero, and if the frequency component of the clock is zero, the clock is described by a perfect line. If frequency error is not zero, there is an expansion of the interval with time. The sender limits the extent of this error by the frequency with which adjustments are sent. The remote clock is partitioned into two components (time received, time not received), so reception of any unique signal suffices to manage the clock. This is a regular clock pulse, because the frequency error in the receiver is the size of the true frequency (because the transition function assumes a static clock value (i.e., zero frequency) with an error that increases with time (Figure 4.33). The period of sent messages determines the extent to which the sender’s clock is accurately modeled, and the centerline of the sent messages expresses the clock frequency.

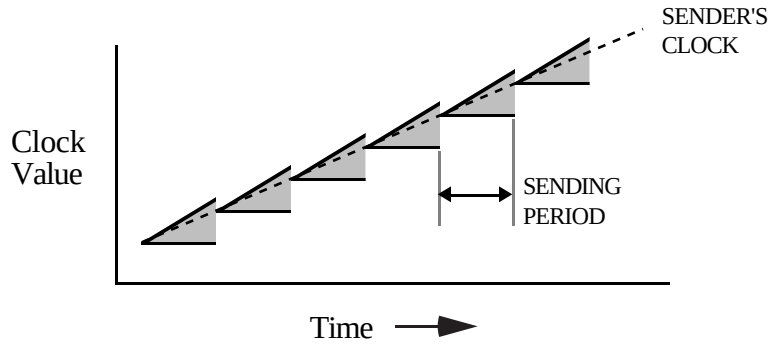


FIGURE 4.33
Regular sender anticipation.

4.4.5. Constraints

Many of the constraints in the Mirage model overspecify components of the NTP protocol. For example, the message size constraints of Mirage indicate that smaller messages could be used in the NTP header, where in the latter a timestamp indicates a 200 picosecond interval within a 136 year span. The overspecification of the timestamps is used by NTP for other consistency checks, e.g., to ensure that incorrect timestamps do not 'wrap around' onto valid stamp values. Consistency checks include measuring positive intervals for client/server responses, verifying reasonable delays, and possible formal authentication of the packet using an auxiliary mechanism.

4.5. Observations

The main observation from the application of the Mirage model to NTP is that NTP makes little use of Mirage components. NTP has a very simple state (time, frequency, drift) that is governed by simple (quadratic) equations, and has a fixed linear computation function. Receive transforms are elaborate, but only insofar as they accommodate multiple Byzantine sources in a way that attempts to discern true time from a perception (through peer selection and filter algorithms); Mirage is intended to maintain the perception only. NTP uses intricate weighting functions to collapse the state space which Mirage preserves (inefficiently, in comparison to NTP).

We have observed an isomorphism between variability in latency, which NTP accommodates, and variability in the computation expansion, which Mirage accommodates. Certain characteristics of the delay and offset measurements in NTP can be predicted from simple models of the network characteristics (Poisson unidirectional latency), with the “Mirage function” embodied as the convolution of the component pdfs.

We were previously unaware of the extent to which Mirage relies on accurate time measurements (relative time), especially in the computation functions. This was however no more so than other protocols similarly rely, even clock synchronization protocols.

Some constraints of Mirage are unnecessary in the NTP protocol, notably those indicating the bit size of the sent and received messages, because NTP uses fixed headers with oversized messages intended to provided informal authentication and error resilience. Other constraints specified in Mirage also appear in NTP, e.g., an NTP *Time Warp* is a violation of the state space collapse being a subset of the existing state space model.

NTP circumnavigates issues of modeling sent messages, as well as sender anticipation, by organizing the time server network into a strict hierarchy. We can add sender anticipation to NTP, which results in a server-initiated periodic ‘chime’, similar to NTP’s *broadcast* mode of operation.

4.5.1. Gains

NTP uses a fixed header description, with fixed message sizes. The message sizes can be reduced to minimal values as specified in Chapter 2. The sizes of the messages are a function of the variability in the remote state, which in NTP is isomorphic to variability in the round trip time, as described earlier. As the variability in round trip time decreases, the need for extended precision clock stamps in the header is reduced; a zero round trip variability indicates that the clock discrepancies are a function only of local drift and error. As local drift and error approach zero, there is less need for clock value correspondence. Zero round trip variability and zero drift and error would abolish the need for communication altogether.

Note that these conclusions require that latency variability and drift disappear; actual round trip latency does not matter, so long as it can be predicted with a probability of 1.

4.5.2. Prior work

Relevant prior work on the analysis of time protocols includes probabilistic clock synchronization [Cr89], optimal clock synchronization [To87], and distributed clock mechanisms [Ma85].

All of these analyses rely on bounds on the maximum round trip latency to determine achievable precision and accuracy, according to the same bounding formula as in NTP (Equation 4.8). A smaller round trip latency results in less error in the computed offset. We have shown that offset error is linearly proportional to the variability in round trip latency, and more precisely to smaller difference between forward and reverse path latency (Equation 4.11).

We do not consider Byzantine failures, or the explicit failure of any components of the protocol in the Mirage analysis. Mirage shows the errors in the perception of remote clocks, even where the remote clock is perfect. Byzantine and fail-stop failures would increase the error in the local perception of the remote clock, although determination of whether stability could be achieved is beyond the scope of this analysis. We limited this analysis to the characteristics of NTP.

The methods used here are similar to those used in statistical timekeeping analysis [Cr89], especially those relating the pdf of latency values to a time protocol. Here we additionally show the way in which a Poisson unidirectional latency variability can cause Erlangian round trip variability, and Gaussian-like measured offset variability, even in the presence of a perfect remote clock (i.e., one with no temporal expansion in state).

The statistical method suggests increasing the number of message attempts to increase the probability of receiving a message with a minimal latency, potentially increasing the link latency due to load increases [Cr89]. Other methods use sets of messages to overcome node failure [To87] or ensure accuracy [Ma85], but indicate nothing of the tradeoff between message size and frequency. These are self-defeating mechanisms, which Mirage counteracts by indicating that more frequent communication allows use of smaller guards and smaller messages.

Mirage does not claim optimality in the number of messages exchanged alone; rather, it optimizes the total communication in message number and length (integrating length over the message set), and considers optimality only insofar as the desired stability can be achieved and maintained. Furthermore, [Cr89] permits failure of the protocol to achieve the desired precision, thus permitting instability, which Mirage prohibits.

4.5.3. Conclusions

The use of Mirage to analyze NTP has highlighted aspects of NTP not required in the specification, and helped refine our understanding of Mirage. NTP specification considers the logical clock, peer selection, and filter algorithms optional to the protocol specification. Mirage cannot model NTP without the use of constraints from these algorithms, so we consider them integral components of the NTP protocol, and would suggest that they be required in the specification.

We showed the equivalence between latency variability and imprecision in the local perception of a remote state. We concluded that the NTP protocol boundaries on measured offset error can be corrected, to denote explicitly that measured error is the result of latency variability alone. This results in more accurate constraints on the measured offset error.

Using message length and frequency tradeoffs in the Mirage communicability formula (Chapter 2), we show the degeneration of NTP to a hardware clock signal, as latency variability approaches zero. The Mirage was used to derive exact effects of latency variability on measured offsets, by applying the discrete Mirage formula to continuous latency distributions using pdf convolutions.

Before this investigation, we were not aware the extent to which the Mirage model relies on time measurements for its analysis. This use is not precluded by the application of Mirage to even clock synchronization protocols, because the assumptions of measurement required within Mirage are similar to the requirements in other clock synchronization model analyses.

NTP was useful as a sample protocol for Mirage analysis, but many of the more interesting components of Mirage were not applicable to NTP. Messages in NTP are static and overspecified as required by Mirage's message constraints. NTP models time as continuous intervals, whereas Mirage permits arbitrary sets in its model. Sender anticipation and guarded messaging can be inferred from some aspects of message processing in NTP, but are not explicitly applicable to this protocol. Finally, some of the most interesting algorithms in NTP, those of peer selection and data filtering, are mechanisms to collapse and project multiple dimensions of the state space that Mirage is intended to preserve.

CHAPTER 5

μ-Net

5.1. Preface to Chapters 5 and 6

The following two chapters comprise a discussion of the application of Mirage to processor-memory interaction as a protocol. This, Chapter 5, is a description of the process of modeling processor-memory interaction as a protocol (Sections 5.2-5.3.), and an elaboration of the various degrees of implementation of the design that the process suggests (Sections 5.4.-5.5).

Chapter 6 compares the feasibility of the degrees of implementation, based on measured opcode statistics (Sections 6.1.-6.3). That chapter also contains the discussion of prior work specific to the processor-memory domain; more general prior work regarding Mirage is contained in Chapter 3, in Section 6.4. Chapter 6 concludes with an evaluation of the utility of this modeling process, and the value of the results it generated (Section 6.5).

A more concise version of the material in these two chapters, describing m-Net and its results, can be found by reading the following sections:

5.2. (inclusive)	Introduction
5.4. (inclusive)	μ-Net - Design
5.5.6.2.	The Total implementation of μ-Net
5.6-5.7. (inclusive)	Implications
6.1.-6.2. (inclusive)	Performance, Feasibility
6.3.2.	Other Observations
6.5. (inclusive)	Conclusions

As a preview, μ-Net is a processor-memory interface design derived using Mirage as a model of the interaction as a protocol. The processor is augmented with a filtering device, to accept only messages (opcodes) with guards (addresses) that match the processor's state (program counter).

The memory is augmented with a mechanism, called a Code Pump, to model the state of the processor (PC) as a set of possible states, using a data structure (we call it the TreeStack). The mechanism sends data messages to the processor ([address, opcode] pairs) that increase the modeled state (i.e., send transformations), using a component called a Diverger. The mechanism receives data from the processor (PC values) that collapse the modeled state (i.e., receive transformation), using a component called a Converger.

This design is very similar to, and a superset of, current research trends in memory anticipation mechanisms, notable that of proactive memory. Performance increases are achieved by using idle bandwidth during round-trip latencies to send anticipatory information. μ-Net sender anticipation as suggested by Mirage.

Measurements indicate that a simple, partial implementation would reduce opcode pipeline gaps and increase processor performance by a factor of 10; a complete implementation would increase performance by a factor of 300 to 3,000, when coupled with a cache. Feasible implementations require a simple stack mechanism and as little as 400 bytes of storage to achieve the same performance over high latencies as a 50 Kilobyte cache alone.

5.2. Introduction

Mirage provides methods for mitigating the effects of latency on communication, provided that the communication is well described by the protocol. In Mirage, the protocol doesn't merely police the information stream, it *is* the information stream. Evaluating Mirage requires the selection of a protocol in which shared state is the goal of the communication, rather than merely a means for connection management. In Chapter 4, NTP was used to describe the abstract components of the Mirage model in real terms, but many aspects of Mirage did not have analogs in NTP. For further analysis of the model, a new domain was chosen so as to exhibit some features of Mirage that existing protocols do not possess. We call the design that results from this analysis ***μ–Net*** (pronounced 'MicroNet').

There are two common interpretations of a protocol - a mechanism to maintain shared state (i.e., the protocol *is* the communication), and a mechanism to manage communication (i.e., the protocol provides bit-transmission). Existing protocols manage only connection information as shared state; most of the data communicated is not part of this shared state. For the shared state to comprise the actual communication, a large state is not required; in NTP as few as 2 variables (current time and rate factor) are shared. In TCP a similarly small number of variables are shared[1], but this state information is used to facilitate other data transmission. Most current protocols are not currently used to manage state spaces directly (as Mirage requires in order to show advantage).

A more complete investigation of Mirage requires the selection of a domain in which communication protocols may not be normally applied, yet one in which state sharing is the goal of the communication. We have chosen a domain in which communication protocols are traditionally not used, that of processor-memory communication. Existing processor-memory communication architectures are shown equivalent to existing protocols, and application of the Mirage model yields novel communication architectures that improve performance where transmission latency is high.

As discussed in Chapters 1 and 2, Mirage adds a parameter of transit time to the conventional parameters of a communication channel, those of bandwidth and channel

---

[1]TCP state information consists of the current connection information: the current window number, one of 11 connection states [Co91b], and endpoint address information.

width (parallelization) (Figure 5.1). Consider an analogous structure, in which the communicating nodes are that of processor/RAM and program memory (read-only) (Figure 5.2); this example is justified by its similarity to the client-server model of computation, in which code is stored on shared remote nodes, and utilized locally. This structure is a variant of the Harvard-style architecture, in which the data bus is local to the CPU and the code bus is a latency communication channel. The benefits of this architecture are not discussed here; it was chosen only as a domain in which to demonstrate the benefits of Mirage. The application of the Mirage model to this domain provides methods for reducing the performance effects of latency.

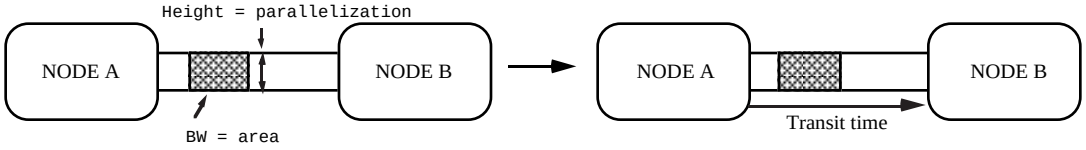


FIGURE 5.1
Mirage extends the channel model to include latency

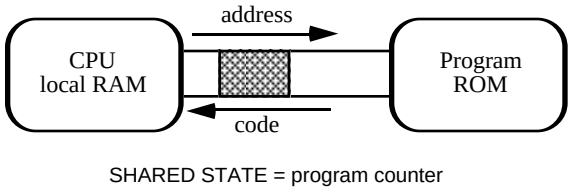


FIGURE 5.2
Communication channel analog of processor/memory interaction

In this domain, performance is defined as the time of execution of a fixed code measure. In some cases reducing the execution time is cosmetically desirable, i.e., the computational outcome remains the same, but a speedup is desired; it is in this way that performance is usually considered. Execution time is as valid a measure of code as any other (i.e., correctness, completeness), and performance can be considered a correctness criterion, especially in time critical environments (see Chapter 3, on Real Time systems).

The characteristics of this architecture can be measured as the channel latency increases. The channel is a logical interaction between the processor/RAM and program memory, where the address of the desired code and the code itself are the communication exchanged across the interface.

5.2.1. The Domain - the processor / memory interface

A processor communicates with memory for two uses - retrieving data with which to operate and retrieving code to direct these operations. In the Harvard architecture, these two communication streams are both logically and physically independent[1]. The domain considered here is a version of this architecture, where the code stream communicates across a distance, but the data stream is local. This resembles a client-server model of computation, where the data (and perhaps I/O devices) are located at the CPU, but the code is archived at remote stations.

The advantages of this example are not of prime importance to us here, although they have been discussed elsewhere (as RPCs, or simply shared code on mounted disks in NFS). In this domain the shared state is concise and the temporal and communicative transformation functions on the state space are well understood. The shared state is the address whose code is being retrieved (i.e., the program counter), and the state space transformations are indicated in the opcode (e.g., jump, jump-to-subroutine), in some structure in the data space (return), or implied (the default for most opcodes). The nature of these transformation functions will be elaborated later; initially current architectures are described, together with their behavior as the code storage area is moved away from the CPU/data area.

The following discussion focuses only on communication between the CPU/data storage (noted as CPU) and the code storage area (noted as Memory). The effects on the communication of program code are measured as the program memory is moved away from the site of its utilization.

5.2.2. Description of current architectures

The processor/memory interface can be considered as a communication channel. In conventional computer designs, a processor communicates with program memory by either a request/response or a timed-response protocol, usually across a bus. The processor places the address of the desired code on a bus, and indicates a memory request by a pulse or level on a signal line (e.g., read/write, memory request, etc.). Memory

---

[1]Sometimes Harvard architectures only indicate separate data and code caches; here we intend that code and data are not only communicated separately, but also stored separately.

responds to this request by placing the opcode at the desired address on a bus, either setting a reply line (memory ready, etc.) or assuming that the processor will wait a fixed number of clock cycles for the reply. These two protocols are shown below (Figures 5.3, 5.4, 5.5, 5.6), both as conventionally shown as a bus interaction (time/wire label) (Figures 5.3, 5.5), and as a protocol time line (time/space) (Figures 5.4, 5.6). We denote the former protocol as ***explicit***; and the latter as ***timer- based***.

Explicit and timer-based protocols differ from the conventional designations of synchronous and asynchronous. Synchronous protocols ensure (and rely) on signals being aligned with the clock pulses, and asynchronous protocols permit clock signal independence. Timer-based protocols are synchronous, but explicit protocols can be either synchronous or asynchronous, depending on whether the reply signal must be aligned with the clock pulse.

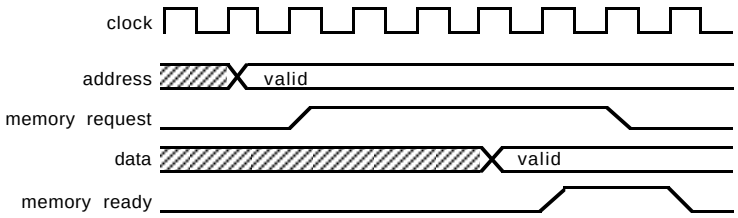


FIGURE 5.3
Explicit processor-memory protocol (voltage/time diagram)

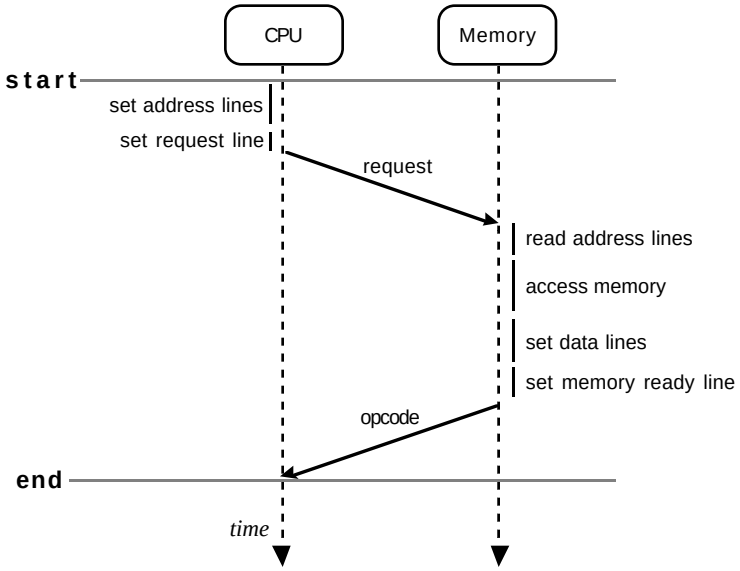


FIGURE 5.4

Explicit processor-memory protocol (as a protocol time line)

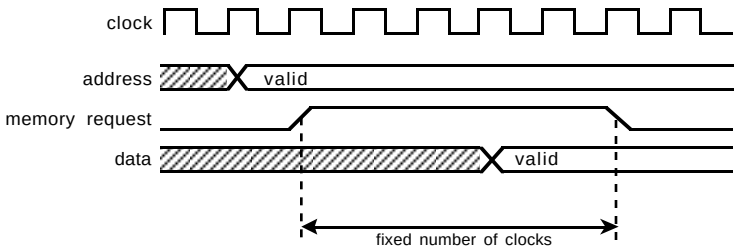


FIGURE 5.5
Timer based processor-memory protocol (voltage/time diagram)

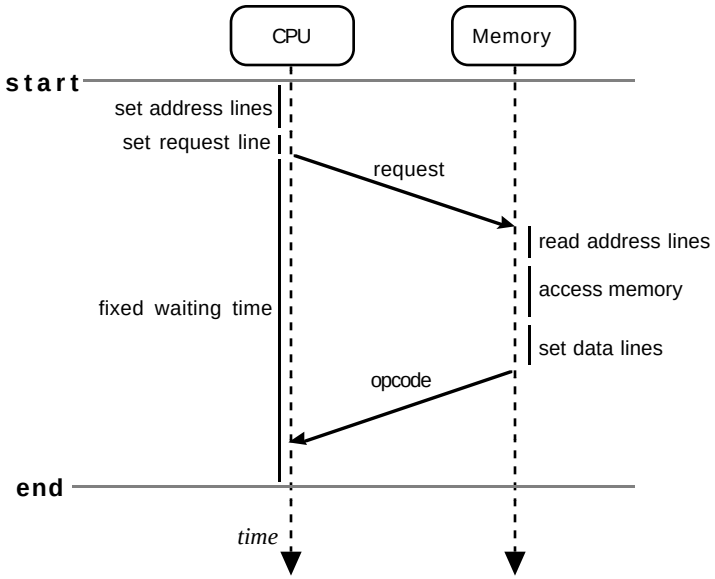


FIGURE 5.6
Timer-based processor-memory protocol (as a protocol time line)

There are usually few or no provisions for a failure of this request-response type of protocol; it is assumed that if this communication fails, little can be accomplished anyway (i.e., if an arbitrary opcode cannot be retrieved, neither can the interrupt handler code)[1]. If an explicit protocol is used, the processor would wait indefinitely; if a timer-

[1]In fault-tolerant systems, an external mechanism monitors opertation and switches to a backup system or halts the current failure in a safe manner; systems do not perform fail-safe functions after their own (arbitrary) failure.

based protocol is used, the processor would read invalid data, because it would not otherwise know that the memory had failed to reply correctly.

Consider the characteristics of this communication as the processor and program memory are moved farther apart. In this description, *CPU* refers to the processor and local RAM memory (read/write data), and *Memory* refers to program memory (i.e., read only). The two components communicate by a high bandwidth path (Figure 5.7), whose latency increases as a parameter of this investigation.

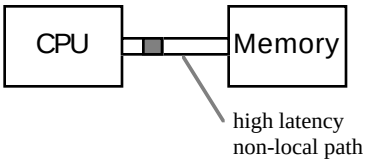


FIGURE 5.7
Processor-memory interaction across a distance

This configuration is usually augmented with a program cache (Figure 5.8). The cache is located with the processor and communicates to it via a low latency, high-bandwidth path. The CPU initiates a request for code to the cache, and the cache either replies directly (in the case of a cache hit) or forwards the request to the program memory over the high latency path. The memory reply is forwarded back to the processor and also copied into the cache for later reuse.

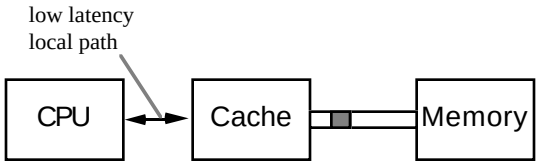


FIGURE 5.8
Processor-memory interaction via a cache

There are extensions to this architecture which support prefetching, where the cache requests program code in anticipation of its use by the processor. The receiver of the data (the cache, ‘speaking’ for the processor) asks for the data before it is needed; this is called a *lookahead cache*.

The Mirage model, when applied to this domain, indicates a new way to design this architecture. In this new design, the sender (memory) anticipates the needs of the receiver (processor). This has been alluded to as *proactive memory*, although we refer to a specific

architecture here (i.e., suitable for implementation), whereas the prior work refers only to a "possible future direction" [Kr91].

In the new architecture, called *μ–Net* [To91b], a *Code Pump* manages the anticipation at the site of the memory, and a *Filter Cache* emulates the function of a conventional cache, isolating the processor from the details of the mechanism (so that the system appears to the processor to be identical to Figure 5.9 (CPU-memory only), except in performance).

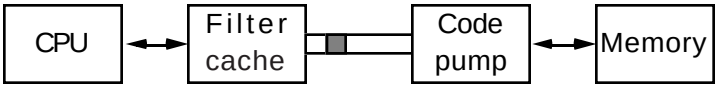


FIGURE 5.9
μ-Net processor-memory interaction

In terms of the earlier protocol figures (Figures 5.4, 5.6), the protocol timelines of the three architectures can be shown symbolically (Figure 5.10). The conventional and cache architectures exhibit the same protocols, except that the cache can omit communication when a hit occurs. A prefetching cache can retrieve multiple replies with a single request. μ-Net permits memory to send replies before requests are received, i.e., the sender anticipates the receiver requests (the gray line in the μ-Net version in Figure 5.10).

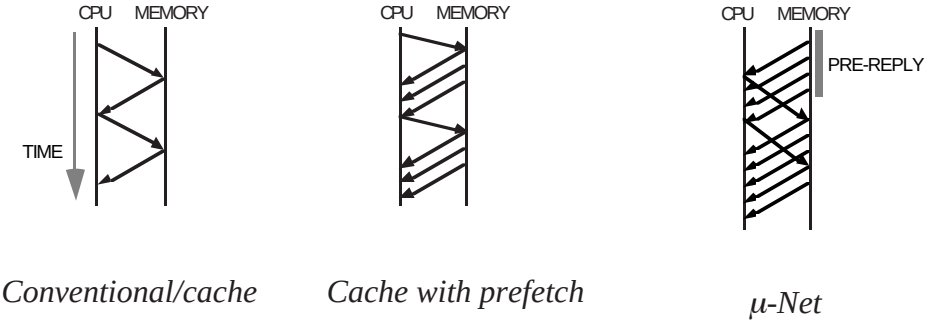


FIGURE 5.10
Protocol timeline comparisons of processor-memory protocols

A conventional architecture with a cache incurs a round trip latency expense (miss penalty) whenever a miss occurs. A prefetching cache periodically requests blocks of code in advance, whereas μ-Net is a self-adapting sender-based version of prefetching

where the memory sends code that the processor might need and concurrently receives updates of the processors actual state.

Cacheing is complementary both to prefetching and to anticipatory replies as implemented in μ-Net. Cacheing reduces access bandwidth to memory. Prefetching increases memory use because fetched code may not be utilized; μ-Net similarly increases memory use by sending sets of codes (isopotent sets), but only one member of each set is used. Cacheing is a way of looking into the past (of code use), making assumptions that code is reused in the future, to reduce memory bandwidth, whereas both prefetching and μ-Net are ways of anticipating future requests (beyond just code reuse) which subsequently increase memory bandwidth.

5.2.3. Effect of latency on existing architectures

The effect of latency on these architectures can be considered by describing the time it takes to execute some number of instructions (*N*). Assume that the processor executes 1 instruction per time unit (*t*) (thus defined). A conventional architecture requires *Nt* time units to execute *N* instructions. This is the optimal case, with respect to the communication latency (i.e., latency is not considered, or zero) (Equation 5.1).

Equation 5.1: $T_{optimal} = N * t$

This formula is augmented to include latency (i.e., as latency increases beyond a negligible amount), in Equation 5.2, where *r* denotes the round trip latency. *Negligible latency* is defined as any time at least one order of magnitude smaller than the execution time of a single instruction, in which case Equation 5.2 reduces to Equation 5.1 (i.e., latency contributions are negligible in comparison to instruction execution).

Equation 5.2: $T_{conventional} = N * (t + r)$

Performance is described as the time required to execute *N* instructions vs. the time required in the optimal case. The ratio of particular execution time to optimal time is defines the *slowdown* (Equation 5.3). This ration in conventional architecture analysis is usually called the *speedup*, but we consider cases where the instances examined are slower than optimal. In the case of a conventional architecture, this is Equation 5.4.

Equation 5.3: $Slowdown = \frac{T_{measured}}{T_{optimal}}$

Equation 5.4: $$Slowdown_{conventional} = \frac{T_{conventional}}{T_{optimal}} = 1 + \frac{r}{t}$$

In the case of the conventional architecture augmented by a cache, the effects of the latency are reduced proportional to the effectiveness of the cache utilization (Equation 5.5). *M* denotes the probability of a miss in the cache. On a cache miss, both the round trip and the execution time are incurred. A cache hit costs only one instruction execution time. Equation 5.6 reduces to Equation 5.4 where the cache is 100% effective. In this case, no communication costs are incurred, because all code accesses are intercepted by the (local) cache.

Equation 5.5: $$T_{cache} = N * (t + r * M)$$

Equation 5.6: $$Slowdown_{cache} = 1 + M * \frac{r}{t}$$

The time to execute *N* instructions in an architecture where the cache prefetches can also be described. Let *k* denote the prefetch length, in instructions. Further characterization of the communication stream will be required to estimate the probability of a prefetch occurring (described later), as used in the equation.

Prefetching is usually implemented in a linear fashion only. When an opcode at address *i* is accessed, address *i*+1 though *i*+*k* are prefetched, where *k* is the linear lookahead, and usually *k* is the number of opcode words in a cache line[1]. A miss occurs when the prefetch fails to anticipate the next opcode used, i.e., when the linear anticipation assumed by most implementations is breached. Misses also occur when the destination of a control transfer instruction (e.g., BRANCH, JUMP, CALL, RETURN) is not in the cache.

At most, prefetches occur at the miss rate of the cache alone (i.e., the same as without prefetching). The actual prefetch miss rate is also at most the rate of occurrence of control instructions, because the opcode after a control transfer may not be in the cache (Equation 5.8).

Control instructions are assumed always to prevent the prefetch of the next opcode. This assumption is valid for JUMP, CALL, and RETURN opcodes, but it is not always valid for BRANCHES. In the case where a BRANCH is not taken, the next opcode is

[1]In an Intel 80386, *k*=4; in most caches the line size is between 4 and 8.

successfully anticipated. Because branches are taken about 50% of the time [He90], the formula can be further refined (Equation 5.9).

For the moment, the probability of a fetch occurring is denoted as *F* (Equations 5.7, 5.10). *F* is always less than *M*, because control instructions are only part of the cause of misses in a conventional cache, so Equation 5.7 is always at least as good as Equation 5.5. Empirical values of *F* and *M* are discussed in Chapter 6.

Equation 5.7: $T_{prefetch} = N(t + r * F)$

Equation 5.8: $F \leq J + B + C + R + I$

where opcode occurrences are denoted by percentages:
J = % JUMPS (direct only)
B = % BRANCHES (direct only)
C = % CALLS (direct only)
R = % RETURNS[1]
I = % other indirect opcodes (JUMPS, CALLS, BRANCHES)

Equation 5.9: $F \leq J + \frac{B}{2} + C + R + I$

Equation 5.10: $Slowdown_{prefetch} = 1 + F * \frac{r}{t}$

In μ-Net, the equations are a little more complicated. The time to execute the *N* instructions depends on the ability of the *Code Pump* to predict correctly the opcodes desired by the processor. The probability of an incorrect prediction is denoted as *P*, and one round trip time will be required for each misprediction, to allow the processor to fetch the desired data which is not already available (Equations 5.11, 5.12). This case reduces to the optimal (Equation 5.1) where *P* = 0, i.e., when the prediction is perfect. An estimate of *P* will be discussed later, with the conditions under which it is smaller than *F* and *M*.

Equation 5.11: $T_{\mu Net} = N * (t + r * P)$

Equation 5.12: $Slowdown_{\mu Net} = 1 + P * \frac{r}{t}$

---

[1]The percent of CALLS and RETURNS are not always equal. Some systems permit direct stack manipulations, or returns which pop out of multiple nesting levels.

5.3. μ-Net

μ-Net [To91b] refers to our choice of the processor-memory interaction as being analyzed by a communication model, Mirage. This is in keeping with the tradition that all protocol research, at one time, must coin a network name suffixed in 'Net'. This was not done with the model name (MirageNet?), so the tradition is upheld in the name of the example.

The μ-Net domain is easily modeled in Mirage, by suitably defining the shared state space and by defining the state transformations of Mirage in terms of this space. In this example, communication is affected by *classes* of opcodes in the instruction stream. The usual techniques of cache prefetching, widening cache line sizes, and branch prediction are confirmed from the Mirage model of μ-Net. Further, μ-Net exhibits new techniques for active opcode anticipation, which are only recently being proposed as new solutions to latency issues in architectural design. Also, some simple data structures in the memory interface can reduce the latency for some kinds of memory access operations; these structures implement isopotent anticipation as described in Mirage.

In order to consider the Mirage model of μ-Net, the components of μ-Net need to be defined in terms of the model components in Mirage. The first and most fundamental of these is the shared state between the sender and receiver.

In the processor/memory interface, what state is shared? Usually, this is minimally the program counter (PC) value. The program counter denotes the address of the next instruction desired, to be placed on the address lines when a memory request is made. During the memory request, that address is received by memory, after which the desired opcode is replied. The state space is thus the space of all PC values. Conventional protocols communicate by understanding the state of the remote node as a point in this space, and moving this point explicitly. Therefore, until the memory receives the new value of the program counter, the previous value remains in effect, as far as the memory is concerned (Figure 5.11).

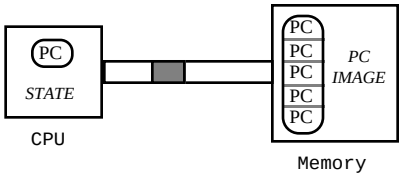


FIGURE 5.11

CPU state and Memory image

In terms of the Mirage model, the state is a PC value, and an image is a set of PC values, i.e., a subset of the entire PC space. Transformations map PC values onto new PC values, or sets onto sets.

The value of the state space changes with the progression of time, the sending of a message, and the receipt of a message. The state space, in this case, is the program counter, because the memory models the processor's current PC value.

This analysis considers the memory's model of the processor, not the processor's model of the memory. This is appropriate because the communication is essentially unidirectional, although there are components in both directions (control in one direction vs. data in the other).

5.3.1. Time transformations

As time progresses, two things happen at the processor. The processor is executing the current instruction, then it must wait for opcodes to be sent from memory, sitting idle until its request for the subsequent opcode is serviced. As a result, the state of the processor (i.e., the PC, the part of the state that the memory is concerned with) is stable during the execution of the instruction, and changes only afterwards.

This assumes, however, that there is no cache at the processor. When there is, the processor may use opcodes already in the cache during the time lapse. The opcodes in the cache have already been sent from memory, so they are already accounted for in the memory's image of the processor's PC.

The passage of time then, in this domain, *denotes the scheduling of the need for opcodes by the processor*. During the time in which the current opcode is executing, the PC image does not change. At the time when the execution changes, the PC image changes to reflect the need for the next opcode.

The simplified model assumes a RISC architecture with a single opcode execution time. Variability in execution times (e.g., in CISC architectures) can be accommodated by tables in the sender when opcode execution times are static, or by a dynamic structure, if execution times vary, as in pipelining. This latter dynamic structure emulates the pipeline activity, in order to predict the need for new opcodes at the pipeline input.

5.3.2. Receive transformations

When messages are received, the state space collapses accordingly. There are cases (in Table 5.1) in which the state space of the program memory can grow, by modeling both arms of a conditional, or by expanding to the limit of the state space (in the case of indirect opcodes). The messages received by the program memory are processor PC values, which collapse the PC value set in the program memory to a single value. In fact, in this case, the PC set is collapsed to the value received, and then expanded to account for the transit time of the message. In this way, the PC set in the program memory always models the PC of the processor, out of sync by the transit time of a message.

5.3.3. Send transformations

When information is sent, the state space expands through the unioning of the sets of state space before the message and that state space as affected by the message. The state space consists only of PC values (and possible PC values), so the actual opcodes sent are ignored, except in the way in which they affect the current PC. The memory has some set of possible PC values; it sends the opcodes as these PCs indicate (i.e., it sends the opcodes at those addresses), and transforms each PC by a function indicated by the opcode. After transmission, the new PC set becomes the union of the prior set and the transformed set. For example, if the opcode is ordinary (OTHER, i.e., not otherwise distinguished hereafter), the PC would increment when the opcode is sent. The resulting PC set is the union of the PC and PC+1. The transformation depends on the opcode sent; some opcodes increment the current PC, some transform it, some expand it to a set of two PCs (i.e., two possible PCs), and some expand the PC to the set of all PCs (Table 5.1).

In the most general case, the PC would be arbitrarily transformed with each opcode, but this is not effective in modeling the evolution of the imprecision of the knowledge in the PC as known by the program memory. Because the program memory knows the contents of the message (i.e., the opcode), it can determine how the message will potentially affect the PC at the processor (i.e., the transformation).

The image of the PC in the program memory is transformed differently for various classes of opcodes sent. The opcodes are distinguished only by the way in which they transform the current PC over time (Table 5.1).

OPCODE	transform: PC →	new PC values are computed based on...
other[1]	PC+1	PC
direct jump	PC'	PC, opcode
indirect jump	{all PCs}	PC, opcode, any CPU register, program or data memory value
direct call	PC'	PC, opcode (prior PC is stored in a known structure in RAM)
indirect call	{all PCs}	PC, opcode, any CPU register, program or data memory value (prior PC is stored in a known structure in RAM)
direct branch	{PC+1,PC'}	PC, opcode
indirect branch	{PC+1,PC'}	PC, opcode, any CPU register, program or data memory value
return	PC'	stored in a known structure in RAM (previously saved)

TABLE 5.1
Opcode time transformations

For the majority of the opcodes (denoted as OTHER), the PC is incremented. In the case of a JUMP, the program counter becomes a new value based on the destination. In the case of a BRANCH, the program counter becomes one of two new possible program counter values, one being the incremented PC, the other being the newly indicated branch destination, thus the model of the PC becomes a set of PCs. This is the expansion of the state space referred to in the Mirage model.

At some later time, at the receipt of a message from the processor, this set is collapsed down to a single member. CALLs behave similarly to JUMPs, except that in addition to performing the transformation indicated, some local state is maintained (in a stack). A RETURN utilizes this local state to perform a JUMP back to the opcode following the origin of the corresponding CALL.

[1]I.e., not otherwise listed in this table.

