AD-A044 313 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
ANALYSIS OF MEMORY ADDRESSING ARCHITECTURE.(U)
JUL 77 D W HAMMERSTROM DAAB07-72-C-0259
UNCLASSIFIED R-777 NL

1 OF 2
AD A044313

REPORT R-777 JULY, 1977 UILU-ENG 77-2224

CSL COORDINATED SCIENCE LABORATORY

12

ANALYSIS OF MEMORY ADDRESSING ARCHITECTURE

DANIEL WAYNE HAMMERSTROM

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) ANALYSIS OF MEMORY ADDRESSING ARCHITECTURE,		5. TYPE OF REPORT & PERIOD COVERED (7) Technical Report,
		6. PERFORMING ORG. REPORT NUMBER (14) R-777, UILU-ENG-77-2224
7. AUTHOR(s) (10) Daniel Wayne Hammerstrom		8. CONTRACT OR GRANT NUMBER(s) (15) DAAB-07-72-C-0259, VNSF-MCS-73-03488 A01
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program		(11) 12. REPORT DATE July, 1977 13. NUMBER OF PAGES 137
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 151 p.		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

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

18. SUPPLEMENTARY NOTES

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

Intelligent Memory
Information Content
Addressing Architecture

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

This research investigates the process of generating memory addresses within computer programs and evaluates methods for making that process more efficient.

The approach taken here is to separate program execution into a computation process and an addressing process. The addressing process generates the memory reference stream for the computation process (and incidentally for itself as well). The memory reference stream of the computation process is then modeled probabilistically and its information content derived. It is shown that the addressing overhead, A, is lower bounded by the information content, H, of the

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

097 700

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

computation process memory reference stream. Techniques are also presented for estimating A and H from traced program execution. The estimation results indicate that $\tilde{A}$ (A estimate) is much greater than $\tilde{H}$ (H estimate) indicating that not only does the addressing process place a great load on the system, but that much of that load is unnecessary.

Several techniques are presented for improving addressing efficiency with standard computer architectures. Significant savings can be achieved here. It was concluded, however, that the greatest reductions in $\tilde{A}$ occur by using second order memories. These memories are defined and their performance analyzed. Using such a memory not only reduces $\tilde{A}$ to approximately $\tilde{H}$, but also has the additional advantage of significantly reducing the bandwidth requirements between the CPU and the memory system.

UILU-ENG 77-2224

ANALYSIS OF MEMORY ADDRESSING ARCHITECTURE

by

Daniel Wayne Hammerstrom

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259 and in part by the National Science Foundation under Grant MCS 73-03488 A01.

ANALYSIS OF MEMORY ADDRESSING ARCHITECTURE

BY

DANIEL WAYNE HAMMERSTROM

B.S., Montana State University, 1971
M.S., Stanford University, 1972

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Thesis Adviser: Professor Edward S. Davidson

Urbana, Illinois

ANALYSIS OF MEMORY ADDRESSING ARCHITECTURE

Daniel Wayne Hammerstrom, Ph.D.
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1977

This research investigates the process of generating memory addresses within computer programs and evaluates methods for making that process more efficient.

The approach taken here is to separate program execution into a computation process and an addressing process. The addressing process generates the memory reference stream for the computation process (and incidentally for itself as well). The memory reference stream of the computation process is then modeled probabilistically and its information content derived. It is shown that the addressing overhead, A, is lower bounded by the information content, H, of the computation process memory reference stream. Techniques are also presented for estimating A and H from traced program execution. The estimation results indicate that $\tilde{A}$ (A estimate) is much greater than $\tilde{H}$ (H estimate) indicating that not only does the addressing process place a great load on the system, but that much of that load is unnecessary.

Several techniques are presented for improving addressing efficiency with standard computer architectures. Significant savings can be achieved here. It was concluded, however, that the greatest reductions in $\tilde{A}$ occur by using second order memories. These memories are defined and

their performance analyzed. Using such a memory not only reduces $\tilde{A}$ to approximately $\tilde{H}$, but also has the additional advantage of significantly reducing the bandwidth requirements between the CPU and the memory system.

ACKNOWLEDGMENT

The author wishes to express his deepest and heartfelt gratitude to his advisor Professor Edward S. Davidson. Professor Davidson's constant patience and insightful guidance were necessary ingredients to this dissertation.

The author also wishes to thank his colleagues at the Coordinated Science Laboratory; Faye Briggs, Rob Budzinski, Joel Emer, Alan Gant, Bill Kaminsky, Balasubramanian Kumar, Trevor Mudge, and Janak Patel for providing an intellectually stimulating environment. Special thanks are also in order to Professor Richard A. Flower whose timely suggestions contributed greatly to this research.

The author, in addition, wishes to thank Ms. Rose Harris for the immaculate typing of this report.

The author especially wishes to acknowledge his wife, Kathy, and sons, Johan and Erik, without whose love and devotion this dissertation would not have been possible.

The author also wishes to thank his parents for their encouragement and patience.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Problem Statement and Objectives

Many researchers have concluded that only a portion of what a computer system does contributes directly to the desired computation. The rest is overhead required of the program to work around awkward hardware. This situation has come about, in part, because of a lack of understanding of the behavior of computer programs and the relationship of that behavior to the system hardware. Essentially, many areas of computer design are approached in an ad hoc manner. One area where this is true is that of CPU/memory communication (including address generation). Since considerable overhead goes into managing this communication, it is desirable to analyze or characterize the essential characteristics of the address generation process more precisely. For example, such characterization is important in the realm of Large Scale Integration (LSI), where tremendous functional capability can exist within a single integrated circuit, but the communication bandwidth between circuits, e.g. CPU and memory, is extremely limited.

In this dissertation the area of CPU/memory communication is examined. In particular, methods are presented for deriving the information content of the CPU/memory reference stream. The information content is then used as a measure of the efficiency of the addressing overhead incurred by a particular CPU/memory addressing architecture. Our principal objectives are to characterize CPU memory referencing behavior

and the addressing process within the CPU that implements the memory reference stream, and to discover computer architectures which perform this memory referencing function much more efficiently.

1.2. Background

Although there is no research known to us in the area of analyzing general purpose computer system addressing and memory referencing, there has been some work done in applying information theoretic concepts to other aspects of computer system evaluation. Constantine [1] has briefly examined the information content of a typical computer word. Hehner [2], and Foster and Gonter [3] have looked at improved opcode encoding, taking advantage of the highly sequential aspects of the opcode stream. Hehner has shown that by using his techniques, program size can be reduced by as much as 75% in some cases. Clark [4] has used frequency based encoding to optimize LIST data structures to minimize access times.

All these researchers have reached the conclusion that, in most of today's general purpose computing systems, significant compaction of program storage space in memory and significant reductions in program execution time can be achieved by more efficient coding and/or improved system architecture.

1.3. Plan of Development

In Chapter 2 an executing program is shown to consist of an addressing process and a computation process. From the addressing process

concept, the addressing overhead and addressing bandwidth requirements can be formally defined. Also the information theoretic concept of nth order entropy is applied to analyze the memory reference stream of the computation process.

It is shown that theoretically this entropy is a lower bound for the addressing overhead and is therefore a good basis for measuring addressing efficiency. Techniques are developed and a system of computer programs presented for determining the estimated addressing overhead and entropy of an actual program execution. Numerical results are then shown that demonstrate the considerable inefficiency that exists.

Chapter 3 briefly examines several techniques for improving addressing efficiency within the context of standard architectures. These improvements include reducing the number of address registers and the maximum displacement to more accurately reflect program requirements, improving the allocation of address registers by the compiler, and improving the indexing capabilities of the architecture.

Chapter 4 presents a radical approach to the design of the memory subsystem. In particular the concept of second order memory is introduced. This memory architecture improves performance by taking advantage of the higher order, predictable behavior of a typical referencing stream. Two examples are analyzed in detail, clearly demonstrating, that if one is willing to pay the price of increased space and difficulty of use, addressing overhead can be reduced by an order of magnitude or more.

Chapter 5 then summarizes the major results and conclusions of this research and presents a few topics on which additional research would be most useful.

CHAPTER 2

INFORMATION CONTENT OF REFERENCE STREAMS

2.1. Introduction

In analyzing the information content of the memory address (reference) trace of a program, an attempt is made to isolate that part of the trace inherent to the computation in the program from the part devoted to address generation performed by the program. The program is accordingly divided into a computation process and an addressing process. The average information content per reference of the computation process, H, is then analyzed.

In this chapter the variable H is defined as well as a method for estimating H. Also defined are variables B, the number of bits of address per reference actually sent to the memory per computation reference and A, the number of bits input to the addressing process per computation reference.

In addition, the methods used by a system of analysis programs to compute values for the above variables are presented.

2.2. The Addressing Process

Before defining the addressing process, the concept of R+D architecture is introduced.

Definition 2.2.1: An R+D architecture forms a memory address by taking the contents of a base register R and adding some displacement D.

Memory address = (contents of R) + D, where

$r = \lceil \log_2 (\text{number of registers}) \rceil$ and

$d = \lceil \log_2 (\text{maximum displacement in locations}) \rceil$. □

Now r+d is the number of bits required in an instruction encoding to name a register and specify a displacement.

The R+D architecture is important because most addressing architectures use variations of R+D addressing. Table 1 shows a few of the addressing modes in common use and some common computers that use them. Paged addressing, for example, requires a page select register which is often loaded implicitly by indirection. Indirection is actually just the automatic reloading of the memory address register. Consequently each form of addressing has its own method for loading into and displacing from explicitly or implicitly addressed base registers. So in reality, the other addressing modes are just variations of the first. Thus it is reasonably general to devote our attention to a strict R+D architecture.

Throughout the rest of this dissertation, the IBM 360 is used for modeling purposes. IBM 360 program traces were readily available to us. Furthermore, the 360 architecture does not have indirection facilities, and since it would have added additional complexity to our models without being useful in our examples, we did not consider indirection (except briefly in the section on compilers in Chapter 3). Future research in this area may very well include the explicit modeling and analysis of multiple level indirect addressing. That extension to the model used here should be straightforward.

Table 1

Addressing Types

- Base + Displacement
  IBM 360, PDP-11
- Paged
  HP 2100, PDP-8
- Indirection
  PDP-11, HP 2100
- Auto-Index
  PDP-11
- PC-Relative
  PDP-11

Table 2

IBM 360 Instruction Types

Type	Memory Addressing Mode
RR - Register Reference	None
RX - Storage Reference, indexed	(B) + (X) + D
RS - Register and Storage Reference	(B) + D
SI - Storage and Intermediate	(B) + D
SS - Storage and Storage	(B) + D, (B) + D

(Where (B) signifies contents of a base register,
(X) signifies contents of an index register, and
D is an integer value representing displacement.)

The IBM 360 addressing architecture lends itself quite well to this analysis, since it has easily delineated index operations and is basically an R+D architecture. Table 2 shows the IBM 360 instruction types and the addressing mode used by each. Therefore by analyzing a more or less pure R+D architecture like that of the 360 we feel we are performing a reasonably general analysis.

Program execution now can be divided into two interleaved processes: the computation process, which performs actual computational work, and the addressing process, which calculates the references to memory needed by itself and by the computation process (see Figure 1). These are modeled as follows.

Let $\tilde{A}$ be the total number of bits flowing into the CPU to perform the addressing function averaged over the total number of references made by the computation process. The following algorithm filters out the addressing process references and defines $\tilde{A}$ for a 360 program trace. (It is possible to specify a general algorithm for filtering out the addressing process for any architecture. However, since all our examples are from the 360, a more specific 360 oriented algorithm is presented. Adaptation of this algorithm for some other computer would not be difficult.)

Filter Algorithm

- Scan the instruction trace in the reverse direction.
- When a register is used for base or index addressing, place it on the active base or index register list, respectively.

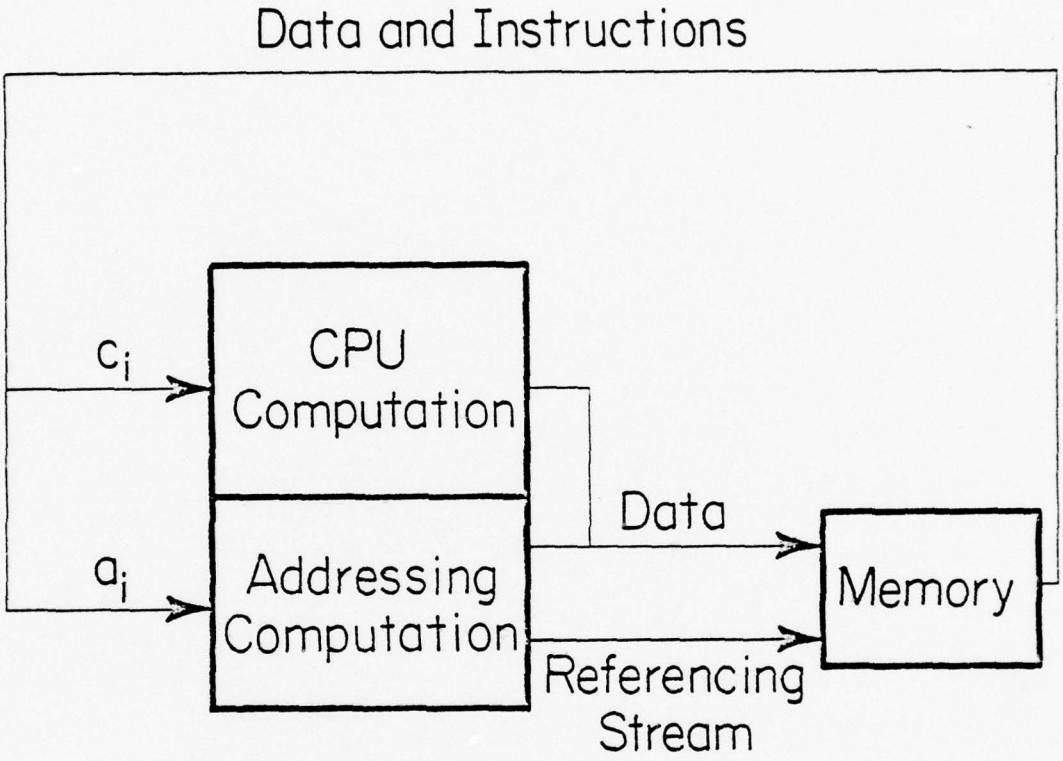


Figure 1. CPU-memory model.

- When an instruction operates on an active register, delete that instruction. If the instruction was a load, deactivate the destination register and place the source memory location on the active storage list.
- When a store into an active storage location is encountered, delete the instruction, deactivate the storage location, and activate the source register.
- When loading or doing arithmetic from any register to an active register, activate the source register, delete the instruction, and if the instruction is a load, deactivate the destination register.
- $\tilde{A}$ is computed as the sum of all bits of deleted instructions, all the bits of data fetched by their associated memory references, all address register (base and index) *select bits and* all displacement bits (see Definition 2.2.1), and then dividing the total by N_c which is the total number of references made by the instructions and partial instructions that remain after filtering (i.e., those not deleted). □

In all but one case we assumed three bits instead of four for address register select, since for all but one of our traces at most seven or eight registers were active at one time for addressing. Consequently, the execution of a single memory reference (RX) instruction usually incurs 18 bits of addressing overhead (3 for index register select, 3 for base register select, and 12 for displacement). Also two bits are added to each computation instruction executed, since there are basically four addressing operations that need to be delineated;

- RX, RS, or SI memory reference,
- SS memory reference,
- Base Register Load, and
- No memory reference (RR instruction).

This algorithm filters out only active addressing activity. This means that if an address register is set up in anticipation of a branch that is never taken (or an operand that is never accessed), that set up operation is not filtered. This is one of the gray areas of algorithm operation, since it is impossible for the algorithm to detect "unfulfilled" addressing process activity.

Definition 2.2.2: For the ith word read from memory by a trace, let

w_i = the number of bits in the word,

a_i = the number of bits in that word which are associated with the addressing process; i.e., those bits deleted by the filter algorithm from the reference word ($a_i = w_i$ in many cases), and

$c_i = w_i - a_i$ = those bits of w_i associated with the computation process, i.e., the number of bits in the referenced word which remain after the filter algorithm is run. □

The computation process then operates given instructions and data, c_i, fetched from the memory. Computational work is performed on the data.

The addressing process operates given instructions and data, a_i, from the stored program in memory. The addressing process is responsible for creating all memory references needed by both processes.

Definition 2.2.3: The addressing overhead is defined as

$$\tilde{A} = \frac{\sum_i a_i}{N_c}$$

where N_c is the number of computation process references. That is, let

N_t = total number of references,

N_a = number of references with $c_i = 0$

(N_a = number of references for instructions rather than number of instructions, since some instructions require multiple references), and

N_{d_a} = number of data references in the addressing process, we get

$$N_c = N_t - (N_a + N_{d_a}).$$

Note that N_c includes all referenced words with $c_i \neq 0$. In an infinite process, we define A as

$$A = \lim_{N_c \to \infty} \tilde{A}. \qquad \square$$

A then is the average number of addressing bits input into the CPU per computation reference. The addressing process can be visualized then as a finite, deterministic encoder which generates the entire memory reference stream.

Definition 2.2.4: B is the number of bits of address which are sent to the memory per computation reference. $\square$

Until Chapter 4 it is assumed, as in conventional computers, that the number of bits being sent to memory consists of fixed size packets of w bits each so that

$$B = \frac{N_t}{N_c} w$$

where $w = \log_2$ (memory size). For the 360, w is assumed to be 32 bits. In Chapter 4 a more general formula for B will be developed, which is compatible with a broader class of architectures.

The next section introduces the concept of entropy and defines the variable H. It is then shown that this variable is a lower bound for A and consequently A vs. H provides a measure of addressing efficiency of the system architecture operating on typical object code compiled for the system.

2.3. Entropy

Entropy is used by information theorists as a measure of the uncertainty of a random variable.

Definition 2.3.1: The entropy, H(S), of a random variable, S, which arises from an unconditional probabilistic process is the average self information:

$$H(S) = -\sum_i p(s_i) \log\ p(s_i)$$

where the self information of the occurrence of element s_i of S is defined as $-\log p(s_i)$, the logarithm of the probability of occurrence of element s_i. □

Thus, for example, if S takes on one of three values with probability 1/2, 1/4, and 1/4, the information conveyed by the occurrence of each of these values is 1 bit, 2 bits, and 2 bits, respectively.

The expected information conveyed by a value of S, the self information of S, is 1.5 bits. A Huffman encoding of the three values which transmits no more bits than the information content is 0, 10, and 11, respectively.

Many processes have considerable structure, i.e., they are very predictable given sufficiently high order conditional probabilities. Extension to higher order requires a more involved, but more useful definition of entropy. Let $m_j \in M$, where M is the set of memory locations accessible by a program and $|M| = 2^w$, i.e., the memory size. Let S be the random variable that represents the memory referencing process. S takes on values which are successively referenced memory addresses, in a probabilistic fashion.