The table distinguishes between direct and indirect types of jumps, branches, and calls, because in the former case the resultant PC can be computed, whereas in the latter the state space becomes completely unpredictable[1] (i.e., it expands completely to encompass the entire state space). Opcodes differ in the way in which they affect the expansion of the state space over time; this is the criterion for partitioning them as described.

In the case where the PC set expands to the limits of the state space (for indirect opcodes), further temporal transformations become impossible to compute. The memory must wait for a message from the processor of the actual PC value chosen, before it can proceed further. Indirect opcodes necessarily cause 'bubbles in the communication loop', where sender anticipation cannot be accommodated. Indirect opcodes, are, in effect, too unpredictable to model.

5.3.4. Guarded messages

In the conventional processor/memory architecture, there is no need to label the opcodes which are sent from memory, because only one memory request is outstanding per unit time. Even in the cache prefetch case, the cache either requests each memory value independently, or an initial request is made and the resulting values are assumed to arrive sequentially.

The opcodes being sent from memory must be labelled, in order to permit the processor to receive them conditionally. Consider the case where the PC modeled in memory contains a BRANCH instruction. Memory sends the next instructions, but more than one instruction is sent (i.e., more than one possible next request, as discussed before). Replies to these requests must be differentiated, so that the processor (which knows its own state) can select the appropriate one. Although there are more efficient labellings, the opcode can be labelled with the part of the state space to which it applies (i.e., with the PC it is located at). A single bit could be used to indicate the two arms of a

---

[1]Indirect jumps can be limited to the set of labels in the program source code, assuming that the compiler ensures such restrictions, and that the executable code contains label information. This will be discussed in the prior work of μ-Net, in Chapter 6.

branch,, but any label thus chosen is limited to some finite partitioning of the state space (i.e., only 1 branch lookahead per bit).

5.3.5. Partitioning the state space (stability)

The state space is not partitioned into sets of PC values, which would be computationally prohibitive. The PC image may indicate two PC values which request the same opcode, which could be sent only once with a guard indicating both PCs. Such an implementation would be excessive, because the resulting reduction in bandwidth would be at the expense of excessive encoding.

Instead, the state space is partitioned into its individual points (single PC values), so each message (opcode) requires a single PC value as a guard. The state space is not separated into “last timestep / this timestep”, as in the union of the two spaces in the Mirage send transformations. μ-Net assumes that messages are not lost, so the prior state (before the sent message) need not be retained. This prevents having to resend opcodes, under the assumption that the communication channel is lossless. If messages always arrive, in some fixed time delay, the part of the state space which corresponds to the prior state can be ignored because the probability density highly favors the send-transformed state.

5.3.6. Isopotent sets

There are two analogs of isopotent sets in this domain. In the first, isopotency occurs when two PC image values recombine, i.e., when instruction streams reconverge. In the second, isopotency indicates the satisfaction of both paths of a conditional by the set of both destination opcodes. This latter isopotency is accommodated in μ-Net by the expansion of the state space when a BRANCH occurs, because subsequent message sets furnish both arms of the branch.

The reconvergence case occurs either during a single timestep (simultaneously), or at differing timesteps. For example, when two current PC image values JUMP to the same location, the streams converge simultaneously. This convergence is managed as a side-effect of our implementation; if two PCs in the image at time t become the same at time t+1, only one remains.

When a forward branch occurs, the streams converge at different time values, because the PC of the branch-taken stream is the same as the PC of the branch-not-taken

stream at some earlier time. This is not handled in μ-Net. In this latter case, isopotency also exists where two PCs in the image become one, but only under message sequences of different length. If information were maintained on the past state in the model (it isn't - see the discussion on Partitioning, above), this recombination would be recognizable. μ-Net cannot take advantage of the case where one stream becomes a time-shifted image of another stream. This inability is not corrected because the complexity in modeling past time images would be prohibitive, and a corresponding benefit may not exist.

As a result, the way in which a set is isopotent depends on the ways in which the members of the transformed set are considered equivalent, i.e., the way in which the messages are equivalent.

5.4. μ-Net - Design

Although Mirage is an abstract model whose direct implementation was argued against, μ-Net can be implemented directly. Models are not usually intended to be implemented, but the domain of this problem has been sufficiently constrained to permit such an implementation to be reasonable, both to give a real impetus to the abstract model and to permit a better understanding of the model's advantages.

Communication in μ-Net is unidirectional - although the processor needs to communicate its state to the memory to specify the desired opcode location, the processor is modeled in the memory, and not the converse. Only the memory is concerned with the state of the remote entity (processor) here. The model of the processor can be implemented by putting a PC in the memory. In fact, a structure is placed near memory which will permit the modeling of more than one possible PC.

5.4.1. The Code Pump

The ***Code Pump*** manages the image of the processor for the memory. It also contains the send and receive mechanisms, insofar as they affect the state space modeled within the Code Pump. The Code Pump implements the time, receive, and send state space transformations, and the state space image (albeit limited) (see detail, Figure 5.12).

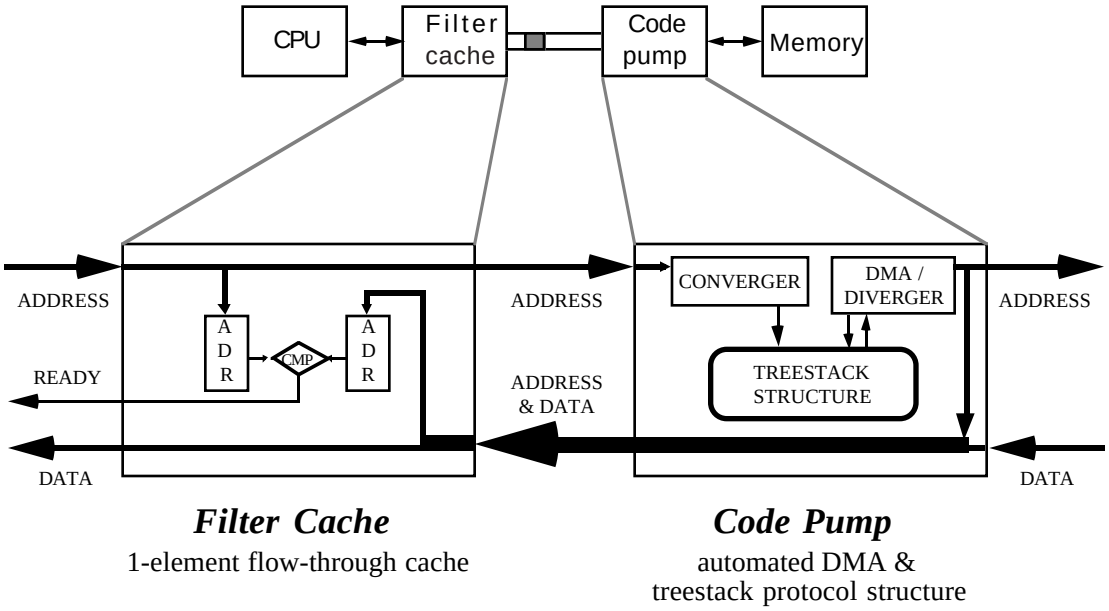


FIGURE 5.12
Detail of Filter Cache and Code Pump of μ-Net

The Code Pump contains three main components: a TreeStack data structure, a Converger, and a DMA/Diverger. The TreeStack structure maintains the image of the processor's PC; its tree-like attribute manages bifurcations in the PC image caused by sending BRANCH opcodes, and its stack-like attribute manages the information of saved and restored PC images which are caused by CALLs and RETURNs, respectively.

The Converger manages the collapse of the PC image in the TreeStack structure, as indicated by the receipt of messages from the processor (i.e., addresses). The DMA/Diverger manages the preparation and sending of messages anticipating the processor's request (DMA), and the expansion of the PC image in the TreeStack, as indicated by the send transformations.

5.4.2. The Filter Cache

The ***Filter Cache*** serves only to make the *Code Pump* mechanism appear invisible. It buffers data coming in from memory, and passes through only opcodes whose label (PC value, i.e., address) match the requested address whose data the processor is waiting for. All other data, for addresses not requested, is thrown away. It thus implements the receiver requirements for guarded message use.

The Filter Cache needs only a small amount of space to operate. If the rate of the processor is known(i.e., the rate at which opcode requests are issued, which is nearly trivial in a RISC processor), the Filter Cache needs only one opcode/address unit of space. The opcode arrives with the address as a conditional label (guard), so that the opcode is used only if the address is pending a request, which is the purpose of the comparator in the filter. The Code Pump knows this rate, so it sends the messages only when needed (see the time transformation).

The Filter Cache is also completely compatible with a conventional cache in parallel at the same location. The Filter Cache's purpose is to manage the incoming stream of new information, and pass the relevant parts on to the processor, whereas the conventional cache passes old information back to the processor. The two caches are complementary.

5.4.3. Degrees of design

The model is implemented directly in μ-Net, so there are some variations to the level of implementation which can be performed. For example, a TreeStack structure cannot be effectively implemented which models the PC after an indirect opcode, because the state space grows to its limits, and further partitioning of the state space to predict the next desired opcode is impractical. The design can be further simplified by not implementing any branch transformations, or not implementing RETURNs,. The following is an enumeration of the various levels of implementation, in increasing order of complexity (Table 5.2).

In the Code Pump implementation, the effects of the sent message on the PC image must be modeled, as well as modeling the old image (i.e., unioning of the unaffected and affected PCs). One simplifying implementation decision is to ignore the possibility of message loss, and omit the old PC image, replacing it with the new one as soon as the message (i.e., opcode) is sent. This works under the assumption that messages are not lost, simply delayed by a fixed time, in a way that simplifies the design to account for a reduction in the probability of states in the old image remaining valid.

The Code Pump can be implemented as a null device, which reduces to a conventional or conventional plus cache architecture. Prediction can be limited to only OTHER opcodes, where the Code Pump requires only a single PC image and an incrementer. This is analogous to an opcode prefetch in current CPUs (680x0, 80x86,

MIPS, SPARC, etc.), but differs in that the PC and incrementer are placed in the memory, rather than in part of the processor (i.e., the PC is on both sides of the interface), thus reducing the comparable latency by half. At this point, the Filter Cache needs only 1 element of space to operate - to hold the opcode/address pair as it is received, and to compare it to the current desired address.

Level of opcodes done	Filter size	Code Pump Components	Storage required (in Code Pump, in PCs)[1]
(none)	(null)	(null)	(none)
other	1 element	increment	1
other, jump	(same)	adder	1
other, jump, call	(same)	(same)	1
other, jump, call, return	(same)	adder, stack	avg. pending depth
all but indirect[2]	2 elements	adder, TreeStack	$L * \log_2\left(\frac{r}{L*2+1}\right)$

TABLE 5.2
Degrees of implementation, and the implications of each

When the Code Pump is augmented to handle JUMP opcode messages, the incrementer is converted to an adder, to accommodate the ways in which JUMP opcodes alter the PC - by direct overwrite, or by PC-relative addressing (i.e., add a constant to the current PC). CALLs can be similarly accommodated with no increase in complexity, assuming we do not store the origin of the CALL opcode.

---

[1] L=limb length, r=round trip time.

[2] The extra bits in the Filter Cache encode the choices made at each branch pending in the round trip communication, and are used only if the guards on incoming messages are so labelled. If the incoming guards are labelled with complete addresses, this storage can be omitted.

In order to accommodate RETURN opcodes, a more complex structure in the Code Pump is required to image the PCs in the processor. When a CALL is encountered, the Code Pump must hold the current PC in storage, so that when the corresponding RETURN occurs, the destination address implied can be determined. The state space image models the way in which a CALL causes the current PC state space location to be moved to a new point, for later return. This is a recursion in state space, such that a CALL is an entry point into a fresh copy of the state space, and a RETURN is the lone exit point from this state space, back to the original. Recursive state space manipulation was not envisaged when Mirage was developed, but it appears a natural extension of the application of the model.

The natural data structure for maintaining this recursive space image of the PC space is a stack. μ-Net proposes to put a stack on the memory side of the interface, to permit the memory to model the PC transformation of a RETURN opcode message. The memory can then proceed to subsequent messages (following the RETURN, in logical sequence), rather than being required to wait for an explicit request from the processor.

There are only two other classifications of opcodes, whose message transformations have not yet been considered: INDIRECT (e.g., indirect branch, indirect jump, and indirect call), and direct branches. INDIRECT opcodes transform the existing PC image to one which encompasses the entire PC space, as noted before. The result of this transformation is to prevent subsequent partitioning of the state space, to permit messages to be determined. If an indirect opcode can indicate a jump to any PC, there is no way to predict the next PC to send without further assumptions. Memory is forced to wait for an explicit request from the processor, because only the receipt of a request message will cause the state space to collapse (to the PC indicated by the request).

The assumptions under which an INDIRECT opcode may be predictable are those which restrict the definition of those opcodes. Analysis of source code, or suitably supplemented object code, can indicate a list of possible indirect jump, call, or branch locations. If the compiler restricts indirect opcodes to jump to computed values which are members of this list, then the opcode could be predicted. Indirect opcodes so restricted are equivalent to a fixed dispatch handler with a passed offset argument. In the handler, the passed argument denotes the requested jump destination. This mechanism requires compiler participation in the restriction of the action of indirect opcodes. μ-Net makes no assumption about cooperation of the compiler, and is intended to be compatible with

existing object code, which is not subject to INDIRECT jump restrictions. True INDIRECT opcodes do not limit the control flow to known compiler labels.

The forced wait for communication from the processor upon execution of an indirect opcode is sensible, because the new PC value depends on information which the memory does not have, such as values in RAM or processor registers. Indirect opcodes provide a transformation of the state space which is too powerful to model. This can be used as an argument for a more restricted form of indirect branching, such as a multiway table jump, especially because indirect opcodes are used mainly for such table jumps anyway. Any form of indirect opcode which permits the PC space to expand in a finite way, but not to its complete limits, would be able to be accommodated by the Code Pump.

One such version of a limited indirection is a BRANCH, in which only two resultant PC values are permitted (one is always PC+1, and the other is specified in the opcode by either a PC offset or a new PC value). When a BRANCH occurs at a single PC value, the transformation becomes a set of two PCs. The Code Pump requires a much more complex, but realizable, structure to implement the image of PCs under BRANCH message transformations. We call this structure a TreeStack.

5.5. Elaboration of degrees of design

These variations on implementation require various degrees of complexity in design of the components of μ-Net. All involve the maintenance of data structures in the Code Pump, where the Diverger manages creation and extension of the data structure, and the Converger manages reduction. The Filter Cache also varies in design, although only minimally so, in comparison to the Code Pump. Here we elaborate on the previous descriptions, and provide implementation designs.

These designs are not intended to be computationally optimal. Delays caused by the complexity of an implementation would further limit potential anticipation. Only the combined recursion and branching anticipation design is succeptible to such complexity; all other designs can be implemented as effectively as existing CPU/memory components.

5.5.1. No opcodes anticipated (Null implementation)

The null implementation uses a Null Filter Cache (i.e., no Filter Cache), and a nearly-null Code Pump. All memory requests from the CPU are forwarded to the Code Pump, where they are latched onto the opcode memory address port (as in a conventional design). Opcodes in reply are sent back to the CPU. The Filter Cache is non-existent, and the Code Pump is a memory address latch. Tables 5.3 and 5.4 describe the Null Filter Cache, and Tables 5.5 and 5.6 describe the actions of the Null Code Pump components (*Converger* and *Diverger*, respectively). Figure 5.14 denotes the data structure maintained. Figure 5.13 denotes the Null design of the Filter Cache.

The Null μ-Net implements the point model of communication. The Code Pump models the CPU as a point in state space, i.e., the last address requested is stored in a latch (PCvalue), representing the last known PC state of the CPU. The latch value is updated only after the current opcode returns to the CPU, is executed, and a new address (PCvalue) is communicated to the Code Pump; the alternation of control between the CPU execution and the Code Pump latching is denoted by the use of the signal variable (Flag).

Current CPU opcode	Action
any opcode	send current PC to Code Pump Converger

TABLE 5.3
Null Filter send actions

Message received	Action
any opcode	send opcode to CPU

TABLE 5.4
Null Filter receive actions

Message received	Action
any PC	overwrite PCvalue set Flag

TABLE 5.5
Null Converger actions

Data area item	Action
PCvalue & Flag set	fetch opcode at PCvalue send opcode to Filter Cache set Flag

TABLE 5.6
Null Diverger actions

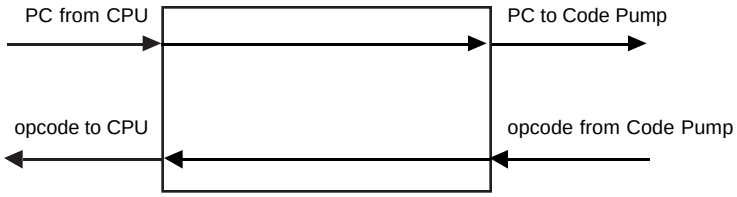


FIGURE 5.13
Null Filter Cache design

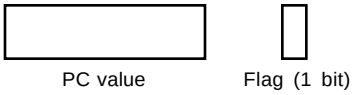


FIGURE 5.14
Null data space

5.5.2. Unit Linear opcodes anticipated

Unit Linear anticipation extends the Null design to accommodate the anticipation of regular (OTHER) opcodes, i.e., those opcodes which transform the PC by a unit addition

(i.e., PC ← PC + 1). The Filter Cache can be more selective in sending PC values to the Code Pump, because when 'other' opcodes are encountered, no PC need be sent.

In the same way as the Null implementation, the Flag variable indicates whether the anticipation may proceed or when it must cease and await resynchronization. The latter occurs whenever the Code Pump Diverger encounters an opcode whose future path cannot be determined.

The Unit Linear Filter Cache has a new set of send actions (Table 5.7), but the same receive actions as the Null Filter Cache (Table 5.4). The Unit Linear Converger is identical to the Null Converger (Table 5.5), but the Unit Linear Diverger is augmented to follow the PC path during pumping of opcodes (Table 5.8). Figure 5.14 denotes the data structure maintained, as before in the Null implementation, because no other data is required. Figure 5.15 shows the modified design of the Unit Linear Filter Cache, to accommodate the required opcode type information.

Current CPU opcode	Action
'other' opcodes	{ no action }
jump, call, return, branch, or indirect	send current PC to Code Pump Converger

TABLE 5.7
Unit Linear Filter send actions

Data area item	Action
PCvalue & Flag set	fetch opcode at PCvalue send opcode to Filter Cache Type of opcode: 'other': PC ← PC+1 jump, call, return, branch, or indirect: reset Flag

TABLE 5.8
Unit Linear Diverger actions

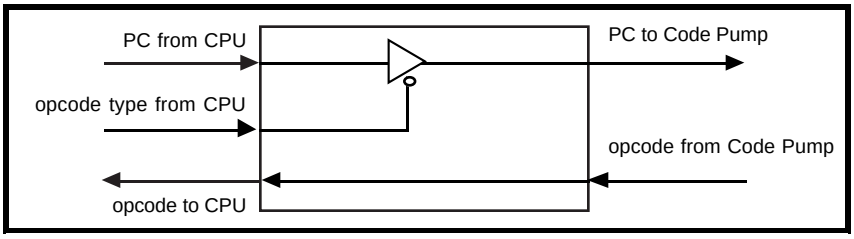


FIGURE 5.15
Unit Linear Filter Cache design

5.5.3. Linear opcodes anticipated

Linear opcodes transform the state space by some constant amount, not just '1' as in the Unit Linear case. These include JUMP and CALL opcodes. Linear anticipation can be accommodated by a minimal modification of the unit linear anticipation design. The Linear Filter Cache receive and Linear Code Pump Converger remain unchanged, and are still the same as in the Null implementation. Filter Cache sent data changes only in which opcodes activate message emission (Table 5.9).

Similarly, the Diverger is extended to add arbitrary fixed offsets, as extracted from within the JUMP and CALL opcodes (Table 5.10). The data space remains unchanged.

Current CPU opcode	Action
'other', jump, call	{ no action }
return, branch, or indirect	send current PC to Code Pump Converger

TABLE 5.9
Linear Filter send actions

Data area item	Action
PCvalue & Flag set	fetch opcode at PCvalue send opcode to Filter Cache Type of opcode: 'other': PC ← PC+1 jump, call: get 'k' from opcode PC ← PC+k return, branch, or indirect: reset Flag

TABLE 5.10
Linear Diverger actions

5.5.4. Recursion and Linear anticipation

Extending the Linear μ-Net design to accommodate recursion (RETURN opcodes) involves extending the data structure to model a kind of 'space embedding.' The data structure is a simple stack of PC values, with the same (single) signal Flag as before (Figure 5.16). The Recutsion Filter Cache receive mechanism remains unchanged (same as in the null design), and the send mechanism is changed to omit messaging upon 'return' opcodes (Table 5.11). The Recursion Converger remains unchanged from the Null design.

The Recursion Diverger includes not only the linear transformation components of the linear anticipation design, but also adds the 'push' and 'pop' operations on the data structure to model the state space embedding (Table 5.12).

Current CPU opcode	Action
'other', jump, call, return	{ no action }
branch or indirect	send current PC to Code Pump Converger

TABLE 5.11
Recursion Filter send actions

Data area item	Action
PCvalue & Flag set	fetch opcode at PCvalue on stack top send opcode to Filter Cache Type of opcode: 'other': PC ← PC+1 jump: get 'k' from opcode send opcode to Filter Cache PC ← PC+k call: push PC+1 onto stack get 'k' from opcode send opcode to Filter Cache PC ← PC+k return: pop top off stack and discard branch or indirect: reset Flag

TABLE 5.12
Recursion Diverger actions

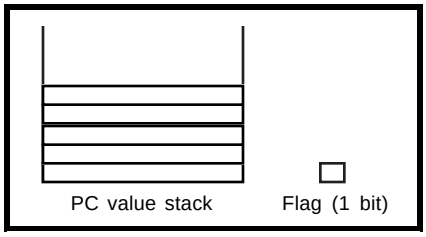


FIGURE 5.16
Recursion data space

5.5.5. Branching and Linear anticipation

Before attempting to augment the Recursion anticipation design to accommodate branching, it is easier to show the extension of the simple linear anticipation for branching alone. Later recursion and branching are combined, but here each is shown as independent extensions to the linear case.

The point model of the PC of the CPU of the linear case is replaced with a branching set model. The single PC value is replaced with a tree of values, each element of which is shown in Figure 5.17. This allows multiple simultaneous active PC values, denoted by the active leaves. ‘Active’ denotes a valid PC model, which was indicated in the prior designs by a set Flag value. Each leaf of the tree can be active or inactive. Interior nodes of the tree are inactive by definition.

Now that branching has been added to the set of opcodes accommodated, the components of the design take on activities denoted by their names. The Branching Filter Cache send portion emits messages indicating either which branch was taken, or how to reactivate an inactive leaf of the data structure (caused by indirect and return opcodes) (Table 5.13). The Branching Filter Cache has a different internal structure, modified to retain PC values in a shift register in order to enable indirect and return opcode messages (Figure 5.18).

The Branching Filter Cache receive portion of the branching design performs the filter function, matching outgoing PC values to incoming (PC,opcode) pairs (Table 5.14). In this way multiple alternate streams of opcodes sent by the Branching Code Pump can be distinguished.

The Branching Diverger extends the state space of the Branching Code Pump’s model of the CPU’s PC value, by splitting a single leaf into a branch with two leaves, in the case where a branch is encountered. Each arm of the branch has its own new PC value, and is labelled with the first PC value encountered on the branch path. The Branching Converger matches incoming branch selection PC values to the set of all branch labels, indicating a node in the tree where the CPU state has been in the past. The nodes in the tree superior to the indicated node are possible subsequent states to the past CPU state, and are kept; all other states denote possible states which the received PC value has invalidated, and are deleted.

Only leaves in the tree indicate currently active paths, and so the ‘DMA[1]’ in the Branching Diverger sends opcodes for each active leaf. Leaves which encounter indirect or return opcodes are inactivated. A separate message type from the Branching Filter Cache reactivates these leaves and deletes the whole remainder of the tree when received.

---

[1]DMA stands for Direct Memory Access, and represents a component similar to the system-level component of the same name.

These activities are indicated in the specifications of the Branching Converger (Table 5.15) and Branching Diverger (5.16).

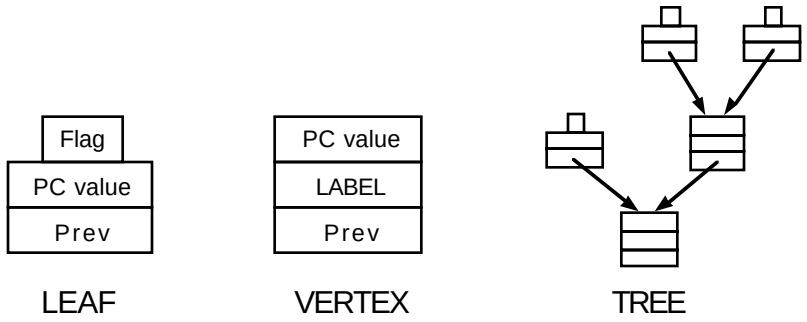


FIGURE 5.17
Branching data space

Current CPU opcode	Action
'other', jump, call, return	{ no action }
branch	send ('B', current PC) pair
indirect	send ('I', current PC, previous PC) triple

TABLE 5.13
Branching Filter send actions

Message received	Action
(PC, opcode) pair	if (current PC = PC) then send opcode to CPU else { ignore pair }

TABLE 5.14
Branching Filter receive actions

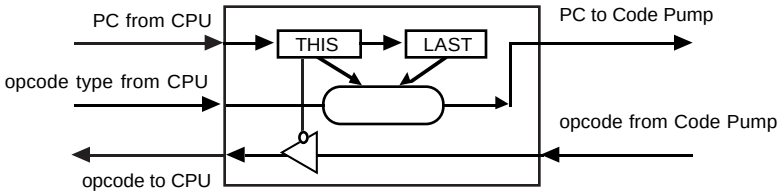


FIGURE 5.18
Branching Filter Cache design

Message received	Action
('B',thisPC)	find thisPC among branch labels delete all but tree superior to found node
('I',thisPC,lastPC)	find lastPC among inactive leaves delete all but found leaf activate found leaf

TABLE 5.15
Branching Converger actions

Data area item	Action

for each active leaf	fetch opcode at leaf_PC send opcode to Filter Cache Type of opcode: ‘other’: leaf_PC ← leaf_PC+1 jump, call: get ‘k’ from opcode send opcode to Filter Cache leaf_PC ← leaf_PC+k branch: mark current leaf as inactive get ‘k’ from opcode add child leaf1 leaf1_PC ← leaf_PC+1 activate leaf1 leaf1_label ← leaf_PC+1 leaf1_prev ← leaf add child leaf2 leaf2_PC ← leaf_PC+k activate leaf2 leaf2_label ← leaf_PC+k leaf2_prev ← leaf return or indirect: inactivate leaf

TABLE 5.16
Branching Diverger actions

5.5.6. Combining Recursion and Branching anticipation

The final and most complete μ-Net design includes anticipation of both branches and recursion. This requires a merging of the aspects of the Recursion anticipation and Branching anticipation, which is called Total anticipation. Indirect opcodes are still not anticipated in the Total version, because they cannot be anticipated at all[1].

First, the data structure is augmented to provide aspects of the stack required for recursion and the internal vertices required for branching (Figure 5.19). There are two types of data structure components, unary and binary elements. A leaf is a unary element with an empty 'next' pointer, and denotes one of the possible CPU PC values being modeled (either active or inactive). A restoration is an inactive unary element, and encodes a past call location, to be used when a return opcode is encountered. A vertex is a binary element. Active leaves denote modeled PC values, and inactive leaves denote pending indirect opcodes, which cannot be modeled.

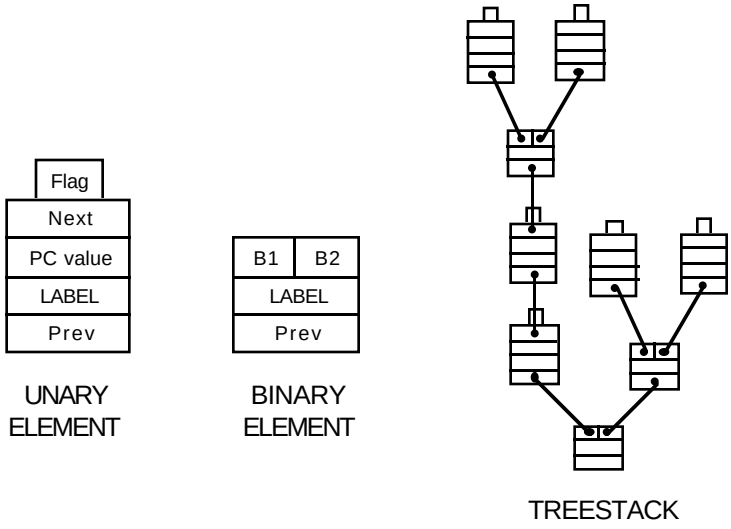


FIGURE 5.19
Total Anticipation data space

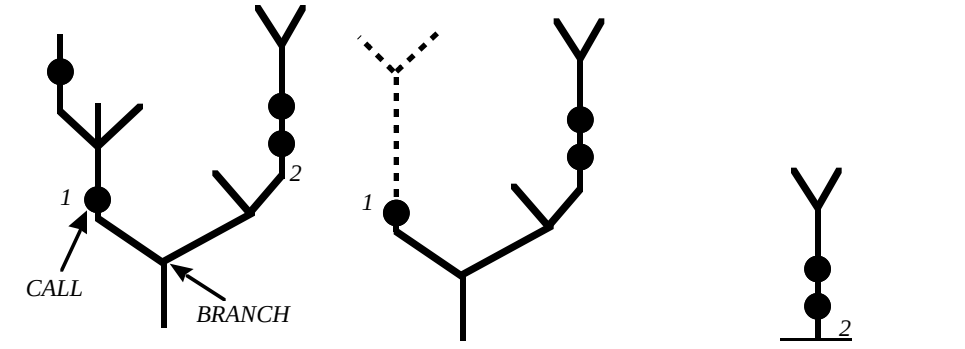
[1]Again, we claim that indirect opcodes which are imited to compile-time jump labels are equivalent to a fixed set of branches, and preclude the intent of an indirect opcode, which is to permit unpredictability.

The TreeStack is a more general structure than either a stack or tree. The Recursion anticipation data space consists of unary elements only, i.e., in a linear stack. The Branching anticipation data space consists of vertices and leaves only, i.e., a simple binary tree, with no internal unary elements.

5.5.6.1. The TreeStack

A TreeStack is a union of the notions of a tree and a stack. It is both a tree of stacks and a stack of trees. The stack implements the maintenance of the pending stack information (as produced by CALLs and consumed by RETURNs), in concert with path divergence information as indicated by BRANCHes. With respect to the stack, a CALL causes a push, and a RETURN refers back to the most recent pending CALL on the path previous to that PC. When all arms of a branch have been RETURNED or if one is selected (via a received message), the unused arms disappear (are pruned); in this way the tree aspect of the structure is managed.

The purpose of a TreeStack is to permit storage of RETURN addresses when a CALL message is sent (to model the recursion of the state space), and to permit the storage of the pairs of resultant PCs when a BRANCH message is sent. RETURNs should cause the structure on the path back to the originating CALL to be popped, unless these structures remain in use by pending BRANCHes. Multiple branches are pruned either when specifying information is received from the processor, or when all arms are popped as a result of RETURNs (Figure 5.20).



Original tree before pruning | *Branches at 1 pruned back via RETURNs, & regrown* | *Branches other than 2 pruned via received message*

FIGURE 5.20
TreeStack structure: RETURNs prune, received messages specify

For example, if a CALL occurs, then a BRANCH, a RETURN from the result of either branch should cause a transformation to the same stored CALL address. Further, when a both arms of the branch have thus been transformed, i.e., when RETURNS have been sent from all PCs in the space of the CALL, that space disappears from the image.[1]

There are other ways to consider the interaction between the sending of code, received updates of the processor's actual state, and the structure of the TreeStack. Sending out code causes the tree to grow, except in the case where a branch arm indicates a RETURN, in which case the branch arm collapses to reflect the effect of the RETURN. Received messages cause the TreeStack to be pruned, such that the received address is the root of the resulting tree, branches superior to that tree location remain, and branches subordinate to that tree location are pruned.

The transformations describing the TreeStack as a general data structure appear in Appendix G. This structure may be of more general use.

5.5.6.2. The Total implementation of μ-Net

The Filter Cache sending mechanism of the Total implementation is changed only slightly from the Branching version, because return opcodes are ignored, and indirect opcodes send triples consisting of an indicator, the current PC, and the previous PC (where the indirect opcode occurred) (Table 5.17). The Total Converger needs the previous PC to match the indirect opcode execution to the correct leaf in the TreeStack, because the path back to the root must be maintained to encode the recursion stack. This requires extending the Filter Cache internally to store PC values through a 2-element shift register (Figure 5.21). The Filter Cache receive design is unchanged (it is repeated here for convenience in describing the Total design) (Table 5.18).

The Total Converger is modified to maintain the recursion stack information as well as the superior tree graph of possible futures of the CPU PC (Table 5.19). Some compression of the paths returning to the root may be possible, by deleting vertices on the path when the vertex is not required to join two converging root paths. Multiple simultaneous root paths are possible, because more than one label can be matched by an incoming branch message.

---

[1]This is similar to virtual/real particle pair interaction, such that eventually the pair collapses. Mirage was conceived of in terms of such physics analogs, as described in Appendix B.

The Total Diverger operates as a combination of the tree maintenance mechanism of the Branching diverger and the stack mechanism of the Recursion diverger (Table 5.20). 'Other', jump, and branch opcodes are accommodated as in the Branching diverger. Call opcodes transform a leaf into an internal unary element, in the fashion of a simple stack push operation.

The return opcode is slightly more complex to accommodate in the TreeStack structure. The problem is that elements on the path back to the corresponding prior call (i.e., elements on the path back to the first internal unary element on the path back towards the root) cannot be popped, because they may encode information which other pending branches still need. Also, when branch resolution is indicated (by a received message), the Total Converger may attempt to find a label in part of the tree which was deleted.

In terms of recursion and branching, consider a call opcode then a branch. One branch may return and continue execution in the prior environment, but the information of the call (i.e., its PC) cannot be destroyed, because the other branch will need it to execute its eventual return. Also, even when all branches in the recursed space return, the tree of branches must be maintained, because incoming branch resolution information indicates which returned path is valid. In other words, because a branch can induce multiple paths back through the embedded spaces, the paths must be maintained.

The Total Diverger accomplishes this by copying the found prior unary node, and replicating it. The replicate (now a leaf) is activated, and the current leaf (where the return opcode occurred) points back to the replicate. The tree can therefore have branches whose limbs recombine, but no circular paths are ever created by these operations, so the notions of 'superior tree' and 'path to root' are maintained.

Current CPU opcode	Action
'other', jump, call, return	{ no action }
branch	send ('B', current PC) pair
indirect	send ('I', current PC, last PC) triple

TABLE 5.17
Total Filter send actions

Message received	Action
(PC, opcode) pair	if (current PC = PC) then send opcode to CPU else { ignore pair }

TABLE 5.18
Total Filter receive actions (same as Branching filter)

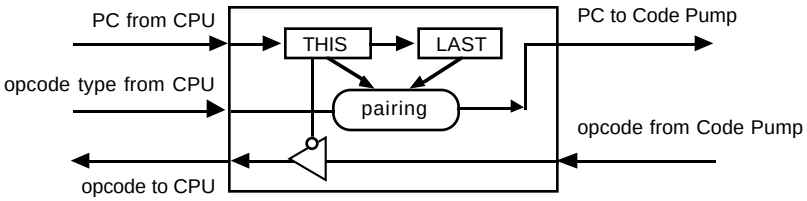


FIGURE 5.21
Total Filter Cache design

Message received	Action
('B',thisPC)	find thisPC among branch labels unmark all elements for each found element: mark all elements in superior tree mark all elements on path to root delete all unmarked elements
('I',thisPC, lastPC)	find lastPC among inactive leaves unmark all elements for each found element: found_PC ← this_PC activate found element mark all elements on path to root delete all unmarked elements

TABLE 5.19
Total Converger actions

Data area item	Action
for each active leaf	fetch opcode at leaf_PC send opcode to Filter Cache Type of opcode: 'other': leaf_PC ← leaf_PC+1 jump: get 'k' from opcode send opcode to Filter Cache leaf_PC ← leaf_PC+k call: get 'k' from opcode create new leaf new_prev ← leaf new_PC ← leaf_PC+k activate new leaf deactivate this leaf leaf_next ← new_leaf branch: replace current leaf with vertex vert_label ← leaf_label vert_prev ← leav_prev vert_b1 ← leaf1 vert_b2 ← leaf2 get 'k' from opcode add child leaf1 leaf1_PC ← leaf_PC+1 activate leaf1 leaf1_label ← leaf_PC+1 leaf1_prev ← leaf add child leaf2 leaf2_PC2 ← leaf_PC+k activate leaf2 leaf2_label ← leaf_PC+k leaf2_prev ← leaf return: find first unary antecedent copy found element new_prev ← found_prev new_PC ← found_PC+1 this_next ← new activate new leaf deactivate this leaf indirect: deactivate leaf

TABLE 5.20
Total Diverger actions

5.6. Implications

The implications of the μ-Net architecture on the performance of the processor/memory system can now be considered. The performance equations defined before are repeated below:

Equation 5.1: $T_{optimal} = N * t$

Equation 5.2: $T_{conventional} = N * (t + r)$

Equation 5.5: $T_{cache} = N * (t + r * M)$

Equation 5.7: $T_{prefetch} = N * (t + r * F)$

where $F \leq J + B / 2 + C + R + I$

Equation 5.11: $T_{\mu\text{-}Net} = N * (t + r * P)$

where T = total time to execute N instructions
t = time for the processor to execute 1 instruction
r = round trip time between the processor and memory
M = probability of a conventional cache miss
F = prob. of a conventional prefetching cache hit
P = probability of the μ-Net Code Pump miss

The values of M, F, and P can be compared. As the time separation between memory and the processor increases, the r component of the equations begins to dominate. The goal is to determine the conditions under which P is less than M or F.