Definition 2.3.2: An n gram, g^n, is an ordered set of n addresses; e.g., if we have two locations a and b, all possible 2 grams would be $g_1^2 = aa$, $g_2^2 = ab$, $g_3^2 = ba$, $g_4^2 = bb$. G^n is the set of all possible n-grams. □

Using the n gram entropies of Shannon [5] the absolute entropy is defined.

Definition 2.3.3: Let m_j be an arbitrary reference to the j^{th} location in memory. Let g_i^{n-1} be the n-1 gram of the previous n-1 references. The zeroth order entropy is

$$H_0(S) = \log_2|M| = w,$$

if all 2^w locations of memory are accessed by the program. Note that $H_0(S)$ actually represents the value $\log_2|M'|$, where M' is the set of referenced memory locations, $M' \subseteq M$. All locations of memory are not always referenced, so in general

$$H_0(S) = \log_2|M'| \leq \log_2|M| = w.$$

The nth order entropy is

$$H_n(S) = \underset{m_j \in M}{-\Sigma} \; \underset{g_i^{n-1} \in G^{n-1}}{\Sigma} p(m_j, g_i^{n-1}) \log_2 p(m_j | g_i^{n-1})$$

and the absolute entropy is

$$H(S) = \lim_{n \to \infty} H_n(S). \qquad \square$$

It should be noted that calculation of H(S) for a finite program trace is impossible, since as n approaches the length of the trace, $H_n(S)$ approaches 0 and is no longer meaningful. Therefore, the variable $\tilde{H}$ will be introduced, which is an estimate of H based on the general structure, but not on complete knowledge of the trace.

Note that we are using base 2 logarithms so that entropy has dimension bits/reference. For example, with the above definition

$$H_1(S) = \underset{m_j \in M}{-\Sigma} p(m_j) \log_2 p(m_j)$$

$$H_2(S) = \underset{m_{j_1} \in M}{-\Sigma} \; \underset{m_{j_2} \in M}{\Sigma} p(m_{j_1}, m_{j_2}) \log_2 p(m_{j_1} | m_{j_2}), \text{ and}$$

$$w \geq H_0(S) \geq H_1(S) \geq \cdots H_\infty(S) = H(S) \geq 0.$$

Note that $G^1 = M$. Entropy is an excellent measure of the unpredictability of a probabilistic process. If S represents an independent uniformly distributed random variable, H(S) = w, and if for some i, $p(m_i) = 1$ (a particular location, i, is always referenced), H(S) = 0, bounding H(S). Entropy then is a measure of the "uncertainty" or "unpredictability" of the random variable S; nth order entropy is a measure

of the unpredictability of the next reference given knowledge of the previous n-1 references. It is important that entropy is a memory allocation independent measure of predictability in contrast to locality which is allocation dependent.

Before presenting the key result of this section, it is useful to describe the abstract model of the addressing process. When a program is being executed by a CPU, all or part of some of the words fetched by the CPU constitute input to the addressing process. That process as it is executed under program control by the hardware can be modeled as a finite-state machine. This machine accepts as input the data words, instructions, and portions of instructions which we defined as the addressing process input in the previous section. In an actual system these codewords flow into the CPU over the data bus. This same bus also transmits the computation process inputs. Hence the variable length property of the input codes is essential even though the bus itself and the data words on it may have constant width. The output of the finite state machine representing the addressing process is the actual string of memory locations (outputs to the address bus) referenced by the computation process only. The state of this machine may be thought of as the cross product of the states of all registers (or partial registers) used by the addressing process.

At this point, the addressing process is abstractly thought of as generating only the references for the computation process. This means that for this section, the variable A represents just that overhead required to generate the computation process memory reference stream.

Of course the actual values for $\tilde{A}$ in the results section will also include some overhead required by the addressing process for generating its own references. This makes $\tilde{A}$ only slightly larger than if such overhead were excluded.

Since our decoder is a finite-state machine, it therefore can be thought of as a decoder which uses a finite number of different codebooks. The current codebook being used is determined by the state of the machine. Codebooks can be changed by certain codewords which are state dependent. Different codebooks then map the same input code set onto the same output set in different ways. What this means is that depending on the initial state of the machine a finite length string of codewords input into the machine could give two or more distinct output strings.

Therefore we have a finite-state decoder accepting an input code set of variable length codewords, and outputting elements from the set of memory addresses (in other words, the memory reference stream of the computation process). The variable-length input codeword set must be uniquely decodable. A code is uniquely decodable if for each source sequence of finite length, the sequence of code letters corresponding to that source sequence is different from the sequence of code letters corresponding to any other source sequence [6].

Recapitulating then, H(S) is the absolute entropy of the stream of computation references, i.e., machine outputs, which are the memory references required by the computation process. A is the average length in bits of the codewords input into our finite-state "addressing" machine per computation reference.

Figure 2 illustrates the entire process. In Figure 2a, a precise diagram of the relationship of the addressing process to the computation process is shown. Figure 2b shows the abstract model of the addressing "communication channel." The decoder is the finite-state machine model of the addressing process. A represents, in bits/symbol, a code for selecting codebooks and inputs to the selected codebook which cause the addressing process to generate the S string. The encoder is then an imaginary process which codes S into an efficient A string. A similar encoding function is normally performed by the compiler.

Theorem 2.3.1: If a finite state deterministic decoder is used to convert A to B, then

$$A \geq H(S). \qquad \square$$

Before Theorem 2.3.1 can be proven two useful lemmas and a theorem are given.

Lemma 2.3.1: Let $p_1, p_2, \ldots, p_M$ and $q_1, q_2, \ldots, q_M$ be arbitrary positive numbers with $\sum_{i=1}^{M} p_i = \sum_{i=1}^{M} q_i = 1$. Then $-\sum_{i=1}^{M} p_i \log p_i \leq -\sum_{i=1}^{M} p_i \log q_i$ with equality if and only if $p_i = q_i \ \forall i$.

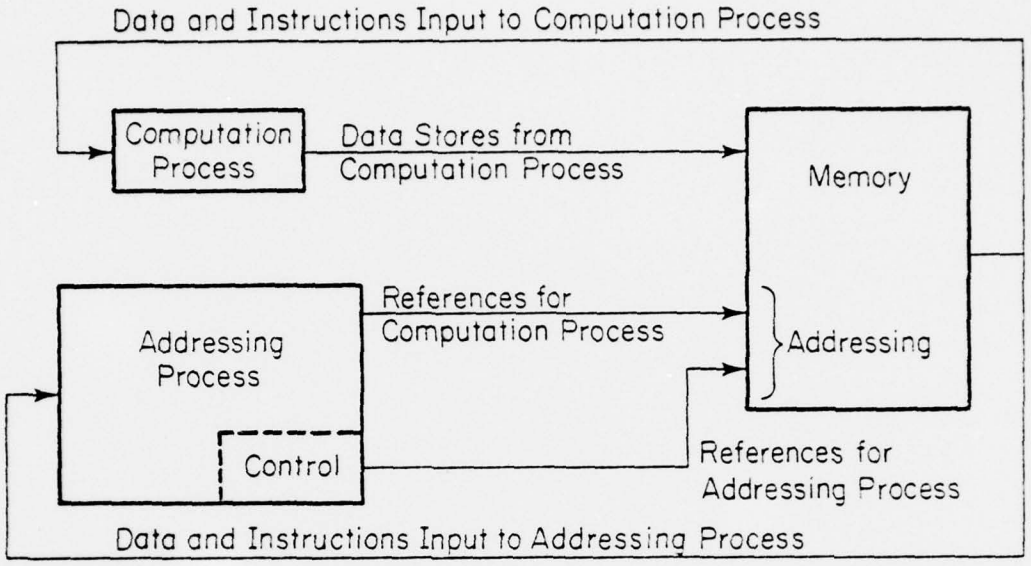
Proof is given in Ash [7]. $\square$

Lemma 2.3.2: For any uniquely decipherable binary code with word lengths $n_1, n_2, \ldots, n_M$

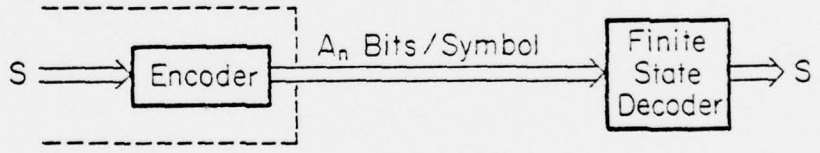
$$\sum_{i=1}^{M} 2^{-n_i} \leq 1.$$

Proof is given in Ash [8]. $\square$

Theorem 2.3.2: Let S be a random variable which represents a source generating symbols. Let A_n be the average number of bits



(a) Addressing Process



(b) Coding Channel

FP-5552

Figure 2. Addressing process and channel.

required to code each source output s_i in binary digits, given that the encoder and decoder have only knowledge of the previous n-1 symbols generated. A_n is then the average code length needed to code the next symbol, knowing the n-1 gram, g^{n-1}, of previous symbols. Then

$$A_n \geq H_n(S).$$

Proof: (Similar to that of Theorem 2.5.1 in Ash [9].) Define

$$A_n = \sum_{i,j} p(m_i, g_j^{n-1}) n_{i|j}$$

where $n_{i|j}$ is the length of the code word assigned to m_i when the n-1 gram g_j^{n-1} has just been transmitted. Then

$$A_n = \sum_j p(g_j^{n-1}) A_n(S|g_j^{n-1})$$

where

$$A_n(S|g_j^{n-1}) = \sum_i p(m_i|g_j^{n-1}) n_{i|j}.$$

Now

$$H_n(S) = -\sum_{i,j} p(m_i, g_j^{n-1}) \log_2 p(m_i|g_j^{n-1})$$

or

$$H_n(S) = \sum_j p(g_j^{n-1}) H_n(S|g_j^{n-1})$$

where

$$H_n(S|g_j^{n-1}) = -\sum_i p(m_i|g_j^{n-1}) \log_2 p(m_i|g_j^{n-1}).$$

First we shall prove

$$A_n(S|g_j^{n-1}) \geq H_n(S|g_j^{n-1}).$$

Multiplying $A(S|g_j^{n-1})$ by $\log_2 2 (= 1)$ we have

$$(\log_2 2) A(S|g_j^{n-1}) = -\sum_i p(m_i|g_j^{n-1}) \log_2 2^{-n_{i|j}}.$$

Letting $q_i = \dfrac{2^{-n_{i|j}}}{\sum_{\ell} 2^{-n_{\ell|j}}}$, by Lemma 2.3.1 we have

$$-\sum_i p(m_i|g_j^{n-1}) \log_2 p(m_i|g_j^{n-1}) \leq -\sum_i p(m_i|g_j^{n-1}) \log_2 q_i$$

or

$$H(S|g_j^{n-1}) = -\sum_i p(m_i|g_j^{n-1}) \log_2 p(m_i|g_j^{n-1}) \leq -\sum_i p(m_i|g_j^{n-1}) \log_2 \left(\frac{2^{-n_{i|j}}}{\sum_{\ell} 2^{-n_{\ell|j}}} \right)$$

$$H(S|g_j^{n-1}) \leq -\sum_i p(m_i|g_j^{n-1}) \log_2 2^{-n_{i|j}} + \sum_i p(m_i|g_j^{n-1}) \log_2 \left(\sum_{\ell} 2^{-n_{\ell|j}} \right)$$

or

$$H(S|g_j^{n-1}) \leq A_n(S|g_j^{n-1}) + \log_2 \left(\sum_{\ell} 2^{-n_{\ell|j}} \right)$$

since

$$\sum_i p(m_i|g_j^{n-1}) = 1.$$

By Lemma 2.3.2, $\sum_i 2^{-n_{i|j}} \leq 1$ for any uniquely decipherable code, then

$$\log_2 \left(\sum_{\ell} 2^{-n_{i|j}} \right) \leq 0$$

since $\log_2 y \leq 0 \quad \text{for} \quad 0 < y \leq 1.$

Therefore

$$A_n(S|g_j^{n-1}) \geq H_n(S|g_j^{n-1}).$$

Since the inequalities hold between A_n and H_n for all j, by multiplying both sides of the inequality by $p(g_j^{n-1})$ and summing over j we get

$$\sum_j p(g_j^{n-1}) A_n(S|g_j^{n-1}) \geq \sum_j p(g_j^{n-1}) H_n(S|g_j^{n-1})$$

or

$$A_n \geq H_n(S). \qquad \square$$

Basically the above theorem is applicable whenever the encoder and decoder know only the previous n-1 symbols. Since CPUs use registers to remember addresses and operands, the CPU can pass information forward indefinitely, beyond any finite n gram that may be stored. Hence the concept of infinite memory encoding (unbounded memory) is necessary to allow modeling of the memory of unbounded history brought forward by the contents of the CPU addressing registers. This CPU behavior is precisely an infinite memory coding situation , since the same finite, arbitrarily long subsequence of instructions can refer to two or more distinct locations.

When a process has tightly coupled codeword groupings, it is possible to beat the standard Huffman encoding for the entire ensemble by Huffman encoding each group and changing codes when a new group is entered. This technique, of course, requires a decoder with memory and intelligence. This process is exactly what occurs in an R+D architecture when base registers are loaded to cover current localities. For Theorem 2.3.2 to apply to our finite-state machine then we must let n go to infinity as stated in the following corollary, thus proving Theorem 2.3.1.

Corollary 2.3.1: For an infinite memory coding system

$$A \geq \lim_{n \to \infty} H_n(S) = H(S). \qquad \square$$

No proof is given since this is the most fundamental result of information theory, i.e., one cannot code at a rate which is lower than the absolute entropy of a probabilistic source. In a sense we have let n go beyond any past knowledge the coder may bring forward.

Corollary 2.3.1 implies that a program must be initial state independent, i.e., it should assume nothing of the initial register

contents (reloading before using). This stems from the fact that the coder must always be able to operate correctly by looking at the last n references as n goes arbitrarily large. This assumption is not unrealistic since all computer systems require initial register contents to be set up for a program by each program itself or by the operating system each time a program is activated.

H(S) is thus a lower bound for A, making A vs. H(S) a good measure of addressing efficiency. Though H(S) is upper bounded by w, A is not upper bounded and could be infinite. (If for example no computation is occurring at all, $N_c = 0$.) Although in practice A is much larger than H(S), one can visualize a machine where $A \cong H(S) = w$. Basically, if we have H(S) = w, a random referencing process, the most efficient addressing architecture would be for the addressing process to fetch the next address directly from the memory each time making $A \cong w$. So even though A can be infinite, A should not be greater than w in a reasonable system.

If the coding process has infinite memory, we cannot guarantee $A_n \geq H_n(S)$ for any finite n. This is due to the fact that the CPU can always have some knowledge of past behavior (> n away) which can be used to code more efficiently than $H_n(S)$, since $H_n(S)$ assumes no knowledge beyond the last n-1 gram of references. It is, however, interesting that with real programs an R+D architecture has difficulty coding much better than H_1 (Chapter 3). Therefore such an architecture does not make effective use of higher order information.

The calculation of $H_\infty(S)$ from a finite trace is of course impossible. The next section presents an algorithmic approach to computing $\tilde{H}(S)$, an estimate of H(S).

2.4. Estimation Theory

Since actual computer programs are not infinite processes, a method is needed for finding an estimated entropy $\tilde{H}(S)$ of an actual reference stream. The following method has been used to find $\tilde{H}(S)$ for IBM 360 program traces. This method tries to take advantage of the "structure" of the program. Basically a model of program execution is derived using this structure, with the entropy $\tilde{H}(S)$ being the entropy of the model.

Note that the $\tilde{H}(S)$ that is calculated from these traces is, like the traces themselves, data dependent. It is precisely valid only for a particular run of a particular program.

Initially, a computation reference trace is created from a filtered 360 instruction execution trace. Given the reference trace, the following constructs are used to create a block graph of program execution:

Definition 2.4.1: A Ramp is a set of sequentially executed, nonbranching instructions terminated by a branch instruction (conditional or unconditional). A ramp may be entered only at the first instruction. □

Definition 2.4.2: A Block is a set of one or more sequentially executed ramps where all entry points to the block are to the beginning of the first ramp in the block and all exit points from the block only leave the last ramp in the block; see Figure 3. □

Blocks and ramps are totally sequential, i.e., no looping is allowed within them.

Blocks actually could be constructed from a trace in one pass, but it has been found that a two pass (ramps, then blocks) distillation

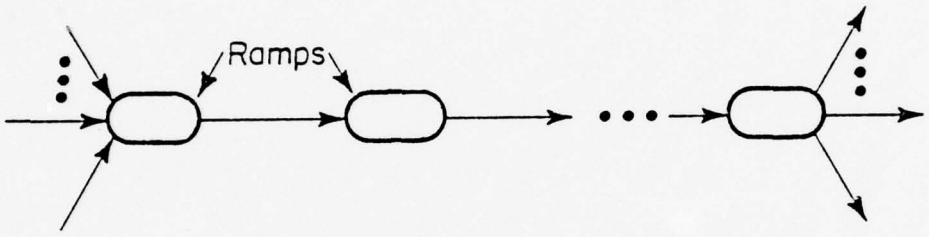


Figure 3. A block.

process works much more smoothly. This is true, primarily because of the large number of references one must keep track of when creating blocks directly from the reference trace.

One problem that can occur, is that a ramp can have an arc entering in the middle. SUPER, the program that determines the blocks from the ramp graph simply moves such an arc to the beginning of the ramp rather than split the ramp. This, however, occurred for only two ramps within one program trace (LIST) and therefore hardly affected the accuracy of $\tilde{H}(S)$.

A block graph is now created (see Figure 4) where each arc from node i to node j is labelled f_{ij}, representing the execution frequency of that arc. Note that

$$\sum_j f_{ij} = F_i, \quad \text{the total execution frequency of node i.}$$

Also an arc is added from the last block executed to the first executed with $f_{last,first} = 1$. This construction results in an irreducible, i.e., any state can be reached from any other, Markov process. The node transition probabilities can be estimated from the arc frequencies f_{ij}. Since

$$\lim_{F_i \to \infty} \frac{f_{ij}}{F_i} = p_{ij}, \quad \text{the transition probability from node i to node j,}$$

$$\tilde{p}_{ij} = \frac{f_{ij}}{F_i}, \quad \text{is an estimated transition probability.}$$

To get a rough idea of the error introduced by such finite sampling, the expected first order entropy calculation has been shown by Basharin [10] to be

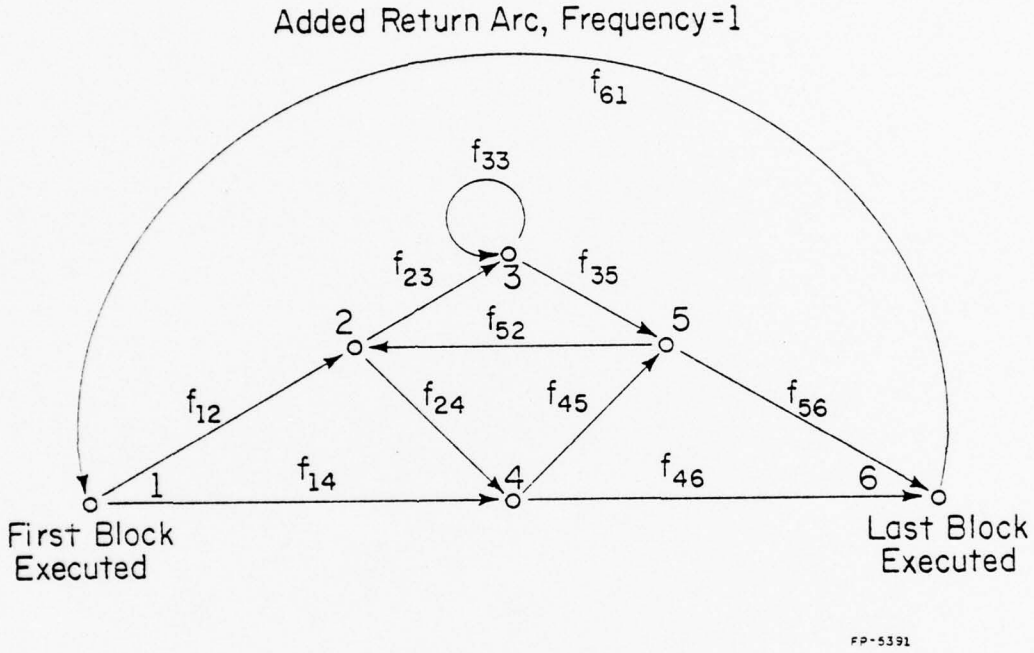


Figure 4. A block graph.

$$E[\tilde{H}_1(S)] = H_1(S) - \frac{q-1}{N-1} \log_2 e + \mathcal{O}(\frac{1}{N^2})$$

where q = number of symbols, $|M'|$, and

N = sample size, number of references in trace.

For our traces, typically $q \approx 1100$ and $N \approx 80{,}000$, which gives an error of about 2%. As n increases in $\tilde{H}_n$ the finite sampling error also increases. This effect poses an upper limit on the order of H which can be established by this method.

After estimating the block transition probabilities $\tilde{p}_{ij}$, we can easily compute the stationary probabilities $\tilde{q}_i$, i.e., the probability of entering state i on an arbitrary state transition, for each block. These exist since the process is irreducible and ergodic. The process is ergodic, i.e., it reaches a unique steady state, since in all our programs there has existed at least one block with an arc to itself. This is a sufficient, though not a necessary condition of ergodicity [11]. Each block is now expanded as in Figure 5 to include both its instruction and data references. Again each arc is labelled with its execution frequency.

Definition 2.4.3: A Program Graph is an expanded block graph, in which each node represents a single reference. □

Definition 2.4.4: Let $\tilde{H}(S)$ be the information content of the program graph as defined above. □

It should be mentioned that $\tilde{H}(S)$, though not computed by letting n approach infinity in $H_n(S)$, does nevertheless represent the uncertainty, given that we know only the program graph of the program run. It does not, however, trace block execution "histories," since such knowledge,

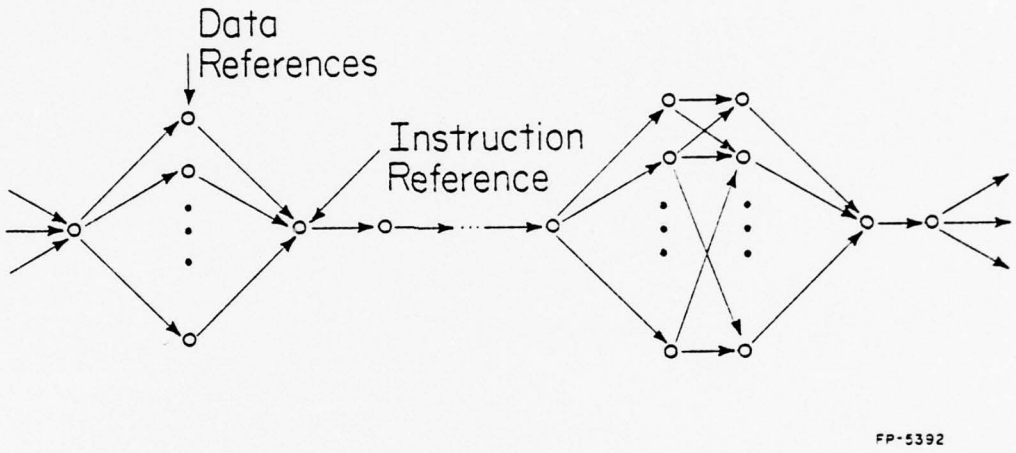


Figure 5. An expanded block.

though it would reduce $\tilde{H}(S)$, would be expensive and unwieldy to obtain, especially considering the necessity of knowing the multitude of data reference "histories." Including such information also runs the risk of incurring unacceptably large finite sampling error. Consequently, $\tilde{H}(S)$ represents a good engineering estimate of $H(S)$.

Within a single block all data and instruction references are dependent on the previous instruction reference. All sequentially accessed instructions then contribute nothing to the entropy, since given the first instruction of the block, they are known with absolute certainty. Data references are also dependent only on the previous instruction and are independent of each other. This assumption adds more uncertainty than is perhaps justified by the model. However, it was found through study of one of the traces (GAUSS) that blocks which had two or more instructions that referenced data from two or more distinct locations were quite rare. Incidentally by assuming data reference dependence we also eliminate any highly correlated data referencing behavior that might occur between successive executions of a block such as in a loop. Consequently data references may be made in a sequential manner from pass to pass within the block but will appear to be independent in the program graph. This again overestimates the uncertainty, but is necessary for a reasonable computation of $\tilde{H}(S)$.

The absolute entropy $\tilde{H}(S)$ of a program graph is given by the following theorem:

Theorem 2.4.1: For a program graph the estimated information content per reference with independent data references is

$$\tilde{H}(S) = \sum_i \frac{r_i}{n_i} \tilde{H}(S|i) = \frac{1}{N} \sum_i \tilde{q}_i \tilde{H}(S|i),$$

where n_i is the total number of references in block i, $N = \sum_j n_j \tilde{q}_j$

$r_i = \dfrac{n_i \tilde{q}_i}{\sum_j n_j \tilde{q}_j}$ is the fraction of total references which are generated by block i, and

$\tilde{H}(S|i)$ is the total information content over all references in block i.

$$\tilde{H}(S|i) = -\sum_j \tilde{p}_{ij} \log_2 \tilde{p}_{ij} + \sum_k \tilde{H}_d(i,k)$$

where the $\tilde{p}_{ij}$ are the estimated block transition probabilities, k spans the data references in block i, and

$$\tilde{H}_d(i,k) = -\sum_\ell \tilde{p}(i,k,\ell) \log_2 \tilde{p}(i,k,\ell)$$

where ℓ spans all nodes associated with data reference k in block i and $\tilde{p}(i,k,\ell)$ is the estimated probability of occurrence of node ℓ during data reference k in block i. □

Before proving Theorem 2.4.1 the following lemma is needed.

Lemma 2.4.1: The entropy per symbol output by a Markov source is given by

$$H_\infty(S) = \sum_i q_{(1,\infty)}(i)\, H(S|i)$$

where

$$H(S|i) = -\sum_j p_{ij} \log_2 p_{ij},$$

and $q_{(1,\infty)}(i) = q_i$, the Markov stationary probability for state i, since the program graph has only one irreducible set of states, which is ergodic. The proof is given in Gallager [12]. □

Proof of Theorem 2.4.1: To evaluate $\tilde{H}(S)$, let us consider each node (reference) in the program graph, and find its entropy. The possible cases are:

i) First instruction in a block: This entropy is based on the first order block transition, $i \rightarrow j$, without considering, e.g., where block i went last time or which block preceded i. The estimated uncertainty of the transition from block i to j is

$$\tilde{H}_{\text{Block } i \rightarrow j} = -\log_2 \tilde{p}_{ij}.$$

Associating this $\tilde{H}$ term with the end of block i and accumulating and weighting by $\tilde{p}_{ij}$ over all j (successor blocks of block i) we get

$$\tilde{H}_{\substack{\text{Block i} \\ \text{Exit Transition}}} = -\sum_j \tilde{p}_{ij} \log_2 \tilde{p}_{ij}.$$

ii) Other instructions of block: Given that we are in block i, these other instructions contribute nothing to uncertainty, since they are fetched sequentially, therefore

$$\tilde{H}_{\substack{\text{Instructions} \\ \text{Within Block i}}} = 0.$$

iii) Data reference at data reference position k in block i: Only consider dependency on previous instruction fetched (equivalent to knowing k); e.g., do not consider instruction references before that, any other data references in the block, or the data element referenced at position k in block i during the last execution of block i. Of all the locations referenced at this position throughout the trace, consider the one numbered ℓ, then

$$\tilde{H}_{\substack{\text{Data Reference } \ell \text{ at} \\ \text{Position k in Block i}}} = -\log_2 \tilde{p}(i,k,\ell).$$

Accumulating and weighting by $\tilde{p}(i,k,\ell)$ over ℓ we have then the entropy associated with the data reference at position k in block i

$$\tilde{H}_d(i,k) = -\sum_{\ell} \tilde{p}(i,k,\ell) \log_2 \tilde{p}(i,k,\ell).$$

Grouping terms, the entropy associated with block i is

$$\tilde{H}(S|i) = \underbrace{-\sum_j \tilde{p}_{ij} \log_2 \tilde{p}_{ij}}_{\substack{\text{Block Transition} \\ \text{Associated with} \\ \text{Exit of Block i}}} + \underbrace{\sum_k \tilde{H}_d(i,k)}_{\substack{\text{Data Reference} \\ \text{Entropy within} \\ \text{Block i}}}.$$

The entropy per reference for block i = $\frac{\tilde{H}(S|i)}{n_i}$, where n_i is the total number of references in block i.

Recall that $\tilde{q}_i$ is the estimated stationary probability of entering block i, given that a block transition is being made. However, n_i is not constant over i. Therefore define r_i as the stationary probability of a particular reference being from block i; i.e.,

$$r_i = \frac{n_i \tilde{q}_i}{N} \quad , \quad \text{where } N = \sum_j n_j \tilde{q}_j.$$

Using Lemma 2.4.1 where $q_{(1,\infty)}(i) = r_i$ and $H(S|i) = \frac{\tilde{H}(S|i)}{n_i}$, we have

$$\tilde{H}(S) = \sum_i r_i \frac{\tilde{H}(S|i)}{n_i}.$$

However since $r_i = \frac{n_i \tilde{q}_i}{N}$,

$$\tilde{H}(S) = \frac{1}{N} \sum_i \tilde{q}_i \tilde{H}(S|i). \qquad \square$$

The above theorem states that by knowing the number of references n_i in block i and the probability of being in block i, it is merely a matter of summing the data entropy contributions and the contribution to entropy made when leaving the block and then multiplying by r_i/n_i to get the total entropy contribution per reference made by block i to the absolute entropy.

Figure 6 shows an example of a program graph and the application of Theorem 2.4.1 to it. Let us assume for brevity that the contribution by data references in blocks 2 and 3 is zero. The following calculations can then be made. The $\tilde{q}_i$ can be computed from steady state Markov analysis of Figure 6a.

$$N = \tilde{q}_1 n_1 + \tilde{q}_2 n_2 + \tilde{q}_3 n_3 = \frac{4}{7}(7) + \frac{1}{7}(4) + \frac{2}{7}(12) = 8,$$

$$\tilde{H}_d(1,1) = 0,$$

$$\tilde{H}_d(1,2) = -(\tfrac{1}{8}\log\tfrac{1}{8} + \tfrac{1}{8}\log\tfrac{1}{8} + \tfrac{6}{8}\log\tfrac{6}{8}) = 1.06,$$

$$\tilde{H}_d(1,3) = -(\tfrac{1}{2}\log\tfrac{1}{2} + \tfrac{1}{4}\log\tfrac{1}{4} + \tfrac{1}{4}\log\tfrac{1}{4}) = 1.5,$$

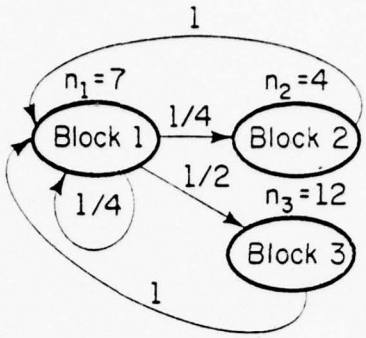
$$\tilde{H}(S|1) = -(\tfrac{1}{4}\log\tfrac{1}{4} + \tfrac{1}{2}\log\tfrac{1}{2} + \tfrac{1}{4}\log\tfrac{1}{4}) + 0 + 1.5 + 1.06 = 4.06,$$

$$\tilde{H}(S|2) = 0,$$

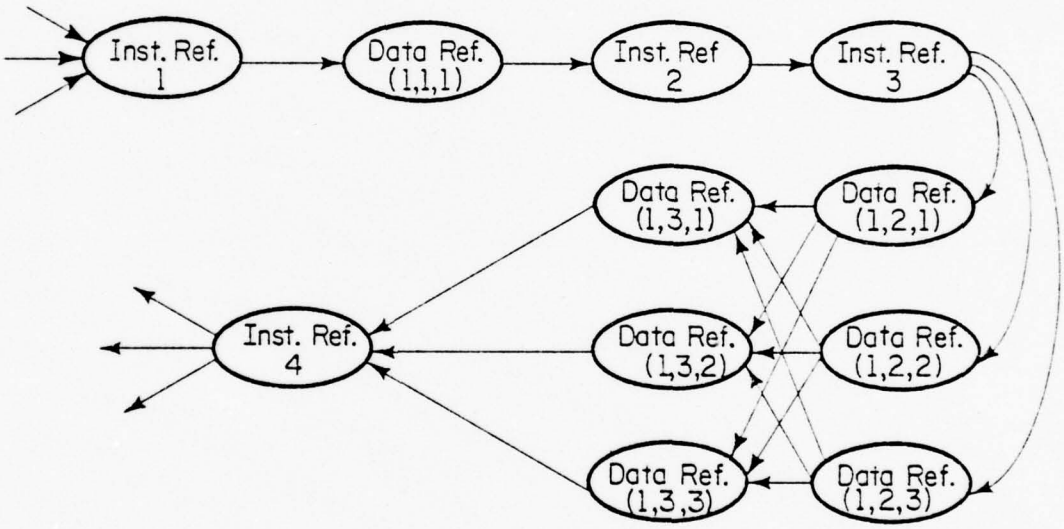
$$\tilde{H}(S|3) = 0,$$

$$\tilde{H}(S) = \tfrac{1}{8}((\tfrac{4}{7})4.06 + (\tfrac{1}{7})0 + (\tfrac{2}{7})0),$$

$$\tilde{H}(S) = 0.29.$$



a. Block Graph



p(1,3,1)=1/2
p(1,3,2)=1/4
p(1,3,3)=1/4

p(1,2,1)=1/8
p(1,2,2)=1/8
p(1,2,3)=6/8

b. Expansion of Block 1

FP-5398

Figure 6. Example of a program graph.

2.5. Analysis Programs

We now describe the system of programs which compute $\tilde{A}$, $\tilde{H}(S)$, $H_0(S)$, $\tilde{H}_1(S)$, and $\tilde{H}_2(S)$ (see Figure 7). Incidentally, no higher n gram entropy than $\tilde{H}_2(S)$ was computed due to the large sampling error that would occur. For example, applying Basharin's formula [10] to $\tilde{H}_3(S)$, (for GAUSS) a sampling error of over 15% would be incurred.

The programs traced are discussed in Section 2.6. Basically the traces themselves are lists of the instructions executed by a particular program when running on an IBM 360 architecture. Each element in the trace contains the hexadecimal code for the instruction executed, any memory references made by the instructions, and for branches, the branch destination. (Note: With the MOVE or other multiple reference instructions, only the first location is given by the trace, since the number of locations referenced is given in the instruction.) For conditional branch instructions the destination is also given according to whether the branch is taken.

FILT1 takes an instruction trace and reverses it so that FILT2 can scan it in a reverse direction.

FILT2 takes the reversed trace and applies to it the filter algorithm given in Section 2.2. The statistics computed by FILT2 are given in Table 3. The output is the reversed computation trace; all addressing instructions having been deleted. The total number of bits deleted (addressing instructions, their references, and addressing portions of computation instructions) is passed to PREP, which when executed will

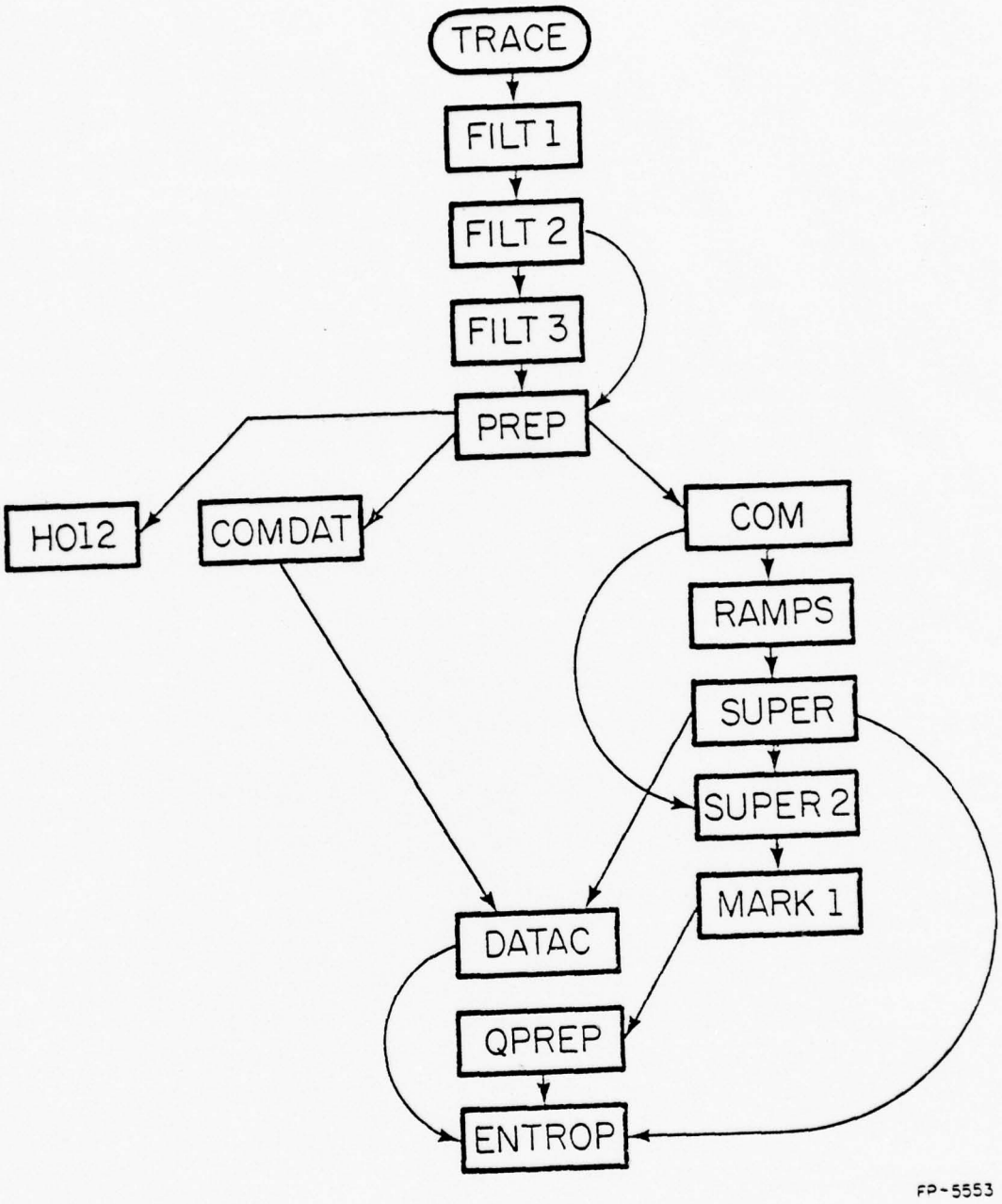


Figure 7. Program system.