The value of P can be further elaborated, in the various levels of implementation of μ-Net. The levels of implementation describe the degree for which various opcode classes are included in the send transformations. Implementations model 'no' send transformations, send transformations on regular opcodes only, on regular and jump opcodes, etc. These correspond to predicting the next opcode (and sending it) after regular opcodes, jumps, calls, etc.

The following formulae describe the ways in which the prediction can alleviate the effects of round trip latency (Equations 5.13, 5.14, 5.15, 5.16). Evaluation of these formula, thus far, depends only upon measurements of the probability of each class of instruction type.

Definitions: O = other instructions
J = jumps
C = calls
R = returns
B = branches
I = indirect (calls, branches, or jumps)
where $N = O + J + C + R + B + I$

Equation 5.13: $T_{null} = N * (t + r)$

Equation 5.14: $T_{other} = N * (t + (J + C + R + B + I) * r)$

Equation 5.15: $T_{other,jump} = N * (t + (C + R + B + I) * r)$

Equation 5.16: $T_{other,jump,call} = N * (t + (R + B + I) * r)$

These formulae require a constant sized Code Pump (i.e., only a finite amount of storage to implement the imaging of the PC of the processor), because they consider only transformations which preserve the size of the image.

If the Code Pump is augmented to accommodate an arbitrary amount of stack space, as required to handle the pending *calls* that could occur during one round trip time, the transformations indicated by RETURN opcodes can be modeled, as shown in Equation 5.17. Because all the types of opcodes whose transformations have been modeled thus far preserve the size of the PC image, issues of bandwidth in the communication can be ignored. Only one PC value is ever active, so the instruction communication consists of a single stream.

Equation 5.17: $T_{other,jump,call,return} = N * (t + (B + I) * r)$

If the Code Pump is further augmented to accommodate an arbitrary amount of TreeStack space, as required to handle the pending *branch executions* that could occur in one round trip time (i.e., the entire branch, not just the calls), the transformations indicated by BRANCH opcodes can be modeled as well.

Recalling the discussion of the Mirage model, there are limitations to this predictability beyond that of the data storage in the Code Pump. When the image space becomes large, the state space must be partitioned coarsely enough to be able to send a small enough number of messages such that the entire space is covered by the guards of

the set of the messages sent; this is from our definition of stability (entropic, as also discussed in Chapter 2).

At this point that limitation becomes applicable. Expansion of the PC image is limited not only by the space needed to represent it (the size of the TreeStack structure), but also by the total number of messages in transit. All possible destinations of all pending PCs due to BRANCH instructions cannot be sent if the round trip time does not permit it. Once a BRANCH is encountered, one round trip time exists to send the messages corresponding to its isopotent set (the set of opcodes of every possible branch destination, in this case).

Partitioning the state space coarsely is not a concern here; there is not enough information in the opcodes of a program to collapse the information efficiently, i.e., to send a single opcode with a label of the five locations where it occurs. This kind of dynamic partitioning is less effective than simply repeating the opcodes for each address sent. Further, no compression of the state space (i.e., encoding of the addresses used as labels) is really necessary, although some useful assumptions can be made. For example, in many CPUs, branches, jumps, and calls are limited to some fixed maximum distance from the current PC, usually less than the limit of the entire PC space (i.e., short jumps vs. long jumps). If a short jump opcodes cause transformations of the PC which are, at most, 8 bits offset from the current PC, only the lowest 8 or 9 bits of address need be sent as the label. This latter minimization of communication is not the same as partitioning the space coarsely; it reflects only a useful encoding of the guards.

A version of Equation 5.17 reflects the use of the bandwidth to accommodate the multiple possible values in the memory's image of the processors PC (Equation 5.18).

Equation 5.18: $T = N * (t + (I + X * B) * r)$

where X = percent of branch possibilities which cannot be communicated

X can be measured directly, from an implementation (either direct or emulated), or can be approximated through the application of some assumptions. Recall the prior discussion (Chapter 2), which defines *communicability* in the abstract model. This refers to the ability to partition the image space coarsely enough, and to send a small enough set of short enough messages, that the entire set that covers the image (the isopotent set) can be communicated in the information separation (bandwidth-delay product) available; if this can be done, and if the set sufficiently constrains the image space, then the image (and thus the communication) is stable (entropically, or otherwise).

In the abstract model state space volumes are introduced to represent images of remote state spaces. Each image either continues in time, or is transformed into a set of images. In μ-Net, each PC value either continues as a single point (via JUMP, CALL, RETURN, or regular opcodes), splits into two points (BRANCH), or becomes a set too large to partition (INDIRECT).

Under the simplifying assumption that branches are indistinct, the ways in which branch possibilities can be communicated can be computed, thus determining the size of a isopotent set which can be communicated per round trip time. This formula was also developed earlier (Chapter 2, Branching Streams).

The ultimate goal is to determine the probability of not predicting an opcode, i.e., the 'miss' rate of the Code Pump, and to determine the storage required to maintain the TreeStack structure.

Let L be the limb length (typically 6 to 8 opcodes in length), let D be the branch degree (usually 2, because only binary branches are modeled), and let *rtt* be the round trip time, as defined in prior discussion of channel utilization. Also let OP_BW be the bandwidth, in opcodes per unit time. In time *rtt*, the original PC of the processor attempts to execute opcodes (Equation 5.19, where CPI = clocks per instruction [He90]). In that time, only a portion of the tree of possible execution streams can be transmitted (Equation 5.20). With respect to an individual execution path, only a portion of the path is sent, i.e., the portion equivalent to the tree depth (Equation 5.21). The ratio of the path length which the CPU attempts to execute in the round trip time to the path length in the tree sent is the probability of success (Equation 5.22).

Equation 5.19: $$opcodes\_executed = \frac{rtt}{t}$$

where $$t = \frac{CPI}{clock\_rate}$$

Equation 5.20: *depth* such that $$rtt * OP\_BW = \frac{D * \left(D^{depth} - 1\right)}{D - 1} * L$$

Equation 5.21: $$depth = \log_D\left(\frac{rtt * OP\_BW * (D-1)}{L * D} + 1\right)$$

Equation 5.22: $$tree\_success = \frac{\log_D\left(\frac{rtt * OP\_BW * (D-1)}{L * D} + 1\right)}{opcodes\_executed}$$

This analysis focuses on prediction failure based on the inability to transmit enough alternates in the branching stream; it ignores the effects of INDIRECT ipcodes, because they are so infrequent (less than 0.3%).

Equation 5.17 refers to Equation 5.23, which depends on the growth of the state space image as determined by the latency, and the management of that growth which is limited by the information separation (i.e., Equation 5.22). The result is the effective communicability in μ-Net (Equation 5.24).

Equation 5.23: $$X = 1 - tree\_success$$

i.e., $$X = 1 - \frac{\log_D\left(\frac{rtt * OP\_BW * (D-1)}{L * D} + 1\right)}{\frac{rtt}{t}}$$

Equation 5.17: $$T = N * (t + X * r)$$

Equation 5.24: $$T = N * \left(t + \left(1 - \frac{\log_D\left(\frac{rtt * OP\_BW * (D-1)}{L * D} + 1\right)}{\frac{rtt}{t}} \right) * r \right)$$

Before this point in the discussion, the round trip latency did not figure into the calculation of performance. In considering the way in which the state space image can expand beyond the ability to manage it, latency becomes an issue. Latency determines to what extent the expansion of the image can be managed. As a result, it plays a role not only in the penalty assessed when prediction fails, but also in the evaluation of the frequency of penalization (X).

In the limit of this equation as the available bandwidth goes to infinity, all isopotent sets can be sent, and X consists of only those instructions whose transformations cannot be modeled, those of INDIRECT opcodes. These opcodes require a round trip latency in which to determine the resultant PC, as computed at the processor (Equation 5.25). This equation represents an ‘Amdahl’s Law’ of communicating interaction, i.e., that the

speedup is limited to the predictible component, so the speedup is limited to the reciprocol of the percentage of indirect instructions 5.26.

Equation 5.25: $T_{optimal\ \mu\text{-}Net} = N * (t + I * r)$

Equation 5.26: $Speedup_{\max} \leq \frac{1}{I}$

In these equations, *OP_BW* is the number of opcodes sent per unit time, and *t* is the time to execute an opcode, the two of which are related via the CPU *clock_rate* and the *CPI* (cycles per instruction). Typical values for *CPI* values are as low as 1.0 for true RISC processors, 1.3 for a heavily pipelined CISC (68040) [Mo89], and up to 14 for other CISCs (68000) [Ma84] (Table 5.21). The *clock_rate* of a CPU is typically 10 to 30 Mhz.

Processor	CPI
Motorola 68000	13.5 [Ma84]
Motorola 68010	11.4 [Ma84]
Motorola 68020	6.6 [Ma84]
Motorola 68040	1.3 [Mo89]
DecStation 3100 (MIPS)	1.87 [He90]
Sun 3/75 (680x0)	10 [He90]
DLX (RISC) [He90]	6.28 [He90]

TABLE 5.21
CPI (*EXEC*) values of common CISC and RISC CPUs

5.7. Conclusions

This chapter presented equations describing the performance of processor-memory protocols, and developed a new protocol called μ-Net based on the abstract Mirage model. μ-Net's design, and the various degrees of its implementation were also

presented. Performance equations require either detailed modeling of the expected opcode stream, or measurement of real opcode streams. Initial modeling here indicates that the performance of μ-Net depends on the internal structure of the stream, specifically, how the stream branches and how long each branch persists.

In order to compare these equations, detailed measurements are required of real opcode streams. These measurements include the expected branch degree, and the expected limb length. Other required measurements include the distribution of the opcode classes. Prior work can indicate the expected values of cache miss (M) or miss with prefetch (F). Description of these measurements and analysis of the results appears in Chapter 6.

CHAPTER 6

μ-Net under a μ-Scope

Evaluation of μ–Net requires measurement of cache miss (*M*), prefetch cache miss (*F*), and μ–Net miss (*P*) probabilities. Cache performance parameters are available in the literature [Sm82]. Some measurements of opcode statistics have also been published [He90], but some of the statistics required to evaluate μ–Net are not available, and were made by direct measurement of opcode execution. The measured design is called μ–Net (MicroNet), so this measurement method is called μ–Scope (MicroScope). A description of μ-Scope appears in Appendix H.

Required measurements which were not available in the literature include statistics of opcode distributions according to the partitioning indicated by μ–Net. μ–Net also requires measurements of the average number of non-control transfer opcodes between control transfer opcodes; the definition of control transfer differs for the various levels of design discussed (Chapter 5). μ-Scope was developed to perform these measurements. Other tools for opcode distribution measurement were considered (as discussed in Appendix H), but were either insufficient for the measurements desired (Pixie) or proprietary and not available to us.

These measurements are made on the dynamic trace of opcode executions of various benchmarking programs. The applicability of benchmarks for general analysis is not advocated here. This selection represents widely available benchmarks which

compiled and ran without explicit error in μ-Scope. Other benchmarks were considered, but omitted because they were not available in either SPARC Assembler or C language source code (which μ-Scope is limited to), or because of system limitations during the preparation of this dissertation (i.e., the benchmark was of limited interest due to its specificity, and there was insufficient disk space and insufficient processing capability to examine all available benchmarks).

The GNU C, T_EX, Linpack, and Dhrystone benchmarks were chosen for these measurements. GNU C is a freely distributed C language compiler; our version (1.35) and the test set weightings used were extracted from the SPEC Benchmark Release 1.0 [Wa90]. T_EX is an embedded text typesetting program; our version was contained in the benchmark distribution of [He90][1]. The C language versions of both Dhrystone and Linpack were obtained from standard Internet libraries[2]. Each benchmark has particular characteristics, listed below:

- GNU C - a large program, complex, recursive; uses many language features
- T_EX - a typical large frequently-used program
- Linpack - a set of routines collected from a mathematical software library
- Dhrystone - a contrived program claimed to represent intense integer computing

Both Linpack and Dhrystone represent intense mathematical computing, but lack sophisticated use of compiler and language constructs. They are often used for estimating measurements of theoretical CPU performance, such as FLOPS or MIPS.

T_EX represents a common but complex application program, exhibiting significant levels of recursion and opcode proportions in general purpose systems. The GNU C compiler represents a typical upper-bound (or at least a reasonable bound) of complexity, both in its static structure and dynamic execution.

---

[1]The software supplement to [He90] is available via anonymous FTP at max.stanford.edu.

[2]These benchmarks are available via Internet e-mail; send queries to netlibd@surfer.epm.ornl.gov.

Some of the measurements performed here have also been presented elsewhere. They are repeated here because it is important to compare the measurements of various characteristics on a common set of benchmarks, and because some necessary measurements were not found in the literature, notably the variability in limb length with various treatments of jump, call, and return instructions, as discussed in detail below.

6.1. Performance gains

The performance gain of μ–Net is measured by the extent to which P is less than either M or F. We need to compare actual measurements in order to make these measurements. As noted before, there is no incompatibility between conventional caches and μ–Net, so the comparison is simplified by the assumption that μ–Net also has a conventional cache inside the Filter Cache. The Filter Cache then behaves like the conventional cache, with the exception of the one entry which represents the latest opcode/address pair as received from the incoming communication stream. Further, if this ‘streaming’ entry is hit, it is copied into a conventional entry, just as when a memory reply occurs after a conventional cache miss.

In prefetching caches, a miss of a particular datum causes a sequence of data to be fetched. This is equivalent to the assumption that all opcodes are of type OTHER (as described in Chapter 5) (i.e., simply increment the PC), and that the memory will send a number of opcodes equivalent to the line size or lookahead of the cache upon a miss.

Communication can be modeled as a branching stream (as described in Chapter 2). The branching of this stream is 2 because BRANCH opcodes accommodate 2 alternates. The limb length of the stream describes the extent to which a particular image value (PC) remains predictable by transforms. In μ–Net, limb length is the average number of non-control flow opcodes between control flow opcodes. Opcodes which alter the flow of control are those that cause branching in the stream (BRANCHes), or that cause a round trip latency penalty because they are not anticipated (INDIRECT, and opcodes not anticipated in a given implementation).

Dynamic opcode traces (i.e., execution traces) indicate the following distributions of opcode classes. I-Call’ and ‘I-Jump’ denote indirect CALL and JUMP opcodes, respectively; SPARC CPUs have no indirect branches (Figure 6.1). Other published measurements [Ka91] of *troff* (GK troff) and *cc* (GK cc) are included for comparison.

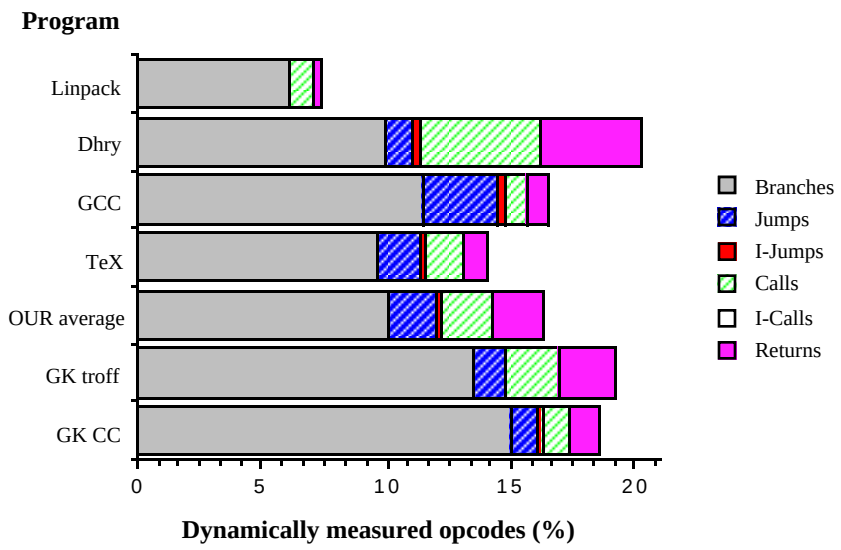


FIGURE 6.1
Dynamic control opcode distributions

In the following discussion, INDIRECT refers to the aggregate of all indirect opcode types. These measurements were made on the actual execution of the benchmarks on a SPARC (SUN-4), which has no indirect branch opcode. The vast majority of the indirect opcodes were indirect JUMPs, due to a preference in the C compiler used[1].

The code was measured by compiling the benchmarks (using optimization), and generating SPARC assembler code. The assembler was modified by an AWK script, which inserted additional instructions to count the various classes of opcodes during execution, as well as measuring the number of OTHER opcodes between control of flow instructions, such a JUMP, BRANCH, CALL, and RETURN. Existing tools, such as PIXIE and PIXIESTATS for the MIPS processor, were not used because the method for determining basic blocks causes errors in the measurement of opcodes due to the effects of indirect JUMPs and CALLs. The SUN equivalents, SPIX and SPIXTOOLS, as well as the SUN SPARC emulator SHADOW, were requested but not available for public use. See Appendix H for a further discussion on the measurement techniques used here.

---

[1]In the test code we measured, the SPARC C compiler generated indirect CALLs only where structures were returned as function values (4 times), and only in the GNU C compiler code. Indirect JUMPs appeared in almost all of the test code, in very small (<0.3%) amounts.

The variance in these measurements is large, due to the arbitrary choice of canonical code examples, and the variability between benchmarks. Estimates of the measurements are summarized in Table 6.1. These are verified by similar, albeit more limited published measurements [Ka91][1].

OPCODE	%
Other	84
JUMP	2
CALL	2
RETURN	2
Indirect	0.3

TABLE 6.1
Approximate dynamic opcode distribution

Comparing μ–Net to an architecture without a cache, μ–Net reduces the effect of latency by the percent of instructions which can be accommodated by the Code Pump (Figure 6.1, Table 6.2).

These comparisons assume that the majority of time spent in execution is due to round trip latency, i.e., that $r >> t$ by at least one order of magnitude, preferably at least two. This necessitates a latency of at least 100 instruction execution times, which in a 1 gigabit/sec pipe, assuming 32-bit instructions, corresponds to a latency of 3,200 bits, or 3.2 μsec, or 960 meters, which is about 6 city blocks. For example, these results apply where the processor and memory are separated by at least 3 city blocks, which is completely reasonable for a client/server system located on a small college campus.

[1]Our method of dynamic tracing, μ-Scope, was developed using methods similar to those which appear in the Katevenis paper. To use a common euphemism, we reinvented their wheel, then checked theirs to tune ours. A further discussion of μ-Scope appears in Appendix H.

Opcode groups anticipated	% not predicted	Speedup	Caveats
none (conventional)	100	1	
Other	16	6.3 x	
Other, JUMP	14	7.1 x	
Other, JUMP, CALL	12	8.3 x	
Other, JUMP, CALL, RETURN	10	10 x	unlimited stack
all but Indirect	0.3	333 x	unlimited TreeStack

TABLE 6.2
Approximate speedup in various degrees of implementation

Measurements of cache utilization have appeared often in the literature [He90] [Sm82]. The miss rate *M* is approximately 3%, which with prefetching (*F*) drops to 1.5%. These assume extremely large (or infinite) caches. A 1K cache has a miss rate of about 20%, a 4K about 12%, a 128K about 5%, and the miss rate approaches 3% only for caches larger than 256K.

Large (infinite) caches miss mostly due to first time code use, whereas smaller caches miss both from first time code use and from most control opcodes, because the cache is so small that changes in control flow are likely to require opcodes not in the cache. A prefetching cache misses only from control flow opcodes because first time code use is predicted by the prefetch.

Changes in control flow occur from JUMPs, CALLs, RETURNs, and INDIRECT opcodes, as well as from half of the BRANCHes, the latter assuming that BRANCHes are taken about 50% of the time [He90]. Such control opcodes comprise 11.3% of the instruction stream (JUMP, CALL, RETURN, INDIRECT, and half of the BRANCHes), so control opcode misses comprise 11.3% of the cache misses, i.e., prefetching caches are estimated to miss 11.3% of the time.

This estimate can be further refined because branches can be separated into backward and forward branches, where forward branches are more likely to cause misses. About 25% of branches are backward (empirically [He90]), because backward branches

are generated for loops, whereas forward branches are generated for IF and CASE statements, which are more common in source code. Backward branches are more likely to hit existing cache entries, so a revised formula for misses in small prefetching caches is 10% (JUMP, CALL, RETURN, INDIRECT, and half of 3/4 of the BRANCHes).

Prefetching reduces the miss rate up to 50% for large caches, compared with non-prefetching caches [Sm82]. Misses occur due to CALLS and RETURNS more than BRANCHES, because most local code remains in the cache. A small cache has a 20% miss rate and 50% of these misses are predictable (due to the opcode distribution difference), so the miss rate of a small prefetching cache is expected to be 10%, equivalent to our approximation based on opcode distributions.

Cacheing and cache anticipation are compared to μ–Net anticipation, in Table 6.3, assuming nothing about branch outcome. Opcodes are called JUMP, CALL, RETURN, BRANCH, INDIRECT, and OTHER for all other types, and labelled respectively J, C, R, B, I, & O [He90]. The table notes the cache size required to approximate various versions of μ–Net, based on which sets of opcodes are anticipated. In the Recursion case, storage is required in the Code Pump of 100 addresses; justification of this figure is discussed later.

μ–Net Version	Opcodes anticipated	miss rate	equivalent cache	μ–Net storage
Unit Linear	O	16%	2K bytes	none
Linear	O+J+C	12%	4K bytes	none
Recursion	O+J+C+R	10%	8K bytes	100 addresses

TABLE 6.3
μ–Net implementations and cache equivalents (no branch assumption)

Assuming branches are taken 50% of the time, i.e., that half the branches are predicted, more dramatic comparisons result (Table 6.4).

A naive comparison of these numbers results in the conclusion that most degrees of implementation of μ–Net are far worse than either caches or prefetching caches. Although cache miss rates are often quoted as 3% (1.5% for prefetching caches), these rates apply only to very large caches. Current microprocessors have small caches (8K for an Intel 80486, 256 bytes for a Motorola 68030, 4K for a Motorola 68040 and Intel

80860 [Mo89]). In comparison, μ–Net beats most of these on-chip caches, if transmission latency is large.

μ–Net Version	Opcodes anticipated	miss rate	equivalent cache	μ–Net storage
Unit Linear	O	11%	8K bytes	none
Linear	O+J+C	7%	16K bytes	none
Recursion	O+J+C+R	5%	50K bytes	100 addresses

TABLE 6.4
μ–Net implementations and cache equivalents (equiprobable branching)

Opcode	% of control	estimated cache miss distribution
OTHER	-	50 %
JUMP	12 %	6 %
CALL	12 %	6 %
RETURN	12 %	6 %
BRANCH	62 %	31 %
INDIRECT	2 %	1 %

TABLE 6.5
Adjusted dynamic opcode distributions (approx. a cache miss stream)

These results can be extrapolated to a system where μ–Net complements cacheing, rather than replacing it. Opcode distributions change with the inclusion of a cache, and measurements including a cache implementation were beyond the scope of this dissertation. One estimate assumes that the proportions of control opcodes (relative to each other) remains the same, between a conventional instruction stream and the stream of cache misses. Assuming that control opcodes increase to a total of 50% of the cache miss stream [He90], keeping all other relative proportions the same, this results in an

adjusted opcode distribution (Table 6.5), and expected speedup (Table 6.6). The combined entries (cache with μ–Net) indicate that a resulting miss is the result of the combination of a cache miss and a failure of μ–Net to have anticipated the request for that opcode (i.e., the miss and failure rates are multiplied).

Implementation	Failure rate	Speedup (ratio to null)
cache alone (large- 256K)	3%	33 x
cache alone (small- 4K)	10%	10 x
prefetch cache (large- 256K)	1.5%	67 x
small cache + μ–Net(O)	10% * 50% = 5%	20 x
small cache + μ–Net(O,J)	10% * 44% = 4.4%	22 x
small cache + μ–Net(O,J,C)	10% * 38% = 3.8%	26 x
small cache + μ–Net(O,J,C,R)	10% * 32% = 3.2%	32 x
small cache + μ–Net(O,J,C,R,B)	10% * 1% = 0.1%	1000 x
large cache + μ–Net(O,J,C,R,B) [BEST]	3% * 1% = 0.03%	3,333 x

TABLE 6.6
Performance increases where latency dominates design parameters

According to this table, a small cache with a Recursion μ–Net (using 400 bytes of internal stack space) uses a total of 4.4K bytes of space, but achieves the miss rate equivalent to a 256K byte cache. Note that the Recursion μ–Net uses the same memory bandwidth as a system without a cache; only the Branching and Total μ–Nets require bandwidth higher than the Null design.

These are overestimates of the expected speedup, because the distributions of cache miss code are different from distributions of overall code use. Misses will be generated disproportionately by JUMPs, CALLs, RETURNs, and INDIRECTs, because these kind of control flow changes are not likely to be contained in code already accessed, due to the principle of locality. Measurements of code distributions in cache misses were

unfortunately beyond the scope of this dissertation, but we do not expect they would significantly change the conclusion that μ–Net/cache exceeds the speedup capability of cache/prefetch or cacheing alone.

More sophisticated experiments are required to extend this analysis to the Branching and Total μ–Net cases, as described later.

6.2. On the feasibility of implementations as architectures

The feasibility of these implementations as realizable architectures depends on the real size of the stack or TreeStack data structures in the Code Pump.

For example, the size of the stack structure in the Recursion μ–Net is equal to the largest number of pending RETURN opcodes during the entire code execution (i.e., the depth of recursion). Cumulative percentages of the CALL opcodes at each level of recursion are plotted in Figure 6.2. For a particular stack size, the plot indicates the probability of not being able to store a CALL address, for use by its corresponding RETURN. For example, 85% of CALL/RETURN pairs can be accommodated in the T_EX benchmark by a stack of size 17, whereas the stack would have to be increased to a size of 50 to handle the same likelihood of overflow in the GCC benchmark.

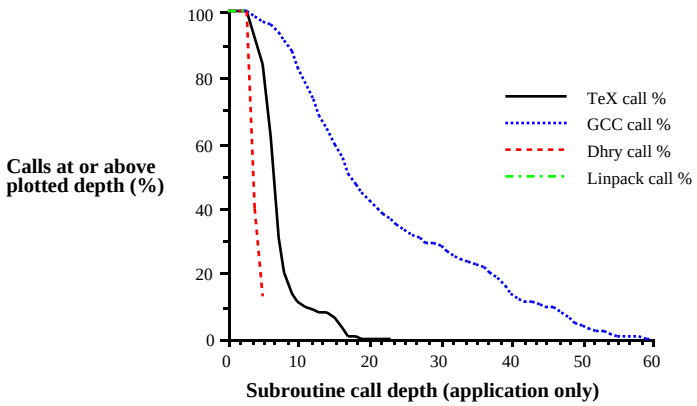


FIGURE 6.2
Percent of CALLs occurring at or above a given depth of recursion

Some of these depth measurements are not very accurate because the depth of pending calls could not be traced within the operating system with μ-Scope (Figure 6.3). This analysis is similar to that presented in [He90], where their results indicate that C compiled code has 5% overflow at a depth of 6, LISP at a depth of 7, and SmallTalk at a depth of 9, so the results depend on the program type as well as language and compiler characteristics.

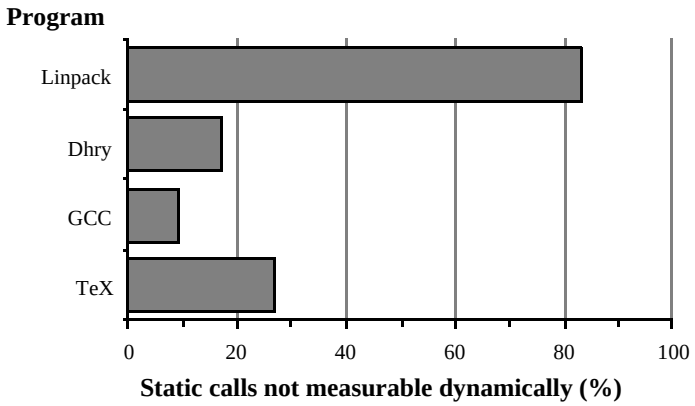


FIGURE 6.3
Percent of CALLS not depth-traced (system calls)

Because only RETURN addresses need to be stored, a stack as small as 10 items will handle all of the Dhrystone and Linpack RETURNs, and miss only 10% of the T_EX RETURNS, although it would fail in 85% of RETURNS in a highly recursive program, such as the GCC compiler. Increasing the stack to as little as 100 values allows the Recursion μ–Net to anticipate 100% of the user-code RETURN opcodes. Thus storage for a stack structure in the Code Pump which is large enough to accommodate all pending levels of recursion should be trivial to implement.

The size of the TreeStack structure also limits the feasibility of implementations. The TreeStack represents not only the information in the stack of pending recursions, but also the information on the expansion of the state space image. In order to estimate the size of this structure, there are two options. The first involves emulating the Code Pump, and determining the maximum extent of the structure as the benchmarks were executed. This level of experimentation was beyond the scope of this dissertation, but may be considered as a future research area.

The second involves some assumptions. Rather than directly measuring the size of the TreeStack, an estimate of its size can be made using the same simplifying assumptions as the branching stream analysis (Chapter 2). Assuming that the state begins in a finite set of values (i.e., a single PC, in this case), and that the sequence of PCs can be expressed as a tree, a snapshot of the image of the remote PC at any given time is expressed by the set of PCs in a level of the tree thus formed.

The branch degree of the tree is 2, modeling only BRANCH opcodes, because INDIRECT opcodes expand the space infinitely (actually, they expand the space by a degree of the size of the PC, at which point that level in the tree covers all possible PCs, as described before). The other relevant tree characteristic is the average limb length, defined as the number of opcodes between those which cause the tree to either branch or terminate (due to a terminated modeling).

For example, in the Total μ–Net, limb length is the number of opcodes between BRANCH or INDIRECT opcodes. In the Branching μ–Net, limb length is measured between BRANCH, INDIRECT, and RETURN opcodes. Figure 6.4 shows the average limb length in each benchmark, as measured for different degrees of implementation. In these graphs, ***linearity*** refers to limb length, i.e., linearities in the flow of control in the opcode stream.

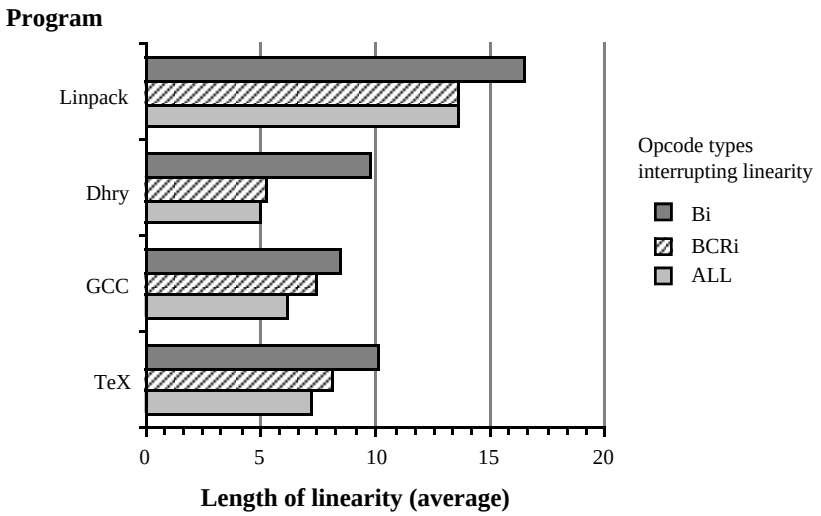


FIGURE 6.4
Average limb length (linearity)

For this graph (Figure 6.4), ALL refers to the Unit Linear μ–Net (i.e., all control opcodes delimit limb lengths). BCRi refers to the Linear μ–Net (BRANCH, CALL, RETURN, and INDIRECT), and Bi refers to the Total μ–Net. Other combinations were not measured, but would be interpolations of these values (e.g., Branching μ–Net and Recursion μ–Net), because the Total μ–Net exhibits the longest possible limb length, and Unit Linear exhibits the smallest. Modeling JUMP instructions (ALL/Unit Linear vs. BCRi/Linear) increases the average length by 10-20% from around 6 to around 7 opcodes, but that modeling CALLs and RETURNs (Recursion) increases the averages by about 40% (or as much as 100%), to around 9 (Figure 6.5).

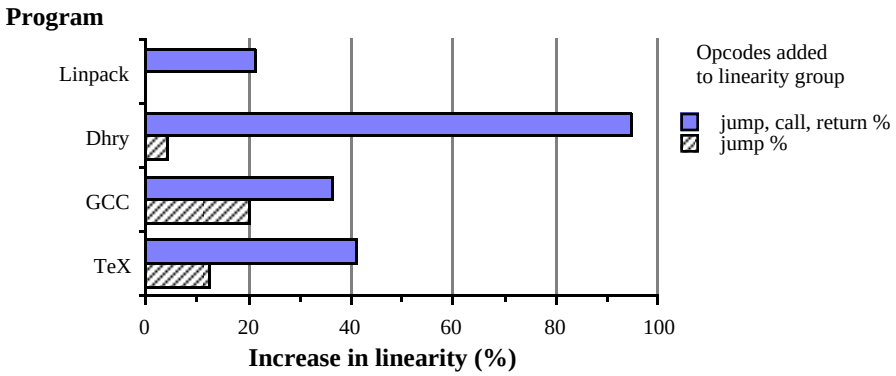


FIGURE 6.5
Percent increase in limb length (linearity), adding calls and returns

The distributions of limb lengths was also measured because the variance in these averages was very large (greater than 100%). These figures plot cumulative probabilities, i.e., for a given length, the ratio of the number of limbs at that length or less to the total is plotted, so that the point indicates “the probability that a random limb is shorter or equal to the limb length indicated” (Figures 6.6, 6.7, 6.8, 6.9). The mode of these distributions can be read directly from these plots, as the 50% value (i.e., 50% of the arm lengths are less than or equal to the plotted value).

In this set of plots, ALL indicates the limb length distribution of the Unit Linear μ–Net, BCRi indicates the limb length distribution of the Linear μ–Net, and Bi indicates the limb length distribution of the Total μ–Net. Linpack exhibits a jump in limb length statistics, presumably because it consists primarily of a set of nested FOR loops, and data from the inner portion of these loops overwhelms the statistics. The scientific programs (Linpack and Dhrystone) exhibit large limb length increases if CALLs and RETURNs are

accommodated within the limbs, because the repetitious structure of the code ensures interleaving of CALL/RETURN opcodes with BRANCHes; if CALLs and RETURNs are then included within the limb, limb length increases.

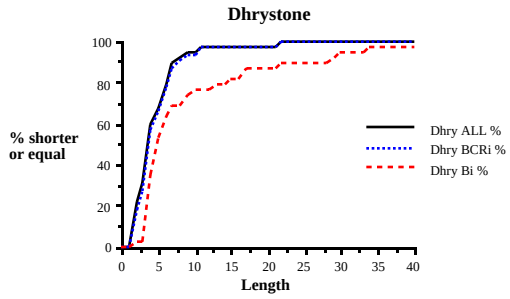


FIGURE 6.6
Dhrystone limb length distribution

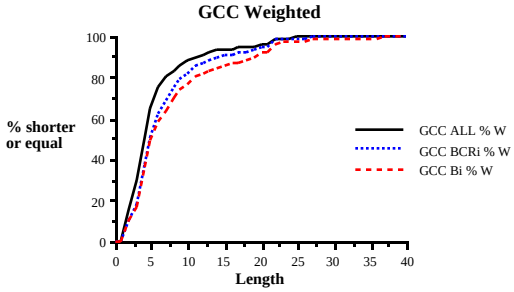


FIGURE 6.7
GCC (weighted) limb length distribution

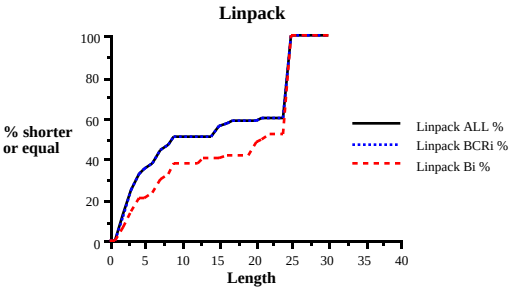


FIGURE 6.8
Linpack limb length distribution

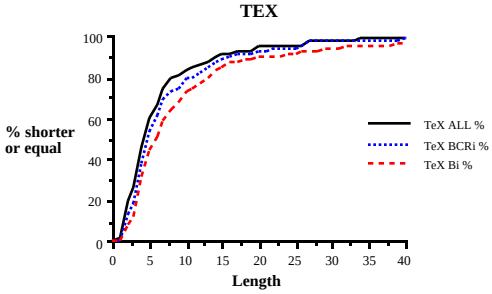


FIGURE 6.9
TEX limb length distribution

Cumulative branch arm length distributions of various benchmark programs

Rather than using the complete distributions shown above to evaluate the effects of branching on channel utilization, the mean and median of these distributions are used (Table 6.7). These measurements can be used to compare the effects of limb length increases on the Total μ–Net implementation. In effect, the performance of the Total μ–Net is compared to a Total μ–Net without recursive accommodation, called *Total-Recursive*, and to a Total μ–Net without either recursive or linear accommodation (i.e., Branching μ–Net without linear accommodation), called *Branching-Linear*.

Now that reasonable values for limb length (Table 6.7), and branch degree (2)have been measured, μ–Net's channel utilization can be evaluated. μ–Net utilization is determined by the miss rate of the μ–Net anticipation mechanism; Equation 6.1 describes

the performance measure, and Equation 6.2 is the miss probability component which has been simplified for a branch degree of 2.

μ–Net implementation	Limbs delimited by	Mean L	Median L
Branching-Linear	BRANCH, CALL, RETURN, JUMP, INDIRECT	7	4.3
Total-Recursive	BRANCH, CALL, RETURN, INDIRECT	8	4.75
Total	BRANCH, INDIRECT	10	5.8

TABLE 6.7
Mean and median limb lengths for μ–Nets implementing branching

Equation 6.1:
$$T = N * \left(t + \left(1 - \frac{\log_D \left(\frac{r * OP\_BW * (D-1)}{L * D} + 1 \right)}{\frac{r}{t}} \right) * r \right)$$

where OP_BW = opcode bandwidth, i.e., opcodes per unit time
D = branch degree (usually 2)
L = limb length, in number of opcodes
N = total number of opcodes executed
T = resulting execution time
r = round trip time
t = time to execute an opcode = $\frac{CPI}{cpu\_speed}$

Equation 6.2:
$$\mu Net\_miss\_rate = \frac{\log_2 \left(\frac{r * OP\_BW}{L} + 1 \right) - 1}{\frac{r}{t}}$$

The anticipatory miss probability is determined by the sum of the probability of an INDIRECT opcode occurring (0.3%) and the probability that the Code Pump has insufficient bandwidth to send all the possibly requested opcodes. Performance is proportional to the channel utilization, which is proportional to hit probability, which in

turn is equal to 1-miss_probility. The performance of the three variations of anticipation (Branching-Linear, Total-Recursive, Total) can be plotted vs. bandwidth (*OP_BW*) round trip latency (*r)*, with branch lengths as indicated above (Table 6.7). Figure 6.10 describes the Branching-Linear μ–Net (limb length L=7), Figure 6.11 describes the Total-Recursive μ–Net, and Figure 6.12 describes the Total μ–Net. These figures allow comparison of mean limb length values.

The graphs show how performance (i.e., channel utilization) decreases as latency increases, and increases as bandwidth increases. They also indicate that small increases in average limb length, as derived from more complete implementations of the Code Pump, achieve large increases in utilization. *Increases in bandwidth compensate for increases in latency, although a linear increase in latency requires a corresponding exponential increase in bandwidth (see shaded areas).*

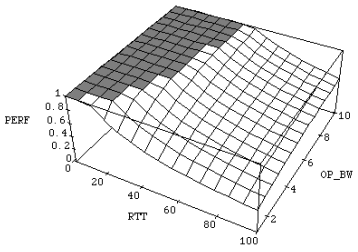


FIGURE 6.10
Mean limb length (L = 7) hit probablity vs. BW vs. rtt.

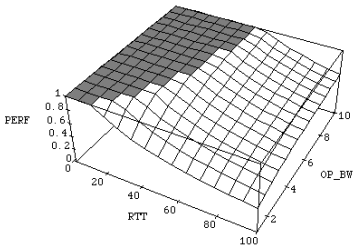


FIGURE 6.11
Mean limb length (L = 8) hit probablity vs. BW vs. rtt.

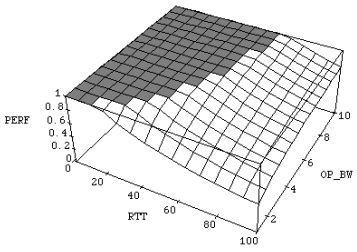


FIGURE 6.12
Mean limb length (L =10) hit probablity vs. BW vs. rtt.

μ–Net anticipatory hit probability vs. bandwidth (OP_BW) and round trip latency (rtt)

In order to better compare these three graphs, 2-dimensional slice of each graph were plotted together. μ–Net anticipation channel utilizations are compared for fixed values of bandwidth (OP_BW = 2), and for the mean and median values of branch length (Table 6.7). μ–Net is compared against a non-anticipatory protocol in which only a fixed deterministic prefix (of length 7) can be prefetched. The non-anticipatory protocol is denoted by the solid curve, and Branching-Linear, Total-Recursion, and Total μ–Net versions achieve increasing channel utilizations, as indicated by wide-dashed, narrow-dashed, and dotted lines (Figs 6.12, 6.13).

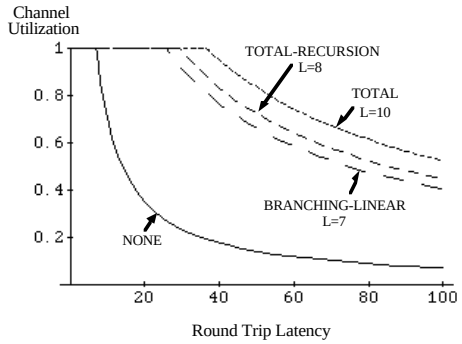


FIGURE 6.12
Mean limb length rtt. vs. utilization

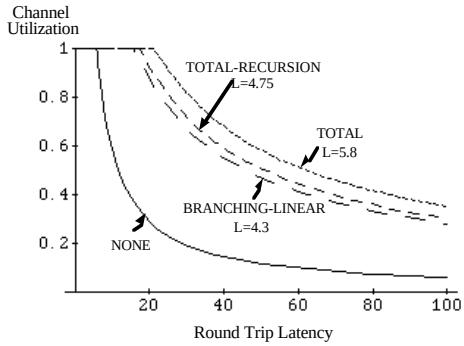


FIGURE 6.13
Median limb length rtt. vs. utilization

A view of the above, for OP_BW=2 (mean, median, for each of 3 implementations)

Examination of the ratio of the μ–Net channel utilizations to the non-anticipation prefetch shows the utilization increases possible (Figs. 6.14, 6.15). *The initial utilization increase is linear, representing the μ–Net anticipation of the entire tree of possible opcode streams. There is a point in the stream when the entire tree cannot be transmitted given the available bandwidth; at this point the utilization increase is logarithmic as latency increases.*

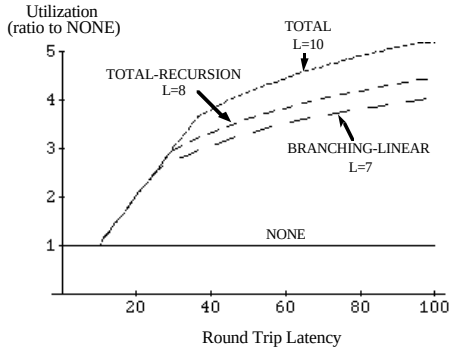


FIGURE 6.14
Mean relative performance

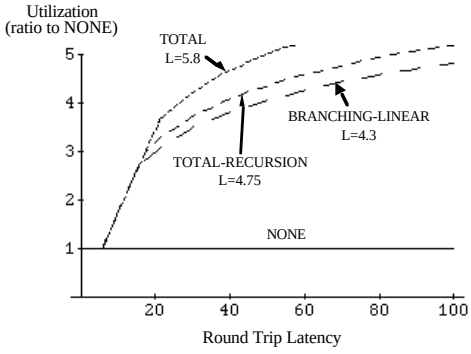


FIGURE 6.15
Median relative performance

Relative performance, vs. non-anticipatory implementation

There may be optimizations to the Code Pump which would further constrain the size of the TreeStack. In some cases, especially those of short branch jumps, the state space collapses back down. If the set of all addresses currently in transit is known, further

messages with those addresses may be omitted. Consider the opcode sequence in Figure 6.16.

```
    -other1-
    conditional branch to X
    -other2-
    -other3-
X:  -other4-
    -other5-
```

FIGURE 6.16
Short forward branch opcode sequence

The PC image, after the execution of the conditional, evolves into two identical but temporally shifted sub-images. One image expresses the branch not taken, the other expresses the branch taken. It would be more efficient for the Code Pump to recognize this shift, sending the opcodes as if the branch were not taken, thus covering all code for both branches. The drawback in this scheme is that the opcodes must be scheduled as if all were executed; they cannot be interleaved as if they were two mutually exclusive opcode sequences, each at the rate of execution, resulting in a higher sent rate (utilizing high bandwidth). As a result, there are times (when the branch jump is taken) when the processor is idle, but this is still often less than the time which would have been incurred by not implementing the branch as a sent message.

For example, if the round trip time is 100 instruction execution times long and the branch jump is less than 100 opcodes long, the above simplification, of sending opcodes as if no branch occurred, remains more effective than waiting for a round trip time before replying with the opcodes. This simplification further reduces the size of the image in the Code Pump, reducing its storage and computational requirements.

A proper analysis of the effective size of the TreeStack structure will require an emulation of the architecture because there are many dependant characteristics of the code stream which affect the results. These include the way in which the size of the TreeStack structure depends on the number of pending branch operations down all possible computational paths, and the ways in which the existence of a code cache at the processor affects the distribution of opcodes requested from memory. An emulation of the TreeStack in the Code Pump was beyond the scope of this dissertation.

6.3. Observations

There are some observations which were discovered through the process of casting the processor/memory communication architecture into the Mirage model. These include a better understanding of the ways in which opcode classes affect the communication channel, and the ways in which the μ–Net solution fails and the reasons why it fails.

6.3.1. Kinds of instructions

Opcodes have been partitioned into classes based on the ways in which they transform the image of the PC in memory. Most opcodes only increment the current PC, and so transform the PC image by moving it through state space in a linear fashion. Other classes of PCs transform the space non-linearly, either by a simple function (old PC → new PC), by a function with memory (old PC → new PC + state memory), or by a set mapping (old PC → set of new PCs).

6.3.1.1. Regular opcodes (OTHER) and JUMPs

Usually the PC image is transformed by a simple mapping function, e.g., by the addition of a constant to the current PC values. For regular opcodes (i.e., OTHER), the constant is implied and fixed, with a value of 1 (i.e., to increment the PC); for JUMPs the constant is contained in a field in the opcode, and is called an *offset* (PC-offset JUMP). In other kinds of JUMPs, the PC values can be completely replaced by the fixed constant in the opcode (direct JUMP).

These opcodes also perform a function in terms of the ways in which they alter the path traversed in state space. Regular opcodes provide the default traversal of the state space, a linear sequence. JUMPs permit discontinuities in the path, and are means of permitting paths to evolve which are not dependant on the natural ordering of the space. Backward JUMPs provide only for code reuse because the code could have been repeated at the location of the JUMP. Forward jumps skip over code which must be otherwise accessible, i.e., they permit use of places in the state space which other paths otherwise ignore. JUMPs permit the efficient use of a linear address space to contain a nonlinear path.

6.3.1.2. BRANCHES

Branch transformations are conditional upon the state of the processor. The PC is incremented by a value which depends on whether the branch is taken; if it is, the increment is contained in the opcode, if not, the increment is fixed and has a value of 1. Branches behave as either a regular opcode or as a JUMP, depending on the processor state. Memory in μ–Net does not model the entire processor state, only its PC value; as a result, BRANCHes induce an ambiguity in the transformation upon the memory's image of the processor's PC.

The processor PC undergoes one of two transformations; here modeled as a set mapping from one PC value to two, so that the size of the PC image doubles as a result. The particular resulting PC remains unknown, but it is one of a finite and explicit set, so the transformation is modeled as an expansion of the image.

Later, when the processor communicates the need for an opcode that is on the resultant path of one of these PC values, the other PC path precedents can be omitted. As memory sees it, the PC image becomes two images, which are collapsed down to a single image after later communication.

The communication which collapses the image can be made explicit, by making the Filter Cache encode and send only branch decisions to the Code Pump, in which case the reduction in the size of the image would be that proscribed by the communication limit (Chapter 2). Instead, the interface is simplified by having the Filter Cache transmit entire addresses (PC values) to the Code Pump, where the decision as to how the image is affected is subsequently determined.

BRANCHes implement a level of context sensitivity, where the actions of the Code Pump are affected by some portion of the state of the processor not modeled. This is a minimal context sensitivity because PC image is partitioned into two resultant PC images as the result of the branch execution. BRANCH context sensitivity derives from the implementation in code of Dijkstra's guarded commands [Di76]; communication stream context sensitivity parallels BRANCHes in Mirage, via *guarded messages*.

6.3.1.3. INDIRECT

Indirect opcodes provide a level of context sensitivity which may be too powerful for most intents. A PC value is transformed by the execution of the indirect instruction into *any* PC value. There is no information in the opcode which restricts this transformation, as occurs in a JUMP or BRANCH (because they contain fixed PC offsets

or fixed new PC values). Whereas a BRANCH is modeled as transforming a PC into one of two resultant PCs, an INDIRECT opcode transforms a PC into the set of *all* PCs.

The net effect of an indirect opcode is to partition the PC image into an infinite number of components, each of which may be the result of its execution. Actually, because the size of a PC is fixed, the size of this set is also fixed, but the partitioning results in a set of SIZEOF(PC) components, which is effectively infinite to the μ–Net model.

In attempting to model the transformation of its PC image by the transmission of an INDIRECT opcode, the Code Pump fails. It cannot sufficiently partition the resulting image (which encompasses the entire PC state space), in order to send messages (opcodes) pertaining to portions of it. Indirect opcodes introduce too much information to model effectively. The Code Pump is required to wait for a message from the Filter Cache, indicating the destination of the indirect opcode. An indirect opcode expands the PC image to completely cover the state space, so the size of this message must be the log of the size of the space (i.e., the size of the PC value), in order to collapse the image down to a single value.

Indirect opcodes permit the PC path to be completely dependant on part of the state of the processor which is not modeled, which means that a round trip time is required in order to obtain that (unmodeled) state information.

These assertions require qualification. As mentioned, there is no information in an INDIRECT opcode, or in the entire opcode stream, which limits the destination of an INDIRECT offset, other than to the raw size of the address (i.e., the entire PC space). In some compilers, INDIRECT address destinations are restricted to explicitly labelled destinations. In this case, μ–Net would suggest that the INDIRECT jump be compiled into a table lookup of existing label destinations, i.e., a fixed JUMP handler. Using such a JUMP dispatch table, the INDIRECT jump’s potential destinations are reduced from infinite (or SIZEOF(PC)) to some small constant. This table also permits the Code Pump to provide anticipation of the fixed branching of the data stream, rather than being thwarted by the (effectively) infinite branching of INDIRECTS.

6.3.1.4. CALL and RETURN

CALLs and RETURNs are, respectively, entry and exit points to recursive state space instances. A CALL behaves like a combination of an entry point to this space (in terms of its effect on the stack in both the Code Pump and at the processor), and a JUMP,

in the way it specifies an address at which to enter this recursive space. Within the recursive space, most opcodes transform the space as usual, not affecting the parent space (i.e., the space of the source of the CALL opcode); a RETURN is the only exception, it being the lone exit point from that space (back to the parent space).

Subroutines, as facilitated by CALLs and RETURNs, provide a structure for code reuse, which does not affect this analyses. They also permit a spatial recursion which is relevant because our modeling of them emulates this recursion in the management of the PC image, via the TreeStack structure.

6.3.2. Other observations

There are a few other observations that can be made. Comparing our method of sender anticipation in the Code Pump vs. the anticipation made by prefetching of branch targets is equivalent to comparing breadth-first search (BFS) vs. depth-first search (DFS) techniques. The BFS method utilizes increased bandwidth requirements of the μ–Net domain, and also avoids the need to backup or cancel the code search, because opcodes are sent as isopotent sets (each level of the BFS).

Also, μ–Net has a problem with pointers. Fortunately, it is not alone in this regard; pointers also inhibit many types of static code analysis as well. Indirect opcodes are code-space pointers, and these opcodes are advantageous only if they outweigh the latency they necessarily induce. Opcodes should be limited to only the required degree of branching of the code stream, i.e., limit indirection to multi-way jumps where possible.

μ–Net indicates the difference between sender-anticipation and prefetch caches. The latter assumes a correlation between past and future code use (via the cache), and anticipation assumes infrequency of some types of control opcodes. μ–Net system uses semantic information about the traversal of the code space, i.e., by actually determining the effect of each opcode on the code space sequencing (i.e., PC values).

6.4. Relation to prior work

μ–Net is a unique system of anticipation, derived from the abstract Mirage model of communication in the presence of latency. Since its invention, the literature has been investigated for comparable processor/memory mechanisms, and very few of these references were applicable.

An analog to the Code Pump was investigated in early microprocessor research, which has since been abandoned (Instruction Issue Logic). Current trends in system architecture research have returned to this area (SPA), but use a restricted form of μ–Net's TreeStack structure for control. Prior research, both past and present, has been developed from empirical evidence alone, whereas μ–Net was derived from the abstract Mirage model.

6.4.1. Instruction Issue Logic (Code Pumping)

'Instruction-issue logic' is the closest precursor to the Code Pump mechanism in μ–Net. One of the first 'instruction-issue logic' implementations is the Fairchild F8 system (1976), another more recent investigation was the Distributed Logic Instruction Issue System (1987). Most of these schemes differ from μ–Net's Code Pump in that multiple branches are not accommodated.

The most recent version of this research is the Sustained Performance Architecture (1991), which is guided by the need to investigate active memory architectures (coined *proactive* therein). All of these versions of implementation can be considered restricted instances of the more general μ–Net concept, however none mentions such an abstract version. There are also ways of extending each of these implementations to accommodate opcodes not currently handled, which would not complicate their design, and were not investigated.

Further, μ–Net is derived from the consequences of applying the Mirage protocol model to the processor-memory interface domain. All these prior designs were created via other means, i.e., engineering issues such as simplified system design (F8), or arbitrary visions of appropriate memory interfaces.

6.4.1.1. Fairchild F8

The Fairchild F8 was a microcomputer system based on a 3850 CPU and 3851 Program Storage Unit (PSU) [Fa76b],[Fa76c],[Os76]. Its goal was to reduce the overall chip count, from 7 in a typical Intel 8080A or Motorola 6800 design to 2, at the expense of partitioning the CPU according to the complexity of its components into a separate ALU and instruction sequencer. Other comparable microprocessor chip sets were partitioned along functional lines. The F8's low chip count helped it lead world sales of CPUs in 1977, when it became a dominant embedded controller in consumer products.

The F8 chip set was complemented by the Mostek 3870, a single chip combination of the 2-chip set which was not extensible, but isolated the CPU/PSU interface from the designer and was thus more accepted in competition with traditional microprocessors.

Although the F8 was first noted as an “offbeat product with a strange set of chips and a ridiculous instruction set”[Fa76c], it is examined here for its CPU/PSU interface, especially in its similarity to μ–Net. The 3870 CPU has no program counter, as in current CPUs; instead, the function of instruction sequencing has been relegated to separate devices on the bus, implemented by instances of the 3851 PSU. The advantages of removing the memory access logic from the CPU are reduced pin count (no address lines needed), and availability of additional real estate on the CPU.

The communication between the CPU and PSU is provided by a set of control lines, ROMC 0-4, and a pair of clock signals. Each PSU has an internal PC, and a factory-preset mask (upper 6 of 16 bits). The PC in each PSU contains the PC that would have occurred in the CPU, at all times. Operations which affect the PC, i.e., load the PC from the data bus, write the PC to the data bus, increment the PC, add the offset on the data bus to the PC, etc. affect all PCs in all PSUs identically. When the CPU, by the control lines, requests an opcode, any PSU whose mask matches the PC’s high bits responds by placing the opcode on the data bus.

This can be interpreted as hardwiring each PSU to respond only to its own mapped address space, even though the entire PC is imaged in every unit. The internal mask acts like an address decoder and tri-state output enable together.

The similarity to the μ–Net design is obvious; each memory unit has its own Code Pump (PSU), which models only increment transformations. Other transformations occur only in direct response to messages from the CPU. The PSUs need only model a single point in state space (i.e., a single PC value) and send a single opcode out, because latency is not a design criterion. Recursion of the state space image is modeled, but only one level of pending recursion; this is sufficient because it was intended for interrupt servicing (where interrupts are not subsequently interruptible, in this design), and it was assumed that further PC storage to accommodate recursion of the normal execution would be provided in software, i.e., in external storage which was not part of the PSU design.

6.4.1.2. Distributed Logic Instruction Issue

The Distributed Logic Instruction Issue System (DLII) [Ha87] implements a Code Pump where conditional branches are modeled, as well as regular (OTHER) opcodes. Only one pending branch is permitted, and opcodes are buffered at the processor, rather than in the communication stream. The design was intended to accommodate access latency induced by bus contention, rather than transmission delay. The system can also be interpreted as a prefetch, where both alternates of a branch are prefetched.

The intent was to move instruction logic to the memory module, as was done in the F8 (on which the idea was based). In contrast to μ–Net, no instruction addresses are transmitted, this being substituted by a bit indicating to which arm of a conditional each opcode belongs (branch taken/not taken). This implements the minimal encoding of message guards because the PC image space is permitted to grow to only two points in space. Consequently, only one bit of reply information from the processor is required to collapse the space to the PC actually used.

DLII partitions opcodes into 5 classes: JUMPs, CALLs, RETURNs, BRANCHes, and regular (OTHER) opcodes. INDIRECT opcodes, and their effect on the instruction stream, were not considered separately. It also recognized the ability to predict the successor to JUMPs and CALLs, and also RETURNs, the latter-most with the addition of a stack in the program storage module (PSM, akin to our Code Pump/program storage). The PSM stack implements the Recursion μ–Net, with one addition.

The PSM also contains two PCs, the set of which is similar to a limited version of our PC image, as maintained by the TreeStack structure in the Code Pump. The instructions accumulate in a dual buffer in the CPU, which executes instructions from one side of the buffer at a time. When a branch is indicated, the CPU switches to the indicated buffer (i.e., either remains or switches), and clears the contents of the alternate, because those opcodes are no longer needed. The PSM is notified of this selection, and the unused PC is cleared and released for reuse.

Subroutine return addresses are contained in the PSM and only the PSM and CPU need know of their existence, so that no other copies of the stack exist. The PSM transmits both the RETURN opcode and the destination address, in effect, translating the opcode into a direct JUMP. This may be a useful modification to the Code Pump design, although it requires modification of the CPU design. Proposed μ–Net designs do not require CPU modification.

The PSM also recognizes short loops, such that repetitions of instruction sequences that are capable of being held completely in the CPU cache are omitted. This is a useful optimization of the Code Pump, but necessitates a more intelligent (or Code Pump/Cache aware) compiler. Again, although there may be benefits, the optimization requires that the compiler writer be familiar with the architecture. μ–Net performs the same functionality and also permits the compiler to be independent of the mechanism (the mechanism is hidden in the abstraction).

Similar to μ–Net, the PSM assumes its memory is read-only code. Opcode loops are permitted to have only one internal branch, or a branch which may jump to the opcode immediately following the loop only; if these conditions are violated, the instruction issue waits. If either instruction stream (of the two possible, emanating from the two possible PC values), encounters a loop start (i.e., set of instructions assumed to remain within the cache), call, return, or subsequent branch, the instruction issue again waits until the opcodes at the CPU are completely consumed.

The restriction that only one return address stack is managed, and that only one pending branch is permitted (and only at the top of that stack), means that here the TreeStack data structure is reduced to a simple stack. The internal entries of the stack contain single PC values, whereas the top of the stack is contained in the pair {PC1,PC2}, i.e., a stack whose top element *only* may contain a tree with one branch *only*.

The analysis of the architecture is similar to that of μ–Net, although indirect opcodes are not considered. Their effect on the architecture is identical to that of loop starts, jumps, etc. in the presence of a pending branch - that the PCU waits until the opcode in question is executed by the CPU before proceeding further. There is no mention that indirect opcodes by their nature require such latency, whereas the latency incurred by other opcodes in this design are the result of the limited implementation of the TreeStack structure.

PSM proponents further acknowledge the speedup that is the result of the parallelism in the anticipation mechanism in the PSU being decoupled from the CPU. Additionally, communication between the PSU and CPU is reduced by as much as half because addresses need not always be sent with the opcodes from the PSU. It is not clear that a complete instance of the TreeStack structure would result in a similar bandwidth savings, because μ–Net's message guards are complete addresses to simplify implementation. The communication per opcode is reduced by half, and the resulting bandwidth (and any additional available bandwidth) is used by the transmission of two

code streams (i.e., two alternative opcodes). This shows a method where the necessary additional bandwidth required is created by the design, whereas μ–Net assumes a further bandwidth surplus exists.

6.4.1.3. The Sustained Performance Architecture

There is an excellent overview of existing instruction sequencing methods which has just recently appeared in the literature [Kr91]. These include methods of prefetching, branch prediction, and cacheing. In the conclusion, the authors allude briefly to their vision of the future of instruction sequencing, where they define *proactive* memory, where "...memory should produce the correct instruction before the execution unit needs it...".

In the event of concurrent potential opcode sequences, "...the program-flow graph can be used during program execution to initiate the prefetch of *all* instruction sequences...". This architecture has been named the "Sustained Performance Architecture." [Do89]

The SPA paper describes the conditions under which opcodes can be predicted by memory, and notes the need for a stack to store return destinations, and also notes that indirect opcodes require round trip communication regardless of any type of prediction. The goal of this architecture, as in μ–Net, is to avoid main memory access latency.

They do not discuss the complexity involved with the maintenance of the PC image in their version of the Code Pump, the Instruction Decode Unit (IDU). They acknowledge the need for a stack structure to store return destinations, but do not note the interaction between pending conditional branches and levels of recursion.

They claim a 30-40% increase in processor performance, for the levels of latency they envision (about 2-8 opcodes), and also claim that sequences of opcodes without control flow changes have an average length of 7 and a median length of 4 (this includes branches not taken). Our measurements differ, but not substantially (average 6, increasing to 9 if jumps, calls, and returns are not considered control flow changes, with respective medians of 4 and 6). They also achieve multiple instruction streams by replication of the IDU; the problem with this design is that an IDU is associated exclusively with a single opcode sequence. Their Program Execution Controller (PEC) is charged with the task of controlling this exclusivity, prefetching the actual instructions from memory. In essence, the PEC is our Code Pump, and each IDU manages a stack in the TreeStack, which is

here restricted to a tree of stacks (i.e., a tree whose elements are stacks). The wiring and control of the IDUs implements a hardwired tree structure.

This is, therefore, a more restricted form of the general TreeStack structure, because it cannot handle branching subsequent to recursion. A branch instruction causes one side of the branching stream anticipation to be relegated to a separate IDU, thus partitioning the anticipation between two IDUs; subsequent rebranching causes the IDUs to organize into a tree structure, as in the TreeStack. This opcode organization is common in OS handler subroutines, a monitor program, or a dispatcher. It is a reasonable tradeoff, but the general TreeStack structure is not acknowledged in the SPA, nor are the reasons for its existence.

Once recursion occurs, a stack is required. In μ–Net the stack structure would be analogously created by adjoining a stack of IDUs, whose interior elements would be inactive. SPA notes the necessity for a stack to manage recursion, and for a tree structure of IDUs to manage branching, but does not indicate where the stack would exist. Were the stack within the IDUs, the occurrence of a CALL would inhibit subsequent branching, because such branch alternates could ‘pop’ at different times. This would cause inconsistencies in the IDU stack. μ–Net explicitly specifies the structure of the mechanism required to handle branching and recursion together.

Further, the PEC is guided by the program call graph, which is loaded at run time. This introduces another level of complexity, especially if two concurrent processors attempt to use the same PEC, or if the call graph of a program were superseded by the call graph of the interrupt handler, or operating system, or other intervening code. μ–Net determines all requisite control information from the instruction stream itself, and can dynamically reallocate resources for multiple pending branches because it is monolithic in design, and manages the data structure centrally.

The SPA appears to be an empirically derived, specific instance of the more general, more abstractly derived μ–Net architecture. This research, although state-of-the-art for architecture design, is presented with no abstract basis. μ–Net was developed concurrently to and without knowledge of the SPA, as the logical consequence of the protocol interaction between the processor and memory and the limitations of such communication as latency grows.

6.4.2. Cache issues

As discussed before, there are similarities between some cache issues and our Code Pump, especially if the Code Pump is viewed as complementary to a cache, with which it should be coupled. Some similarities exist between the anticipatory nature of the Code Pump to that of cache prefetching, as implemented explicitly in hardware, or in software, or as partially effected by widening cache lines.

6.4.2.1. Prediction/prefetching

As was discussed in the general Mirage model, the expansion of state spaces can be further specified by attaching a probability distributions function (PDF) to the expansion. If guarded messages are sent, utilization of the communication channel can be optimized by sending messages which affect the most likely subspace, as indicated by the guard. The limit of this analysis is explicit prediction, i.e., assuming all branches are taken, not taken, or that their behavior is static over time (i.e., if a branch was taken last time, it will be taken this time). The assumptions of probability behavior and extrapolation from a set of events to a future event depends on whether the average is temporal or ensemble, as discussed earlier.

As also mentioned before, ensemble averages predict individual behavior from behavior of a group, i.e., within a single execution of a program, if most branches are not taken then it is likely a given branch is not taken. In branch prediction, this translates to the assumption, given statistics gathered over all executions, that branches in general are not taken, and governs the use of that assumption in the absence of temporal information.

Temporal averages predict future behavior of a branch based on its own past behavior, which may have been measured during prior executions or at some earlier time in the current execution. Temporal average information translates into the assumption that whatever way a branch went in the past, it is very likely to do the same in the future. Tag bits associated with each branch encode the last path through it, and are used by prefetching mechanisms to predict the current most likely path. This information usually encodes the last two execution paths because it would be undesirable to override the dominant probability due to a single prior event of lower probability.

Branch prediction achieves [Le84] 90-95% accuracy, using temporal average information. Indirect jumps can also be predicted, using temporal averages only because

ensemble averages are meaningless. These are less effective, achieving between 50-75% [Wa91] accuracy, but these estimates are completely useless if the prediction fails, because failure does not imply a result address.

6.4.2.2. Wide lines

Wide cache lines have a similar effect to linear prefetching. Consider a line size which accommodates 4 opcodes. A miss on an opcode fetch from the cache generates a request of that opcode and the opcodes that surround it in the line. If the instructions execute linearly, the miss on the first opcode causes the next 3 to be fetched as well.

The normal cache fetch of one opcode causes a prefetch of the next three opcodes. Such a mechanism would be optimal where the prefetched opcode sequence was at least as large as the expected length of a linear sequence of opcodes; this is borne out by empirical results, as the analysis of cache line sizes shows the most effective size between 4-16 opcodes [Sm82], whereas some measurements claim a mean linearity of 6 and a median of 4 [Kr91], and our results indicate a mean of 7 and a median of 4.

Cache line size is related to prefetching, but a prefetching cache is more effective than an equivalent non-prefetching cache with twice the line size. Doubling the line size is beneficial only where opcodes execute in linear order, otherwise the width of the lines causes larger sets of opcodes to be swapped out when a miss occurs. In addition, increasing the cache line size only prefetches within the line (i.e., prefetches the subsequent opcodes from the head of the line), which indicates that a miss penalty is incurred when each new line is loaded. Prefetching can retrieve the next cache line when the last opcode in an existing line is used, removing this penalty.

6.4.2.3. Software

Software prefetching has also been suggested as a means to avoid memory access latency [Ca91]. The system assumes a model where the processor prefetches its own opcodes (as in standard caches and prefetching caches), rather than having them sent ahead by the memory (as in μ–Net). It differs from both regular and prefetching caches in the way in which opcode fetches are guided.

Regular caches reuse code already executed, and prefetching caches retrieve opcodes in linear sequence only. Software prefetching uses compile-time information to anticipate the future needs of the processor, and initiates the prefetching by an explicit

instruction. Pseudo-opcodes in the stream control the prefetch mechanism, and are executed by the cache mechanism, and never reach the processor.

Software prefetching has been suggested as a general scheme for guiding prefetching. A more specific kind of software prefetching was developed for the TI-ASC supercomputer [Wa72]. Its compiler (for FORTRAN) generates *prepare-to-branch* instructions, in order to quicken loop execution.

By comparison, long cache lines are an implied prefetch, but sustain a hit penalty when the first item in a line is accessed, and prefetch only where access proceeds in sequence. Hardware prefetching, as it is commonly implemented, fetches a line whenever the address preceding that line is accessed, i.e., whenever address (i) is accessed in the cache, address (i+1) is tested and fetched if missing. The hardware avoids the hit penalty of long cache lines, but again works only where accesses are sequential.

Software prefetching avoids the need for sequential access as a precondition to successful prefetching. The compiler initiates a fetch in advance of code use by inserting explicit cache load instructions; this works because it has prior knowledge of static code sequences, regardless of their order, at compile time. As a result, an execution cycle is spent communicating an upcoming sequence of address accesses.

The mechanism requires a cache supporting a list of pending access requests, which in our system corresponds to the sequence stored in the round trip latency. This form of prefetching has been implemented only for data accesses, where ‘DO-loop’ structures inform the compiler of data access patterns which can be easily translated into prefetch instructions. Software prefetching can be applied to code prefetching as well, using a static opcode lookahead in the compiler (i.e., peephole optimizer). This would not exceed μ–Net’s performance, as μ–Net can look ahead into the instruction stream at runtime almost as easily as software prefetching can do at compile time, and μ–Net incurs no runtime penalty for prefetching. Software prefetching incurs a penalty of one instruction execution time for each prefetch, in order to communicate the prefetch information which μ–Net extracts from the code sequence at runtime.

6.4.2.4. Prefetching vs. cacheing

Prefetching has been compared to caching, although the two can be complementary, because cacheing assists in reducing latency in accessing instructions already issued, whereas prefetching assists during its first use. Another way to view the comparison is that caches look into the past, whereas prefetching looks into the future.

Cacheing and prefetching have been compared as alternates [Le87]. In the cases measured, prefetching performs at least as well as cacheing, in cases where memory access has high latency.[1] This makes sense because caches cannot remove the first hit penalty, whereas prefetching can work both in first use and reuse cases.

6.4.2.5. Prefetch v.s pre-reply

Prefetching is distinct from the pre-reply proscribed in μ–Net. Prefetching is processor-directed, receiver based anticipation, whereas pre-reply is memory-directed, sender based anticipation. Some versions of prefetching [Le87] perform as detailed a management of future state as the Code Pump, but there are several reasons for our placing the Pump at the memory side of the communication channel.

μ–Net's partitioning permits a reasonable distribution of work. The anticipation mechanism is off-loaded from an already overloaded side of the channel; other research in general protocols (e.g., the Universal Receiver Protocol [Fr89]) suggests that such balanced systems are more effective because neither side of the mechanism is unduly overloaded.

Also, receiver-based anticipation involves twice the managed lookahead of the sender-based equivalent because the sender can collapse half the lookahead into the image. In effect, the receiver would work with a single large tree of possibilities, whereas the sender works only with parts of some branches.

6.4.2.6. Guarded messages

Guarded messages permit multiple streams of opcodes to be issued by the memory, where only one stream is actually used. A similar form of conditional labelling of opcodes occurs in pipelined processing, where "conditional branches can be converted to guarded jumps..." [Hs86], and store instructions are converted to guarded stores, to avoid delaying the store in the pipeline.

These techniques are usually associated with pipeline scheduling, either with additional hardware support or compiler participation. Guarded communication differs

---

[1]Clearly caches outperform prefetching where memory access is restricted by bandwidth, because cacheing reduces memory bandwidth use, whereas prefetching increases memory bandwidth.

from guarded execution, but the principle of using extra power (bandwidth or pipeline stages) to overcome latency is the same.

6.4.3. Other related architectures

Prefetching has been examined in other architectures. The IBM Stretch implements branch prediction, where failed predictions are backed over later on. The IBM 360/91 prefetches both arms of a conditional, but down only 2 branch arms, and neither prefetched arm is executed.

Prefetching down both arms of a conditional branch has been called “branch bypassing” and “multiple prefetching.” [Li88]. Predictions from existing code estimate that branch bypassing benefits vary with the level of bypassing performed [Ri72]. Code execution is speeded by a factor of $\sqrt{j}$, where j pending branches are bypassed, at a cost of 2^j. These performance increases are the result of empirical estimates, based on FORTRAN and CDC-3600 assembler, and are not the direct result of utilizing latency for prefetches. The more primitive languages tested in this research exhibit branching largely as the result of loop statements of explicit conditionals in the source code, rather than being generated by the compiler to express complicated nested structures or data access mechanisms.

6.4.3.1. IBM Stretch (7030) - one of the first prefetch

One of the first computer architectures to implement opcode prefetching was Project Stretch, which resulted in the design of the IBM 7030 computer [Bu62]. At the time (1955), IBM had produced several computers, including the 650, 704, and 705. This project was intended to stretch (thus the name) the capabilities of the existing technology, in order to design a computer with a performance of 100x its immediate predecessor.