Table 3

Statistics Computed by FILT 2

Total Instructions Read

Instructions Deleted

Index Instructions Deleted

Base Instructions Deleted

Average Number of Active Addressing Registers

Maximum Number of Active Addressing Registers

Active Register Deletions

Number of Index Register Loads (Register Reference Instructions)

Number of Base Register Loads (Register Reference Instructions)

Number of Index Register Loads (Memory Reference Instructions)

Number of Base Register Loads (Memory Reference Instructions)

Number of Register Reference Instructions in Filtered Trace

Number of Memory Reference Instructions in Filtered Trace

Number of SS Type Instructions in Filtered Trace

Number of RS and SI Type Instructions in Filtered Trace

Arithmetic Instruction Deletions:
- Register Reference Instructions Operating on Index Registers
- Register Reference Instructions Operating on Base Registers
- Memory Reference Instructions Operating on Index Registers
- Memory Reference Instructions Operating on Base Registers

Number of Stores from Activated Registers

Number of Load Multiple Instructions

use this information to compute $\tilde{A}$ after calculating the number of computation references.

FILT3 then reverses the resulting computation trace so that it can be scanned in the forward direction.

PREP takes the computation trace and, by simulating the operation of an IBM 360 CPU, creates a file of all memory references to 32 bit words. (Since the actual references exist on the trace, information regarding the addressing process is not needed by PREP.)

All instructions for which the number of locations referenced depends on the data within those locations (for example, the EDIT instruction) are deleted. This is a reasonable approximation, since from the trace it is impossible to determine what references are made and it was found that such instructions occurred rarely in our traced programs.

Whenever a branch occurs in the instruction stream, a special symbol is placed after the reference for the branch in the output reference stream. This information is used by COM and COMDAT to determine ramp termination. Also, for each reference, a pair of values is output by PREP. The first value is the actual location referenced in base 10, and the second is the type of reference: instruction, data read, or data write. PREP also computes some statistics (see Table 4) and $\tilde{A}$ from the information passed to it by FILT2. COM creates a ramp trace from the the memory reference trace. Each element in the ramp trace contains the starting address of the ramp (unique to each ramp) and the length of the ramp in references. Data references are not considered here, since they

Table 4

Statistics Computed by PREP

The Maximum Address Referenced in Memory

The Minimum Address Referenced in Memory

Number of References Output

Supervisor Calls Encountered

Total Number of Instructions Interpreted

Number of Instructions Not Interpreted

$\tilde{A}$

will not be needed to create the block graph, and will be separately handled later.

RAMPS creates the ramp graph from the ramp trace. A hash coded data structure is used as an associative memory (addressed by ramp starting addresses) to build a file representing a sparse ramp transition matrix. In the ramp transition matrix each entry, r_{ij}, represents the number of times the direct path from ramp i to ramp j is traversed. Also various statistics about the ramps are collected; these are listed in Table 5.

SUPER takes the ramp graph (the output from RAMPS) and determines the blocks by the following algorithm:

Block Algorithm: Note: The Block Algorithm operates on the ramp graph. The nodes of the graph are numbered from 1 to n. Each node J can have an arbitrary number of successors, which point to other ramps. All nodes have at least one successor, except the last ramp executed in the trace which has none. The algorithm is therefore started with the first ramp executed (J = 1) and terminated when the ramp with no successors is encountered. The algorithm traverses all ramps at least once. Its basic operation is to encounter a node (ramp) and stack all successors if more than one exists, and traverse the arc leaving the node if only one exists. When a node which has been previously traversed or has multiple successors is encountered, the algorithm returns to the stack, obtains another arc and begins again. Blocks constitute collections of ramps that have no arcs entering the block except to the first ramp in the block and no arcs leaving the block except from the last ramp. While traversing the graph,

Table 5

Statistics Computed by RAMPS

Maximum Number of Distinct Ramps Read

Total Number of Ramps Read

Maximum Frequency of a Ramp

Average Frequency/Ramp

Maximum Outdegree/Ramp

Average Outdegree/Ramp

Total Number of Arcs in Ramp Graph

the nodes are labeled with the current block number. Let BLOCKNUM be the current block number. Let RAMP [I] be a pointer to the first ramp in block I. Initially ∀I LABEL [I] ← 0. There also exists another stack which can have values placed on it from P by PUSHSTACK ← P and removed from it and placed in P by P ← POPSTACK. Initially, BLOCKNUM ← 1, RAMP [1] ← 1, and J ← 1 (a pointer to the first ramp).

```
Step 1: WHILE LABEL [J] = 0  (unlabeled) DO
            BEGIN
            LABEL [J] ← BLOCKNUM;
            IF J has only one successor THEN J← successor of J
                ELSE BEGIN (If J has multiple successors a new block
                     must be started.)
                     IF J has no successor THEN J← POPSTACK; terminate
                          program when stack underflows.
                        ELSE BEGIN
                             PUSHSTACK ← all successors to J;
                             J ← POPSTACK;
                             END;
                     IF LABEL [J] = 0 (unlabeled) THEN
                             BEGIN (start a new block)
                             BLOCKNUM ← BLOCKNUM + 1;
                             RAMP[BLOCKNUM] ← J;
                             END;
                     END;
            END;
```

PUSHSTACK ← J;

GO TO STEP 2; (Program drops through to Step 2 whenever an already labeled node is encountered.)

STEP 2: J ← POPSTACK;

Search RAMP [I] list until an element I is found such that J is the starting ramp for block I, i.e. RAMP [I] = J;

(If a node is not an initial ramp for a block, it and all its successors must be relabeled.)

IF such a J is found THEN GO TO Step 2 (it is a complete block).

ELSE BEGIN

BLOCKNUM← BLOCKNUM + 1; RAMP[BLOCKNUM] ← J;

IF LABEL [J] = 0 (unlabeled) THEN GO TO Step 1;

ELSE BEGIN

K ← LABEL [J];

WHILE LABEL [J] = K DO

(Keep relabeling until either multiple successors or a different labeling, evidence of another incoming arc, is encountered.)

BEGIN

LABEL [J] ← BLOCKNUM;

IF J has more than one successor or no successors THEN GO TO Step 2

ELSE J ← successor of J;

END;

END;

END;

GO TO Step 2. □

The block algorithm produces a labeling (by block number) as well as a list of pointers (RAMP [I]) to the starting node (ramp) of each block. An output file is created giving the block number, the length of each block in ramps, the number of references in each block, and the actual starting address of the beginning ramp in each block.

SUPER2 takes the ramp trace from COM and the output file of SUPER and creates a block trace, that is, each element in SUPER2's output file is a block number indicating the execution of that block.

MARK1 takes the block trace generated by SUPER2 and creates the block graph in exactly the same manner as RAMPS created the ramp graph. The output of MARK1 is a sparse block transition matrix where each element f_{ij} represents the transition frequency from block i to block j. Statistics on the blocks are also collected by MARK1 (see Table 6).

QPREP takes the output file from MARK1 and creates two files. One is a Markov transition matrix ($\tilde{p}_{ij}$) for the block graph, and the other a vector of the stationary probabilities ($\tilde{q}_i$), these are estimated as follows:

$$\tilde{p}_{ij} = \frac{f_{ij}}{F_i} \quad , \quad \text{where} \quad F_i = \sum_j f_{ij}$$

$$\tilde{q}_i = \frac{F_i}{F_T} \quad , \quad \text{where} \quad F_T = \sum_i F_i .$$

Now, before ENTROP can act, the data access probabilities must be computed.

COMDAT works almost exactly like COM, except that each element of the COMDAT output file besides containing the starting address for

Table 6

Statistics Computed by MARK1

Maximum Number of Different Blocks Read

Total Number of Blocks Read

Maximum Frequency of a Block

Average Frequency/Block

Maximum Outdegree/Block

Average Outdegree/Block

Total Number of Arcs in Block Graph

each ramp and the ramp length also contains an ordered list of the data references made by each particular execution of each ramp.

DATAC computes the data access probabilities $\tilde{p}(i,k,\ell)$ (see Theorem 2.4.1) for each block, i, in the block graph. It uses the output file of SUPER which contains the block numbers, the starting address of each block and the length of each block in ramps. It reads the modified ramp trace output of COMDAT, creating a list for each reference, k, in the block, i. It then computes for each value of ℓ, the frequency $f(i,k,\ell)$, which is the frequency of the ℓth distinct data address at data reference (position) k within block i. The relative frequencies, $\tilde{p}(i,k,\ell)$, are computed as follows;

$$\tilde{p}(i,k,\ell) = \frac{f(i,k,\ell)}{F(i,k)} \quad , \quad \text{where } F(i,k) = \sum_{\ell} f(i,k,\ell).$$

ENTROP obtains the $\tilde{q}_i$ and $\tilde{p}_{ij}$ from the QPREP output file, the n_i from the SUPER output file, and the $\tilde{p}(i,k,\ell)$ from the DATAC output file. ENTROP then calculates $\tilde{H}(S)$ by applying Theorem 2.4.1.

H012 computes $H_0(S)$, $\tilde{H}_1(S)$, and $\tilde{H}_2(S)$ by reading the computation process memory trace (output file of PREP). A hash coded data structure is created, content addressable by memory address, which contains memory locations accessed by the trace. For each element of the table the absolute frequency, F_i, is kept as well as a list of successors to it and the successor frequencies, $f(i,j)$. Then

$$\tilde{p}(i,j) = \frac{f(i,j)}{F_T} \quad \text{, where} \quad F_T = \sum_i F_i$$

$$F_i = \sum_j f(i,j)$$

and

$$\tilde{p}_i = \frac{F_i}{F_T} .$$

So

$$H_0(S) = \log_2 \text{ (number of entries in the table)},$$

$$\tilde{H}_1(S) = -\sum_i \tilde{p}_i \log_2 \tilde{p}_i$$

and

$$\tilde{H}_2(S) = -\sum_{i,j} \tilde{p}(i,j) \log_2 \tilde{p}(i|j)$$

which is

$$\tilde{H}_2(S) = -\sum_{i,j} \tilde{p}(i,j) \log_2 \tilde{p}(i,j) + \sum_{i,j} \tilde{p}(i,j) \log_2 \tilde{p}_j$$

or

$$\tilde{H}_2(S) = -\sum_{i,j} \tilde{p}(i,j) \log_2 \tilde{p}(i,j) - \tilde{H}_1(S)$$

since

$$\tilde{H}_1(S) = -\sum_j \tilde{p}_j \log_2 \tilde{p}_j = -\sum_{i,j} \tilde{p}(i,j) \log_2 \tilde{p}_j .$$

The next section describes briefly the programs traced and the results of the application of the above system of analysis programs to their traces.

2.6. Results

The results of applying our system of programs to several execution traces are shown in Tables 7 and 8. The programs were traced using a program, Trace 360, obtained from the University of Waterloo, on the IBM 360/75 operated by the Computing Services Office of the University of Illinois. They are as follows:

GAUSS: A Gaussian elimination on a 14 X 14 matrix. This program was compiled by the IBM FORTRAN G compiler.

Table 7

Information Content and Trace Size

	GAUSS	ERROR	EIGEN	LIST
Number of Computation Instructions	46,441	13,762	40,864	42,705
Number of Instructions Deleted by FILT2	15,561	239	13,138	17,350
Number of Computation References	75,546	23,306	64,169	65,825
$\tilde{A}$ - Addressing Overhead (bits/computation reference)	17.2	10.0	17.0	24.1
$\tilde{H}(S)$ - Entropy of Computation Reference Stream (bits/computation reference)	1.64	.010	.495	.329
$\tilde{H}_I(S)$ - Instruction Fetching Component of $\tilde{H}(S)$	.079	.0021	.030	.040
$\tilde{H}_D(S)$ - Data Fetching Component of $\tilde{H}(S)$	1.56	.0079	.465	.289
Combined Stream $H_0(S)$	10.36	10.74	10.80	11.27
Combined Stream $\tilde{H}_1(S)$	7.88	9.33	8.62	9.58
Combined Stream $\tilde{H}_2(S)$	1.86	1.32	1.51	.845
Instruction Referencing Stream $H_0(S)$	9.25	10.42	10.21	10.64
Instruction Referencing Stream $\tilde{H}_1(S)$	6.36	9.96	8.37	9.24
Instruction Referencing Stream $\tilde{H}_2(S)$	.074	.006	.080	.098
Data Referencing Stream $H_0(S)$	9.35	8.43	9.24	9.77
Data Referencing Stream $\tilde{H}_1(S)$	7.86	6.40	6.47	7.58
Data Referencing Stream $\tilde{H}_2(S)$	3.12	2.53	2.71	1.90

Table 8

Ramp/Block Structure

	GAUSS	ERROR	EIGEN	LIST
Number of Ramps/Program	115	143	207	606
Maximum Ramp Length	46	312	185	29
Average Number of Data Accesses/Ramp	3.55	25.42	4.39	1.29
Average Number of Successors/Ramp	1.12	1.06	1.21	1.14
Number of Blocks/Program	31	23	72	143
Average Number of Ramps/Block	3.71	6.22	2.88	4.24
Average Number of Successors/Block	1.48	1.39	1.61	1.51
Number of Blocks Executed	7,746	121	3,209	5,705

ERROR: An IBM 360 floating point benchmark used by the University of Illinois Computing Services Office. This program was also compiled by the FORTRAN G compiler.

EIGEN: A program which finds the eigenvalues of a 14 × 14 matrix using two routines (TRED1 and TQL1) from the Eigensystem Subroutine Package (EISPACK) of the National Activity to Test Software project.

LIST: A symbol manipulation program which builds lists from an input stream and then traverses these lists. This program was compiled by the IBM PL/1 compiler.

The programs traced here have quite different structures, yet the main numerical results were quite consistent for all of them. There were some variations from program to program, but these numerical differences did not contradict any of the broad qualitative or comparative conclusions which can be drawn from the data. Therefore one might expect that the conclusions drawn here are of general value.

In all cases only a portion of the total program execution was traced, albeit a rather lengthy portion. This was done for two reasons; first, it gives us more of a "snapshot" of program behavior over a long enough interval to be meaningful rather than a very long term average, and second, large amounts of computer time and storage resources are required to compute $\tilde{H}_2(S)$ and $\tilde{H}(S)$.

For all programs, the addressing overhead $\tilde{A}$ was much larger (at least an order of magnitude) than $\tilde{H}(S)$, the information content of the computation reference trace. This large $\tilde{A}$ suggests that a good

portion of the total bit stream flowing into the CPU is devoted to addressing. We can compute the actual percentage from the following definition. Let

$$\tilde{A}_T = \frac{\text{Total Number of Bits Input to CPU}}{\text{Number of Computation References}} .$$

$\tilde{A}_T$ represents all instruction and data fetches. (The data leaving the CPU consists of data being stored and the address or reference stream.) So we have

$$\tilde{A}_T = \frac{\text{(Total Number of References in unfiltered Trace-Number of Unfiltered Stores)} \times 32 - \text{(Number of Register Reference Instructions)} \times 16}{N_c} .$$

This variable can be computed from the original trace. For example, LIST had $\tilde{A}_T$ = 41.2 bits/computation reference. This means that $\tilde{A}$ is 58.5% ($(\tilde{A}/\tilde{A}_T) \times 100\%$) of the total number of bits flowing into the CPU, or 58.5% of the input bandwidth for the LIST program. This is a large value and gives a good indication of the load which the addressing process places on the system. The small size of $\tilde{H}(S)$ indicates that considerable improvement may be possible in this area.

Much of the difference between $\tilde{A}$ and $\tilde{H}(S)$ is due to the architectural limitations of both CPU and memory (as will be shown in later chapters). It is not difficult to see why $\tilde{A}$ is so large, since for every memory reference, 18 bits are required to specify a mode, a register, and a displacement, and most instructions (at least for our traces) were memory reference instructions. Of course it is possible to reduce the number of bits used for register select and displacement, but then

register reloading would increase (base register loading is exclusively address processing). This trade-off is modeled in Chapter 3.

Other interesting results, in Table 8, include the small number of blocks that exist in each program and the fact that the average number of successors for each block is less than 2. (It can be less than 2, since blocks need only have one successor, since incoming arcs may split blocks.) This fact is further evidence of considerable structure within the control flow of a program, and is reinforced by the small contribution the block graph uncertainty, $\tilde{H}_I(S)$, makes to the total uncertainty. As expected, data referencing, $\tilde{H}_D(S)$, is the most significant contribution to $\tilde{H}(S)$. This results primarily from indexing (vector) operations. Consequently with a more sophisticated model which could capture the regular behavior of vector accessing (indexing), smaller values of $\tilde{H}(S)$ could be obtained.

Incidentally, LIST, which has a markedly different character than the other test programs also has significantly different generated statistics. For example, $\tilde{A}$ is significantly greater than for the other test programs. Also, a larger number of ramps and blocks occurred in LIST. These differences are due primarily to the symbolic character manipulation required by LIST. Referencing under these conditions, though not necessarily less uncertain ($\tilde{H}(S)$ is not much different than for the other traces), does require more overhead in the 360 architecture which is simply not as efficient in performing this type of task.

The most interesting result involves the low order n gram entropies. For all four programs $H_0(S)$ and $\tilde{H}_1(S)$ are about the same, with

$\tilde{H}_1(S)$ being very close to $H_0(S)$. (Note: 2^{H_0} does not indicate absolute filtered program size, but rather the number of memory words required to hold just those instructions and data actually referenced by the computation process during the particular trace of the program.) It is, however, significant that not only are the values for $\tilde{H}_2(S)$ fairly close to each other for all traces, but they are significantly smaller than the corresponding value of $\tilde{H}_1(S)$. What this means is that if one has knowledge of the second order probabilities, $p(i|j)$, of the program, one then has considerable knowledge of the referencing behavior as well. This result is intuitively supported by the following observations:

i) instructions are generally fetched sequentially.

ii) when branching does occur there are usually only two possible targets with one being more probable than the other, and

iii) most memory reference instructions always reference the same location each time the instruction is executed. (Vector or indexed addressing occurred relatively infrequently in our traces, though it did add considerable uncertainty when it did occur.)

Also included in Table 7 are the low order n gram entropies for the instruction reference stream and the data reference stream. Again the results are as expected, $\tilde{H}_2(S)$ being quite small for instruction referencing and relatively large for data referencing.

2.7. Summary and Conclusions

Using information theory, the information content, H(S), of a memory reference stream has been estimated. Also the memory addressing overhead, $\tilde{A}$, was defined. In the last section it was seen that with the programs we have traced, $\tilde{A} \gg \tilde{H}(S)$. The obvious question then is how can $\tilde{A}$ be reduced and what costs are incurred to obtain that reduction? Granted, building a special purpose computer to run a particular program would no doubt enable a tremendous reduction in $\tilde{A}$ but at a great cost. However, there are techniques which can economically be applied to various aspects of computer system design to reduce $\tilde{A}$ significantly for general applications. The following chapters examine some of these approaches and their expected effectiveness based on the traces used.

CHAPTER 3

MINIMIZATION OF $\tilde{A}$

3.1. Introduction

In Chapter 2 the variables A and H(S) were defined; A being the addressing overhead incurred by a program and H(S) the uncertainty or entropy of the memory reference stream for the computation portion of a program. It was shown that $A \geq H(S)$. Actual calculations using program traces demonstrated that $\tilde{A}$ (A estimate) was, in fact, significantly greater than $\tilde{H}(S)$ (H(S) estimate). This means that there is considerable room for improvement. The question this chapter examines then is, can $\tilde{A}$ be reduced and if so, by how much and how easily?

Various methods for improving $\tilde{A}$ are briefly discussed, including adjusting the displacement and the number of address registers, improving compilation procedures and inclusion of architectural features that improve a CPU's addressing capabilities. Though many of these suggestions are applicable to almost any architecture, they are mostly aimed at an R+D architecture like the IBM 360.

3.2. R+D Optimization

In an R+D architecture, the primary contribution to $\tilde{A}$ is incurred in two ways; first, by the register select and displacement bits required by all memory reference instructions, and second, by the instruction and data bits required to load and manipulate the address

(base) registers. There is an obvious trade-off here, since if we use fewer registers and a smaller maximum displacement the r+d contributions to $\tilde{A}$ will be smaller. With fewer registers, however, they will have to be loaded more often, possibly increasing $\tilde{A}$. In this section an approximate model of this trade-off is proposed. The model is now developed.

Definition 3.2.1: A Register Fault occurs when a base register must be loaded in order to make a memory access. □

Definition 3.2.2: The Register Fault Rate, denoted by α, is the probability that a register fault will occur on the next reference. □

$\tilde{A}$ can be approximated then as

$$\tilde{A} \cong \frac{a_\alpha + a_\beta + a_\gamma + a_\delta + a_\varepsilon}{N_c}$$

where $a_\alpha = (2r+d + 32 + 2)RX_{BL}$, the total number of bits due to data register loads. RX_{BL} is the number of base register loads computed by the filter program, 32 bits are loaded, 2 opcode bits are used to name the addressing mode, r+d bits are required to access the operand to be loaded, and r bits are required to specify the destination register.

$a_\beta = (2 + r+d)(RX_F + RSI_F) + (2 + 2r + 2d)SS_F$, the total bits due to memory reference instructions.

$a_\gamma = (12 + r)(RR_{XA} + RR_{XL}) + (16 + r+d)(RX_{XA} + RX_{XL}) + 3RX_F + 32(RX_{XA} + RX_{XL})$, the total bits due to all indexing operations. The 12 and 16 are the number of bits in these instructions except for r and d fields.

$a_\delta = 2RR_F$, the total bits due to register reference instructions.

$a_e = (2r+d+2)RX_S + (r+d+2)LM + (16+d)RX_{BA} +$

$16(RR_{BA}+RR_{BL}) + 32(3LM+RX_S+RX_{BA})$, the total bits due to miscellaneous operations such as base register stores.

In the formulas above

r = number of bits for base register selection,

d = number of bits to specify displacement,

N_c = number of computation references,

addressing process statistics:

RR_{XL} = number of loads to index registers (register reference),

RR_{BL} = number of loads to base registers (register reference),

RX_{XL} = number of loads to index registers (memory reference),

RX_{BL} = number of loads to base registers (memory reference),

RR_{XA} = number of index register arithmetic instructions (register reference),

RR_{BA} = number of base register arithmetic instructions (register reference),

RX_{XA} = number of index register arithmetic instructions (memory reference),

RX_{BA} = number of base register arithmetic instructions (memory reference),

RX_S = number of stores from activated registers,

computation process statistics:

RR_F = number of register reference instructions,

RX_F = number of memory reference instructions,

SS_F = number of storage to storage instructions,

RSI_F = register storage and storage immediate instructions, and

LM = number of load multiple register instructions. On the average 3 registers are loaded per instruction.

Now define $\tilde{\alpha}$ (α estimate) as the fraction of references devoted to base register load fetches

$$\tilde{\alpha} = \frac{RX_{BL}}{N_c + RX_{BL}} .$$

$\tilde{A}$ can then be written as

$$\tilde{A} \cong \frac{a_\beta + a_\gamma + a_\delta + a_\varepsilon}{N_c} + \left(\frac{\tilde{\alpha}}{1-\tilde{\alpha}}\right)(2r+d+32+2).$$

The number of opcode bits for each addressing operation is assumed to be 2 bits, since there are basically 4 operations to be delineated

- RX, RS, or SI memory reference,
- SS memory reference,
- Base Register Load, and
- No memory reference (RR instruction).

(Note: these opcode bits are not explicit within the 360, but we assume their existence both here and in the filter algorithm, since they represent an approximate quantity of information that is required by the addressing process for its proper operation.)

The above model does appear to be quite complicated. However, by more accurately characterizing the addressing process, (for the proper $\tilde{\alpha}$) direct comparisons can be made to the actual trace results. In the model, all contributions to $\tilde{A}$ remain the same except for their r and d

components and the load rate $\tilde{\alpha}$. What is now needed is a model of the $\tilde{\alpha}$ vs. r+d process.

The first major problem we encounter concerns the allocation of registers within the program. This in itself is a difficult problem, which will be discussed later in this chapter. A simple model though can be created by making certain assumptions. First, consider the input stream to an addressing encoder as being the memory references generated by a Markov process. Let us assume that we know the probabilities, $p(m_i | g_j^{n-1})$, of this process, where m_j is a memory location and g_j^{n-1} is an n-1 gram of the n-1 previous locations referenced. These are called the nth order conditional probabilities, since they are used to calculate the nth order entropy $H_n(S)$. Assume for this model that no higher order behavior exists, that is that $H_n(S) = H_\infty(S)$. This assumption results in an nth order process (n-1th order Markov), since the $p(m_i | g_j^{n-1})$ are independent.

These references then enter a finite-state encoder which contains 2^r base registers of width w, and depending on the contents of the base registers, outputs a stream of codewords of the set $\{1t_r t_d, 0t_r t_d t_w\}$ where

$1t_r t_d$ represents register select plus displacement, and

$0t_r t_d t_w$ represents register select plus displacement and register load (t_r is loaded with t_w then displaced from with t_d)

where

t_r is a register select word of r bits,

t_d is a displacement word of d bits, and

t_w is a register load word of w bits.

This stream can then be decoded by a CPU-like mechanism containing addressing registers, thereby generating the original reference stream. The communication rate (in bits/computation reference) using an R+D coding mechanism can now be defined.

Definition 3.2.3: Let R be the coding rate of the above addressing architecture, i.e.,

$$R = (1+r+d)(1-\alpha)+(1+r+d+w)(\alpha)$$

or

$$R = (1+r+d) + \alpha w. \qquad \square$$

R is a little more convenient to use than A for assessing the efficiency of R+D coding, since it does not contain other miscellaneous operations that are included in A (e.g. indexing) and does not include the opcode bits required to delineate these other operations.

α_n then represents the proportion of register load insertions needed by a nth order process. These loads can be allocated by using the following guidelines (given the particular state of the coder, i.e. the contents of the set of base registers);

- Only reload a register when a fault, m^*, occurs (location m^* is referenced but not within range of any register).
- Reload one register per fault.
- The register to be reloaded, R_j, is selected to be the one with greatest expected next use time.
- The address loaded into R_j must include m^* in its range and should be chosen to maximize its expected reload time.

In a first order model, the set of referenced words are allocated to memory such that $m_i < m_j$ if $p(m_i) \geq p(m_j)$ (see Figure 8), this is called

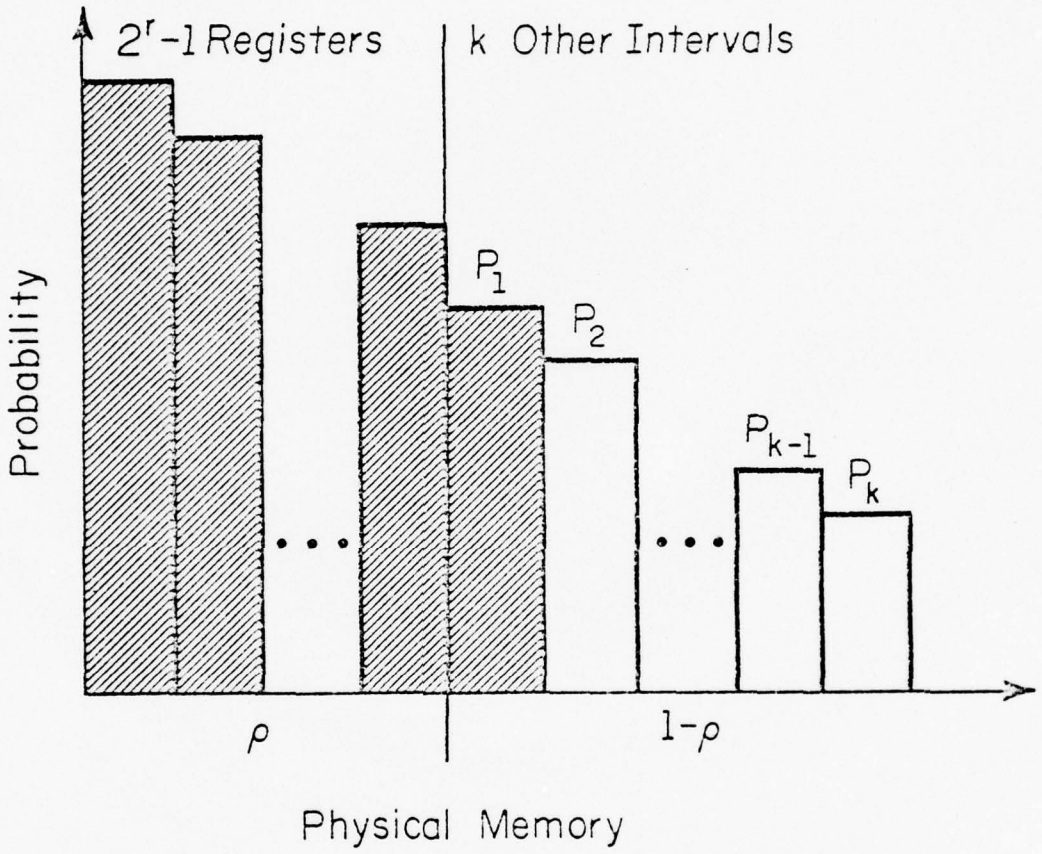


Figure 8. H_1 allocation model.

an H_1 allocation. Since higher order models become rather difficult, we shall assume a first order process for the model. Furthermore, we assume for simplicity, that the H_1 probability distribution is forced into discrete intervals, each 2^d locations wide. These restrictions allow a simpler model, but the α_1' obtained will be greater than the actual α_1, since less loading freedom is available. Using the register load guidelines it is easy to see that 2^r-1 registers should be loaded to cover the intervals with largest probability of reference and that the last register should be used to cover one of the "other intervals." The registers are said to be in state i when the last register covers the "other interval" associated with P_i. Let ρ = the probability of a reference within the range of the 2^r-1 fixed registers. Then $1-\rho$ is the probability of a reference within the "other intervals," i.e., $1-\rho = \sum_{i=1}^{k} P_i$. Therefore, the probability of being in state i is $\frac{P_i}{1-\rho}$, and the probability of a fault on the next reference, if the register is in state i, is $1 - \rho - P_i$ so

$$\alpha_1' = \sum_{i=1}^{k} (P_i - \frac{P_i^2}{1-\rho}).$$

Since the $\tilde{P}_i$ can be determined from the traces, $\tilde{\alpha}$ and $\tilde{A}$ can be computed for various values of r+d.

The model tends to overestimate $\tilde{\alpha}$ for the following reasons:

i) The registers contained only integer multiples of the displacement. This allowed a much simpler model, but also overestimates $\tilde{\alpha}$, since less freedom is available on register loading to cover current references and future probable references.

ii) In an actual program only the data references and branch fetches would require the use of the R+D architecture, since sequential instruction fetching is done by the program counter. Our first order model modeled all references, therefore in actual programs, $\tilde{\alpha}$ would be less than $\tilde{\alpha}_1$.

iii) It was assumed that S was a first order process (zeroth order Markov) which, as shown in Chapter 2, is far from true. This leads to more overestimation, since if a process is of higher order, a policy for memory and register load allocation can be implemented which uses this higher order information for further improvement. This is especially important when well-defined groupings exist, since the multiple codebook characteristic of the R+D architecture makes efficient use of such behavior.

There is one other simplification which contributes to the underestimation of $\tilde{\alpha}$. It has been assumed that 2^r registers are always available to the addressing process, however in an IBM 360, these registers are shared with the computation process and there could be fewer registers available for addressing at times when 2^r are needed, thereby increasing the fault rate over that required if all 2^r registers were available all the time. Incorporating the computation process into our model though, would have made the analysis much too difficult. This model therefore assumes a constant 2^r registers always available for addressing.

Figures 9 and 10 show the curves of $\tilde{A}$ versus the sum r+d for our four traces. The sum r+d was used since there was little variation of $\tilde{A}$ when r and d were varied within a constant r+d. We began with

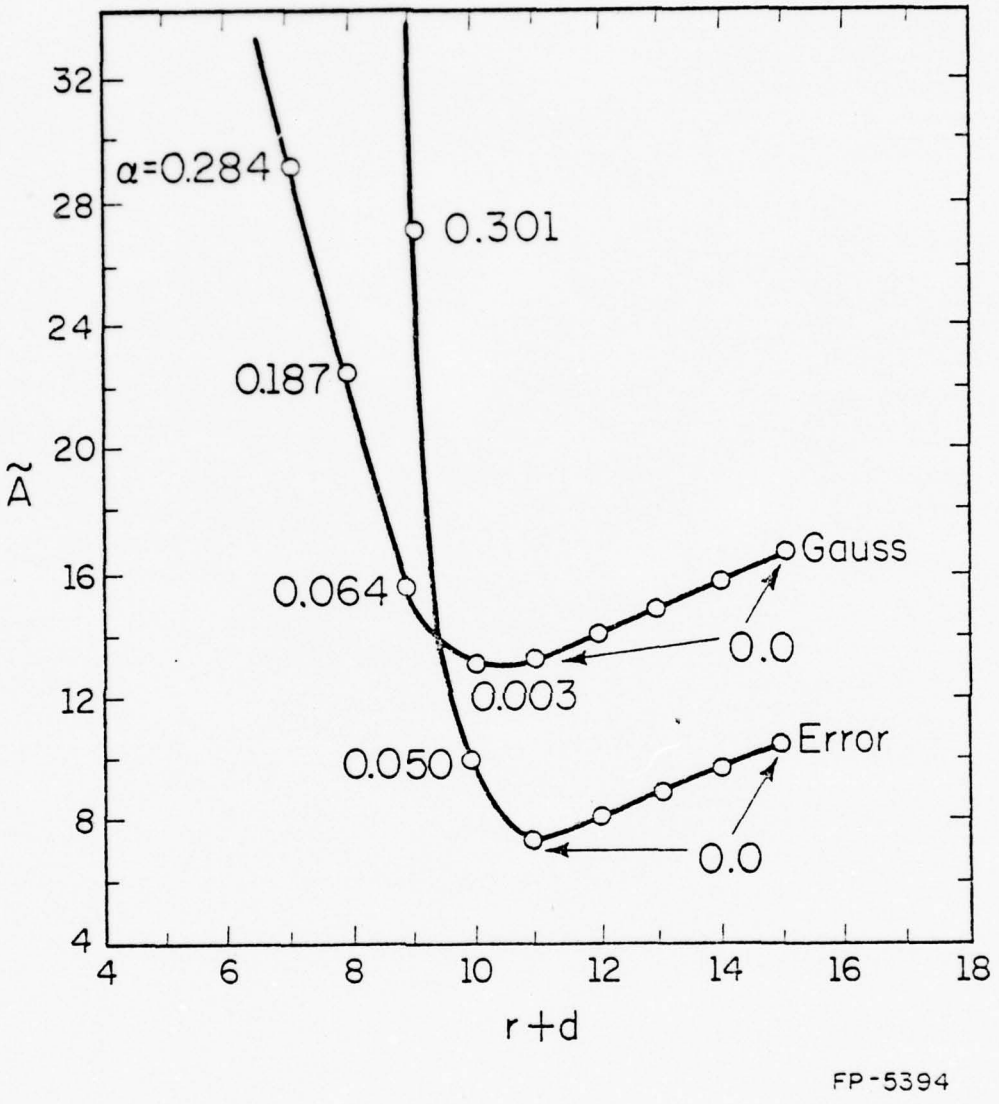


Figure 9. R+D trade-off (GAUSS,ERROR).