The IBM Stretch has several similarities to the methods shown here. It had two prefetch units, one performs operand prefetch up to 6 instructions linearly consequent to the current PC, which are then in various stages of decoding. The other, called the Lookahead Unit, prefetches operands for up to 4 of these opcodes, and provides the required interlocking to maintain execution integrity.

Thus the Lookahead Unit prefetches data, whereas the instruction unit prefetches opcodes. The instruction unit models branch instructions as regular opcodes, and forces a flushing of its cache when this assumption fails (i.e., when the branch is taken, because it

assumes branches are not taken). This system was designed to compensate for the disparity in access time of the memory and execution time of the CPU. The high memory delay was due to the storage technology of the time (2.1 μs core memory, 1.5 μs add time via 10-20 ns gate times), and the propagation time through a series of switches and registers which manage access to components (much like a single bus controller).

The addition of prefetching was examined, varying the amount of prefetch [Bu62]. The result was a marked performance increase out to a prefetch of 8, with further prefetching of little advantage. This result is similar to our analysis, which concluded that prefetching was useful only to the average (?? expected) branch arm length. The performance increase found with prefetch was between 20% and 200%, depending on the benchmark application measured.

6.4.3.2. IBM 360/91 - dual prefetch

The IBM 360/91 [An67] instruction unit attempts to fetch opcodes faster than they are used, so that when branches occur, the gap can be accommodated by emptying the buffer while the branch is resolved. The design goal was a buffer (linear lookahead) of 6 instructions, because memory access is 6x slower than instruction execution; this goal was extended to a 10-instruction prefetch on startup, and the implementation provides for a total of 16 opcodes of prefetch.

Branches in the opcode sequence cause both paths to be prefetched: the not-taken path continues down the lookahead of 16, whereas the branch-taken path is prefetched with a lookahead of 4. Forward branches are handled this way, as are backward branches beyond the scope of the lookahead buffer. Backward branches which refer to existing lookahead entries (i.e., which refer to targets less than 15 opcodes away), signal a ‘loop mode’ in the prefetch unit, and instruction prefetching ceases. The loop mode permits unimpeded use of instructions in the prefetch buffer, providing enhanced performance for some code; this feature occurs in the cache and instruction fetch units of some current microprocessors, including Motorola’s 680x0, (68010 and later), and latter versions of Intel’s 80x86 CPUs as well.

The 360/91 thus implements a Code Pump where regular opcodes are modeled, as well as 1 pending branch. The TreeStack structure consists only of the tree component, whose main trunk (root trunk and major branch) are limited to a total length of 16, and whose minor branch arm is limited to a length of 4. Other opcodes halt the prefetching

because they are not modeled; this includes jump and call instructions which could have been modeled with only an additional adder in the instruction unit, as in μ–Net.

6.4.3.3. Rope multiple prefetch

Pipelined and very long instruction word (VLIW) CPU research has lead to the need to overcome memory latency as well. The Ring Of Prefetch Elements project (ROPE) is aimed at alleviating memory latency and the difficulties in prefetching caused by conditional branches, without the need for complex hardware scheduling [Ka86], [Ka85]. It also provides a new prefetch mechanism which supports multi-way branching, which we discussed in μ–Net as a viable alternative to most uses of indirect opcodes.

ROPE recognizes that “caches offer no speedup for loops larger than the cache size,” and so looks for other ways to alleviate memory latency. They abandoned the μ–Net methods, claiming that the prefetching of all branch destinations is prohibitive in communication costs; we accept that conclusion because we are designing an architecture where the communication bandwidth is very high.

ROPE reduces memory latency and removes the need to flush the pipeline when conditional jumps fail, because each opcode prefetch path is supported by a separate prefetch unit. Control passes cyclically around the ring, where each opcode is fetched in turn by its corresponding unit. Explicit prefetch instructions (similar to those of software prefetching), initiate the fetching of branch targets in two or more prefetch units. When a branch occurs, control can pass either to the next unit in the cycle, or it can jump to the unit fetching the destination of the branch; control is automatically assumed by the appropriate prefetch unit because all scan their condition masks to the datapath, and only one subsequently replies with an opcode.

ROPE handles regular opcodes, and binary branches (multiway as well, with some modifications). Jumps and calls are not handled, although they could have been. There is no local memory to the prefetch element ring because it would have to be distributed among the elements but commonly accessible, so a shared or central stack cannot be managed, and return opcodes cannot be accommodated.

The utility of multiway branching is supported by the measures that code is 15-33% branches, and many optimization algorithms (trace scheduling, and percolation scheduling) tend to cluster these decision points. The ability to evaluate simultaneously several binary branches in a multiway branch can reduce the branch occurrence by as much as the fanout of the multiway, in some cases.

The ROPE research measured an increased execution rate of up to 5x conventional architectures, which is consistent with a 6.3x increase predicted for architectures which prefetch only regular opcodes (Unit Linear μ–Net, see Table 6.2).

Both the ring architecture and cyclic transfer of control would evolve naturally from the μ–Net architecture, if we assume that prefetch requests managed by the Code Pump occurred through a pipelined set of memory access registers. μ–Net provides a mechanism for the management of both multiple pending branches and multiple pending levels of recursion, whereas ROPE provides for only a single pending branch. ROPE also requires compiler cooperation via the insertion of software prefetch instructions, whereas μ–Net is self-managing from existing code.

6.4.3.4. Access / execute architectures

Access/execute architectures decouple instruction fetching from ALU operation; in this way, they are similar to the partitioning exhibited in μ–Net [Sm84], [Be91]. There are various implementations of A/E architectures, most notably the IBM RS/6000, the Intel i860, and other less general purpose systems. There are two aspects to these designs: first, instructions and data are communicated between the instruction unit and the ALU via queues, and second, most utilize multiple functional units, usually complementary integer and floating point units. Many of the performance increases cited result from the combination of parallelism and pipelining which this organization achieves, but we are concerned only with their functional partitioning.

The queue communication between the ALU and instruction fetch unit (IFU) is similar to the buffering provided by the communication channel in μ–Net, although in the A/E mechanism the buffer length varies with load. Due to the restricted communication across the partition, branches were computed by the IFU wherever possible; a side-effect of this decision was the ability to permit the ALU to continue ahead of the IFU, which in turn permits the IFU to reduce opcode fetch delays associated with the branch. In our design, this was an engineered result.

Further, only branch decisions and actual opcodes are sent across the partition, as our design confirms to be required. Some studies also examined the effect of the A/E architecture on memory latency tolerance, but most focus on dedicated compiler issues, rather than generalizations to hardware implementation, as in μ–Net.

6.4.3.5. IBM RS/6000

The IBM RS/6000 System [IB90], [Ba90b], [Oe90] implements dual branch stream lookahead, and provides 1 level of pending recursion in the prefetch processor, called therein the “Branch Processor.” The branches provided by the CPU permit indirect jumps, calls, and conditional branches, as well as instructions as complex as indirect conditional calls; this is a result of a design where a branch can be conditional or not, indirect or not, and provide a return address or not.

The Branch Processor has been designed so that it is logically independent from the rest of the processor, so that all branch functions use and modify registers and condition codes local to it which would facilitate its replication in the Code Pump of μ–Net.

The processor provides some level of instruction prefetch, but there is insufficient information in the published literature to determine its exact extent. The instructions are placed into a 32 entry 2-way TLB, so it appears the lookahead is limited to 16 instructions in each of 2 possible paths, at best.

The RS/6000 implements a TreeStack 2 levels deep, where the first level is a simple stack entry, and the second level permits one pending branch. The prefetch appears to provide a total of 16 outstanding opcodes, cumulative to all levels of the TreeStack. The result is a very limited and restricted implementation of prefetching, which is of limited use in highly latent systems.

6.4.4. Remote Evaluation

Remote evaluation (REV) is a version of remote procedure call (RPC), which examines variations which avoid the conventional RPC tradeoffs of performance vs. generality of interface [St90]. RPC traditionally permits the movement of work to a remote location, for execution there, with collected and returned results. REV also permits the movement of code to the data, which reduces memory bandwidth requirements for some applications, notably data collection and reduction in database systems. This research, in conjunction with file server systems such as NFS, provide justification for the domain chosen for μ–Net, where read-only code is temporally remote from the processor/RAM set.

6.4.5. Multiple alternates

μ–Net (and Mirage, in the 'abstract') and its version of exploring multiple possible states of a remote node differs from conventional forms of the exploration of alternates. Exploring branch alternatives is commonly done depth-first, such that some path in the tree of possible executions is fetched in advance of the decision of which branch is actually desired.

Similarly, the Code Pump advances beyond the known state of the receiver (i.e., last state communicated). The way in which it advances, suggests a breadth-first exploration instead, as a direct result of the description of the stability of the communicating system, and the conditions under which such stability can be ensured (Chapter 2).

Another method of anticipating multiple paths is to instantiate each path uniquely, then remove those which are not used later. The idea, described in [Sm89], is to spawn multiple processes, each exploring a different path, such that results are collected from the process which terminates first, the others being destroyed at that time. In μ–Net, this is analogous to the way in which the Code Pump sends multiple instruction streams, in the hope that one stream will be of use.

The difference between this exploration of alternates and μ–Net's is that [Sm89] relies only on the members of the set being mutually exclusive, whereas μ–Net relies on the entire set covering the space of alternates. This cover-set principle is used in the stability equations, where stability is based on the ability to send messages to the entire space of alternatives (communicability). Once messages have been sent to the entire current state, the Code Pump progresses to the next state (or set of states) and can ignore the possibility of backing up to a previous state. The Code Pump doesn't wait for any single alternate to terminate; it waits for additional bandwidth with which to send further messages, or for state resolution information from the remote state.

6.5. Conclusions

Now that μ–Net has been described (Chapter 5) and measured via μ-Scope (here, in Chapter 6), some conclusions can be discussed. μ–Net was intended to provide a vehicle for the description and elaboration of components of Mirage that were not exemplified in

the discussion of existing protocols (Chapter 4). To that end, issues of communicability, guarded messages, and isopotency were all applicable to μ–Net, and so the purpose was served.

Beyond its use as an example, μ–Net also exhibits the advantage of the Mirage model as a method for reexamination of existing disciples with a new viewpoint, another goal of any model. We have applied for a patent for the designs of μ–Net [To91b], as the result of these investigations. The application of the Mirage model to the domain of processor/memory communication led to the independent discovery of 'proactive' memory, and the description an instance (μ–Net) which generalizes current proactive memory research. μ–Net also indicated the the complementary nature of caches and this proactive memory, because caches handle reuse of code, whereas proactive memory handles first use.

6.5.1. Notes for designers

As a design for implementation, μ–Net provides a design for shared code systems to reduce the effects of latency. μ–Net requires that the executed code be read only (i.e., not self-modifying), and that the bandwidth-delay product of the communication network be an order of magnitude larger than the local memory available at the processor (i.e., workstation). For a 33 Mhz 32-bit RISC processor, and a 1 Gigabit channel, this implies a distance of about 3 city blocks between the workstation and the server, so that the speed of light propagation is 6 blocks. Further, μ–Net reduces the effects of latency, but does not reduce the bandwidth requirements on the code memory; μ–Net requires that the access bandwidth of the code memory is the same as the bandwidth of the communication channel.

μ–Net is a feasible design, given current technology. Preliminary measurements indicate that a substantial gain in speed (8x) is possible, assuming transmission latency dominates execution time by at least two orders of magnitude. The implementation of μ–Net mechanism is reasonable because even a design which models only JUMP, CALL, RETURN, and OTHER opcode types can achieve substantial speedup, with as little as 400 bytes (100 addresses) of additional storage. Extension of this design to handle a limited amount of BRANCH pre-reply would further increase performance, but measurements have not yet been made which allow us to predict the extent of this increase. The limit of the performance has been measured at near 330x.

The limitation of implementation is due to both the space complexity of the TreeStack, and the time complexity of the associative leaf- and node-matching required within the Total implementation. The set of active leaves is known, so the size of the leaf-matching required in Converger can be sufficiently restricted, and a reasonable implementation may be possible. Internal node-matching required for the TreeStack to be collapsed are more difficult to estimate, but the virtue of the design is that a delay in pruning is managed by an overgrowth in the TreeStack, and a subsequent halting in the Diverger. The system is thus self-controlled, and the communicability is constrained by the ability of the implementation to keep up with the communicating entities.

μ–Net indicates that INDIRECT opcodes are an impediment to an anticipation mechanism. Indirect opcodes are very infrequent (0.3%), have a very high penalty (1 round trip latency), and μ-Scope measurements indicate that these opcodes usually implement a table-lookup, and could be replaced by a dispatching procedure that avoids their use. Further, the removal of these opcodes may simplify processor design as well.

6.5.2. Notes for researchers

As a research vehicle, μ–Net demonstrates the utility of the abstract Mirage model. Memory management methods were confirmed by thinking of the μ–Net domain in terms of the Mirage model. μ–Net indicates a proactive memory version of Amdahl's law limiting parallelism speedup, and can also be viewed as another interpretation of the missing class of MISD in Flynn's taxonomy.

6.5.2.1. Memory management

Possible partitions of the state space permit stability, as discussed in Chapter 2. There are three obvious ways in which to partition the state space, and they have architectural analogs which were obvious upon such examination.

The state space of opcodes (i.e., the address space of a system) can be partitioned many ways, but a few are most obvious. The first is according to a finite state space volume, i.e., to partition the space uniformly, regardless of the probability density within that volume. This loosely corresponds to the way in which paged memory models handle the address space. The second corresponds to a partition which separates volumes of equal density, i.e., such that the probabilities (i.e., integral of the PDF within the partition) are uniform among the partitions. There is unfortunately no clear architectural

analog of this partitioning. The last most obvious partitioning is based on semantic information, in a way which is an attempt to emulate equiprobable partitions, under the assumption that the programmer semantically partitions the problem nearly equally. This is analogous to segmented memory, where partitioning of the address space corresponds to the semantic partitioning of the program.

6.5.2.2. Opcode anticipation Amdahl's Law

Communication is limited by interrupts or indirect opcodes which cause arbitrary discontinuities in flow path. This can be considered a communications version of Amdahl's Law, which describes the limit in execution speedup under unlimited bandwidth, space, and lookahead power, in the presence of time delays. It states that indirect instructions must incur a round-trip time cost, whereas any other instruction can be predicted, given sufficient capability. The limitation in speedup is expressed in Equation 6.3.

Equation 6.3: $$SPEEDUP_{\max} = \frac{1}{percent\_indirect}$$

6.5.2.3. Flynn's taxonomy

Finally, it is possible that μ–Net's contributes to Flynn's taxonomy. There have been other claims of discovery of a real instance of the 4th category of the taxonomy, MISD. In some cases, A/E (Access/Execute) architectures are claimed to exhibit MISD characteristics, because many employ multiple ALU units, such that two instructions are executed on different streams of data within the CPU during a single cycle. The claim is that because the data set in the CPU (i.e., one data item per ALU) doesn't change during the execution of two opcodes, this is MISD; by that argument, any MIMD system can be considered MISD where the multiple data elements are considered a single set.

Another proposed MISD contender is that of VLIW (very long instruction word) machines. They operate as if multiple instructions are send in parallel to the processor. Unfortunately, the scheduling of these instructions is static; opcodes are paired for storage in the instruction word at compile time, not dynamically adjusted during execution, as our streams are.

We believe μ–Net is one of the closest attempts to a true MISD architecture, from the memory's point of view. The Code Pump sends multiple instructions to the processor,

where, although only one is executed, the set permits execution at rates which a single instruction stream could not achieve. Our speedups are the result of sending a set of instructions to the processor, so μ–Net relies on the MISD nature of the system for performance.

CHAPTER 7

Conclusions

This dissertation is the first evaluation of Mirage as an abstract model. We have discussed the impetus for the model and described the components of the model. Mirage has been described by its application to an existing protocol to exemplify its abstract components, and has been used to design a new protocol to illustrate components not manifested in existing protocols.

The Mirage model has been useful in refining our questions as to why existing protocols may fail in the gigabit, wide-area domain. Mirage has also presented one possible solution to the predicted failure. Further, the model was used to design a novel processor-memory interface, for which a patent has been applied.

7.1. Review

Before we elaborate on these conclusions, a review of previously presented conclusions may be useful, especially because of the conglomeration of discussions contained herein.

7.1.1. New questions

Mirage began with a discussion based on the question of whether existing protocols would 'fail' in gigabit, wide-area networks. We concluded that existing protocols will exhibit a performance failure in domains where information separation is high, due to their inability to accommodate variability in the communicated data stream.

Existing protocols are largely based on connection establishment and management, whereas we define the protocol as facilitating the communication of the data itself. This leads to our description of the limitation of interaction in the presence of latency, based on the limitations of the channel, and the extent to which the interaction is known to be constrained.

> "What we have here is a failure to communicate" - the Warden, in *Cool Hand Luke*

This discussion led to the formulation of the domain in which the Mirage model is applicable. This domain is based on a set of tenets, listed in Chapter 2, and repeated below.

TENET 1: Communication is logical information synchrony among information separated entities

TENET 2: A protocol is a mechanism for maintaining communication

TENET 3: Information separated entities are separated in time*space, in units of *pending-information*

TENET 4: Bandwidth-delay product is a measure of *information separation*

7.1.2. The model

The Mirage model was described in terms of transformations on state space subsets. This description led to our definitions of ***stability*** and ***communicability***. Communicability in turn led to the development of ***guarded messages*** to partition the state space, and ***isopotency*** as the description of the way in a set of *physical* guarded messages represents a single *logical* message.

Mirage describes how error and latency are conjugates, and that the tradeoff between them is determined by the extent of existing constraints, the variance in behavior of a remote entity, the latency with which that behavior is measured, and the precision to which that behavior is modeled.

7.1.3. Existing protocols

Existing protocols were used to exemplify the components of the abstract Mirage model. Describing the model components of Mirage as they applied to NTP showed the violation of layering in the protocol, because the internal, optional algorithms were required for the description of the state space transformations and partitioning. Some of the constraint conditions were also modified as the result of this work.

Analysis of NTP also indicated that Mirage applied not only to fixed latency variable state systems, but also fixed state variable latency systems. This demonstrated the equivalence between variance in state and variance in latency.

Finally, the measurements of NTP were verified from the Mirage model by using the probability density function (pdf) interpretation of the Mirage constraint equations, where the set-notation version of the temporal transformation is expressed in terms of pdf convolutions.

7.1.4. New protocols

Mirage was also applied to a new domain, that of processor-memory interaction, to exhibit some of its components which existing protocols did not manifest. The result, μ–Net (MicroNet) showed the use of guarded messages, isopotency, and anticipation of the Mirage model. Measurements indicate that the simple branching stream model of Mirage was reasonable for a real instance of a protocol, and that an implementation of μ–Net was feasible.

Comparisons of the anticipation of μ–Net indicate that a version with as little as 400 bytes of storage can reduce the effects of latency as well as a 50K byte cache. μ–Net reduces the penalty of latency using data management, anticipation, and increased bandwidth utilization (both channel and memory bandwidth).

The application of Mirage has resulted in a novel design, for which a patent has been applied. Various levels of implementation were described, corresponding to various

partitions of the state space, as proscribed in the Mirage *communicability* and *stability* criteria.

Furthermore, existing research in anticipatory memory interfaces was extended through the application of Mirage. Finally, we developed a formula describing the speedup limitations, with respect to existing protocols (a Mirage version of Amdahl's Law).

7.2. Evaluation

The description of Mirage and the applications to which Mirage was applied were presented here in chronological order. This was an exercise in the creation of an abstract model to describe a phenomenon, although the phenomenon was expected and not observed. We predict that the performance failure of existing protocols will be due to an increase in information separation. Under this assumption, the Mirage model is an effective model, both to describe the failure and to indicate methods to avoid it.

This research differs from many other gigabit protocol analyses because it is based on initial principles of communication and interaction, rather than as an extension to existing protocol instances. Mirage shows that layering is detrimental to high speed protocols, not only on performance grounds, but because it is semantically incompatible with state space partitioning required for communicability and stability constraints.

7.2.1. And the answer is...

We began this discourse with a set of 6 questions, discussed in Chapter 1, and repeated, and addressed below.

1) Existing protocols show substantial drops in channel utilization in domains with high bit-latency.

The performance failure in existing protocols is due to their inability to anticipate sufficient information to occupy the round trip latency. The drop in channel utilization is the result of this inability; protocols that are able to anticipate can surpass the performance of existing protocols. The limitation of anticipation protocols is the result of an unmanageable expansion in the state of the receiver, and thus represents a constraint of the communication itself, rather than the protocol mechanism chosen.

2) The advantages to modeling the endpoints of the link, rather than the channel itself.

If the channel is modeled, as in Shannon's communication model [Sh63], transmission errors can be accommodated (i.e., corrected). If the endpoints are modeled (Mirage), latency can be accommodated (i.e., compensated via anticipation).

3) Why we conclude that the sender should anticipate the receiver.

Sender anticipation is the consequence of partitioning the state space of the sender's perception of the receiver. Communicability is possible when the partition is efficient, and when that partition is used by the sender to maintain stability.

4) How this results in a tradeoff between error and bit-latency.

The tradeoff between error and latency is a tradeoff between communicability and stability. If a larger state space (i.e., larger error) is considered the stable set, then more latency can be tolerated. Reduction of errors is facilitated by constriction of the stability state, corresponding to a lower bit-latency over which state expansion must be compensated.

5) Why achieving increased channel utilization necessitates avoiding layered protocols, i.e., why we need to look inside packets.

If the partitioning cannot be determined, latency cannot be tolerated. Layering prohibits the efficient partitioning of the state space via semantic information of the temporal transformation expansion. In effect, layering clouds the description of the expansion of the perception of a remote state; if the expansion is hidden, it cannot be predictably anticipated or managed.

6) There is a limit to how well we can get around things, which is a function of:

a) variability in the receiver state
b) bit-latency
c) power of the sender to accommodate this variability
d) ability of the channel to accommodate this variability

The limit to "how well we can get around things" is communicability, which is a measure of the extent to which stability can be maintained. Communicability decreases with bit-latency increases or with increases in receiver variance, because either permits the perception of the remote state to expand more rapidly. The power of sender accommodation is determined by the degree to which an efficient partition of this

perception can be determined. Channel accommodation is determined by bandwidth, in its ability to transfer the entire isopotent set in the latency given.

7.2.2. Is Mirage useful?

Mirage provides a model for understanding the effects of latency on communication. It differs substantially from existing models, and was useful both in the description of an existing protocol and in the design of a new protocol. Mirage reduces to conventional models where information separation approaches zero, because it is an extension of finite state models into a state space subset model.

> "It is better to debate a question
> without settling it
> than to settle a question
> without debating it."
> - Joseph Joubert, 1754-1824,
> in *In Search of Schrödinger's Cat [Gr84]*

The most important result of this work is the questions that were asked. Mirage is a model in which the effects of latency can be considered, and in which performance failures of existing protocols can be considered. Issues of layering, stability, and the limitation of communication in the presence of high bandwidth-delay products can be discussed though the use of the Mirage model.

7.3. Future Directions

This dissertation presented the Mirage model, and provided some examples of its use in protocol analysis and design. The model can be further developed as an abstract medium for protocol science. Mirage can also be applied to other protocols, to analyze their effectiveness or examine their limitations in the presence of latency. The protocol developed as part of this research, μ-Net, can also be further investigated, and an implementation developed.

7.3.1. Abstract studies

Mirage was described as an abstract model for latency. The Mirage model concept transcends the actual model description; it is the incorporation of imprecision and

temporal constraints to existing models. The description in this dissertation was presented as an extension to existing finite state machine models (and elaborated in Appendix E), but can be described in terms of other existing models as well. One such example uses Petri Nets as the corresponding existing model (Appendix F). The Mirage model can also be interpreted as an extension of Shannon's communication model, as a temporal extension to that work. Preliminary description of this interpretation appears in Appendix A.

Mirage was developed by applying intuition from analogs in physics, notably particle interactions, to communication issues. The correlation between Mirage and physics may also be of more formal interest. A discussion of some analogs used in the development of the model appears in Appendix B.

The TreeStack is another abstract component of Mirage which may have more general application. Further investigation of this data structure, and whether it exists or is useful in other domains, may prove useful.

7.3.2. Protocol studies (analysis)

Mirage can be used to analyze protocols that are more intricate than NTP. Current protocol research is focusing on issues of flow control, both anticipatory and reactive. Application of Mirage to flow protocols may assist in this analysis.

7.3.3. Implementation studies (design)

Mirage can also be used to design other new protocols, as it was used herein to develop μ-Net. Future work on μ-Net includes emulation to measure the feasibility of a Total anticipatory design, to determine the space requirements for a given latency. An implementation of μ-Net, either in software or directly in hardware, would also be useful in further analysis of the benefits of this protocol.

CHAPTER 8

Bibliography

[Ag83] Aggarwal, S. and Kurshan, R.P., "Modeling Elapsed Time in Protocol Specification." In *Protocol Specification, Testing, and Verification.* North-Holland, Rudin, H. and West, C.H., (Eds), pp. 51-62, 1983.

[An67] Anderson, D.W., Sparacio, F.J., and Tomasulo, R.M., "The IBM System/360 Model 91 Machine Philosophy and Instruction-Handling." *IBM Journal of Research and Development* (Jan. 1967), pp. 8-24.

[An88] Anderson, D.P., "Automated protocol implementation with RTAG." *IEEE Transactions on Software Engineering V. SE-14*, N.3 (Mar. 1988), pp. 291-300.

[As56] Ashby, W. Ross, *An Introduction to Cybernetics*, Methuen, London, (1956).

[Ba90a] Balraj, Timothy S. and Yemini, Yechiam, "PROMPT - A Destination Oriented Protocol for High Speed Networks." In *Participant's Proceedings, International Workshop on Protocols for High-Speed Networks*, IFIP WG 6.1/WG 6.4, Palo Alto, CA, Nov. 1990.

[Ba90b] Bakoglu, H.B. and Whiteside, T., "IBM RISC System/6000 Hardware Overview." *IBM Journal of Research and Development* (1990).

[Be87] Bertsekas, Dimitri and Gallager, Robert, *Data Networks,* Prentice-Hall (1987).

[Be91] Benitez, Manuel E. and Davidson, Jack W., "Code Generation for Streaming: an Access/Execute Mechanism." In *ASPLOS-IV*, ACM, 1991, pp. 132-141.

[Bo78] Bochmann, Gregor V., "Finite State Description of Communication Protocols." *Computer Networks*, N.2 (1978), pp. 361-372, Pub. by North-Holland.

[Bu62] Buchholz, Werner, (Ed), *Planning a Computer System: Project Stretch,* McGraw-Hill (1962), pp. 229-248.

[Ca70] Carr, C.S., Crocker, S.D., and Cerf, Vint G., "HOST-HOST Communication Protocol in the ARPA Network." In *Spring Joint Computer Conference*, AFIPS, 1970, pp. 589-597.

[Ca91] Callahan, David, Kennedy, Ken, and Porterfield, Allan, "Software Prefetching." In *ASPLOS-IV*, ACM, Also: SIGARCH Computer Architecture News, V. 19, N. 2; SIGOPS Operating Systems Review, V.25 Special Issue; SIGPLAN Sigplan Notices, V. 26, N. 4, Apr. 1991, pp. 40-52.

[Ch81] Choi, Tat Y. and Miller, Raymond E., "Protocol Analysis and Synthesis by Structured Partitions." *Computer Networks*, N.6 (1981), pp. 367-381, Pub. by North-Holland.

[Ch86] Cheriton, David R., "VMTP: A Transport Protocol for the Next Generation of Communication Systems." In *Communications Architectures and Protocols*, ACM Sigcomm, ACM Press, Stowe, VT, Also Computer Communication Review, V. 16, N. 3, Aug. 1986, pp. 406-415.

[Ch88a] Chesson, Greg, Eich, Brendan, Schryver, Vernon, Cherenson, Andrew, and Whaley, Al, XTP Protocol Definition Revision 3.0. Tech. Rept. Protocol Engines, Inc., 1900 State Street, Suite D, Santa Barbara, CA 93101, Jan., 1988.

[Ch88b] Cheriton, David, VMTP: Versatile Message Transaction Protocol. Tech. Rept. RFC-1045, DARPA Network Working Group Report, Stanford University, Feb., 1988.

[Ch89] Cheriton, David R. and Williamson, Carey L., "VMTP as the Transport Layer for High-Performance Distributed Systems." *IEEE Communications Magazine* (Jun. 1989), pp. 37-44.

[Ch91] Cheriton, David, *Dissemination-Oriented Communication Systems*. Mar. 1991, Presented at the DARPA Networking Principal Investigator's Meeting, Monterey CA.

[Cl87] Clark, David D., Lambert, Mark L., and Zhang, Lixia, NetBlt: A Bulk Data Transfer Protocol. Tech. Rept. RFC-998, DARPA Network Working Group Report, MIT, Mar. 1987.

[Cl89] Clark, D., Jacobson, V., Romkey, J., and Salwen, H., "An analysis of TCP processing overhead." *IEEE Communications Magazine V. 27* (Jun. 1989), pp. 23-29.

[Co91a] Comer, Douglas E., *Internetworking with TCP/IP - Principles, Protocols, and Architecture,* Prentice-Hall, Englewood Cliffs, NJ, Vol. 1 (1991).

[Co91b] Comer, Douglas E., *Internetworking with TCP/IP - Principles, Protocols, and Architecture,* Prentice-Hall: Englewood Cliffs, NJ, Vol. I (1991), pp. 199.

[Cr89] Cristian, Flaviu, "A Probabilistic Approach to Distributed Clock Synchronization." In *Ninth IEEE International Conference on Distributed Computing Systems*, IEEE, Jun. 1989, pp. 288-296.

[De73] DeWitt, Bryce and Graham, Neill, (Eds), *The Many-Worlds Interpretation of Quantum Mechanics*, Princeton University Press, Princeton, NJ (1973).

[Di76] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, NJ (1976).

[Do77] Doyle, Jon, Truth Maintenance Systems for Problem Solving. Tech. Rept. MIT AI Lab TR-419, MIT, Cambridge, MA, Sept., 1977.

[Do79] Doyle, Jon, "A Truth Maintenance System." *Artificial Intelligence V. 12,* N.3 (1979).

[Do89] Dollas, Apostolos and Krick, Robert F., "The Case for the Sustained Performance Computer Architecture." *Computer Architecture News V. 17,* N.6 (Dec. 1989), pp. 129-136.

[Do90] Doeringer, Willibald A., Dykeman, Doug, Kaiserswerth, Matthias, Meister, Bernd Werner, Rudin, Harry, and Williamson, Robin, "A Survey of Light-Weight Transport Protocols for High-Speed Networks." *IEEE Transactions on Communications V. 38,* N.11 (Nov. 1990), pp. 2025-2039.

[Fa76a] Farber, David J. and Pickens, John R., "The Overseer: A Powerful Communications Attribute for Debugging and Security in Thin-Wire Connected Control Structures." In *IEEE International Communications Conference,* IEEE, 1976.

[Fa76b] *F8 Guide to Programming*, Fairchild Camera and Instrument Corporation, Mountain View, CA, 1976.

[Fa76c] *F8 User's Guide*, Fairchild Camera and Instrument Corporation, Mountain View, CA, 1976.

[Fe90a] Feldmeier, David C., "Multiplexing Issues in Communication System Design." In *Communications Architectures and Protocols,* ACM Sigcomm, ACM Press, Also Computer Communications Review, V. 20, N. 4, Sept. 1990, pp. 209-219.

[Fe90b] Ferrari, Domenico and Verma, Dinesh C., "Real-Time Communication in a Packet-Switching Network." In *Participant's Proceedings, International Workshop on Protocols for High-Speed Networks,* IFIP WG 6.1/WG 6.4, Palo Alto, CA, Nov. 1990.

[Fe90c] Feldmeier, David C. and Biersack, Ernst W., "Comparison of Error Control Protocols for High Bandwidth-Delay Product Networks." In *Participant's Proceedings, International Workshop on Protocols for High-Speed Networks,* IFIP WG 6.1/WG 6.4, Palo Alto, CA, Nov. 1990.

[Fr89] Fraser, A.G., "The Universal Receiver Protocol." In *Protocols for High-Speed Networks*, Rudin, Harry and Williamson, Robin, (Eds), IFIP, North-Holland, 1989, pp. 19-25.

[Ga65] Gamow, George, *Mr. Tompkins in Paperback,* Cambridge University Press, NY (1965).

[Gi79] Gifford, D.K., "Weighted Voting for Replicated Data." In *Symposium on Operating System Principles,* ACM Sigops, Dec. 1979, pp. 150-159.

[Go] Gouda, M.G., Sabnani, K.K., and Netravali, A.N., *Self-stabilizing and Correctness Properties of a New Class of High Speed Transport Protocols*. (To appear).

[Go86] Gove, Philip Babcock, (Ed), *Webster's Third New International Dictionary of the English Language, Unabridged,* Merriam-Webster, Springfield, MA (1986).

[Go88] Gotzhein, Reinhard, "Knowledge-oriented consideration of communication protocols." In *Protocol Specification, Testing, and Verification*. North-Holland, Aggarwal, S. and Sabnani, K., (Eds), pp. 295-306, 1988.

[Go90] Golestani, S. Jamaloddin, "A Stop-and-Go Queuing Framework for Congestion Management." In *Communications Architectures and Protocols,* ACM Sigcomm, ACM Press, Also Computer Communication Review, V. 20, N. 4, Sept. 1990, pp. 8-18.

[Gr84] Gribbin, John, *In Search of Schrödinger's Cat,* Bantam Books, Toronto (1984).

[Gr87] Graham, William R., "Research and Development Strategy for High Performance Computing." In *Report of the Executive Office of the President*. Office of Science and Technology Policy, Nov., 1987.

[Ha28] Hartley, R.V.L., "Transmission of Information." *Bell System Technical Journal V. 7* (1928), pp. 535-563.

[Ha84] Halpern, Joseph Y. and Moses, Yoram, "Knowledge and Common Knowledge in a Distributed Environment." In *Symposium on Principles of Distributed Computing,* ACM Sigops-Sigact, ACM Press, Aug. 1984.

[Ha87] Halang, Wolfgang A., "A Distributed Logic Program Instruction Prefetching Scheme." *Microprocessing and Microprogramming V. 19* (1987), pp. 407-415.

[Ha88a] Halsall, Fred, *Data Communications, Computer Networks, and OSI,* Addison Wesley, Second (1988).

[Ha88b] Hawking, Stephen W., *A Breif History of Time,* Bantam Books, Toronto (1988).

[Ha91] Halliwell, Jonathan J., "Quantum Cosmology and the Creation of the Universe." *Scientific American* (Dec. 1991), pp. 28-35.

[He90] Hennessy, John L. and Patterson, David A., *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann, San Mateo, CA (1990).

[Hi56] Hill, T.L., *Statistical Mechanics,* McGraw-Hill, NJ (1956).

[Ho78] Hoare, C.A.R., "Communicating Sequential Processes." *Communications of the ACM V. 21,* N.8 (1978).

[Hs86] Hsu, Peter Y.T. and Davidson, Edward S., "Highly Concurrent Scalar Processing." *IEEE Computer Architecture* (1986), pp. 386-395.

[IB90] *Assembler Language Reference for IBM AIX Version 3 for RISC System/6000,* IBM Corporation, Austin, TX, 1990.

[II87] III, Jacob J. Wolf and Ghosh, Biswadip, "Modeling Very Large Area Networks (VLAN) Using and Information Flow Approach." In *Symposium on the Simulation of Computer Networks,* IEEE Computer Society, IEEE Computer Society Press, Aug. 1987, pp. 36-44.

[IS88a] ISO, LOTOS - a formal description technique based on the temporal ordering of observational behavior. Tech. Rept. ISO-8807, International Organization for Standardization, 1988.

[IS88b] ISO, Estelle - a formal description technique based on an extended state transition model. Tech. Rept. ISO-8807, International Organization for Standardization, 1988.

[Ja55] Jacobson, Homer, "Information, Reproduction and the Origin of Life." *American Scientist V. 43,* N.1 (Jan. 1955), pp. 119-127.

[Ja84] Jay, Frank, (Ed), *IEEE Standard Dictionary of Electrical and Electronics Terms,* IEEE Inc., NY, ANSI/IEEE Standard 100-1984 (1984).

[Ja88a] Jacobson, V. and Braden, R., TCP Extensions for Long-Delay Paths. Tech. Rept. RFC-1072, DARPA Network Working Group Report, Lawrence Berkeley Labs and Information Sciences Institute, CA, Oct., 1988.

[Ja88b] Jacobson, Van, "Congestion Avoidance and Control." *ACM Computer Communication Review* (Oct. 1988), pp. 314-329.

[Ja89] Jain, Raj, "A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks." *ACM Computer Communication Review V. 19,* N.5 (Oct. 1989), pp. 56-71.

[Ja90] Jain, Raj, Myths About Congestion Management in High-Speed Networks. Tech. Rept. DEC-TR-726, Digital Equipment Corp., Littleton, MA, Oct. 1990.

[Je85] Jefferson, David R., "Virtual Time." *ACM Transactions on Programming Languages and Systems V. 7,* N.3 (Jul. 1985), pp. 404-425.

[Ka85] Karplus, Kevin and Nicolau, Alexandru, "Efficient Hardware for Multi-way Jumps and Pre-fetches." *IEEE Micro V. 18* (1985), pp. 11-18.