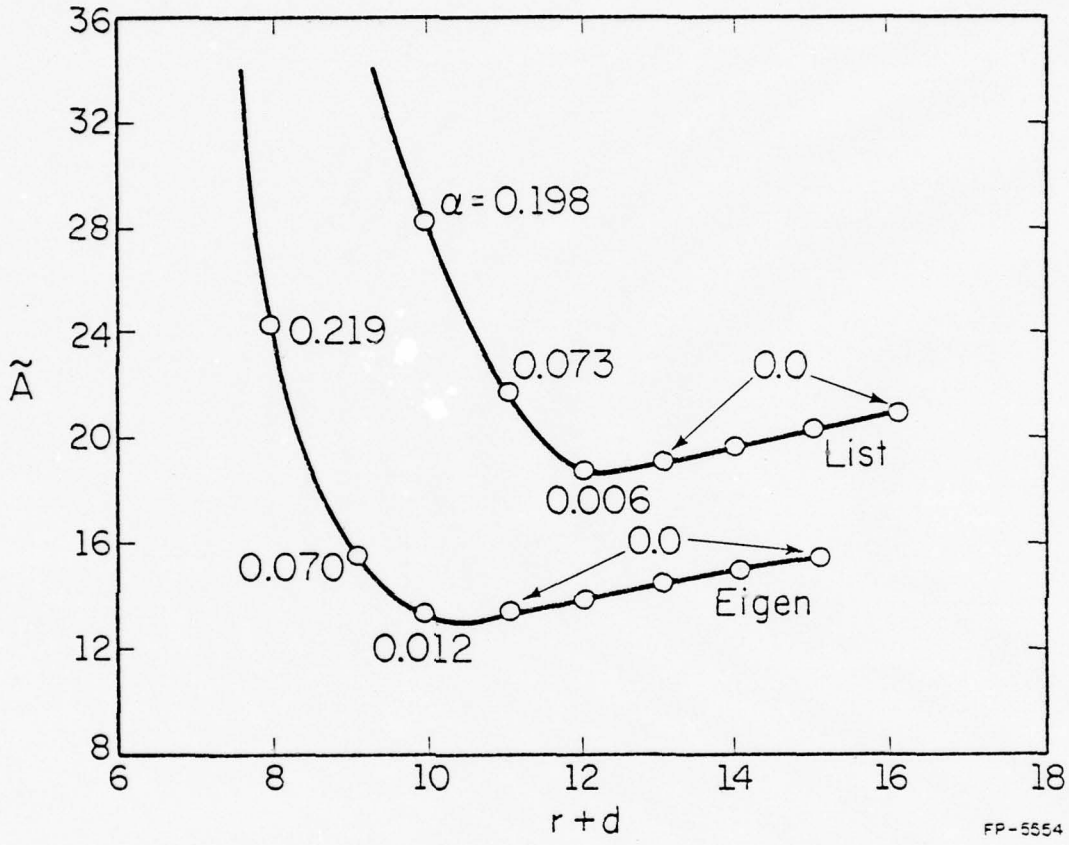


Figure 10. R+D trade-off (LIST,EIGEN).

r+d = 15 bits for GAUSS, ERROR, and EIGEN, since the maximum number of address registers used by those programs was 7, $r = \lceil \log_2 7 \rceil = 3$ and d = 12. LIST, however, used a maximum of 11 registers for addressing so that data set began with r+d = 16.

The results are encouraging. Namely, r+d can be reduced significantly before faulting occurs at all, and though the curves rise steeply to the left (indicating that the basic working space is too small) the minimum $\tilde{A}$ occurs at 10 or 11 bits (12 for LIST) for the program traces. This means that with EIGEN, for example, one could use 8 registers and a maximum displacement of 7 bits instead of the original 12. Thereby $\tilde{A}$ is decreased from 24 to about 13 bits/computation reference.

Note that r+d could be reduced further (moving up the curve to the left) to get approximately the original values of $\tilde{A}$. With GAUSS if r+d = 8, $\tilde{A} \approx 19$, which is not much greater than the original $\tilde{A}$ of 17.2. This could involve, for example, 4 address registers and 6 bits for displacement. Although there is a significant fault rate at this point, which slows down the addressing process, the RX instruction size has been reduced by seven bits. This gives the designer seven more bits to implement (2^7 times) more complex and helpful instructions, thus possibly speeding up the computation process. In order to preserve design flexibility, one must be careful not to go too far to the left on the design curve. However, as discussed above the $\tilde{\alpha}$ used to compute these curves is overestimated. Another reason to avoid going too far to the left is that by reducing d too much, the directly accessible sections of

memory become more fragmented and difficult to use. Although this is true, there is an associated advantage, since one is forced to encode the reference stream more tightly around existing spatial localities, therefore removing seldom referenced information which takes up valuable CPU/memory bandwidth and contributes little or nothing to the CPU's computational efficiency.

The two main conclusions that can be drawn concerning these results are that the IBM 360 r and d values are too large, at least for our traces, and the original compiler generated allocation for the programs was inefficient. The latter can be seen from the fact that all $\tilde{A}$ calculations are slightly less than the actual $\tilde{A}$ values, since the model computes $\tilde{\alpha} = 0$ for all traces at r+d = 15, whereas for the actual execution of the programs on the 360 $\tilde{\alpha} \neq 0$. GAUSS, for example, had $\tilde{\alpha} = .034$. This discrepancy is important for two reasons. First, though .034 seems small, it can be seen from Figures 9 and 10 that $\tilde{A}$ is quite sensitive to small changes in $\tilde{\alpha}$. Second, if our simplistic model, which tends to overestimate $\tilde{\alpha}$, predicts an $\tilde{\alpha} = 0$, then it should be easily obtainable by a real system. The reason for $\tilde{\alpha} = .034$ is then poor register allocation by the compiler on the 360.

We finally ask, how good a coding mechanism is the R+D architecture or, in fact, multiple codebook encoding in general? Table 9 shows the calculation of the estimated rate, $\tilde{R}$ (using the value for r+d that gives minimal $\tilde{A}$ with w = 32), for each trace. $H_0(S)$ and $\tilde{H}_1(S)$ are repeated here for convenience. For all the traces, $\tilde{R}$ is fairly close to $H_0(S)$. This is the chief advantage of R+D addressing,

Table 9

Coding Rate for Trace Programs

	GAUSS	ERROR	LIST	EIGEN
r+d	10	11	12	10
$\tilde{R}$	11.09	12.09	13.19	11.38
$H_0(S)$	10.36	10.74	11.27	10.80
$\tilde{H}_1(S)$	7.88	9.33	9.58	8.62

namely, the fact that one can code more or less independently of w, i.e. the coding rate depends more on $H_0(S)$ and less on the total address space 2^w. Though our programs do not code at a rate less than $H_0(S)$, $H_0(S)$ is not an absolute lower bound for this type of multiple codebook encoding. To lower the coding rate further, more and more knowledge of the memory reference process and its past behavior must be used by the CPU. For standard computer architectures though, R+D encoding is reasonably efficient and easy to implement.

$\tilde{R}$ could also be improved by using variable length encoding of instruction addressing fields. For example, by using variable length register names; placing commonly used addresses in the register with the shortest name. Displacement could also be encoded with variable length codes, though either extra bits must be available to give the length of the displacement field or the set of allowable displacements must form a uniquely decipherable codeword set. Frequency based displacement encoding was used on the Burroughs B1700 [13], ostensibly to reduce the coding rate to $\tilde{H}_1(S)$. This technique, however, would not be much more efficient than a properly tuned R+D architecture, since for all our traces, the difference between $\tilde{H}_1(S)$ and $H_0(S)$ is usually only a few bits, and in the B1700, two bits are used just to denote field length.

It can be concluded from the above results that there are wasted bits in the IBM 360 architecture. However, with proper design, an R+D architecture can code fairly efficiently. A technique has been presented for analyzing such architectures. This architecture is easy

to use and can be embellished further. It was seen that the allocation of address registers must be done well, since $\tilde{A}$ is sensitive to small changes in $\tilde{\alpha}$. In the next section, the problem of better compiler allocation of address registers is discussed.

3.3. The Address Register Allocation Problem

As was demonstrated in the previous section, if one could allocate the address registers more optimally, α could, in many cases, be reduced. Except for the work on index register allocation by Horowitz et al. [14] , we are not aware of any work in this area.

Each variable has a context or scope, which is that portion of the program where the use of that variable is defined. A variable's context could arise explicitly from a declaration statement or implicitly by actual use of the variable, depending on the syntax of the particular language being used. There are two types of variable allocation problems: intracontext and intercontext. Intracontext allocation is concerned with the allocation of address registers (to allow access to all variables active within the current context) and address register loads within a context. Intercontext allocation is concerned with the allocation of address registers and address register loads when switching from one context to another. There are two classes of languages: Non-Block Structured Languages and Block Structured Languages. They differ mainly by their emphasis on either intercontext or intra-context allocation. In Non-Block Structured Languages, such as FORTRAN or COBOL, the context is generally the entire program (sometimes

including subroutines). In this environment, allocation is almost exclusively intracontext and must be done on a large scale with a large number of variables. In Block Structured Languages, such as ALGOL and PASCAL, variables are declared within blocks, which can be fairly small and can be nested to an arbitrary level. The range of the block then becomes the context for the variables declared within the block. Intracontext allocation is still necessary, but if the blocks are small enough, i.e., if the program is well structured, intercontext allocation will clearly be a dominant problem.

Section 3.3.1 looks at intracontext allocation and Section 3.3.2 deals with intercontext allocation. Though both types of allocation are theoretically similar, they are dealt with separately, since they are generally implemented quite differently.

3.3.1. Intracontext Allocation

The intracontext allocation problem has two stages. In the first, the program is laid out in memory, each program word and each data word being assigned to a unique location. This process will not be discussed any further here, since it is examined in more detail in Section 4.4. It will be assumed then that before the second stage is entered the program and data are allocated to memory to take advantage of as much second order information as possible; i.e., that there is a large degree of spatial locality (the possible successors to a referenced word are spatially close to that word).

In the second stage, the address register load instructions and all displacements are inserted into the program in such a way as to

reduce the register fault rate α. To be able to allocate optimally, one would need to know program loop frequencies which are data dependent. Of course, it is always possible to restructure a program adaptively each time it is run, optimizing the allocation for a restricted set of data. This approach will not be considered here.

Register load instruction placements in the code and register displacements must be chosen so that any possible data reference or branch destination is covered by some active address register. This is, of course, possible if we place a load before every instruction that references memory. However, we would like to place our load instructions such that for any possible path through the program, including large numbers of loops or cycles, α is minimal. Such an objective is open-ended, since loops could be iterated indefinitely. Also, depending on the data, it may be that such an objective could be inconsistent. This objective though is a good starting point for the development of heuristic algorithms.

It is possible to estimate the complexity of a brute-force algorithm to solve the address register load allocation problem. Assume there are n memory reference instructions in the program. There are then 2^n possible allocations of register load instructions (one before each memory reference instruction), 2^r possible register allocations for each node and 2^d possible displacements for each memory reference instruction. Therefore there are $\mathcal{O}(2^{n+r+d})$ possible solutions to the allocation problem. This can be quite large, since n is usually several hundred.

Multiple codebook ideas can be used to develop heuristics for intracontext allocation. One example is as follows; the memory references can be spatially clustered heuristically into clusters of size 2^d. (The clustering being done to minimize the probability of leaving a cluster.) The compiler can then traverse the program such that each possible path is encountered. An active cluster is one which has an address register pointing to it. There are then 2^r active clusters at any time. A location fault occurs when a memory reference instruction is encountered that references a word which is not in an active cluster. At this point an address register load is inserted by the compiler, activating the faulted reference's cluster. The register to be loaded, i.e., the cluster to be deactivated by the loss of the register holding its base address, is chosen to be one whose next use is estimated to be the furthest away from the current instruction. This algorithm could be greatly improved with more accurate estimates of loop and branch frequencies.

The main point of this section is to define and demonstrate the complexity of the problem of intracontext address register allocation. It can be seen that any significant increase in R÷D architecture efficiency over that shown in Section 3.2 would not be easy to effect by this means.

3.3.2. Intercontext Allocation

Because it is more data dependent, compiling in a Block Structured Language environment would seem more difficult. This is not entirely true, since now the primary burden of coding falls onto the user,

providing that the context switching mechanisms of the architecture operate efficiently. Even here we are basically operating in a multiple codebook coding environment. When a context is entered, we code within that context, if a lower level context is then entered we are coding within both contexts, etc. The user more or less determines how many variables he will declare at each level, and the context switching rate (by his choice of block size). The architecture should provide an efficient coding mechanism both within the context and for context switching. One can easily see that structured programming techniques should also lead to improved address encoding, since they stress the use of highly local modules thereby reducing the context switching rate.

The 360 is inefficient when executing Block Structured Languages for two reasons. First it has more displacement than is necessary for intracontext addressing. For example, Wichman [15] has shown that most blocks rarely need more than seven bits to specify all variables. Secondly, intercontext switching requires loading a base register. In fact, for nested contexts, all contexts should have active base registers, leading to a large number of base register loads in a Block Structured environment. Another example of 360 inefficiency is in array allocation. Local arrays cannot be allocated until a block is entered, since the array dimensions need not be supplied until that time. Currently, most compilers handle this problem by allocating a register each time a block is entered, this register points to a specially allocated, variable size area of memory which is used for block operand storage. Usually the allocated data area does not even contain the arrays themselves,

but rather dope vectors (Gries [16]) which contain pointers to the actual data area. The array buffer is then allocated dynamically by special code inserted into the program by the compiler. This technique requires multiple level indirection, a difficult matter for the 360 which must reload address registers continually.

Various approaches have been used to make the above operations more efficient. The most common technique is to use a form of intermediate storage. This approach places another level of memory between primary memory and the register set. It is not, however, a cache memory, since a cache still requires full addressing. Intermediate storage, through the use of a reduced name space, allows one to reference with reduced addressing requirements. Some schemes have had separate, intermediate stores for operands, vectors, and instructions. A good example of this type of architecture is the descriptor based or tagged architectures (Welch [17]).

All such schemes basically have the following characteristics;

i) a small size intermediate storage area between memory and registers, which reduces the mapping requirements of the instruction set which operates on this store, and

ii) the existence of either an automatic mechanism which loads this store on a demand basis (such as the Name Store of the MU5, Ibbett [18]), or the use of specific instructions (block moves for example) to do the job.

A detailed analysis of the above technique is beyond the scope of this section, however, we can conclude that with this technique, it is

possible to address somewhat more efficiently in Block Structured Languages. There is no reason why such an approach should not be codable with $\tilde{R} \approx H_0(S)$. For most of our traces this would be an improvement from the original $\tilde{A}$. For example, there would be significant improvement for LIST ($\tilde{H}_0 = 11.27$, $\tilde{A} = 24.1$), which incidentally is our only Block Structured Language trace and has the highest $\tilde{A}$.

The main conclusion one can derive from this and the previous section, is that coding on a standard register plus displacement architecture cannot easily be driven below $H_0(S)$. Consequently, due to limitations in register allocation procedures and in standard R+D architectures, improvement beyond that shown in Section 3.2 would be difficult and expensive. This holds true regardless of the class of language being used. The incorporation of higher order information into standard R+D architectures should therefore not be considered further.

In the next section several general improvements to IBM 360 addressing architecture are assumed for one of our traces. Results show some approximate gains in addressing efficiency which can easily be achieved.

3.4. Architectural Improvements

So far in this chapter, two areas of $\tilde{A}$ reduction for more or less standard R+D architectures have been examined, that of R+D optimization and the related area of improved address register allocations. In this section indexing and other miscellaneous contributions to $\tilde{A}$ are

explored. Also, an example of the improvement that can be easily obtained by the above techniques and some ideas on different ways of implementing standard addressing architectures are presented.

In some programs such as GAUSS, indexing overhead contributes significantly to $\tilde{A}$. This contribution can be reduced by adding special indexing capabilities. (Note: Some programs do not have a large number of index operations, so depending on the job mix, it may not be cost effective to add indexing instructions and hardware. Also for some job mixes the increased indexing efficiency may save fewer bits than the number of bits added by the new opcodes.)

To demonstrate the amount of indexing overhead that indexing operations incur, a matrix multiply, which is similar to the kind of operations used by GAUSS, is analyzed. Table 10 (flowchart in Figure 11) lists a short IBM 360 assembly language program that multiplies two matrices, A ($n \times m$) and B ($m \times t$), placing the result in C ($n \times t$). The computational instructions are starred in Table 10 with the remaining instructions being indexing or addressing overhead. The program is by no means optimal, but is intended as an example of the overhead that can be incurred in vector and matrix operations on the 360.

Since the inner loop occurs $n \cdot m \cdot t$ times, the next loop $n \cdot t$ times, and the outer loop n times, $\tilde{A}$ can be computed by summing the bits in the addressing/indexing instructions, their data fetches, and the addressing portions of the (starred) computation instructions

Table 10

Matrix Multiplication Program

Register Contents:

R1 - BA, Matrix A's Base Address
R2 - BB, Matrix B's Base Address
R3 - BC, Matrix C's Base Address
R4 - I
R5 - J
R6 - Constant "4"
R7 - IB
R8 - T*4
R9 - M*T*4
R10 - IA
R11 - IC
R12 - Partial Result Register
R13 - Running Sum Register
R14 - Unassigned
R15 - Constant "0"
R16 - Program and Data Base Register

Defined Memory Locations:

(R16) + 0 - Base Address for Matrix A,
4 - Base Address for Matrix B,
8 - Base Address for Matrix C,
12 - N
16 - T
20 - M
24 - "0"
28 - "4"
32 - Start of Program.

Comment: The 360 has byte (8 bits) addressing, but words are 4 bytes long, therefore all addresses are in multiples of 4.

Table 10 (continued)

Contribution in bits to $\tilde{A}$	Code			Comments
224		LM	R1, R6, 0(R16)	R1 ← BA; R2 ← BB; R3 ← BC; R4 ← N; R5 ← T; R6 ← "4"
64		L	R15, 0(R16)	R15 ← "0";
16		LR	R7, R15	IB ← 0;
16		LR	R10, R15	IA ← 0;
16		LR	R11, R15	IC ← 0;
16		LR	R13, R15	R13 ← 0;
16		LR	R8, R5	R8 ← T*4;
16		MR	R8, R6	
16		LR	R9, R8	R9 ← M*T*4;
64		M	R9, 20(R16)	
18nmt	LOOP:	L*	R12	R12 ← A[IA]*B[IB];
18nmt		M*	R12	
0		AR	R13	R13 ← R13 + R12;
16nmt		AR	R10	IA ← IA + 4;
32nmt		BXLE	R7, R8, LOOP	IB ← IB + T*4; IF IB ≥ M*T*4 THEN GOTO LOOP;
18nt		ST*	R13, 0(R11,R3)	C[IC] ← R13;
0		LR*	R13, R15	R13 ← 0;
16nt		AR	R11, R6	IC ← IC + 4;
32nt		BCT	R4, ALT	I ← I-1; IF I > 0 THEN GOTO ALT;
64n		L	R4, 16(R16)	I ← T;
32n		BCT	R5, ALT	J ← J-1; IF J > 0 THEN GOTO ALT;

Table 10 (continued)

32		BC 15, DONE	Terminate
64nt	ALT:	L R10, 12(R16)	IA← (N-I)*M*4;
16nt		SR R10, R4	
16nt		MR R10, R6	
64nt		M R10, 20(R16)	
64nt		L R7, 16(R16)	IB← (T-J)*4;
16nt		SR R7, R5	
16nt		MR R7, R6	
32nt		BC 15, LOOP	

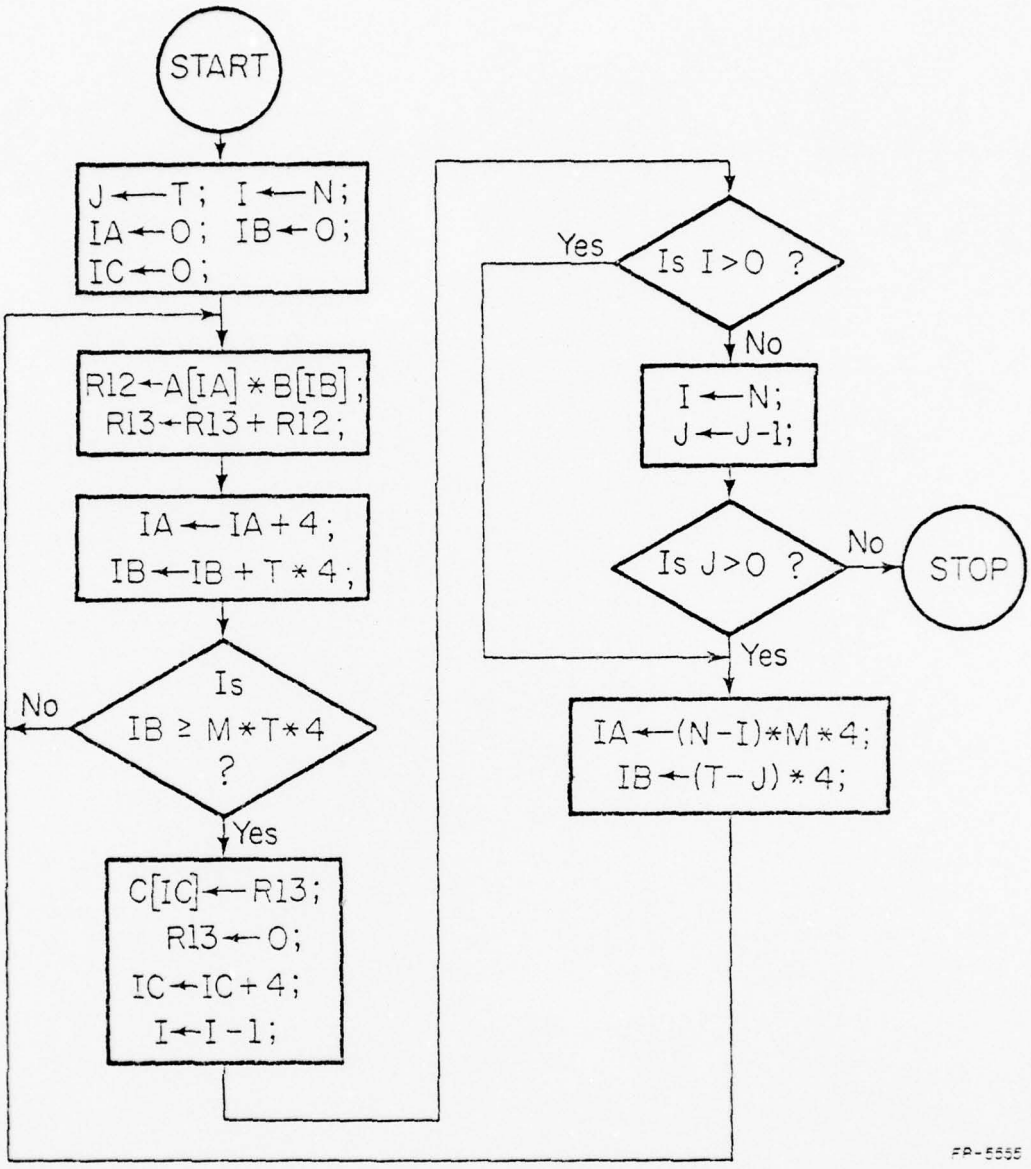


Figure 11. Matrix multiplication program flowchart.

$$N_c = 5mnt + 3nt$$

$$\tilde{A} = \frac{496 + 84nmt + 354nt + 96n}{N_c} \text{ , and}$$

$$a_Y = 17(mnt + 5nt + 7) + 64(3nt + n + 3) + 3(2nmt + nt)$$

(a_Y is the indexing overhead which is defined in Section 3.2). Letting $n = m = t = 14$ as in GAUSS we have

$$\tilde{A} = 21.2 \text{ bits/computation reference, } \tilde{A}_T = 42.6$$

($\tilde{A}$ being 49.7% of $\tilde{A}_T$), and $a_Y/N_c = 8.3$.

Incidentally, this data is somewhat close to that for GAUSS where $\tilde{A} = 17.2$ and $a_Y/N_c = 5.0$.

Most of the overhead computation in the example is devoted to incrementing and checking subscripts and mapping those subscripts into a one-dimensional array. An obvious solution to this problem is to provide a set of two-dimensional matrix operations (with a vector being a matrix with one dimension being unity). Such operations can be easily implemented on a microprogrammable machine. Furthermore, since we are proposing to reduce displacement by at least 3 or 4 bits, the extra opcodes required can be easily accommodated. Such matrix instructions would no doubt cover most array operations, since few arrays have more than two dimensions. (Wichman [19] has discovered that 99.5% of all arrays in the ALGOL programs he examined had one or two dimensions.) With a matrix multiply operation, the addressing uncertainty (given the dimensions and base addresses of the arrays) is close to zero, and so is the addressing overhead. In this case then the CPU is being given specific knowledge of

the addressing characteristics of the program so that it can predict addressing exactly without further information. Creating instructions to perform the desired task may seem an extreme solution. However, matrix and vector operations are not so specialized. Common matrix operations occur in many different classes of program environments. Of course high level languages and compilers would have to be adapted to allow the user to take advantage of the matrix facilities of the machine.

It is not unreasonable to assume that nearly all of the indexing overhead in GAUSS could be covered by such special operations. Such operations then would reduce a_γ/N_c from 5 to nearly zero.

Furthermore if we reduce such wasteful operations as base register stores and base register arithmetic, a_ε/N_c (≈ 3) can also be forced to nearly zero. In Section 3.2, $\tilde{A}$ was reduced by about 25% (from 17.2 to about 13) by reducing r+d. Gathering together the above improvements, one can effect a total, prorated decrease in addressing overhead to approximately;

$$\tilde{A} = (17.2 - a_\varepsilon/N_c - a_\gamma/N_c)*(1 - .25) \approx 7 \text{ bits/computation reference,}$$

which is an improvement of about 58% in addressing efficiency. This is a significant improvement and was fairly easily obtained even within a standard architecture. $\tilde{A}$ is thereby brought down to approximately $H_0(S)$. (Actually $\tilde{A}$ is less than $H_0(S)$ for GAUSS, because of our being able to take advantage of the higher order structure of the index operations. This, of course, is not always possible.)

AD-A044 313 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
ANALYSIS OF MEMORY ADDRESSING ARCHITECTURE.(U)
JUL 77 D W HAMMERSTROM DAAB07-72-C-0259
UNCLASSIFIED R-777 NL

2 OF 2
AD A044313

END
DATE FILMED
10-77
DDC

It appears though that any improvement beyond this point would be difficult and costly. In other words, further improvements would be a matter of marginal returns on further investments, at least when dealing with additional CPU improvements. In the next chapter a totally different approach to $\tilde{A}$ reduction is taken by examining new types of memory architectures.

Before concluding this section, however, a brief comment is in order concerning the parallelism inherent in the addressing and computation processes. These processes operate more or less independently of each other, except when the computation process sends the results of certain operations to the addressing process (for branch decision making) and receives instructions and operands from the memory through the addressing process. Many large systems have tended to split processing along these lines, though they have not explicitly defined the addressing process as such. One exception to this is the work of Flynn [20], in which he explicitly defines three separate machines which execute in tightly coupled parallelism. These machines, however, do not correspond exactly to the definitions of computation and addressing used here.

Consequently, though we do not actually reduce $\tilde{A}$ by the splitting up of these processes, there is still a great potential for significant speed gains by taking advantage of such parallelism. For example, one could define a machine which contains an address processor and a computation processor. By extension there could exist a pool of computation processors served by a large number of address processors.

In conclusion, it was shown in this section, that there is room for significant improvement in addressing overhead in the 360 implementation. This improvement can be achieved fairly easily and tends to increase overall architecture efficiency. Such improvement, however, is only possible up to a point. Beyond this point, the capabilities of the compilers and architectures become unreasonably strained for marginal further gains.

3.5. Summary and Conclusions

In this chapter the problem of $\tilde{A}$ reduction with standard architectures was examined. It was demonstrated that in the R+D architecture of the IBM 360 the number of registers and the displacement can be reduced significantly. It was also shown that the R+D architecture is reasonably efficient, especially when executing a program which has tightly bound groupings of referenced locations. Currently, address register allocation is done poorly and can be easily improved to a point, but going beyond that point is an exceedingly difficult problem.

Other improvements are possible in the indexing overhead, which can be fairly large for some programs. The inclusion of matrix operations in the CPU instruction repertoire can certainly increase addressing efficiency in certain program environments essentially using the opcode field to give the CPU knowledge of commonly used accessing patterns.

Section 3.3 examined the possibility of further reduction of α through better register allocation. This section concluded that such

reduction of α was indeed difficult. Hence further reduction of $\tilde{A}$ by more optimal register allocation should not be considered.

The main conclusion that can be drawn from this chapter is that it is relatively easy to reduce $\tilde{A}$ significantly until it is reasonably close to $H_0(S)$. Note also, that many of our changes result in fewer fetches. Therefore not only are we reducing the addressing overhead and the CPU/Memory bandwidth, but also the number of memory accesses. However, $H_0(S)$ is still much larger than $\tilde{H}(S)$ or even $\tilde{H}_2(S)$. Therefore, reducing $\tilde{A}$ further would require major and expensive changes in an R+D CPU architecture. However, rather than concentrate any more on the CPU, we now focus our attention on the improvement of memory architecture. The nature of these changes and reasons for them, as well as some experimental results are presented in Chapter 4.

CHAPTER 4

SECOND ORDER MEMORIES

4.1. Introduction

In Chapter 2 the addressing overhead, A, was defined and then obtained for various programs. It was shown that A is lower bounded by the information content of the computation process referencing stream, H(S), and that with real programs $\tilde{A}$ (estimated A) is one to two orders of magnitude greater than $\tilde{H}(S)$ (estimated H(S)). In Chapter 3 various methods of reducing A to increase addressing efficiency were examined, from improved and more finely tuned architecture to compiler design. All these improvements, however, assumed a standard random access memory architecture. In this chapter we examine second order memory architectures, that attempt to take advantage of $\tilde{H}_2(S)$, which is much smaller than $H_0(S)$ and $\tilde{H}_1(S)$, and for our traces was always less than 2.

4.2. Theory of Higher Order Memories

The main responsibility of the addressing process is to map (decode) the input stream (A bits/computation reference) into the output stream of memory references (B bits/computation reference). Let us define some related terms, and then B, more precisely.

Definition 4.2.1: An interaction with memory involves the sending, in parallel, of a packet of v address bits to the memory. □

Definition 4.2.2: A location fault occurs when more than one interaction with memory is required to fetch (or store) the desired word for the

computation process. The location fault rate, λ, is defined as:

$$\tilde{\lambda} = \frac{N_t}{N_c}$$

where N_t is the total number of interactions made, including addressing overhead, and N_c is the number of computation references made also

$$\lambda = \lim_{N_c \to \infty} \tilde{\lambda}. \qquad \square$$

Definition 4.2.3: The average number of bits flowing from the CPU to the memory is

$$\tilde{B} = \tilde{\lambda} v$$

or

$$B = \lambda v$$

where

$$v \geq 1. \qquad \square$$

Generally $v = w = \log_2$ (Memory Size), however a key point in Definition 4.2.3, is that smaller packets, $v \ll w$, can be sent, but then normally $\lambda > 1$. As we shall see in this section, certain memory architectures exist which allow a small v without a corresponding increase in λ. Note that when $v = w$, $B = \frac{N_t}{N_c} w$ as previously defined.

$\tilde{B}$ tends to be much larger than $\tilde{A}$. For example the LIST program had $\tilde{A}$ = 24.1 and $\tilde{B}$ = (total references/computation references)*32 = 47.3 bits/computation reference. If we could reduce $\tilde{B}$ we could reduce the mapping requirements. In fact, if we could reduce $\tilde{B}$ to below the presently observed $\tilde{A}$ we could eliminate much of the CPU's addressing responsibilities as well as reducing $\tilde{A}$, thereby increasing addressing efficiency. To accomplish this we need to take advantage of the higher order behavior

of the referencing stream. This speculation motivates the investigation of higher order memories.

Definition 4.2.4: An nth order memory, M_n, $(n \geq 1)$ has the property that the next location accessed by the memory depends nontrivially on the previous n-1 locations accessed in addition to the newly arriving addressing information. □

For each M_n one may use any integer length of packet, v. With $v = w$, M_1 represents a standard random access memory which requires w bits per interaction. With $v = 1$, the optimal form of M_1 is a random access memory with Huffman encoded addresses. Higher order memories begin to look desirable because of the following theorem and corollary.

Theorem 4.2.1: With $v = 1$, an nth order memory that retains and uses information regarding exactly the previous n-1 references in addition to its normal storage

$$B_n \geq H_n(S) \qquad (n > 0).$$

Proof: Assume that memory allocation is fixed, i.e., that a single codebook is being used. Assume also that no addressing information other than the previous n-1 references is retained. By Theorem 2.3.2 then we have

$$\sum_{i,j} p(m_i, g_j^{n-1}) n_{i|j} \geq H_n(S).$$

Since B_n is just the average number of bits required by the memory to code reference m_i (sending 1 bit at a time, i.e. $v = 1$) given the n-1 gram g_j^{n-1}, we have

$$B_n = \sum_{i,j} p(m_i, g_j^{n-1}) n_{i|j} \geq H_n(S). \qquad \square$$

Corollary 4.2.1: For $v > 1$

$$B_n \geq H_n(S).$$

Proof: Since $B_n \geq H_n(S)$ for $v = 1$, then any other v requires multiples of one bit. Consequently, coding could be at least as efficient as with $v = 1$, but never better. □

Though coding would tend to be more efficient with smaller packet size, it is not always more efficient. Of course with $v = 1$, coding is as efficient as with any $v > 1$, but any $v' > v > 1$, does not imply that one can always code more efficiently with v than with v'. This seeming anomaly is due to the fact that one must always send an integer number of packets for each reference, which may lead to wasted address bits when v is poorly chosen.

In our actual 360 programs, the $\tilde{B}$ calculated is greater than the idealized encoding of the previous theorem since not only are we dealing with an M_1 memory (with $v = w$), but also we must perform extra fetches occasionally to keep the addressing process supplied with information to implement its mapping function. If we eliminate this function then A becomes approximately equal to B. This approach actually degrades a standard 360 architecture. However the 2nd order entropy, $\tilde{H}_2(S)$, was significantly smaller than $H_0(S)$ and $\tilde{H}_1(S)$, and was, in fact, only 1 or 2 bits/computation reference. Consequently, if we can build an M_2, we have a tremendous opportunity to reduce B and therefore A as well, since the number of possible second order successors to any reference is much less than 2^w. Consequently, v could be reduced well below w without much increase in λ.

4.3. Examples of Second Order Memories

The next questions then are how is an M_2 constructed and can it actually be used efficiently? Recall that second order memories access the next location based on the current access and some bits provided by the CPU. Basically then we need to design a simple sequential machine within the memory. The only work in this area that we are aware of is that of Sholl [21] and Ouchi [22]. Sholl developed a Direct Transition Memory which is a finite-state machine used for microprogram control. It, however, considered only control flow. Ouchi's Orthogonal Storage Ring Memory is an efficient technique for using shift registers as random access memory. Much of the efficiency of the OSR memory arises from being able to capture some of the first and second order behavior of the program, but a highly complex layout is required within the memory chips.

In this section four examples of second order memories are presented with a discussion of the advantages and disadvantages of each. In succeeding sections the two most promising examples are examined in greater detail.

First, however, some discussion of the general problems that occur with M_2 memories is needed. We are assuming that memory inputs are chosen from a set of constant length codewords. This assumption is necessary from a hardware point of view, but does create some difficulty since most memory references made by the computation process have only one successor, but a few may have several hundred. Another difficulty is that in most cases, the M_2 must be created by using a RAM coupled with an in-memory mapping mechanism. The specific nature of this mapping

mechanism generally leads to inefficiencies in the way the RAM is used and also in the mapping process itself. One advantage of these memories though, is that of eliminating the mapping hardware within the CPU. Thus although some form of index registers would still be needed, the requirement of having base registers and operations on them is no longer valid.

For the proper operation of a true M_2, the memory must receive some information regarding the behavior of the program being executed. This information allows the M_2 to perform better than the M_1 and usually consists of the successor set for each location referenced. The obvious source of this successor information is the compiler or assembler, perhaps modified by a relocating loader at load time, or with a suitable linking mechanism in a paging system. For most instructions, since most are fetched sequentially only one successor exists. Even for branches and subroutine calls sets of possible successors can be derived. Since most instructions access data from the same location these too are not difficult. The successor identification problem becomes sticky though when indirection and indexing come into play. For Non-Block Structured Languages, the compiler should be able to at least identify the largest set of possible successors with the help of user specified parameters. For Block Structured Languages, the problem appears difficult, but not impossible. Section 4.5 contains a more detailed discussion of this problem and some possible solutions.

Following are four examples of second order (M_2) memories.

4.3.1. CCD Memories

A CCD (Charge Coupled Device) shift register memory is a very simple M_2, since the next location accessed depends on the previous location accessed. The problem with CCDs is that location faults would be numerous.

Since H_2 is typically greater than 1, the average uncertainty to be resolved at each reference is greater than two choices. This means that even with optimal allocation, which is extremely difficult, we would have a large λ. Furthermore, since CCD's typically have a linear or multilinear organization, the penalty incurred per location fault may be very large. Consequently, a one dimensional shift register M_2 is not considered further.