[Ka86] Karplus, Kevin and Nicolau, Alexandru, "Getting High Performance with Slow Memory." In *31st IEEE Computer Society International Conference,* IEEE, 1986, pp. 248-253.

[Ka91] Katevenis, Manolis and Tzartzanis, Nestoras, "Reducing the Branch Penalty by Rearranging Instructions in Double-Width Memory." In *ASPLOS-IV,* ACM, 1991, pp. 15-27.

[Kl75] Kleinrock, Leonard, *Queueing Systems: Theory,* Wiley, NY, Vol. 1 (1975).

[Ko90] Ko, Keng-Tai, Mishra, Partho P., and Tripathi, Satish K., "Predictive Congestion Control in High-Speed Wide-Area Networks." In *Participant's Proceedings, International Workshop on Protocols for High-Speed Networks,* IFIP WG 6.1/WG 6.4, Palo Alto, CA, Nov. 1990.

[Kr85] Kritzinger, Pieter S., "Analyzing the Time Efficiency of a Communication Protocol." In *Protocol Specification, Testing, and Verification.* North-Holland, Yemeni, Y., Strom, R., and Yemeni, S., (Eds), pp. 527-539, 1985.

[Kr91] Krick, Robert F. and Dollas, Apostolos, "The Evolution of Instruction Sequencing." *IEEE Computer* (Apr. 1991), pp. 5-15.

[La82] Lam, Simon S. and Shankar, A. Udaya, "An Illustration of Protocol Projections." In *Protocol Specification, Testing, and Verification.* North-Holland, Sunshine, C., (Ed), pp. 343-360, 1982.

[La86] Lam, Simon S., "Protocol Conversion – Correctness Problems." In *Communications Architectures and Protocols,* ACM Sigcomm, ACM Press, Stowe, VT, Also Computer Communication Review, V. 16, N. 3, Aug. 1986, pp. 19-29.

[La90] Larus, James R., Abstract Execution: A Technique for Efficiently Tracing Programs. Tech. Rept. #912, Computer Sciences Technical Report, University of Wisconsin-Madison, Madison, WI, Feb. 1990.

[Le84] Lee, Johnny K.F. and Smith, Alan Jay, "Branch Prediction Strategies and Branch Target Buffer Design." *IEEE Computer V. 17,* N.1 (Jan. 1984), pp. 6-22.

[Le87] Lee, Roland L., Lew, Pen-Chung, and Lawrie, Duncan H., "Data Prefetching in Shared Memory Multiprocessors." In *International Conference on Parallel Processing,* IEEE, 1987, pp. 28-31.

[Li87] Lin, FJ., Chu, P.M., and Liu, M.T., "Protocol Verification Using Reachability Analysis: The State Space Explosion Problem and Relief Strategies." In *Communications Architectures and Protocols,* ACM Sigcomm, ACM Press, 1987, pp. 126-135.

[Li88] Lilja, David J., "Reducing the Branch Penalty in Pipelined Processors." *IEEE Computer* (Jul. 1988), pp. 47-55.

[Ma84] MacGregor, Doug, Mothersole, Dave, and Moyer, Bill, "The Motorola MC68020." *IEEE Micro V. 4,* N.4 (Aug. 1984), pp. 101-118.

[Ma85] Marzullo, Keith and Owicki, Susan, "Maintaining Time in a Distributed System." *ACM Operating Systems Review V. 19,* N.3 (Jul. 1985), pp. 44-54.

[Me76] Merlin, Phillip M. and Farber, David J., "On the Recoverability of Communication Protocols." In *IEEE International Communications Conference,* IEEE, 1976, pp. 21-26.

[Mi80] Milner, R.E., *A Calculus of Communicating Systems,* Springer-Verlag, Berlin, Germany (1980).

[Mi85] Mills, David L., Network Time Protocol. Tech. Rept. RFC-958, DARPA Network Working Group Report, M/A-COM Linkabit, Sept., 1985.

[Mi88] Mills, David L., Network Time Protocol (Version 1), Specification and Implementation. Tech. Rept. RFC-1059, DARPA Network Working Group Report, University of Delaware, Jul., 1988.

[Mi89a] Mills, David L., Internet time synchronization. Tech. Rept. RFC-1129, DARPA Network Working Group, University of Delaware, Oct., 1989.

[Mi89b] Mills, David L., Network Time Protocol (Version 2), Specification and Implementation. Tech. Rept. RFC-1119, DARPA Network Working Group Report, University of Delaware, Sept., 1989.

[Mi90a] Mills, David L., "Internet Architecture Workshop: Future of the Internet System Architecture and TCP/IP Protocols." *ACM Computer Communication Review V. 20,* N.1 (Jan. 1990), pp. 6-17, Report of workshop chair..

[Mi90b] Mills, David L., Network Time Protocol (Version 3), Specification and Implementation. Tech. Rept. RFC-xxxx, DARPA Network Working Group Report, University of Delaware, Jul., 1990.

[Mo82] Moli, Gesualdo Le, Palazzo, Sergio, and Andreoni, Gaetano, "A Model of Entity for the Definition of Protocols, Services, and Interfaces." In *Protocol Specification, Testing, and Verification*. North-Holland, Sunshine, C., (Ed), pp. 249-258, 1982.

[Mo89] *MC68040 User's Manual,* Motorola, 1989.

[Ne90] Netravali, Arun N., Roome, W.D., and Sabnani, K., "Design and Implementation of a High-Speed Transport Protocol." *IEEE Transactions on Communications V. 38,* N.11 (Nov. 1990), pp. 2010-2024.

[Ni91] Nicholson, A., Golio, J., Borman, D.A., Young, J., and Roiger, W., "High Speed Networking at Cray Research." *ACM Computer Communication Review V. 21,* N.1 (Jan. 1991), pp. 99-110.

[Oe90] Oehler, Richard and Groves, Randy D., "IBM RISC System/6000 Processor Architecture." *IBM Journal of Research and Development V. 34,* N.1 (Jan. 1990), pp. 23-36.

[Ok86] Okumra, Kaoru, "A Formal Protocol Conversion Method." In *Communications Architectures and Protocols,* ACM Sigcomm, ACM Press, Stowe, VT, Also Computer Communication Review, V. 16, N. 3, Aug. 1986, pp. 30-37.

[Os76] Osborne, Adam, *An Introduction to Microcomputers: Some Real Products,* Adam Osborne and Associates: Berkeley, CA, Vol. 2 (1976), Chapter 2.

[Pa90a] Partridge, Craig, "How Slow is One Gigabit Per Second?." *ACM Computer Communication Review V. 20,* N.1 (Jan. 1990), pp. 44-53.

[Pa90b] Partridge, Craig, Workshop Report: Internet Research Steering Group Workshop on Very-High-Speed Networks. Tech. Rept. RFC-1152, DARPA Network Working Group Report, BBN Systems and Technologies, Apr., 1990.

[Pa91] Partridge, Craig, *Late Binding (working title),* Ph.D. dissertation, in progress, Harvard University, 1991.

[Pe62] Petri, C.A., *Communication with automata,* Ph.D. dissertation, Darmstadt Institute of Technology, Bonn, Germany, 1962.

[Pe77] Peterson, J.L., "Petri Nets." *ACM Computing Surveys V. 9,* N.3 (Sept. 1977), pp. 223-252.

[Pe80] Pease, M., Shostak, R., and Lamport, L., "Reaching Agreement in the Presence of Faults." *Journal of the ACM V. 27,* N.2 (1980), pp. 228-234.

[Pe89] Penrose, Roger, *The Emperor's New Mind,* Oxford University Press, Oxford (1989).

[Pl80] Plotkin, G.D., "Dijkstra's Predicate Transformers and Smyth's Powerdomains." In *Abstract Software Specifications*. LNCS 86, Bjørner, D., (Ed), 1980.

[Po80] Postel, Jon, User Datagram Protocol. Tech. Rept. RFC-768, DARPA Network Working Group Report, USC Information Sciences Institute, Aug., 1980.

[Po81a] Postel, Jon, Transmission Control Protocol. Tech. Rept. RFC-793, DARPA Network Working Group Report, Information Sciences Institute, CA, Sept., 1981.

[Po81b] Postel, Jon, Internet Protocol - DARPA Internet Program Protocol Specification. Tech. Rept. RFC-791, DARPA Network Working Group Report, USC Information Sciences Institute, Sept., 1981.

[Po81c] Postel, Jon, Internet Control Message Protocol. Tech. Rept. RFC-792, DARPA Network Working Group Report, USC Information Sciences Institute, 1981.

[Po83] Postel, Jon, Time Protocol. Tech. Rept. RFC-868, DARPA Network Working Group Report, USC Information Sciences Institute, May, 1983.

[Po88] Postel, Jon, "Summary of National Communcation Initiative Issues." In *Report on the Advanced Computer Communication Workshop*. Information Sciences Institute, pp. 43-44, CAMar., 1988.

[Ra78] Randell, R., Lee, P.A., and Treleaven, P.C., "Reliability Issues in Computing System Design." *ACM Computing Surveys V. 10,* N.2 (Jun. 1978), pp. 123-166.

[Ra87] Raveche, Harold J., "A National Computing Initiative." In *SIAM Workshop Report*. SIAM, Phila, PA1987.

[Ra88] Ramakrishnan, K.K. and Jain, Raj, "A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with a Connectionless Network Layer." In *Communications Architectures and Protocols,* ACM Sigcomm, ACM Press, 1988, pp. 303-313.

[Re91] Renteln, Paul, "Quantum Gravity." *American Scientist V. 79,* N.6 (Nov.-Dec. 1991), pp. 508-527.

[Ri72] Riseman, Edward M. and Foster, Caxton C., "The Inhibition of Potential Parallelism by Conditional Jumps." *IEEE Transactions on Computers* (Dec. 1972), pp. 1405-1411.

[Ro90] Rose, Marshall T., *The Open Book: A Practical Perspective on OSI,* Prentice Hall (1990).

[Sa89] Sabnani, K. and Netravali, A., "A High-Speed Transport Protocol for Datagram / Virtual Circuit Networks." In *Communications Architectures and Protocols*, ACM Sigcomm, ACM Press, Also Computer Communications Review, V. 19, N. 4, Sept. 1989, pp. 146-157.

[Sa90] Sanders, Robert M. and Weaver, Alfred C., "The Xpress Transfer Protocol (XTP) - A Tutorial." *ACM Computer Communication Review V. 20,* N.5 (Oct. 1990), pp. 67-80.

[Sc82a] Schwartz, R.L. and Melliar-Smith, P.M., "From State Machines to Temporal Logic: Specification Methods for Protocol Standards." In *Protocol Specification, Testing, and Verification*. North-Holland, Sunshine, C., (Ed), pp. 3-19, 1982.

[Sc82b] Schwartz, Richard L. and Melliar-Smith, P.M., "From State Machines to Temporal Logic: Specification Methods for Protocol Standards." In *Protocol Specification, Testing, and Verification*. North-Holland, Sunshine, C., (Ed), pp. 3-19, 1982.

[Sh63] Shannon, Claude E. and Weaver, Warren, *The Mathematical Theory of Communication,* University of Illinois Press, Urbana, IL (1963).

[Sh82] Shankar, A. Udaya and Lam, Simon S., "On Time-Dependent Communication Protocols and Their Projections." In *Protocol Specification, Testing, and Verification*. North-Holland, Sunshine, C., (Ed), pp. 215-233, 1982.

[Sh85] Shankar, A. Udaya and Lam, Simon S., "Specification and Verification of Time-Dependent Communication Protocols." In *Protocol Specification, Testing, and Verification*. North-Holland, Yemeni, Y., Strom, R., and Yemeni, S., (Eds), pp. 215-226, 1985.

[Sh88] Shimony, Abner, "The Reality of the Quantum World." *Scientific American* (Jan. 1988), pp. 46-53.

[Si82] Simon, Gerald A. and Kaufman, David J., "An Extended Finite State Machine Approach to Protocol Specification." In *Protocol Specification, Testing, and Verification*. North-Holland, Sunshine, C., (Ed), pp. 113-133, 1982.

[Si89] Simpson, J.A. and Weiner, E.S.C., (Eds), *The Oxford English Dictionary*, Clarendon Press, Oxford, Vol. III, 2nd (1989).

[Si91] Sidhu, Deepinder, Chung, Anthony, and Blumer, Thomas P., "Experience with Formal Methods in Protocol Development." *ACM Computer Communication Review V. 21*, N.2 (Apr. 1991), pp. 81-101.

[Sm82] Smith, Alan Jay, "Cache Memories." *ACM Computing Surveys V. 14*, N.3 (Sept. 1982), pp. 473-530.

[Sm84] Smith, James E., "Decoupled Access/Execute Computer Architectures." *ACM Transactions on Computer Systems V. 2*, N.4 (Nov. 1984), pp. 289-308.

[Sm89] Smith, Jonathan M., *Concurrent Execution of Mutually Exclusive Alternatives*, Ph.D. dissertation, Available as Computer Science Department technical report CUCS-241-89, Columbia University, 1989.

[St87] Stallings, William, *Handbook of Computer-Communications Standards: The Open Systems Interconnection (OSI) Model and OSI-Related Standards*, MacMillan, Inc., Vol. 1 (1987).

[St88] Stankovic, John A., "Misconceptions About Real-Time Computing: A Serious Problem for Next Generation Systems." *IEEE Computer* (Oct. 1988), pp. 10-19.

[St90] Stamos, James W. and Gifford, David K., "Implementing Remote Evaluation." *IEEE Transactions of Software Engineering V. 16*, N.7 (Jul. 1990), pp. 710-722.

[Su88] Sun Microsystems, Inc., RPC: Remote Procedure Call Protocol Specification. Tech. Rept. RFC-1057, DARPA Network Working Group Report, Jun., 1988.

[Su89] Sun Microsystems, Inc., NFS: Network File System Protocol Specification. Tech. Rept. RFC-1094, DARPA Network Working Group Report, Mar., 1989.

[Ta88] Tanenbaum, Andrew S., *Computer Networks*, Prentice-Hall, N J , Second (1988).

[To87] Toueg, Sam and Srinkanth, T.K., "Optimal Clock Synchronization." *Journal of the ACM V. 34*, N.3 (Jul. 1987), pp. 626-645.

[To89] Touch, Joseph D. and Farber, David J., "Mirage: A Model for Ultra-High-Speed Protocol Analysis and Design." In *Protocols for High-Speed Networks*, Rudin, Harry and Williamson, Robin, (Eds), IFIP, North-Holland, Available as Univ. of Penn. Dept. of Computer and Information Science Tech. Report MS-CIS-89-79 / DSL-1, 1989, pp. 115-134.

[To90a] Touch, Joseph D., Mirage: A Model for Latency in Communication. Tech. Rept. MS-CIS-90-74 / DSL-3, Dept. of Computer and Information Science, University of Pennsylvania, (dissertation proposal), Oct., 1990.

[To90b] Tokoro, Mario, Computational Field Model: Toward a New Computing Model / Methodology for Open Distributed Environment. Tech. Rept. SCSL-TR-90-006, Sony Computer Science Laboratory, Inc., Tokyo, Japan, Jun., 1990.

[To91a] Touch, Joseph D. and Farber, David J., *Mirage: An Introduction*. Feb. 1991, (submitted to ACM Computer Communication Review).

[To91b] Touch, Joseph D. and Farber, David J., *An Active Instruction Decoding Processor-Memory Interface*. Sept. 1991, Patent applied for, Univ. of Pennsylvania.

[Ve86] Venkatraman, R.C. and Piatowski, Thomas F., "A Formal Comparison of Formal Protocol Specification Techniques." In *Protocol Specification, Testing, and Verification*. North-Holland, Diaz, M., (Ed), pp. 401-420, 1986.

[Wa72] Watson, W.J., "The TI ASC: A Highly Modular and Flexible Super Computer Architecture." In *AFIPS Fall Joint Computer Conference*, AFIPS, AFIPS Press, Montvale, NJ, Dec. 1972, pp. 221-228.

[Wa81] Watson, Richard W., "Timer-Based Mechanisms in Reliable Transport Protocol Connection Management." *Computer Networks*, N.5 (1981), pp. 47-56, Pub. by North-Holland.

[Wa89] Watson, Richard W., "The Delta-t Transport Protocol: Features and Experience." In *Protocols for High-Speed Networks*. Elsevier Science (North-Holland), Rudin, H. and Williamson, R., (Eds), pp. 3-17, 1989.

[Wa90] *The SPEC Benchmark Report*, Waterside Associates, Freemont, CA, Jan. 1990.

[Wa91] Wall, David W., "Limits of Instruction-Level Parallelism." In *ASPLOS-IV*, ACM, Apr. 1991, pp. 176-188.

[Wi48] Wiener, Norbert, *Cybernetics: or Control and Communication in the Animal and the Machine*, MIT Press, Cambridge, MA (1948).

[Wo90] Woodside, C.M., Ravinadran, K., and Franks, R.G., "The Protocol Bypass Concept for High Speed OSI Data Transfer." In *Participant's Proceedings, International Workshop on Protocols for High-Speed Networks*, IFIP WG 6.1/WG 6.4, Palo Alto, CA, Nov. 1990.

[Ye87] Yeh, Y.S., Hluchyj, M.G., and Acampora, A.S., "The Knockout Switch: A Simple, Modular Architecture for High-Performance Packet Switching." *IEEE Journal on Selected Areas of Communications V. SAC-5*, N.8 (Oct. 1987), pp. 1274-1283.

[Zh89] Zhang, Lixia, *A New Architecture for Packet Switching Network Protocols*, Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, Jul. 1989.

[Zh90] Zhang, Lixia, "VirtualClock: A New Traffic Control Algorithm for Packet Switching Networks." In *Communications Architectures and Protocols*, ACM Sigcomm, ACM Press, Also Computer Communication Review, V. 20, N. 4, Sept. 1990, pp. 19-29.

APPENDIX A

Mirage & Shannon

In developing a new model for communication protocol analysis, we should examine the seminal work in the area, by C. Shannon. His work explains the operation of many communication models. This is a comparison of Mirage to it. It is also hoped that the Mirage model will reduce to Shannon's, in the case where latency is negligible relative to the other communication parameters.

A.1. The channel

Shannon's mathematical model of communication defines channel bandwidth and capacity, and analyzes the capacity of the channel under the constraint of transmission error [Sh63]. In his model, nodes are connected by channels characterized by bandwidth alone (latency is ignored). His analysis determines the amount of information transmitted across a channel, given the transmission errors of that channel.

In this model, the channel can viewed as a pipe between the communicating nodes (Figure A.1). Bandwidth is a unit of volume of flow in this pipe – bits wide times signal duration. Note that the propagation (latency) of this volume as it traverses the pipe is ignored – Mirage adds this factor, in its extension of this model.

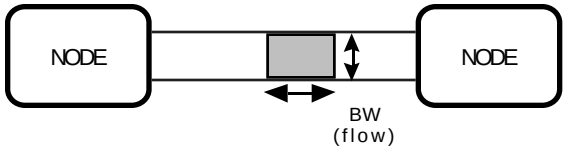


FIGURE A.1
Shannon's communication channel

Mirage adds a spatial measure to the connectivity measure of Shannon's communication theory. In addition to width, the connecting channel thus has a length (Figure A.2).

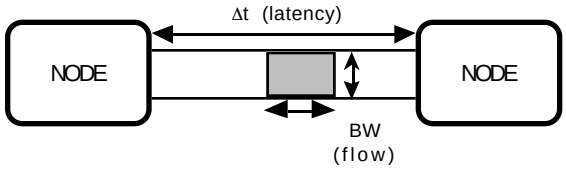


FIGURE A.2
Mirage's communication channel

One the test of the Mirage model is that it reduce to Shannon's where latency is negligible. Mirage adds a latency measure to the channel characterization, but this can be ignored if the state transformation equations ignore the latency measure, so the reduction holds.

A.2. State transformations

Shannon's model is based on a denoting the state of a node as a point in state space, implying that the values at the node are known precisely at remote nodes. This is implicit in the communication model, which attempts to emulate the transitions of the transmitter by equivalent transitions in the receiver (Figure A.3).

The communication is based on a model of the channel as it corrupts information that traverses it. Each action of a participant alters the state space point by moving it to a new point. Sending and receiving data are both modeled as motions of single points in state space. The elapsing of time is not modeled in this scheme.

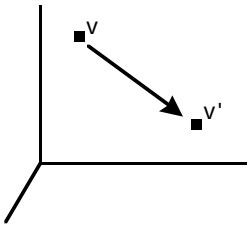


FIGURE A.3
State space point transformation

In Mirage, the sender models the receiver as a set of points in state space. The endpoints of the channel are considered, rather than the channel itself. Each node in the system models each other, to some extent. These models are sets of points, which, in an appropriately transformed space, comprise a volume. Transformations on that volume represent the functions of the communicating system (Figure A.4). When a node sends a message, its model of the state of the destination of that message expands; when a node receives a message, its model of the source of that message contracts. Time causes the model of the remote node to expand, reflecting the increased uncertainty in the knowledge of the remote state.

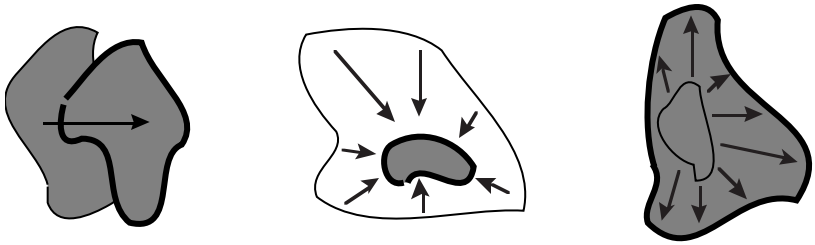


FIGURE A.4
Visualization of state space volume transformations

A.3. Levels of communication

In the introduction to Shannon's work, W. Weaver describes three levels of communication [Sh63]. These levels define the layering of the communication problem, each level being dependent on the successful communication of information at the

previous level. Associated with each level is a problem, which determines the extent to which the communication at that level can succeed (Table A.1).

The first level is called the **PRECISION** level, and it is associated with the *technical* problem of determining the transmitted symbol from the received signal. Communication at this level assures that a mapping is established between the signals on the opposite ends of the channel. The extent to which the symbol association is *repeatable* determines the most fundamental limit of communication.

The second level is called the **ACCURACY** level, and it is associated with the *semantic* problem of identifying the meaning of the symbol received. This determines the *correctness* of the received information, relative to the intended information sent.

The third level is called the **BEHAVIORAL** level, and it is associated with the *effectiveness* of the communicated message. Effective communication is *correlated with the desired behavior* of the receiver, i.e., if the receiver acts as if it received the correct information, then we infer that it has.

Level Name	Defined Characteristic	Net result on communication
technical	precision	repeatable
semantic	accurate	correct
effective	correlate to desired behavior	reaction

TABLE A.1
Weaver's 3 levels of communication

Shannon's work focuses on the technical problems at the precision level, although there are applications of his theory to the other levels as well. Each of these levels is concerned with errors in communication, either in reliability, correctness, or resulting behavior.

A.3.1. Extensions for time

In Mirage we are interested in extending Shannon's theory to its application in high speed, wide area networks. We have already discussed that the major effort here is to

sensitize the problem to communication latency, as such, we consider how to extend Shannon's model to account for latency, as it already accounts for error.

One constraint of our extension is that, where latency is negligible, it collapses to the original model; thus it will be an extension to the model. Other constraints are that the model be useful, i.e., that it describes the new domain effectively and that it enables the derivation of protocols that account for this increased latency. We also prefer the model to exhibit these characteristics by an extension that is minimal, but this is not addressed here.

A.3.2. Time vs. error

One of the fundamental results of Shannon's theory is that any amount of channel error (below 100%) can be removed by sufficient encoding. Given encodings over arbitrarily long sequences of transmitted symbols, the effective error of the channel can be reduced as low as desired (but never removed completely). The effect of error compensation and reduction is to require encoding, which requires delaying the symbol stream by the length over which encoding is performed. As such, error reduction is traded for an increase in propagation delay.

Mirage examines the complement of this, where latency is reduced by increasing the error across the channel; the error will be exhibited by the imprecision of information about remote nodes in the network. Error and latency are thus conjugate spaces, where each may be traded for the other, and some minimal product persists. The error thus introduced will be constrained, in a 'well-behaved' way, which represents the evolution of imprecision of information caused by elapsed time and other causes.

We define three additional levels of communication, associated with the introduction of measured latency in the channels, called ***lag*** and ***stability***. The lag level is associated with the *timeliness* problem, or how to communicate some amount information within some time delay. The stability level is associated with the *synchronization* problem, or whether two sets of information can be synchronized to within some error in the given time lag (Table A.2). Each level has a corresponding Weaver level, as shown.

Level Name	Defined Characteristic	Net result on communication	Corresponding Weaver level
timeliness	effect within Δt	lag	Technical (repeatable)
synchrony	synchronize to within Δt	stability	Effectiveness (reaction)

TABLE A.2
Mirage's 2 levels of latency

A.4. Observations

This gives a hint at the justification for seeking additional models for protocol analysis. Current models yield situations in the emerging high speed, wide-area domains where utilization of the communication capacity can be low. New models may explain this phenomenon more precisely, and perhaps indicate methods that alleviate such degradation.

In Mirage, there are three forms of communication: ***real***, ***virtual direct***, and ***virtual indirect***. Real communication corresponds to Shannon's communication, where information is transmitted, and the intent of the sender is decrypted by the receiver. Virtual indirect communication is derived information about a set of nodes, given global constraints on the state spaces of all nodes combined with real communication from some other set of nodes. This is also known as inferred or derived communication, and corresponds to common knowledge.

Mirage uses a third form of communication, that of virtual indirect. This is information derived from local constraints about the state evolution of a node and the absence of other communication from that node. Virtual indirect information is contained in the state evolution function of a node's individual perception, i.e., in the time transformation function of Mirage.

APPENDIX B

Mirage & Physics

Mirage was originally conceived of in terms of quantum physics analogs [To89]. These analogies helped develop the Mirage model, so it is useful to present some of these discussions here, for historical purposes.

B.1. Origins of the analogy

The origins of Mirage began with discussions of state space evolution, and with the imprecisions in that space introduced by communication latency. This latency corresponds to a latency of interaction, which governs the degree of coupling of systems separated in time. This is loosely analogous to particle interaction by force-carrier exchange. Mirage is an attempt to integrate ideas from particle interaction of quantum physics and information theoretic analysis to develop a communication protocol model, as depicted in Figure B.1.

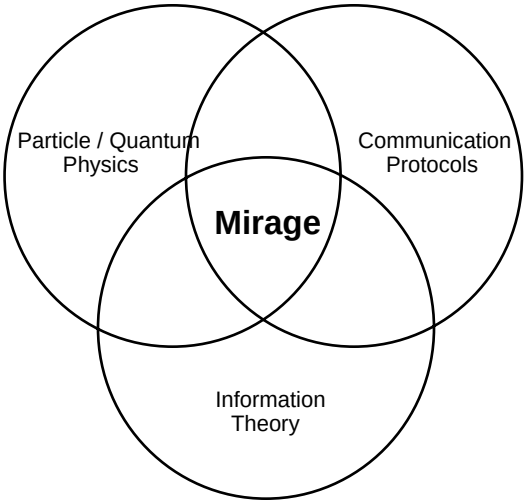


FIGURE B.1
Mirage's relationship to other sciences

B.1.1. Field interaction as communication

A protocol can be considered analogous to field interaction as explained by particle exchange in quantum physics (Table B.1). In this analogy, a field is communication, i.e., action at a distance. The mechanism of interaction is field quantum exchange, analogous to packet exchange. Traditional particles in the field are nodes, thus emphasizing the blurred distinction between particles and field quantum, i.e., between data packets and nodes. Uncertainty of particle interaction corresponds to latency of interaction, and the effects of high speed extend the model of interaction, in the manner of relativistic effects.

Physics	Protocol / Network
field	communication
field quantum	packet
particles	nodes
relativistic effects	high speed
uncertainty	latency

TABLE B.1
Physics analogs of protocol components

In physics, interaction between particles is accomplished by the exchange of other, force-carrying particles. A matter particle emits a force particle by creating it from nothingness (and is recoiled as a result); that force particle is absorbed, causing an impulse where absorbed. If the force particle has high mass, it is hard to exchange over long distances, due to the high temporary energy debt cause by the creation of the force particle. Force particles are thus virtual, i.e., measurable only by their effect. This is discussed in further detail in [Ha88b].

B.1.1.1. Four physical forces

There are, in physics, four forces: electromagnetism, the strong nuclear and weak nuclear forces, and gravity. The strength of the force and distance over which it acts is governed by the mass of the force-carrying particle. Electromagnetism is effective over infinite distances, but affects only charged particles. Photons carry the electromagnetic force, and are bosons (spin-0).

The weak nuclear force governs radioactivity, and is effective over very small (nuclear) distances, and affects only matter particles (fermions, i.e., spin-1/2 particles, not bosons, i.e., integral spin particles). This force is carried by spin-1 vector bosons.

The strong nuclear force holds the nucleus together, and is thus effective over only nuclear distances. It is carried by gluons, which are bosons.

Gravity affects all particles, is weak, and effective over infinite ranges. Further, it is always attractive. Gravitons, which are bosons, are proposed to carry the gravitational force.

B.1.1.2. Communication forces

Communication is also effected by an exchange. Intuition is that the larger the packet of an exchange, the smaller the effective distance of the data of the packet, because interaction is restricted by latency. Larger packets incur higher latencies.

B.1.2. Further references

The analogies between fields and communication have been examined before, in the Computational Field Model (CFM) [To90b], as also discussed as Prior Work in Chapter 3. CFM equates a distributed system with a field and particles, context-switch overhead with inertia, and communication bandwidth with force. It is used to develop a

self-optimizing process migration system. Mirage differs from CFM by using physics analogies to guide protocol design, where the analogy ‘homomorphism’ is given semantic justification.

With regard to the remainder of the discussion, there are a few notable references. First, [Kl75] contains a good description of the difference between probability density and distribution functions. A good overview of quantum concepts is contained in [Re91], [Ha91], [Sh88]. Original discussions of the Many-Worlds principle are contained in [De73], and an introduction to this principle is given in [Ha91]. An overview of quantum principles for non-scientists is given in [Ga65]. Lastly, an excellent historical overview and presentations on quantum principles are given in [Pe89], although this book is more commonly presented as a discussion of the ‘mental’ capabilities of discrete systems.

B.2. Existing analogies from physics

The existing analogies between communication and physics include similarities between entropy and information, uncertainty and stability, and the Hamiltonian function and state change functions.

B.2.1. Entropy

Entropy in physics is related to information in communication, as first noted by John von Neumann [Ha28]. The two are inverses, so that the negative of entropy is proportional to information; information is thus sometimes also called ‘negentropy.’ Both are proportional to the logarithm of possible state space, and are additive where systems are combined. The use of entropy in communications has thus become common.

Some discussions consider physics entropy and information entropy similar but otherwise unrelated, whereas others consider the two identical. We consider them identical for the following reason.

In physics, specifically thermodynamics, entropy is a measure of disorder. The units are calories/degree. Calories are a measure of work, energy, or heat (equivalently). Temperature is defined as energy per degree of freedom, i.e., a measure of energy extracted when degrees of freedom are unified.

In information, entropy is a measure of the log of the number of possible states, or the average number of bits required to specify a state within a partition. Entropy thus

measures imprecision of state, or disorder among elements of a partition that information removes.

In both cases, entropy measures disorder, and the amount of 'work' required to compensate for the disorder. In physics, 'work' is work, heat, or energy, whereas in communication 'work' is information, or bits. Thus we consider physics entropy equivalent to communication entropy because we consider work equivalent to information. The only difference is that in physics the degrees of freedom are continuous[1], and in information they are binary.

B.2.2. Uncertainty

One result of the true equivalence between physics entropy and information entropy observation concerns uncertainty. The Heisenberg uncertainty principle is characterized by units of 'action.' [Gr84] An 'action' is defined as energy*time, or work*time; in communication, this is bits*time, or bit-latency, the unit of 'distance' in Mirage. Uncertainty is a measure of imprecision of state in physical systems, and bit-latency governs the imprecision of state in communicating systems.

Further, 'actions' share a property with entropy – that of observer invariance [Gr84]. In relativistic physics, some measures are not observer invariant; the length of an object depends on the relative velocity between the observer and the object. Time-of-traversal, or distance/time is invariance, because the object shortening is always coupled with a corresponding decrease in the relative time frames.

B.2.3. Hamiltonian function

The *Hamiltonian function* describes the state transformation of a physical system [Pe89]. The quantum equivalent of the Hamiltonian is the *wave-function*. In Newtonian (conventional) state space, the Hamiltonian of object location and velocity is denoted by a pair of partial derivatives, Eqs. B.1 and B.2.

---

[1]In thermodynamics, degrees of freedom are continuous-valued; in quantum physics, they are quantized, further blurring this distinction between information entropy and physics entropy.

Equation B.1: $\dot{p}_i = -\frac{\partial H}{\partial x_i}$

Equation B.2: $\dot{x}_i = \frac{\partial H}{\partial p_i}$

'H' denotes the Hamiltonian, 'p' denotes the momentum, and 'x' denotes the position. The Hamiltonian encodes both the position and momentum. This will be augmented in the quantum description of the equivalent of the Hamiltonian. The pair of equations obey the Heisenberg uncertainty principle, in which the error in position times the error in momentum is always larger than a fixed constant (Eq. B.3).

Equation B.3: $\Delta x \Delta p \geq \hbar$

B.3. Quantum analogies

Although there are some analogies between Newtonian physics and telecommunications, there are others involving quantum physics which have not yet been exploited.

B.3.1. State sets / multiple worlds

In quantum physics, the state space of a system has a single dimension for each system variable, and is called a *phase space*. One point in phase space thus denotes an entire configuration of the system; multiple points denote copies (or possible copies) of versions of entire systems [Pe89]. This latter phenomenon has evoked the title *Multiple-Worlds,* in which each world contains one system [Gr84]. In Mirage, a node models a remote state as a set of possible states, or worlds.

One implication of having multiple simultaneous possible states is that of *state collapse,* in which a single definite state among the possible is denoted. The denotation occurs because of some external event. In Mirage, this event is the reception of a message from the remote state being modeled.

The canonical physics example of simultaneous state is Schrödinger's Cat experiment [Gr84]. In this experiment a delayed choice is modeled in one of two possible

ways – either there are two possible worlds in which the cat is correspondingly dead and alive, or the state of the cat (dead, alive) is superimposed in the single world of the experimenter. In the former case, the state of the cat is known precisely in whatever world exists; it is the lack of information in the experimenter that causes the confusion in the choice of the correct world. In the latter, so-called *Copenhagen interpretation*, the cat exists in two superimposed states, and the delayed opening of the box causes the state vector to collapse.

Mirage is based on the Multiple-Worlds interpretation of quantum interaction, although we often denote the choice of the correct world as 'state collapse', because we model state as an expanding set that message reception collapses. The distinction is that the collapse occurs in the mind of the observer only in Multiple-Worlds, whereas that collapse is a property of the actual state of the system in the Copenhagen interpretation.

One interesting characteristic of the Multiple-Worlds interpretation is that the state of the system can be determined by the actions of the observer, in certain cases. When physical experiments were designed that emit particles (i.e., light quanta), and then the experimental apparatus is modified while the quanta are in transit, the results of the experiment change. The answer depends on the questions.

Compare the observer-creation of results with a game of Twenty-Questions, in which the participants agree not to select a goal object. The participants create replies that are random, but necessarily consistent with previous replies and some possible object. The result is that the random choices and the questions asked determine the final object, delaying the choice of the goal object (as in the delayed-choice cat experiment).

B.3.2. State set collapse

The collapse of the state set occurs in the perception of the observer or in the actual system state. In either case, the evolution of the state space set is governed by the so-called *wave-function, ψ,* and the Hamiltonian describing this evolution in Newtonian physics becomes Schrödinger's wave equation, Eq. B.4. In this equation, the partial of the wave-function with respect to time is the same as the Hamiltonian of the wave-function [Pe89] (with appropriate constants).

Equation B.4: $$i\hbar \frac{\partial}{\partial t}|\psi\rangle = H|\psi\rangle$$

The wave-function thus denotes the evolution of the system over time, even when a set of states describes the possible system states. The wave-function describes the Time Transformation of Mirage.

B.3.3. Virtual pairs

One model for interaction in quantum physics uses *virtual* particles. A particle is virtual if it can be measured only indirectly, through its effect on other, real (directly measurable) particles. One form of virtual particle is a member of a virtual pair, a particle and its negative image, which can be temporarily created in a vacuum by creating a temporary energy debt in the vacuum.

A second form of virtual pair denotes the two possible paths of a single particle in space. Consider the canonical double-slit experiment, using electrons rather than light. A single electron is a real particle, that should go through only one of the two slits, by Newtonian laws.

In quantum physics, the real electron becomes a set of virtual, mutually-exclusive electrons. These virtual particles travel through all paths in space-time from the emitter to the detector. Two of these paths go through the two slits. The virtual electron going through one slit interacts with the virtual electron going through the other slit, forming an interference pattern. The seeming paradox is that a single real, measurable particle apparently must go through two paths in space-time simultaneously for the interference pattern to occur.

> "Any other situation in quantum mechanics, it turns out, can always be explained by saying, 'You remember the case of the experiment with the two holes? It's the same thing.'"
>
> - Feynman, quoted in [Gr84]

In Mirage, this latter form of virtual, mutually-exclusive particles corresponds to the possible paths in *state-space*-time. These virtual paths and particles are denoted more explicitly in the application of Mirage to Petri Nets, in Appendix F.

B.3.4. Feynman path integrals

Real particle paths are the integral of the mutually-exclusive virtual particle paths and interactions therein. Richard Feynman described these *path integrals* in quantum physics [Gr84].

In quantum interactions, the virtual splitting is not restricted, so a real particle path is described by the integral of an infinite number of virtual particle paths. This introduces an infinity that can be removed by Feynman's technique of *renormalization*.

Mirage uses a direct analog of Feynman paths in the description of the computation function governing Time Transformations, i.e., in the wave-equation. Mirage does not require renormalization, because the remote state being modeled is equivalent to a Turing Machine (TM). Over a finite time interval, a fixed number of TM state changes occur, and each state change is finitely bounded, so the resulting number of possible state paths of the TM is also bounded. This prevents the need for renormalization, because infinite path lengths and numbers are not possible.

B.4. Observations from the analogy

There are a few useful observations from these analogies between physics and communication, and particularly involving quantum interactions. Some of these observations have been presented in the Mirage model description (Chapter 2), in Prior Work (Chapter 3), and in the Mirage extension of Petri Nets (Appendix F). Other observations include the relationship between error and latency, and the interpretation of stability as denoted by these analogies.

B.4.1. Error and latency as conjugates

Error and latency are conjugates, in which the units of the product of such conjugates are 'actions', as described before [Gr84]. The limitation of the product in communication is the bit-latency of the channel. This limit determines the smallest error in stability, in the absence of other constraint information.

B.4.2. Stability

Stability in Mirage consists of either traditional stability, or entropic stability (Chapter 2). Traditional stability guarantees constraint of state evolution over time to a fixed subset of possible states. Entropic stability guarantees the evolution of the size of the possible states over time, but doesn't restrict the state values to a fixed set. The Time Transformation indicates the evolution of the state space over time.

In physics, the Hamiltonian function denotes each point in phase space as a vector to the subsequent point, i.e., 'H' defines a *vector field* on phase space. Stability exists when a phase space region is closed with respect to the vector field, i.e., no vector exits the region [Pe89].

The Liouville theorem indicates that the volume of a region of phase space remains constant, but permits the size of the region to grow. The *volume* of a region is a measure of the number of possible states in the region, whereas the *size* of the region is a measure in relation to the extremes of the dimensions. The components in the region can disperse throughout phase space, but the total number of components cannot decrease; this latter view is described as the *incompressibility of the vector field flow*.

By the Liouville theorem, if no vector exits a region, no vector can enter either. This implies that stability violates the theorem. An example of this paradox is shown in the *Hawking box*.

A Hawking box is a box in thermal equilibrium, containing one black hole [Pe89]. The Hamiltonian in the box has vectors converging in the hole, i.e., there is a confluence (compression, merging) of flow lines. By the Liouville theorem, there must be a corresponding divergence of flow lines, because flow lines remain constant in overall number (incompressible flow of the Hamiltonian).

Hawking himself noted that, "I am indeed claiming that it is an *objective* (sic) quantum-mechanical process of state-vector reduction…which causes the flow lines to bifurcate…"[Pe89] – implying that flow lines can converge (lose information) or diverge (create alternatives).

This may seem contradictory with Mirage's interpretation of information creation as divergence, and information reception (state-vector reduction) as convergence. This states that state vector-reduction causes flow lines in the Hamiltonian to bifurcate, whereas we claim that the reduction causes state alternatives to collapse (reducing the entropy of the state).

Divergence of flow lines corresponds to a collapse of the state space, because 1 previous state with imprecision becomes one of 2 with precision, i.e., 1 large space becomes 1 of 2 small spaces. The large → small transformation represents the collapse of the space and the increase in information (decrease in entropy). The 1 to 1 of 2 transformation corresponds to the bifurcation of the flow lines.

Mirage thus resolves the paradox with the creation of information by an external party, i.e., the user. A similar paradox removal involves the introduction of information to a formerly closed system by biological participants [Ja55].

APPENDIX C

Upper Bound

This is an analysis of the comparison of the exact and continuous channel utilization formulae from Chapter 2. It will show that the complete discrete formula of channel utilization under finite branching of the message stream (Equation C.1, as repeated below) is bounded by the continuous form (Equation C.2, also as repeated below). We claim that the continuous form is an upper bound for the complete discrete form.

Equation C.1: $$\%util = \frac{L + \lfloor tree\_depth \rfloor * B + \frac{D^{frac(tree\_depth)} - 1}{D - 1} * B}{rtt}$$

Equation C.2: $$\%util = \frac{L + tree\_depth * B}{rtt}$$

In order for Equation C.2 to be an upper bound to Equation C.1, Equation C.3 must hold. This simplifies (via Eqs. C.4, C.5) to Equation C.6. If Equation C.6 holds under the domain of the original formula, then the upper bound is proven.

Equation C.6 has at least two zeroes, at the endpoints (x=0, x=1) of the domain. It is also positive at arbitrary points within the domain, and continuous within the domain. Its derivative (Equation C.7) has only one zero, i.e., there is only one zero of the slope in the domain. Equation C.6 has two zeroes at the endpoints, is continuous, positive between

those endpoints, and its slope has only one zero (a maximum), thus we conclude that it holds for all points in the interior of the domain.

Equation C.3: $$\frac{L + tree\_depth * B}{rtt} \geq \frac{L + \lfloor tree\_depth \rfloor * D + \frac{D^{frac(tree-depth)} - 1}{D-1} * D}{rtt}$$

Equation C.4: $$tree\_depth \geq \lfloor tree\_depth \rfloor + \frac{D^{frac(tree\_depth)} - 1}{D-1}$$

Equation C.5: $$frac(tree\_depth) \geq \frac{D^{frac(tree\_depth)} - 1}{D-1}$$

Equation C.6: $$x - \frac{D*\left(D^{x} - 1\right)}{D-1} \geq 0$$

where $x \in [0,1)$
$D \geq 2$

Equation C.7: $$x' = \log_D\left(\frac{D-1}{\ln(D)} + 1\right)$$

Equation C.6 is presented graphically below (Figure C.1), denoting the difference between the upper bound and the exact formula. ‘Fraction’ denotes FRAC(*tree-depth*); note that as branch degree increases from 2, the deviation increases, and the upper bound becomes less exact, especially for larger values of FRAC(*tree-depth*).

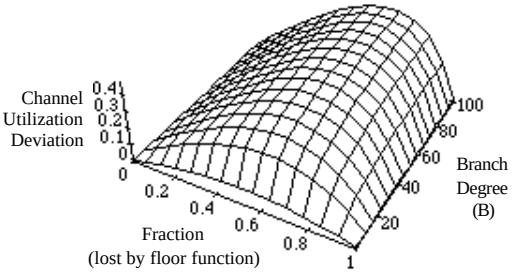


FIGURE C.1
Error between upper bound and exact channel utilization

APPENDIX D

The Liouville Theorem

The Liouville theorem is a result of statistical mechanics, which states constraints on the temporal evolution of state space volumes in models of closed systems [Hi56]. The terminology used here is different from that of statistical mechanics (SM), where the theorem is usually presented. What Mirage call state space, SM calls phase space. Mirage refers to a volume in that state space as representing a subset of that space that denotes a set of possible system values; SM refers to the probability density function as describing this set, and its distribution in phase space.

In our notation, the Liouville theorem states that the volume of state space representing the local state of a node cannot add or delete points. The theorem claims that every point in the original state space lies on a unique trajectory of points, of a system as it evolves in time. A system's trajectory is completely described by one such point, because other constraints on the system completely determine its subsequent state from its current.

This indicates that, whereas the points in the original volume may later spread out in space, the overall number of points (the integral of this volume) remains constant. If the integral were to decrease or increase, it would imply that two trajectories merged, indicating that the system had been incompletely described by the state space variables and transformation rules. A closed system is permitted to move through state space, but

never to bifurcate into two system instances. Volumes in the state space represent ‘competing’ instances of a system, but only one actually exists. Each system instance has its own unique trajectory, so whereas the volume can move through state space (by translation or deformation) the integral (number of possible state values in the volume) must remain constant.

The theorem applies only to closed systems; the integral of the volume of state space representing remote nodes is not part of such a system, and it thus is not a violation to speak of that volume as expanding (increasing the integral) or collapsing (decreasing the integral). The theorem is meant to convey the notion that information, in a closed system, is neither created nor destroyed over the long term. Within a node, however, information can be created. This occurs whenever I/O occurs at the node, such that the state of the system is now enhanced by external information.

The quantum aspect of this phenomenon is necessitated by the discrete values of the variables in our system – the integral is more properly a (discrete) summation. The theorem applies to the quantum domain in a similar, though distinct fashion. The implications of maintaining this theorem as it applies to Mirage will be discussed more completely in subsequent research.

APPENDIX E

Mirage in Set Notation

Mirage uses state space volumes to describe the possible states of a remote node. One instance of this model is a description in terms of sets of states, so that each set represents a 'volume' in state space. This set notation can be extended by including a probability for each state. The continuous form of state sets with probabilities is a probability density function (pdf). The following is an elaboration of the definition of the state space volume description of Mirage from Chapter 2, in terms of sets.

E.1. Definitions

Consider the set of nodes in the network. These nodes are herein completely connected, each pair (*i*,*j*) connected with a finite maximum communication bandwidth ($BW_{i,j}$) and a finite minimum communication delay ($\Delta t_{i,j}$). The following definitions will be used; the 'equations' below represent these definitions.

M_i denotes a node's finite **storage**. This storage is used both to denote the node's dedicated local storage and perceptions of the storage of remote nodes.

S_i denotes the local **state** of the node *i*, the local component of its storage.

$P_{i,j}$ denotes node *i*'s **perception** of node *j*, which is some subset of the set of all states of node *j*, namely S_i (Eq. E.1).

V_i denotes node *i*'s **view** of the network, comprised of its own local state S_i and the set of perceptions $P_{i,j}$ of the other nodes (*j*) in the network (Eq. E.2).

The only constraint thus far is that the size of the node be sufficient to store its view (Eq. E.3). Note also that mutually recursive knowledge is permitted, provided that the recursion is bounded and finite, as required by the fixed size of local storage at each node.

Equation E.1: $P_{i,j} \subseteq Powerset\left(S_j\right)$

Equation E.2: $V_i = S_i \cup \left(\bigcup_{j \neq i} P_{i,j} \right)$

Equation E.3: $\left|V_i\right| \leq \left|M_i\right|$

E.2. Time Transform

Time transforms the perception by expanding it, thus reducing the precision in the knowledge of the state of the remote node. The temporal transformation of a perception of a remote node *P* over the interval Δt is denoted by a function $C[\Delta t, P](t)$; this function describes the known bounds on the state space evolution as a function of time. A node's view thus changes over time as its local state changes, and as the time transformations of its perceptions change (Eq. E.4).

Equation E.4: $V_i(t + \Delta t) = S_i(t + \Delta t) \cup \left(\bigcup_{j \neq i} C_j\left[\Delta t, P_{i,j}\right](t) \right)$

The extent to which the remote node is correctly modeled depends on the precision of this function, as characterized by the amount of state space expansion per unit time, a

form of induced entropy[1], *E*. The ratio of the volumes describes the expansion, equivalent to the entropy increase (Eq. E.5). The imprecision describes the difference between node *j*'s actual state and node *i*'s model of that state. The entropy change per unit time is a measure of the minimum bandwidth required to compensate for the entropy change (Eq. E.6).

Equation E.5: $$\Delta E_{i,j} = -\log_2\left(\frac{|old\_volume|}{|new\_volume|}\right) = -\log_2\left(\frac{|P_{i,j}(t)|}{|C_j\left[\Delta t, P_{i,j}\right](t)|}\right)$$

Equation E.6: $$Bandwidth = \frac{\Delta Entropy}{\Delta t}$$

The computation function, *C*, which describes the evolution of the space over time as viewed at a distance, is a combination of the remote space evolving over time and the messages that it can receive over that time. Analysis of this function can be complex, since all possible permutations of messages and computing intervals must be accounted for. The computation function encodes known internal computation at the node *j*, known bounds on the information received by node *j*, and known message emissions from node *j* (i.e., all known constraints on node *j*).

The formula is derived by taking the union over all possible time intervals (so we include inaction over the remainder of each interval), of the way in which any point (union over all points *p* in *P*) can be transformed by any computation function *c* in *C*, such that the computation function can occur in the specified interval (Eq. E.7).

Equation E.7: $$C_j\left[\Delta t, P_{i,j}\right](t) \subseteq \bigcup_{t' \le \Delta t}\left(\bigcup_{p \in P_{i,j}(t')}\left(\bigcup_{\substack{c \in C_j \\ time(c) \le t'}} c(p)\right)\right)$$

Equation E.7 denotes the computation constraint in terms of state space volume transformation. In the case where the time transformation is expressed by a probability density function (pdf), this function reduces to a convolution of the entire set of remote nodes (*p*'s) over the set of probability density functions (pdfs) of the transformations of individual messages that can be received (*c*'s) and the a time transformation pdf

[1]The notion that state reduction and volume ratios are related to entropy is not new; it has been discussed before, in [ShWe63] and [Ha28].

(Equation E.8). This reduction to convolutions requires that the time transformation is time invariant, i.e., it depends on the interval of elapsed time, but not the absolute time at which the interval occurs (Equation E.9).

Equation E.8: $C_j\left[\Delta t, P_{i,j}\right] = P_{i,j} \otimes C_{i,j}$

Equation E.9: $\forall_{t_1,t_2} P_{i,j}\left(t_1 + \Delta t\right) = P_{i,j}\left(t_2 + \Delta t\right)$

E.3. Receive Transform

Receiving information collapses the perception of a remote node to a subspace of its former volume. R_k denotes a message sent from node *k*, when it is received (i.e., this notation is used only within the node receiving that message).

R_k affects node *i*'s perception of node *k*, denoted by $P_{i,k}:R_k$. The total view of node *i* is affected only in the perception of the source of the message, so that the effect of R_k on node *i*'s view contains the unchanged local state and other perceptions, and the transformed perception of the message source (Eq. E.10).

Once constraint on the receive transform is that the transformed perception is a subset of the original perception, otherwise the perception was in error (Eq. E.11).

There is a limit to the amount to which the message can reduce the volume of the perception (Eq. E.12) (on average), since the volume reduction caused by the incoming information is bounded by the information content of that message, since volume reduction is equivalent to reduction in entropy.

Equation E.10: $V_i:R_k = S_i \cup \left(\bigcup_{j \neq k, j \neq k} P_{i,j}(t) \right) \cup P_{i,k}:R_k$

Equation E.11: $P_{i,k}:R_k \subseteq P_{i,k}$

Equation E.12: $\left|R_k\right| \geq \log_2\left(\frac{\left|P_{i,k}\right|}{\left|P_{i,k}:R_k\right|} \right)$

E.4. Send Transform

Sent messages expand the state of the message source by expanding the volume of the source's perception of the recipient of the message. A transmitted message is denoted as T_k, i.e., as a message transmitted to node *k*. This notation is used in the context of the source of the message, i.e., within node *i*.

T_k affects a node's view by transforming the perception of the remote node it is sent to (Eq. E.13), using notation similar to that of received messages. Again, the information contained in the sent message is limited by the transformation it effects, in terms of the relative volumes of state spaces indicated (i.e., entropy) (Eq. E.14).

Equation E.13: $V_i{:}T_k = V_i \bigcup P_{i,k}{:}T_k$

Equation E.14: $|T_k| \geq \log_2\left(\frac{|P_{i,k}|}{|P_{i,k}{:}T_k|}\right)$

E.5. Observations

The set notation interpretation of the abstract Mirage model is largely composed of additional notational constructions. The result of this exercise supplants this notation with formal relationships among bandwidth, message length, and the volume ratios of the original and transformed components of the state spaces. The description of the time transformation in terms of a computation function emphasizes the dependence communication on the application-layer semantics of the function of the remote node. The computation function description also led to the observation that the probability density function analog of the time transformation is pdf convolution, if the time transformation is time-offset invariant.

APPENDIX F

Mirage & Petri Nets

To introduce the effects of the Mirage model on protocol analysis, we demonstrate its effect on the Timed Petri Net model. The Mirage model suggests a version of Petri Nets where tokens replicate, and later annihilate each other, similarly to the interactions of virtual pairs of particle in quantum physics.

F.1. Petri Net Analogs

As another instance of the Mirage model's application, we have selected Timed Petri Nets [Me76]. Here these nets are extended to describe the state expansion and collapse that is integral to Mirage, while preserving the graphical/formal properties of the original network.

A Petri Net is a network of nodes, called *places* and *transitions*, and *arcs*. The net is bipartite with respect to the sets of places and transitions; arcs connect places to transitions or transitions to places. Nodes are depots for objects called *tokens*, and transitions denote rules for the movement of tokens from places at the input arcs to places at the output arcs. A transition can move tokens if it is *enabled*, if there is at least one

token matching the source of each input arc to the transition. Once enabled, the transition consumes these enabling tokens, and places a token at the places indicated by the output arcs [Pe77], [Pe62]. Timed Petri Nets include temporal constraints on the persistence of enabled transitions and the propagation of tokens along arcs [Me76].

To compose a Petri net analog of the Mirage state space subset model, consider the Petri Net (PN) normally associated with a protocol in a node. Places in this net represent conditions in the protocol, and the transitions represent transformation of sets of conditions into other conditions (Figure F.1). A marking of this net represents a particular, unique state of the node operating that protocol (Figures F.2A-E).

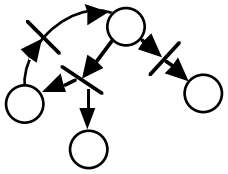


FIGURE F.1
Petri Net (unmarked)

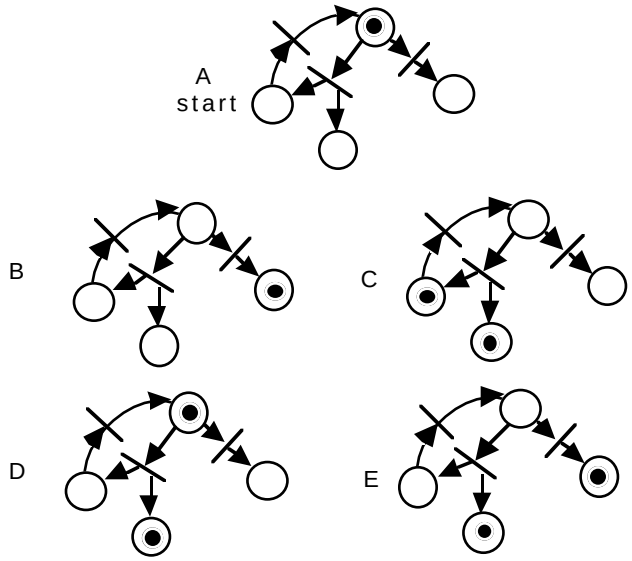


FIGURE F.2
Petri Net markings

Mirage needs to represent subsets of states, i.e., more than just single states. For a particular protocol PN, there is an associated finite state machine, called a token machine (TM) of that net (Figure F.3). The states of the TM correspond to the unique markings,

and the transitions in the TM represent the transformation of one marking into another, by action of the movement of tokens associated with the firing of active transitions of the PN. The state space of a Petri Net is the set of markings of that net.

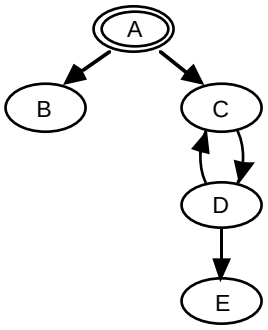


FIGURE F.3
Token machine of a Petri Net

Consider now the PN whose places correspond to the states of the TM, and whose transitions correspond to the arcs of the TM (Figure F.4). This is also a valid model of the protocol; we call this a meta-Petri net, or MPN.

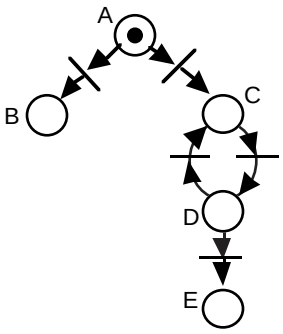


FIGURE F.4
Meta-Petri Net of a Token Machine

The protocol would begin in a single marking in the original PN, so there would be one token in the initial marking of the MPN, indicating that marking. Tokens in the MPN thus indicate entire markings in the original PN, which denoted states of the protocol. Note that in taking the TM of a PN into a MPN, the number of tokens of the MPN must be conserved after each firing in the MPN, because a PN firing results in only one resulting PN marking, even though the choice of which firing occurs from a given marking can be nondeterministic.

The conservation of tokens results from the construction of the machine. No transition contains more than one output, so tokens are never replicated.

For these diagrams, we consider only markings where a place is in one of two states – marked, containing a token, or unmarked, empty, a *binary* Petri Net. This simplifies the diagrams; Petri Net places normally are defined to contain an any number of tokens, in which case a different net could be developed which observes token conservation after some point, which this does not. In the (binary) example, state C goes to state D only once; the conventional (non-binary) PN would go to a next state that was a version of C with two tokens in the lowermost place, which is distinct from marking C.

Mirage implements transformations on the MPN that will enable the MPN to model multiple PN markings, by allowing it to violate the token conservation, under certain conditions. In the Mirage MPN, the number of tokens in the marking of a MPN reflects the entropy of the state of the node that the MPN is modeling.

F.1.1 Communication channel

So far, a Petri Net is considered a model of the entire network. In designing the Mirage-based transformations on the Meta-PN, the nodes are considered as separate subnets in the original Petri Net. One way of separating the network Petri Net is to consider a partition of the MPN into node nets and communication channel nets (Figure F.5).

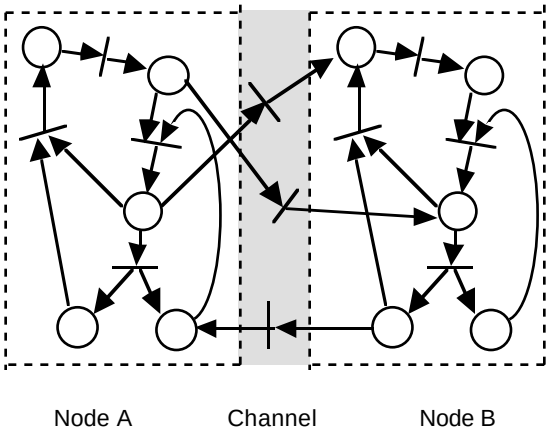


FIGURE F.5
Network MPN partitioned into channels and nodes

In partitioning the MPN, the resulting partial MPNs are incomplete, as they have input and output arcs which are not connected. The input and output arcs of the node MPN's are their interface to the communication channel. One of the simplest partitionings results in a channel MPN consisting of only transitions (i.e., having no places) (also Figure F.5).

The partitioning is required so that we may differentiate the state of a node from the its communication with other nodes; the reason for this difference will be shown later. Tokens in the node MPN denote the state of the node, whereas tokens in the channel MPN denote communication among the nodes.

F.1.1.1 Basic block

Further manipulation of the MPN requires the use of some definitions. We define a *basic block* as a subgraph of a MPN that is completely within the subgraph of a node, and which has one entry for tokens from that node, and one exit for tokens to that node (Figure F.6). It can have any number of token entries and exits to the communication channels to other nodes. The conservation of state tokens represents the point transformation model in Shannon's theory, where one point in state space (i.e., a single token in a single place in the MPN) is transformed into another point. Communication tokens have no similar constraint.

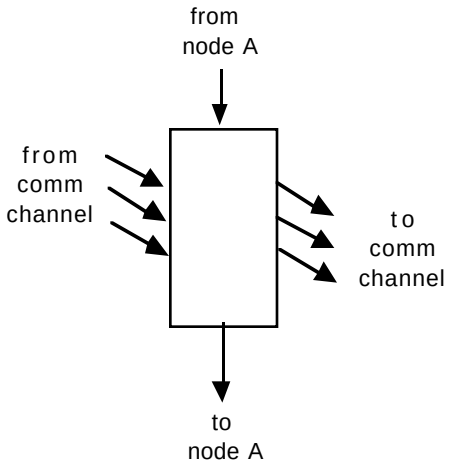


FIGURE F.6
Basic block

F.1.1.2 Virtual tokens

Mirage uses a MPN where indeterminism exists, where an advantage can be gained by running the MPN into the 'future', with virtual tokens. A *virtual token* is one of a set created which causes indeterminism in a MPN. Rather than having a one token continue on a single path, a virtual token is created for each path possible (Figure F.7). These tokens then belong to a *codependent* set. At some later time, if any of these tokens is to be considered real, *all codependents of that tokens set and all ancestors of all tokens in that set* must be destroyed (Figure F.8). A token is *real* if it is the lone token in a MPN.

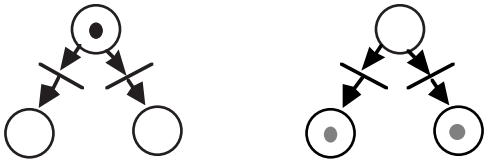


FIGURE F.7
Token virtualization

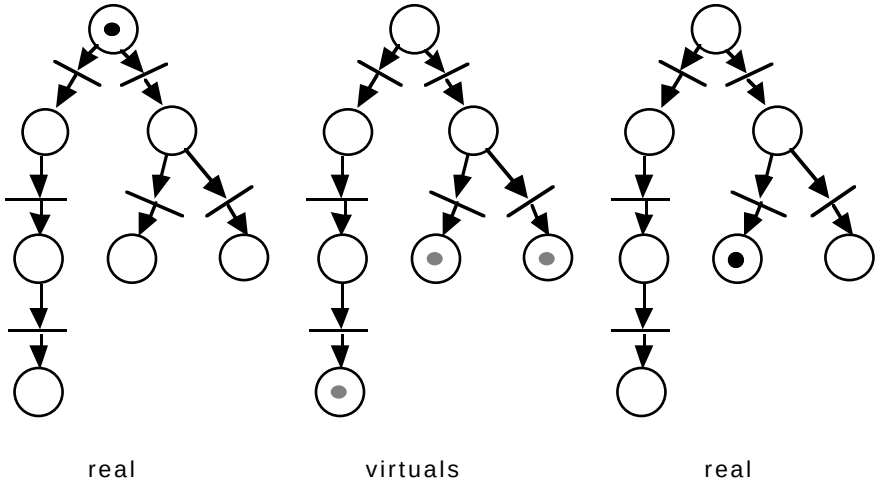


FIGURE F.8
Token realization

In token virtualization and realization, and the definition of removal of codependents is similar in principle to the visual process of a Feynman diagram. This is not a coincidence, as these operations were patterned after concepts from quantum physics. The number of simultaneous virtual tokens is also related to the size of the partition $\mathbb{K}$ in the Mirage stability criterion.

Token virtualization and realization can be introduced by a graph transformation in the MPN. This transformation introduces two tokens for each entering token, yet allows only one token to be emitted. In this way, virtualization and realization are implemented in the MPN.

The transformation works by the following principle. There exists a nondeterministic bifurcation into two basic blocks before the transformation (Figure F.9). After the transformation, two tokens, X and Y, are created. They affect their respective basic blocks, and exit the blocks accordingly. At that point, they are virtual tokens. They enter a mechanism that permits the first virtual token to pass, and be 'measured' (acted on by the rest of the T-MPN). That passage creates the measurable 'hat' token, and creates a token of the complement. The complement annihilates the opposite codependent token, thus maintaining the singular realization constraint (Figure F.10).

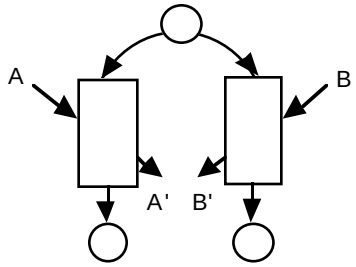


FIGURE F.9
MPN subgraph (before transform)

For example, if Y gets to the gate mechanism first, it passes, creating Y-hat, and X-bar. X-bar annihilates the next X token to reach the gate, causing the codependent virtual token to be destroyed. The rest of the T-MPN is permitted to act on the 'hat' tokens only. The feedback mechanism permits only one pair of virtual tokens from participating in the gate at one time.

Note that the messages emitted in the transformed MPN are guarded messages, where the conditional label on the message indicates which of the virtual tokens caused that message. Guarded messages are denoted by the form X:A, where X is a guard condition and A is a message.

In the pre-transformed subgraph, it is assumed that the inputs to the basic blocks have not yet arrived, so the token waits at the root (top) of the graph, and no further communication ensues. The transformation permits the messages to be sent in anticipation with appropriate guards. Later, the inputs match the virtual tokens at the *meet*

points, where token realization is modeled. The inputs arriving first permit realization of the virtual token on their side of the subgraph.

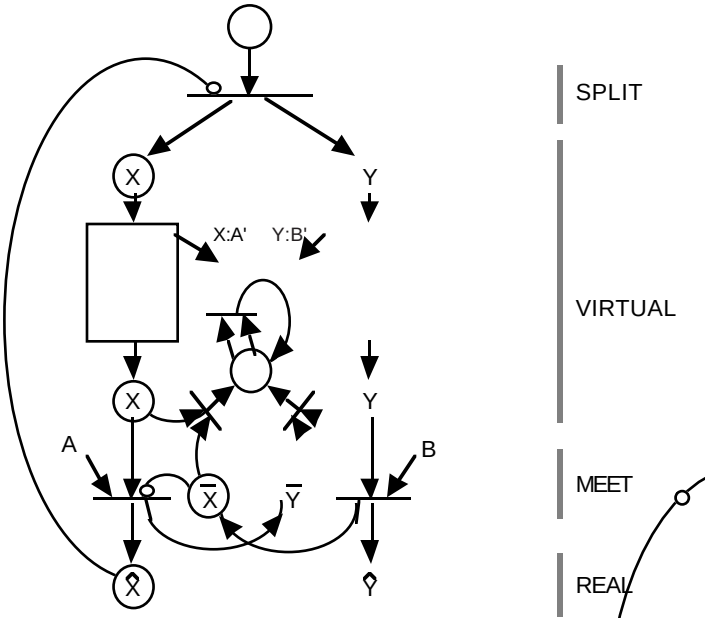


FIGURE F.10
MPN subgraph (after transform)

F.1.2 Equivalences to Mirage transformations

Now we show the equivalent transformations on the MPN of time, transmission, and reception of information in the network.

F.1.2.1 Time

Time is simply the token virtualization of a subgraph of a MPN of a particular node, representing the state of the remote node. This indicates that, as far as the state of the local node is concerned, the remote node is seen as being in multiple states at once. This reflects the increase in entropy of a system when information about the state of that system is absent, and the system is known to evolve over time.

Note that the only restriction imposed by Mirage on the virtualization is that the virtual tokens remain in the MPN representing a single remote node. As soon as any of these tokens interacts with a token from the representation of another node, or the state of the local node, it must be realized, and all codependents of the virtual disappear. Local

interactions involve real tokens, but remote node knowledge looks into the future with virtual tokens.

The other restriction of time virtualization is the existence of indeterminism in the MPN graph. This requires a structure as was shown for the virtualization process above. The increase in the number of tokens in the MPN indicates the increase in entropy of the state of a node, due to the increase in the state space of the perception of a remote node.

F.1.2.2 Send

Sending information similarly causes token virtualization, due to the possible loss of the transmitted message. The node sending the information must operate under the dual assumptions that the message has been correctly received and also that it has been lost (considering non-Byzantine errors only). This involves a recursive form of token virtualization, where one virtual token remains where the original real token was, and the codependent token flows through the graph ahead (Figure F.11). The virtualization is called *recursive* because one of the virtual tokens is placed where the original real token had been, and that virtual token is in a position of being re-virtualized (although this is possible, there may be other considerations).

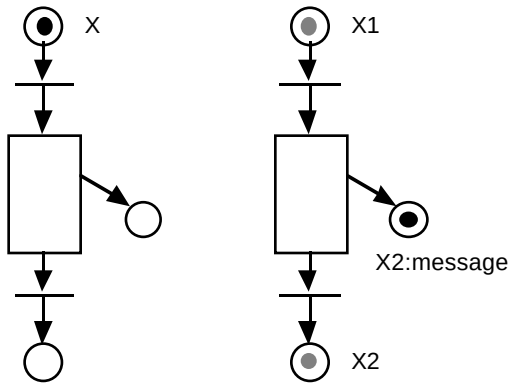


FIGURE F.11
Sending introduces virtualization

This form of virtualization does not require indeterminism in the MPN; it introduces it to account for the potential loss of information by the channel. The basic block must have at least one output, to describe the information sent into the channel. Before the transformation, a simple message would be sent; after the transform, a guarded message is sent instead.

One consequence of this transformation is that certain PN models of protocols are changed into machines that continually emit guarded messages. This is especially true in request/reply protocols based on finite databases, whose MPN is a very shallow, highly branched tree of such basic blocks. This may hint at an analytic justification for the efficiency of 'blasting' protocols in such domains, where continual emission of the database is preferable to an explicit request/reply.

F.1.2.3 Receive

Reception of information causes collapse of the state space of a remote node, from which information was received. This implies that the MPN of the representation of the remote node contained a set of virtual tokens, and that these tokens are meeting the arriving messages. When a message arrives, the state of the remote node is realized, and virtual tokens disappear (Figure F.12).

The decrease of the number of tokens caused by realization indicates the reduction in entropy caused by the reception of information. This decrease affects the entropy of the perception of the node from which information was received.

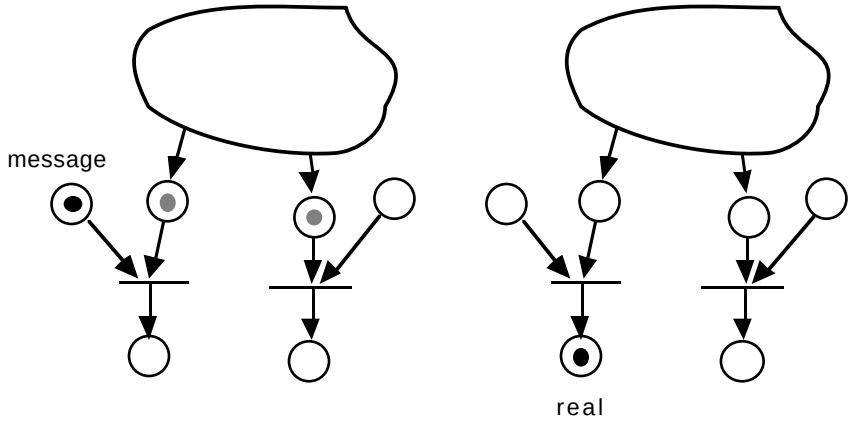


FIGURE F.12
Reception causes realization

F.2. Capacity in a PN

Even though we can define a communication channel as a PN, the notion of channel capacity has a time dimension, so we must use Timed Petri Nets, where there are

temporal constraints on the maximum and minimum times that conditions may persist at a transition before it fires, with similar bounds on the firing action itself. The definition of channel capacity would then be a function of the number of unique markings of this PN, in a given amount of time.

One definition of the maximum channel capacity involves operations on the MPN. The MPN can be extended into a Timed-MPN by analysis on the time conditions of the original net. The bandwidth of the T-MPN is related to the average number of branches in a cycle in the T-MPN, divided by the time to complete the cycle. The number of branches indicates the communication choices in the cycle, a measure of its information capacity. This capacity, over the time taken to communicate it, defines the bandwidth. The MPN must have been initiated from any of the possible markings of the input to the channel.

If we restrict a channel to a set of transitions, the capacity of the channel is defined as the number of patterns ($N^{\#transitions}$ – for Boolean PNs this becomes $2^{\#trans}$) divided by the sum of times of transition for each pattern. The time of transition for a pattern is defined as the time of the slowest enabled transition in the pattern.

The channel has input and output patterns, the capacity of the channel can be determined by considering the unique output patterns derived from the set of unique input patterns. These input/output sets can be analyzed by conventional Shannon techniques.

APPENDIX G

The TreeStack

The TreeStack is an abstract data structure that combines the characteristics of a stack and a tree into a monolithic entity. It was developed to facilitate the μ-Net description (Chapter 5), but may have more general application. We have not investigated the TreeStack as a formal data structure, nor have we determined whether it is novel; such investigations will be included in future work.

This is a description of the TreeStack as a formal data structure. Basic operations that define access to the structure are enumerated. One defining characteristic of the TreeStack is that it reduces a conventional tree or a conventional stack, under particular constraints.

G.1. Components

A TreeStack consists of three basic elements: a binary node, a unary node, and a leaf (Figure G.1). Binary nodes represent branchings, unary nodes represent stack elements, and leaves are the only terminal elements.

FIGURE G.1
TreeStack components

G.2. Operations

The operations of the TreeStack consist of the operations on a stack (Push, Pop) and the operations on a tree (Branch, Select Subtree). If Push and Pop operations are prohibited, then a conventional binary tree results. If Branch and Select Subtree operations are prohibited, then a conventional stack structure results.

G.2.1. Push

The ***Push*** operation indicates a recursion entry point, whose exit is the corresponding later Pop. A Push operation replaces a leaf with a unary node copy of that leaf, and places the leaf as a child of the unary node (Figure G.2). A Push is defined only as a transformation on a leaf. The cost of a Push is $O(1)$.

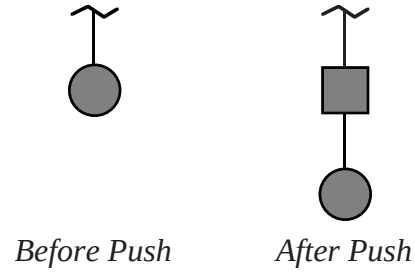


FIGURE G.2
TreeStack Push operation

G.2.2. Pop

A ***Pop*** operation indicates the exit of a recursion. Pops are defined only on subtrees terminating in a unary node & leaf pair. The leaf is discarded, and the unary nodes internal value is used to create a new replacement leaf, which is attached where the pair had been (Figure G.3). The cost of a Pop is $O(1)$.

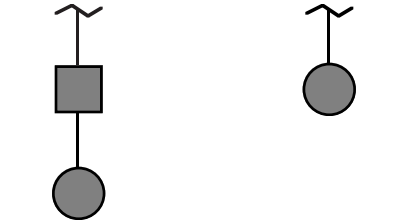


FIGURE G.3
TreeStack Pop operation

G.2.3. Branch

A ***Branch*** operation allows equivalent alternates to be represented in the data structure. A Branch replaces a leaf with a binary node, whose two children are two new leaves which are copies of the replaced leaf (Figure G.4). The cost of a Branch is $O(1)$.

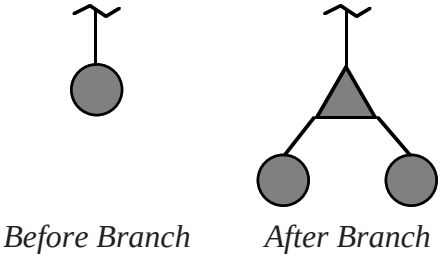
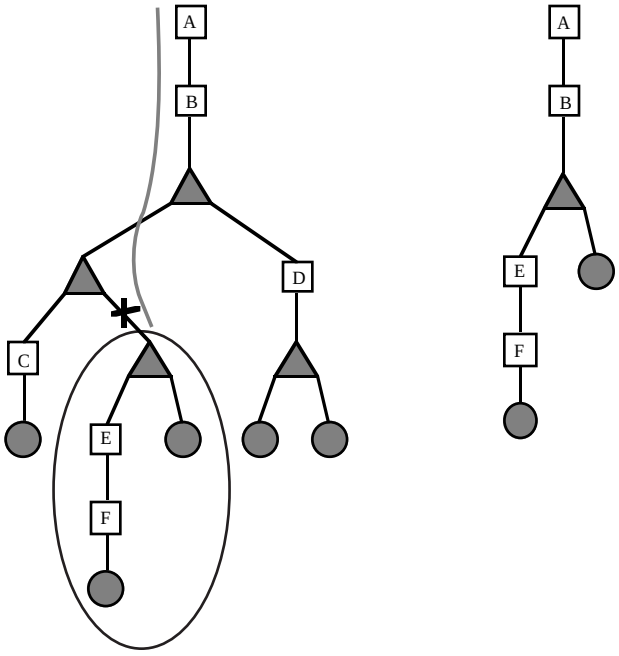


FIGURE G.4
TreeStack Branch operation

G.2.4. Select subtree

Subtree selection indicates a substructure of the TreeStack, which contains all the equivalent alternates of the children of the selected internal node of the structure. In a conventional tree a subtree is selected. In a TreeStack, a selection (i.e., the **X** in Figure G.5) indicates the extraction of the superior tree (i.e., children, indicated by the dotted oval), along with a path of unary nodes back to the root (the dotted path). This path denotes the recursion return information encoded in the TreeStack, such that the resulting extracted substructure is self-contained. The cost of a Branch is between $O(\log(N))$ and $O(N)$, where N is the number of overall elements in the tree, depending on whether the TreeStack is dominated by (respectively) binary or unary nodes.



Subtree selection indicated *Subtree selection extracted*

FIGURE G.5
TreeStack Subtree Selection operation

G.2.5. Equivalence transforms - Twinning and UnTwinning

It is possible that a TreeStack structure evolves to a state where a Pop cannot be executed, even though a corresponding Push exists. Consider the case where a Push is followed by a Branch. The subsequent Pop on either leaf of the binary node is not defined. There is an equivalence between TreeStack substructures which permits a Pop to occur, given some intermediate transformations.

A unary node whose child is a binary node is equivalent to a binary node whose children are two identical copies of that unary node (Figure G.6). Conversion from the unary node/binary child to a binary node/pair of unary children is called ***Twinning***, and the reverse transformation (where possible) is called ***UnTwinning***. The cost of either transform is $O(1)$.

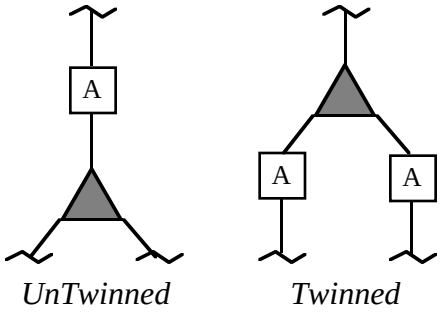


FIGURE G.6
TreeStack Twinning and UnTwinning

G.2.6. Canonical forms

There are two canonical forms of the TreeStack, one that is maximally Twinned, the other that is maximally UnTwinned. A Twinned TreeStack is efficient at Push and Pop operations, but has a high Branch cost and is inefficient in its space requirements, whereas an UnTwinned Stack is efficient in its use of space, Push, and Branch operations, but costly for Pop operations. The costs for Subtree Selection are equivalent in either form.

G.2.6.1. Twinned TreeStack

The result of Twinning is to move the binary nodes closer to the root, eventually becoming a tree of binary nodes originating at the root. The ‘leaves’ of the binary node tree are strings of unary nodes, terminating in leaves. The binary tree structure represents an encoding of the choices of alternates, i.e., alternate worlds, and the unary strings represent a conventional stack local to each world. Thus the composite structure of the TreeStack can be transformed into a conventional tree of conventional stacks (Figure G.7). The cost of transforming an arbitrary TreeStack to canonical form is at most $O\left(2^N\right)$, i.e., if the original structure is a stack of trees.

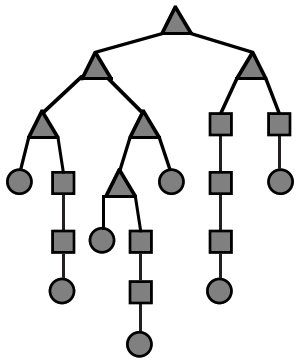


FIGURE G.7
TreeStack canonical form - maximally Twinned

A Twinned TreeStack is inefficient in its use of space, because stack elements are replicated among the tree alternates, to maintain strict independence among the alternates. A Pop operation costs $O(1)$ because Twinning transformations are not required at the time of the Pop, but Branch operations are very costly, i.e., $O\left(2^k\right)$ where k is the depth of the individual stacks of the alternates. In that case the entire recursion stack must be replicated in its entirety, even though much of it may never be accessed independently of other alternates.

G.2.6.2. UnTwinned TreeStack

Twinning is costly, because stack components may be unnecessarily replicated. In the case where the operations consist of a sequence of Pushes, followed by a sequence of Branches, the structure becomes a stack whose top element is a tree. Conversion of this structure to canonical form replicates elements exponentially in the depth of the stack. If

the structure is left in original form, replication of internal unary nodes occurs only where Pops require them, and so some economy of time and space is achieved. UnTwinning (to the extent possible) decreases the cost of access and storage of the TreeStack (Figure G.8).

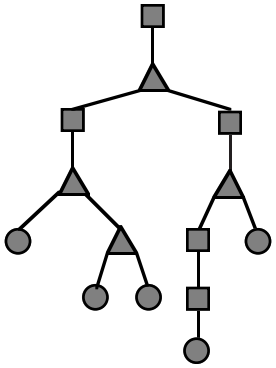


FIGURE G.8
TreeStack form - maximally UnTwinned

An UnTwinned TreeStack is efficient in its use of space, because the stacks of overlapping alternates are maintained as a single structure until Twinning is required. Branch operations cost $O(1)$, but Pop operations cost $O\left(2^k\right)$ where k is the depth of the path back to the root, because of the Twinning required back along that path.

G.2.7. The Graft transform

One optimization uses an alternate storage method. In the TreeStack described thus far, a stack whose last element is a tree can cause unnecessary replication when one leaf of the tree Pops multiple levels of recursion, but where the other leaves do not Pop (Figure G.9). In the equivalence transformation, a set of k Pops would cost $O\left(2^k\right)$ in space and time replication (Figure G.10). Space optimization would suggest that the equivalent structure that was not accessed should be UnTwinned (Figure G.11).

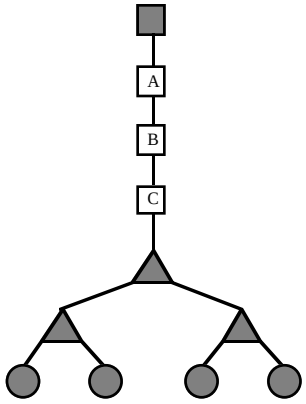


FIGURE G.9
Multiple Pops in max-Twinned TreeStack (before Pops)

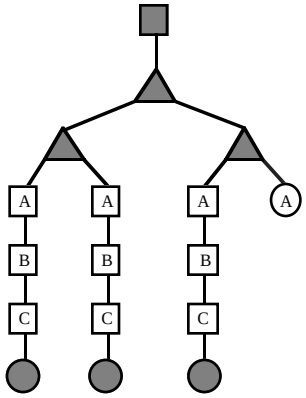


FIGURE G.10
Multiple Pops after Twinning and Pops

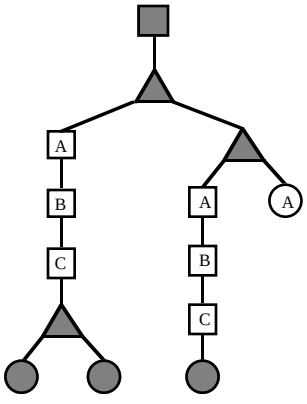


FIGURE G.11
Multiple Pops after subsequent UnTwinning

Rather than performing the transformations only to undo some portion of them later, we have developed an equivalent encoding for a Pop through binary nodes, called a Graft. A leaf of an arbitrary TreeStack is Popped by traversing the TreeStack back towards the root, through binary nodes, stopping at the first unary node found. That node is then preceded by a binary node, whose one child is the unary node, and whose other child is a leaf copied from the data of the unary node (Figure G.12). The leaf where the Pop occurred is considered a null unary node (lined out in the figure), whose child is the leaf thus grafted in (the arrow in the figure). The cost of a Graft is $O(\log(N))$, due to the search for the unary node.

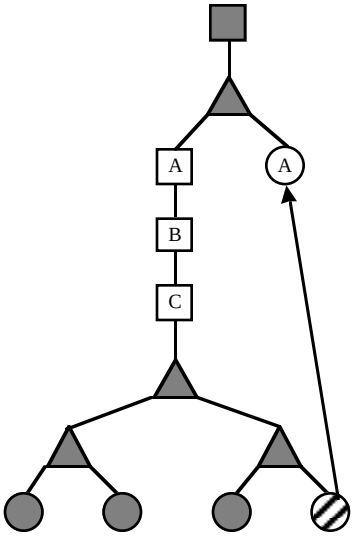
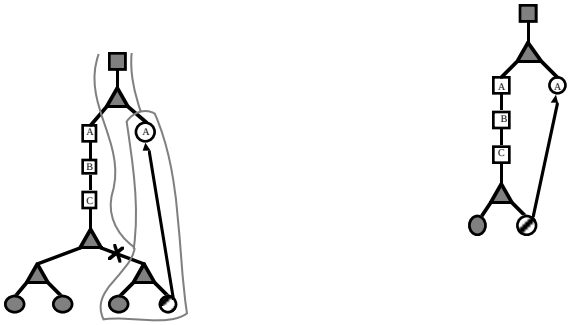


FIGURE G.12
A Graft

The Graft has the advantage of permitting a Pop to occur while retaining the original branching structure that led to the leaf that popped, so that a Select Subtree can choose a subtree within that popped structure. The null unary node permits the grafted leaf to appear as a child within the selected subtree, even though that leaf is actually an alternate on a path back to the root that would normally have been removed in the subtree selection (Figure G.13). The disadvantage is that the entire tree remains after all leaves have popped, thus relying on the Select Subtree operation to prune the TreeStack structure down to its currently accessible elements. We have not analyzed the increased cost of a Grafted TreeStack subtree selection operation, or the reduction in cost due to the omission of equivalence transformations.



Graft with selection indicated *Extracted subtree selection*

FIGURE G.13
A Graft Subtree Selection

APPENDIX H

μ–Scope Methods

Measurements were required to analyze the μ-Net variations, to specify parameters of the branching stream model and to determine the requirements for reasonable implementations. These measurements included limb length, recursion depth, and opcode type distributions. Existing tools performed only some of these measurements, so a method was devised to perform all the required measurements in a single environment.

H.1. Existing Tools

There are several tools that could have been used for some of these measurements. They can be grouped into four basic classes, as listed below:

- Hardware
- Software emulation
- Block tagging
- Opcode interleaving

Hardware methods were considered initially for these measurements. Most current microprocessors provide sufficient internal status on external pins that opcodes can be distinguished from other information on the data bus. A device can be designed which monitors the bus and these signal pins, and increments local counters as required. A design that does not affect performance can be implemented with existing PLA technology. The advantage to this method is that all opcodes executed are measured, the measurements are done in real time, and any executable code can be measured. The disadvantage was that our existing hardware base did not support student hardware access, and that reasonable software schemes would be more portable and more quickly implemented.

Software emulation was not possible, because of the size of the benchmarks to be tested. Software emulation reduces execution time by a factor of 50-200 (approximately), and some benchmarks used required several minutes of direct execution time. Emulation would have required hours of CPU time, which was not available here. Further, measurements we required would have necessitated modification of any existing emulator, or the design of an entire new emulator. The former was precluded by the lack of access to the source code of commercial emulators, such as SHADOW[1], and the latter was precluded by time limitations.

Current opcode measurement techniques have focused on block tagging methods. In these schemes, basic blocks are determined by a preprocessor, and opcodes are inserted which measure the execution of the basic blocks only. After execution of the tagged code, statistics on individual opcode types are computed from block execution statistics and the opcode listings of each basic block. This technique adds only 10-20% to the execution time and code size of the measured program. Block tagging measurement tools include PIXIE[2] and SPIXTOOLS[3]; the former was not used because we had only one DECSTATION 5100[4] computer, and the latter was not publicly available[5].

---

[1]SHADOW is a SPARC emulator of SUN MICROSYSTEMS. We requested access to SHADOW from SUN in Feb. 1991, either in source or executable only form, and were told that SHADOW is a commercial product, and so the source code was not available, even for educational uses with our offered nondisclosure.

[2]PIXIE is a DIGITAL EQUIPMENT CORPORATION program which does block tagging on MIPS object code.

Another version of block tagging is called Abstract Execution (AE) [La90]. AE both saves execution time of the measured code, adding 50-80% to the execution time, and compresses measured data by several orders of magnitude. In AE, 'interesting' events are measured during an initial execution of the program. This trace is used to direct more detailed measurement of 'interesting' portions of the original program; the final measurements are scaled according to the trace proportions to produce the final output. AE thus focuses detailed measurements on statistically significant components of the original program. It was not used here because opcode interleaving provided sufficient measurements with tolerable degradation. AE's major benefit is that unlikely portions of the source code don't waste measurement resources, and that huge amounts of measured data are compressed. We are not concerned with resource optimization, and do not measure data address access, so we do not require trace data compression.

Block tagging was insufficient to perform the measurements we require. Block delineation is performed by static code analysis, where destinations of branch, jump, and call opcodes determine block beginnings, and any control transfer (branch, call, jump, or return) determines a block end. Dynamic opcodes are not analyzed in this method, because the destination of dynamic control transfers (call, jump, branch) cannot be predicted before runtime without additional assumptions. Further, the basic block defines the limb length; its measurement cannot be sufficiently modified to accommodate basic blocks whose interiors contain jumps, calls, or returns, as some of our measurements required.

Instruction interleaving was the final classification of measurement methods. Interleaving can be static, i.e., by modification a-priori of the object code, or dynamically, using software interrupts.

---

[3]SPIXTOOLS is a program of SUN MICROSYSTEMS, which performs block tagging on SPARC object code.

[4]The DECSTATION 5100 is a product of DIGITAL EQUIPMENT CORPORATION.

[5]Again, SUN MICROSYSTEMS was contacted in Feb. 1991 concerning access to SPIXTOOLS for these experiments. We were told that SPIXTOOLS had not yet been released as a product, and would not be available, even under a nondisclosure agreement, for our research. Because of 'proprietary access', we felt it inappropriate to compare our results to even published measurements made with this inaccessible tool, since such experiments are by definition "not repeatable."

Software interrupts can be used dynamically to interleave the opcode stream with execution of the measurement code. In a SPARC the dual instruction pointers can be used to replace the next opcode with a trap instruction, where the software interrupt caused by the trap causes the measurement code to execute. This method also slows execution by two orders of magnitude.

We chose static instruction interleaving on a Sun SPARCstation, due to accessibility of the workstation (five were used for these experiments). The method was chosen to permit measurement of limb length statistics under various definitions of basic block delineations, and to perform more conventional opcode occurrence statistics in a dynamic instruction sequence.

H.2. The method

μ–Scope (i.e., MicroScope) is a sequence of steps that performs dynamic opcode traces on a SPARC CPU. The current method is designed any language, provided that intermediate SPARC assembler code is accessible by μ–Scope.

If the source code is in C, the MSCOPE.h C-language file is included at the beginning of the source code, to prevent warning messages; equivalent assembler directives could have been determined from this C-code and included in the final assembler output.

The SPARC assembler output is passed through two AWK scripts. The first script, INSERT_SYMBOLS, adds temporary placeholders for various measured opcode types and limb sequences. Every type of opcode measured is followed by a placeholder. Limb lengths are measured by inserting limb length calculation placeholders whenever an opcode occurs which terminates a limb of a particular type. For example, branches cause all types of limbs to terminate, whereas returns cause only some categories of limb measurements to indicate a limb end. The placeholders are pseudocodes for operations, which include increment, increment by 2, decrement, and index; the latter pseudocode increments a value at some offset in an array (with bounds checking).

The second script, INSERT_OPCODES, replaces inserted placeholders with SPARC assembler that tabulates measurements as indicated. Opcode type occurrences cause global counters to increment, whereas limb length occurrences cause an increment in a histogram of limb lengths of the type of limb terminated, and a resetting of that limb

length counter. The limb counters allow limb lengths to be calculated across calls, subroutines, and jumps, as some types of limb measurements required. Opcode type occurrences add 3 opcodes for every opcode occurrence; limb occurrences add up to 16 opcodes for each limb end occurrence. In the benchmarks we measured, these two scripts added an average of 7 to 10 opcodes for each original opcode, i.e., the object code was increased by 7-10x, and execution time was similarly increased.

This method of opcode interleaving was proposed independently, but is a specific example of a general technique of source code debugging and measurement. We noted an identical method published concurrently with our development [Ka91]; we then contacted the authors of this work, who provided us with extensive source of their tools. These tools and AWK scripts were used to debug and verify our techniques. Our scripts are believed more efficient and compact, and perform types of measurements of particular interest herein, whereas their scripts measure other data not required here. We use their compensation for SPARC “annulled” instructions in our method.

The final SPARC assembler code contains all measurements code inline with the original opcodes, and a final call to the output routine is also included. This routine is written in C-language code (MSCOPE.c), and is compiled separately, and linked after the assembly of the SPARC code. It computes and outputs the statistics in the desired format.

Other files developed included an AWK script to measure the static opcode distributions of SPARC assembler (STATIC_COUNT), and various scripts to take given dynamic and static output files and merge the statistics. These latter scripts were written on a personal computer, in a script not suitable for linear ASCII output, and are not included here.

H.3. Observations

There were several observations made in the process of the development of μ–Scope. First, the entire technique required approximately two weeks of programming, and the resulting measurements increased static code sizes and execution times each by 7-10x. The largest file was the GNU C compiler, which took approximately 10 hours of SPARC CPU time on our workstations, which was reduced to 2.5 hours by distribution of the process across 5 CPUs (parallelization was not evenly distributed due to quantization of the compilation phases). The original C compiler object code was 1 Megabyte in size, whereas the self-measuring object code was 8 Megabytes.

Static opcode distributions were compared to dynamic opcode distributions, and were nearly equivalent. We attribute this equivalence to size of the selected benchmarks, such that initialization code did not unduly skew the static distributions in comparison, and to the uniformity of the dynamic executions in traversing a uniform distribution of component blocks of static opcodes.

Indirect opcodes occurred less than 0.3% in the dynamic opcode measurements, and not at all in some of the benchmarks. Inspection of the SPARC assembler code indicated that the GNU C compiler (an unmodified version of which is the usual compiler on our systems) generated indirect opcodes only where C source code required a subroutine to return a C-language data structure rather than a conventional predefined C-language data type. Indirect calls were used there; the SPARC has no indirect branches, and indirect jumps were not found in any benchmark.

The UNIMP SPARC opcode, representing unimplemented instructions, was also generated by the compiler in response to the same condition. The UNIMP instruction was inserted to force a software interrupt when the destination of the indirect instruction failed to modify the executable code; this device permitted detection of the use of a non-standard C-language feature in environments that may not have supported it. A sufficient supporting environment would overwrite the UNIMP instruction, thus preventing the interrupt, whereas a deficient environment would result in a software failure.

Finally, the use of “annulled” instructions complicated the design of μ–Scope substantially. These architectural methods, along with internal pipelining, inhibit measurements of this kind. Future architectures that include hardware for direct

measurement of opcode streams would be useful, and have been proposed, but unfortunately have not affected consumer-level CPU designs.

H.4. AWK scripts and C-language code listings

The following are the AWK scripts and C-language code listings. They are available from the author for research use.

H.4.1. MSCOPE.h - C-language 'include' file

```
/* Joe Touch 5/11/91 - added for tracing */

#include <stdio.h>

extern long JTbranch;
extern long JTjump;
extern long JTret;
extern long JTcall;
extern long JTjumpI;
extern long JTcallI;
extern long JTreccall;
extern long JTtotal;
extern long JToldall;
extern long JToldbcr;
extern long JToldb;
extern long JTlinearall[1024];
extern long JTlinearbcr[1024];
extern long JTlinearb[1024];
extern long JTcalldepth;
extern long JTcallall[256];
extern long JTretall[256];

/* Joe Touch -end */
```

H.4.2. INSERT_SYMBOLS - AWK, inserts signals in SPARC assembler

```
# branches (always direct)

/^    b((n?(e|z))|((g|l)e?u?)|((c|v)(c|s))|(n|pos)|(neg)),a / {
      branch = 1;
      annul = 1;
      }
/^    b((n?(e|z))|((g|l)e?u?)|((c|v)(c|s))|(n|pos)|(neg))   / {
      branch = 1;
      }
```

```
/^    fb((u?(g|l)e?)|(n?(e|z))|(ue?)|(n|o|lg)),a      / {
      branch = 1;
      annul = 1;
      }
/^    fb((u?(g|l)e?)|(n?(e|z))|(ue?)|(n|o|lg))  / {
      branch = 1;
      }
/^    cb((0?1?2?3?)|(n)),a    / {
      branch = 1;
      annul = 1;
      }
/^    cb((0?1?2?3?)|(n))      / {
      branch = 1;
      }

# jumps

/^    ((jmp)|((f|c)?)b(a?)),a / {
      jump = 1;
      ba_annul = 1;
      }
/^    ((jmp)|((f|c)?)b(a?))   / {
      jump = 1;
      }

# calls

/^    (call|jumpl)      / {
      call = 1;
      }
#/\#PROLOGUE\# 0/ {
#     local_call = 1;
#     }

# returns
/^    ret((l|t)?)/      {
      ret = 1;
      }
/^    call  \.stret4/   {
      call = 0; # cancel call
      ret = 1; # treat as a return
      }

/\.proc/ {
      new_proc = 1;
# insert local call code after the next label
      }

(new_proc == 1) && /.+\:/ {
      new_proc = 0;
      print $0;
      printf "JT_OP inc _JTreccall\n";
      printf "JT_OP inc _JTcalldepth\n";
      printf "JT_INDEX _JTcalldepth ZERO _JTcallall 0xFF\n";
      next;
      }
```

```
# check for indirect calls and jumps

((jump == 1) || (call == 1)) {
     if ($0 ~ /\%[gGlLiIoO]?[0-9]+.*/)
          indirect = 1;
     }

# use the following flags instead of repeating elaborate conditionals

(branch == 1) || (call == 1) || (ret == 1) || (jump == 1) {
     delaynext = 1;
     }

(branch == 1) || (call == 1) || (ret == 1) || (jump == 1) {
     resetALL = 1;
     }

(branch == 1) || (call == 1) || (ret == 1) || (indirect == 1) {
     resetBCR = 1;
     }

(branch == 1) || (indirect == 1)) {
     resetB = 1;
     }

# modes - branch, call, jump, ret, annul, delaynext, delayed, etc.

# early abort

/    unimp /     {
     print $0;
     next;
     }

($0 !~ /^  ?!/) && ($0 !~ /:/) &&
($0 !~ /^( |( ))*\./) && ($0 !~ /\=/) {
     if (delayed == 1) {
          print $0;
          if (annul == 1)
               printf "JT_OP dec _JTtotal\n";
          if (ba_annul == 1)
               printf "JT_OP inc _JTtotal\n";
          ba_annul = 0;
          annul = 0;
          }
     if (branch == 1) {
          printf "JT_OP inc _JTbranch\n";
          }
     if (jump == 1) {
          if (indirect == 1) {
               printf "JT_OP inc _JTjumpI\n";
               }
          printf "JT_OP inc _JTjump\n";
          }
```

```
      if (call == 1) {
            if (indirect == 1) {
                  printf "JT_OP inc _JTcallI\n";
                  }
            printf "JT_OP inc _JTcall\n";
            }
      if (local_call == 1) {
            printf "JT_OP inc _JTreccall\n";
            printf "JT_OP inc _JTcalldepth\n";
            printf "JT_INDEX _JTcalldepth ZERO _JTcallall 0xFF\n";
            }
      if (ret == 1) {
            printf "JT_OP inc _JTret\n";
            printf "JT_OP dec _JTcalldepth\n";
            printf "JT_INDEX _JTcalldepth ZERO _JTretall 0xFF\n";
            }
      if (resetALL == 1) {
            printf "JT_INDEX _JTtotal _JToldall _JTlinearall 0x3FF\n";
            resetALL = 0;
            }
      if (resetBCR == 1) {
            printf "JT_INDEX _JTtotal _JToldbcr _JTlinearbcr 0x3FF\n";
            resetBCR = 0;
            }
      if (resetB == 1) {
            printf "JT_INDEX _JTtotal _JToldb _JTlinearb 0x3FF\n";
            resetB = 0;
            }
      if (delayed != 1) {
            if ((delaynext == 1) && (ba_annul == 0))
                  printf "JT_OP add2 _JTtotal\n"
            else
                  printf "JT_OP inc _JTtotal\n";
            print $0;
            }
      delayed = delaynext;
      delaynext = 0;
      branch = 0;
      call = 0;
      jump = 0;
      ret = 0;
      local_call = 0;
      indirect = 0;
      next
      }

# take care of comments, assembler directives...
      {
      print $0
      }
```

H.4.3. INSERT_OPCODES - AWK, signals become SPARC assembler

```
/JT_OP/    {
# JT_OP $2=(inc|dec|add2) $3=counter
     printf "\tsethi\t%%hi(%s),%%g7\n",$3;
     printf "\tld\t[%%g7+%%lo(%s)],%%g6\n",$3;
     if ($2 ~ /inc/)
           printf "\tinc\t%%g6\n"
     else if ($2 ~ /dec/)
           printf "\tdec\t%%g6\n"
     else if ($2 ~ /add2/)
           printf "\tadd\t%%g6,0x2,%%g6\n";
     printf "\tst\t%%g6,[%%g7+%%lo(%s)]\n",$3;
     next
     }

/JT_INDEX/ {
# JT_INDEX $2=current $3=old $4=array $5=boundmask
# insert code to increment the offset based on the current totals
     printf "\tsethi\t%%hi(%s),%%g7\n",$2;
     printf "\tld\t[%%g7+%%lo(%s)],%%g6\n",$2;
# suntract the previous value, if not ZERO
     if (!($3 ~ /ZERO/)) {
           printf "\tsethi\t%%hi(%s),%%g7\n",$3;
           printf "\tld\t[%%g7+%%lo(%s)],%%g7\n",$3;
           printf "\tsub\t%%g6,%%g7,%%g6\n";
     }
# mask out the bound
     printf "\tand\t%%g6,%s,%%g6\n",$5;
# multiply by the size of a long int
     printf "\tsll\t%%g6,0x2,%%g6\n";
# g6 now has the length - add the base
     printf "\tset\t%s,%%g7\n",$4;
     printf "\tadd\t%%g7,%%g6,%%g7\n";
# g7 now has the index location - increment it
     printf "\tld\t[%%g7],%%g6\n";
     printf "\tinc\t%%g6\n";
     printf "\tst\t%%g6,[%%g7]\n";
# insert code to reset the old total to the new total
# do if prev value not ZERO
     if (!($3 ~ /ZERO/)) {
           printf "\tsethi\t%%hi(%s),%%g7\n",$2;
           printf "\tld\t[%%g7+%%lo(%s)],%%g6\n",$2;
           printf "\tsethi\t%%hi(%s),%%g7\n",$3;
           printf "\tst\t%%g6,[%%g7+%%lo(%s)]\n",$3;
     }
     next
     }

     {
     print $0
     }
```

H.4.4. MSCOPE.c code - output desired statistics

```
/* Joe Touch 5/11/91 - added for tracing */

#include <stdio.h>

long JTbranch = 0;
long JTjump = 0;
long JTret = 0;
long JTcall = 0; /* all calls */
long JTreccall = 0; /* calls counted in recursion*/
long JTjumpI;
long JTcallI = 0;
long JTtotal = 0;
long JToldall = 0;
long JToldbcr = 0;
long JToldb = 0;
long JTlinearall[1024];
long JTlinearbcr[1024];
long JTlinearb[1024];
long JTcalldepth = 0;
long JTcallall[256];
long JTretall[256];

/* call with a string naming a file to be dumped into */
extern JT_dump();

/* Joe Touch -end */

/* Joe Touch - 5/11/91 added for tracing */

#include "JT.extern.h"

JT_dump(filename)
char *filename;
     {
      FILE *outfile;
      int i;
      double allsum, bcrsum, bsum;
      double psumall = 0, psumbcr = 0, psumb = 0;
      double callpsum = 0, retpsum = 0, retsum, callsum;

      outfile = fopen(filename,"w");

      bsum = JTbranch + JTjumpI + JTcallI;
      bcrsum = JTbranch + JTcall + JTret + JTjumpI;
      allsum = JTbranch + JTcall + JTret + JTjump;
#define JT_PERCENT(x) (((double)(x)) * 100.0 / ((double)(JTtotal)))

      fprintf(outfile,"DYNAMIC CODE MEASUREMENTS\n");
      fprintf(outfile,"Branches = %10d\t%7.3f%%\n",
           JTbranch,JT_PERCENT(JTbranch));
      fprintf(outfile,"Jumps = %10d\t%7.3f%%\n",
           JTjump,JT_PERCENT(JTjump));
      fprintf(outfile,"I- Jumps = %10d\t%7.3f%%\n",
           JTjumpI,JT_PERCENT(JTjumpI));
```

```
fprintf(outfile,"Calls = %10d\t%7.3f%%\n",
     JTcall,JT_PERCENT(JTcall));
fprintf(outfile,"I - Calls = %10d\t%7.3f%%\n",
     JTcallI,JT_PERCENT(JTcallI));
fprintf(outfile,"Rec Calls = %10d\t%7.3f%%\n",
     JTreccall,JT_PERCENT(JTreccall));
fprintf(outfile,"Returns = %10d\t%7.3f%%\n",
     JTret,JT_PERCENT(JTret));
fprintf(outfile,"Others = %10d\t%7.3f%%\n",
     JTtotal - (JTbranch + JTcall + JTret + JTjump),
     JT_PERCENT(JTtotal - (JTbranch + JTcall + JTret + JTjump)));
fprintf(outfile,"INDIRECT = %10d\t%7.3f%%\n",
     JTjumpI + JTcallI,JT_PERCENT(JTjumpI + JTcallI));
fprintf(outfile,"- ALL = %10d\t%7.3f%%\n",
     JTbranch + JTcall + JTret + JTjump,
     JT_PERCENT(JTbranch + JTcall + JTret + JTjump));
fprintf(outfile,"- BCRi = %10d\t%7.3f%%\n",
     JTbranch + JTcall + JTret + JTjumpI,
     JT_PERCENT(JTbranch + JTcall + JTret + JTjumpI));
fprintf(outfile,"- Bi = %10d\t%7.3f%%\n",
     JTbranch + JTcallI + JTjumpI,
     JT_PERCENT(JTbranch + JTcallI + JTjumpI));
fprintf(outfile,"Total = %10d\n",JTtotal);


if (allsum == 0.0)
allsum = 1e8;
if (bcrsum == 0.0)
bcrsum = 1e8;
if (bsum == 0.0)
bsum = 1e8;
callsum = JTreccall;
retsum = JTret;
for (i=0; i<255; i++) {
callpsum += JTcallall[i];
retpsum += JTretall[i];
}
fprintf(outfile,"depth partial sum - number greater/eq\n");
fprintf(outfile,"DEPTH\tCALL\tRET\tcall%%\tret%%\n");
for (i=0; i<255; i++) {
fprintf(outfile,"%d\t%d\t%d\t%5.1f\t%5.1f\n",i,
      JTcallall[i], JTretall[i],
      100 * callpsum / callsum,
      100 * retpsum / retsum);
callpsum -= JTcallall[i];
retpsum -= JTretall[i];
}

fprintf(outfile,"\n\nlen partial sum - number less/eq\n");
fprintf(outfile,"Len\tALL\tBCR\tB\tALL%%\tBCR%%\tB%%\n");
for (i=0; i< 1024; i++) {
psumall += JTlinearall[i];
psumbcr += JTlinearbcr[i];
psumb += JTlinearb[i];
```

```
       fprintf(outfile,"%d\t%d\t%d\t%d\t%5.1f\t%5.1f\t%5.1f\n",i,
             JTlinearall[i],JTlinearbcr[i],JTlinearb[i],
             100.0 * psumall / allsum,
             100.0 * psumbcr / bcrsum,
             100.0 * psumb / bsum);
       }
       fclose(outfile);
      }

/* Joe Touch end */
```

H.4.5. STATIC_COUNT - get static SPARC opcode distributions

```
# branches - ALL SPARC branches are direct
/    b((n?(e|z))|((g|l)e?u?)|((c|v)(c|s))|(n|pos)|(neg))(,a)?    / {
           branch++
           }
/    fb((u?(g|l)e?)|(n?(e|z))|(ue?)|(n|o|lg))(,a)?   / {
           branch++
           }
/    cb((0?1?2?3?)|(n))(,a)? / {
     branch++
     }

# jumps can be direct or indirect

/    ((jmp)|((f|c)?)b(a?))(,a)?    / {
     jump++;
     if ($0 ~ /\%[gGlLiIoO]?[0-9]+.*/) {
           i_jump++;
           print "I-JUMP: ",$0
           }
     }

# calls can be direct or indirect

/    (call|jumpl)      /     {
     call++;
     if ($0 ~ /\%[gGlLiIoO]?[0-9]+.*/) {
           i_call++;
           print "I-CALL: ",$0
           }
     }

# returns

/    ret((l|t)?)/      {
     ret++;
     }

($0 !~ /^!/) && ($0 !~ /:$/) && ($0 !~ /^\./) {
     total++;
     }
```

```
END   {
      if (total != 0) {
            all = branch + jump + call + ret;
            bcr = branch + call + ret + i_jump;
            b = branch + i_jump + i_call;
            indirect = i_jump + i_call;
            other = total - all;
            printf "STATIC CODE MEASUREMENTS\n"
            printf "Branches =\t%10d\t%7.3f%%\n",\
                  branch,branch/total*100;
            printf "Jumps =\t%10d\t%7.3f%%\n",\
                  jump,jump/total*100;
            printf "I- Jumps =\t%10d\t%7.3f%%\n",\
                  i_jump,i_jump/total*100;
            printf "Calls =\t%10d\t%7.3f%%\n",\
                  call,call/total*100;
            printf "I - Calls =\t%10d\t%7.3f%%\n",\
                  i_call,i_call/total*100;
            printf "Returns =\t%10d\t%7.3f%%\n",\
                  ret,ret/total*100;
            printf "Others =\t%10d\t%7.3f%%\n",\
                  other,other/total*100;
            printf "INDIRECT =\t%10d\t%7.3f%%\n",\
                  indirect,indirect/total*100;
            printf "- ALL =\t%10d\t%7.3f%%\n",\
                  all,all/total*100;
            printf "- BCRi =\t%10d\t%7.3f%%\n",\
                  bcr,bcr/total*100;
            printf "- Bi =\t%10d\t%7.3f%%\n",\
                  b,b/total*100;
            printf "TOTAL =\t%10d\n",total;
            }
      }
```