4.3.2. Segmented Counter Accessed Memory (SCAM)

The chief limitation of the CCD memory was that it had only one dimension. By using a RAM with several counters (incrementing/decrementing registers) concatenated together and pointing to the desired location, one can create a pseudo n-dimensional torroidal shift register memory. The only bits that need be sent to the memory are those required to decode the next shift direction. The memory then can be visualized as having memory locations laid out on an n-dimensional torroid in n+1 space with a read head, residing above the last location referenced, capable of moving one step in any one of 2n directions (depending on which counter is selected and the count direction). Figure 12 shows an example of such a memory. The memory surface appears torroidal, since the memory is

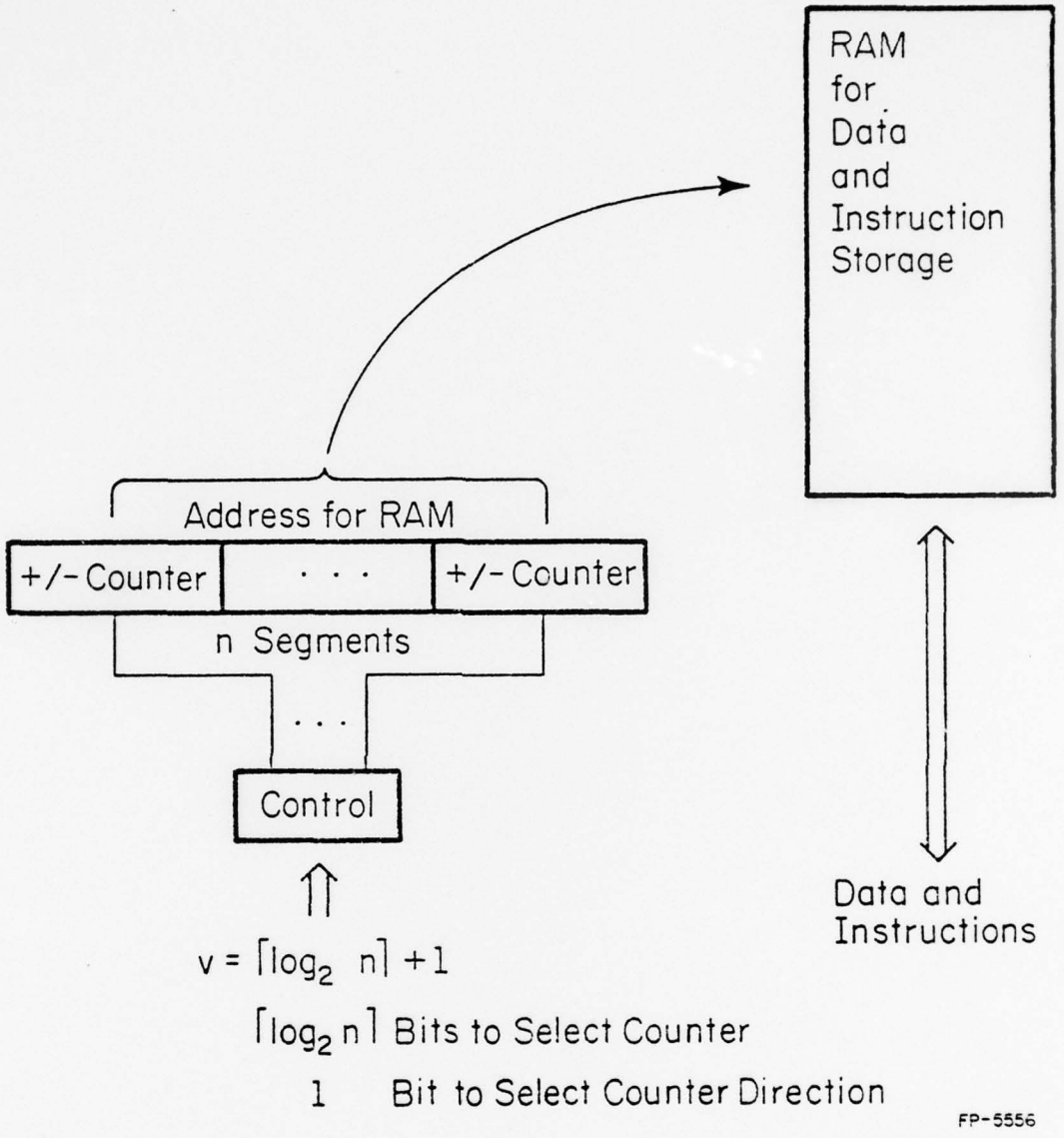


Figure 12. Segmented Counter Accessed Memory (SCAM).

finite and circular in all directions. This design should reduce the location faulting encountered in the CCD memory. Another advantage is that the packet size, $v = 1 + \lceil \log_2 n \rceil$, is independent of the memory size (though λ may increase with increasing memory size). Good allocation, however, again becomes important and difficult. Also because of the CPU having to keep track of the "read head" location and move it accordingly, paging and loading would be non-trivial. Penalties for location faults can be quite large if n is kept small.

Thus, although this M_2 does have some advantages, we feel its disadvantages are sufficient not to warrant examination of this type of memory architecture any further at present.

4.3.3. Segmented Random Access Memory (SRAM)

Although this memory is not a true M_2, it has enough "second order" features to merit its study.

It is not necessary to know the successors of a location in order to allocate an SRAM. Optimal allocation, however, is extremely difficult and requires such knowledge plus that of the conditional probabilities, $p(m_i | m_j)$. There are heuristic allocations though, such as the H1 allocation, that do a fairly good job.

Basically an SRAM has an in-memory address register (Figure 13) pointing to a RAM. The address register consists of k segments of a bits each. For each interaction $v = \lceil \log_2 k \rceil + a$ bits. These bits access one of the segments and replace its contents. Multiple interactions (at most k) are required whenever multiple segments of the address register must be changed for the next reference.

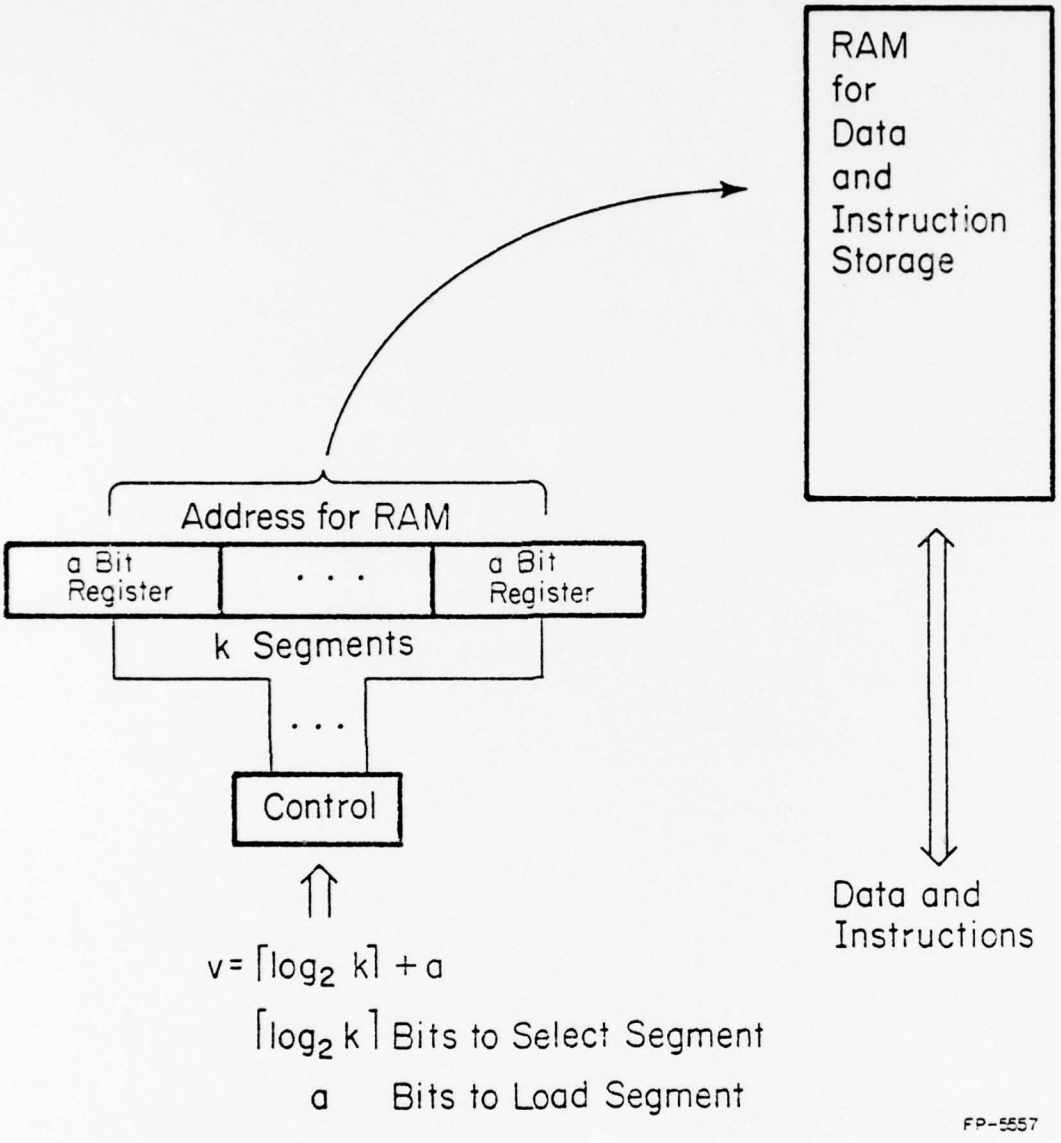


Figure 13. Segmented Random Access Memory (SRAM).

This type of memory is a generalization of a type of architecture used in dynamic 4k and 16k MOS RAMs, where the address is multiplexed into the memory chip in two halves to save pins, but built in "page mode" addressing allows a row (left half of address) to be repeatedly accessed with only the column (right half of address) being specified for each reference.

Section 4.4 analyzes this memory in more detail, including results of an actual simulation of memory operation.

4.3.4. Successor Accessed Memory (SAM)

A Successor Accessed Memory is perhaps the most ideal M_2 of all those we have examined. It gives the best performance, but is also the most difficult to use and requires the most complex address mapping hardware within the memory.

Basically a SAM has an in-memory RAM (see Figure 14), which stores data and successor pointers. It should be noted here, that large reductions in storage requirements are possible through more efficient pointer encoding. The scheme presented in Figure 14 is a straightforward approach that allows a simple analysis. Pointers to up to 2^v successors of the last reference are loaded into registers. To access a SAM, v bits are sent to select one of 2^v successor registers, the next location is accessed and the pointers to its successors are placed in the registers. This register loading could be overlapped with memory/CPU data communication.

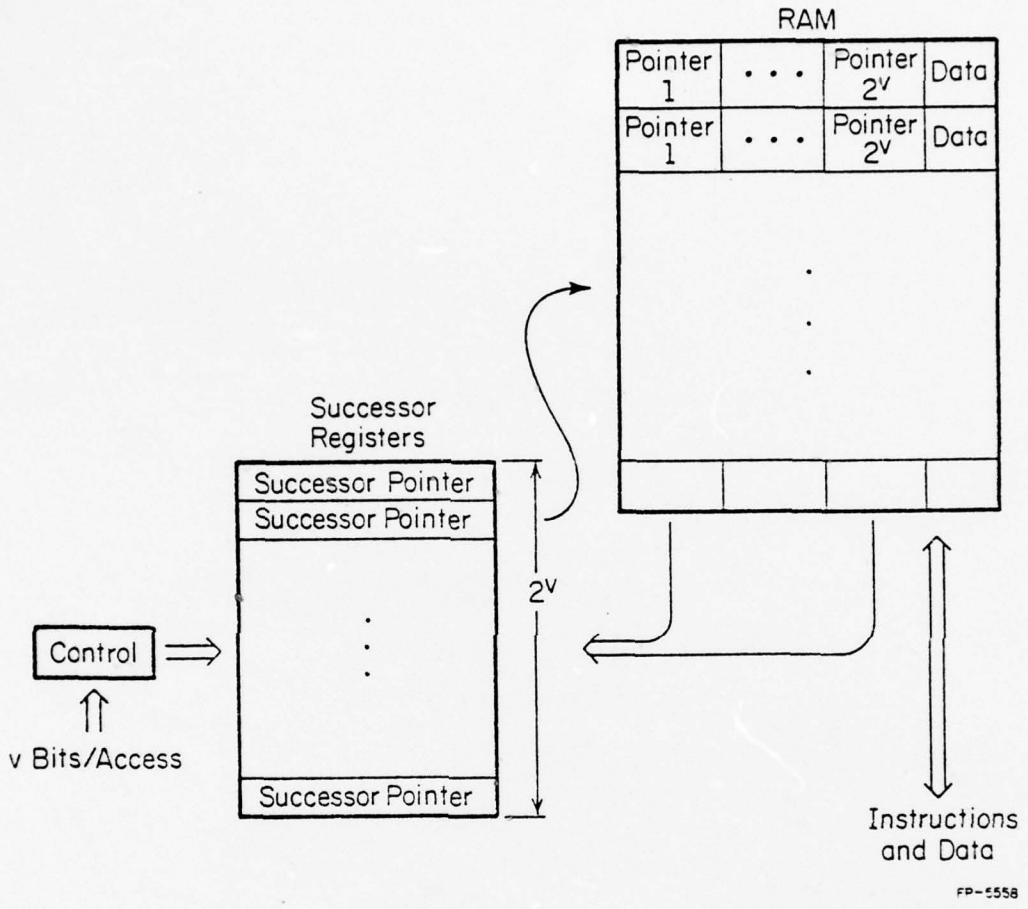


Figure 14. Successor Accessed Memory (SAM).

The major disadvantage with the SAM approach occurs when the number of successors exceeds 2^v. A reasonably fast solution to this problem and simulation of this memory with our program traces is presented in Section 4.5.

4.4. Analysis and Simulation of the Segmented Random Access Memory (SRAM)

As mentioned previously an SRAM consists of a random access memory with an in-memory address register. This register has k segments of a bits each and contains the address of the location being accessed in the memory. Prior to each access one or more of these segments may be changed. For each such interaction $v = \lceil \log_2 k \rceil + a$ bits need be sent. For SRAMs then

$$\tilde{B} = \tilde{\lambda}(\lceil \log_2 k \rceil + a).$$

Consequently if we have programs that exhibit sequential behavior and if we can allocate the memory well, we should expect a fairly small $\tilde{\lambda}$ for a rather small a. Due to the allocation dependent properties of this memory it becomes more difficult to analyze. However, by assuming certain basic accessing patterns the following simple results can be obtained:

Theorem 4.4.1: For pure serial access of an SRAM, i.e., if location i is accessed, $p(i+1|i) = 1$, then

$$\lambda_S = \sum_{i=1}^{k} 2^{(i-k)a}.$$

Proof: If we assume a k segment register with a bits for each segment and we are accessing successive locations n,n+1,n+2,..., then out of every

2^{ka} integers we change the high order segment 2^a times, the second highest 2^{2a} times and the lowest 2^{ka} times which can be written as

$$\lambda_S = \frac{2^{ka} + 2^{(k-1)a} + \ldots + 2^{2a} + 2^a}{2^{ka}}$$

$$= 1 + 2^{-a} + 2^{-2a} + \ldots + 2^{-(k-2)a} + 2^{-(k-1)a}$$

$$= \sum_{i=1}^{k} 2^{(i-k)a}. \qquad \square$$

Lemma 4.4.1: Let N(i) be the number of storage locations that are i interactions (segment loads) away from any current location, then

$$N(0) = 1$$

$$N(1) = k(2^a-1)$$

and $$N(i) = \binom{k}{i}(2^a-1)^i \quad , \quad 1 < i \leq k.$$

Proof: In i interactions, i segments can be changed, each segment allowing 2^a-1 new locations. Therefore there are $(2^a-1)^i$ possible addresses. However we can change any i of the k segments or $\binom{k}{i}$ combinations, resulting in $\binom{k}{i}(2^a-1)^i$ total possibilities. When i = 1 the one possible address which requires no change does require at least one interaction, so

$$N(0) = 1$$

$$N(1) = k(2^a-1), \quad i = 1. \qquad \square$$

Theorem 4.4.2: For random and independent access, i.e. $p(x) = \frac{1}{2^{ka}}$ and $\forall x,y\ p(x|y) = p(x)$, then

$$\lambda_R = \frac{1}{2^{ka}}\left[1 + \sum_{i=1}^{k} i\binom{k}{i}(2^a-1)^i\right].$$

Proof: At each reference there are N(i) locations i accesses away. The probability of any member of that set occurring is

$$N(i)p(x).$$

Therefore the expected number of loads = $\sum_{i=1}^{k} i\,p(x)N(i)$, or by using Lemma 4.4.1 and the fact that $p(x) = \frac{1}{2^{ka}}$, we have

$$\lambda_R = \frac{1}{2^{ka}}\left[1 + \sum_{i=1}^{k} i\binom{k}{i}(2^a-1)^i\right]. \qquad \square$$

Table 11 gives the results of the application of Theorems 4.4.1 and 4.4.2 to various memory configurations with ka = 16. These results are intuitively consistent. Serial accessing does well and, in fact, has a lower B with more smaller segments. Table 11 seems to indicate that sequential accessing of SRAMs generates nonzero overhead ($\lambda_s > 1$). However, by loading in a zig-zag manner, i.e., changing between incrementing and decrementing the lower order segments, each time a higher order segment is changed (Gray code), $\lambda_S = 1$.

Looking at the data for random accessing though, it can be seen that for $k' > k$, $B_{(k')} > B_{(k)}$. This result is again as expected, since there is no structure at all in the reference stream. An actual reference stream fortunately contains considerable structure, so we would expect an improved B when using this type of memory architecture, which, as we shall see from the simulation results, is indeed true.

It is sufficient to examine only second order information (1st order Markov process) when simulating the performance of an M_2, since the cost of a transition from state i to state j is dependent on

Table 11

SRAM Analysis Results

	k = 2, a = 8	k = 4, a = 4	k = 8, a = 2	
B_S, λ_S	9.035, 1.004	6.400, 1.067	6.667, 1.333	(Serial)
B_R, λ_R	17.925, 1.992	22.500, 3.750	30.000, 6.000	(Random)

being in state i and not on how one gets to i. Consequently we did not use higher order graphs in our simulation studies. Higher order graphs could, however, provide more information for improved performance of allocation algorithms. However, due to their complexity such algorithms were not considered further.

Before actually simulating SRAM operation, the problem of memory allocation must be investigated. It is important that such a memory have its contents ordered so as to minimize λ by taking advantage of the second order effects of the reference stream.

The SRAM allocation problem is similar to, but more general than, the program restructuring problem as examined by Ferrari [23], since we are trying to improve "spatial locality" by the proper allocation of memory words.

Before formalizing the allocation problem, the H_2 graph is introduced.

Definition 4.4.1: The H_2 graph is a directed graph consisting of a set of nodes V and a set of directed arcs E. Each node, $v(i) \in V$, is a word (data or instruction) referenced by a program. Each node is labeled with the stationary probability of its occurrence, q_i, where $\sum_i q_i = 1$. Each arc, $e_{ij} \in E$, represents a directed path from node i to node j, indicating that at least once within the reference trace node j immediately follows node i. All arcs are labeled with the probability of transition, p_{ij}, from node i to node j. Only these arcs are included with $p_{ij} > 0$, and $\sum_j p_{ij} = 1$. □

Let Γ be an "allocation," i.e.,a mapping of program words (nodes) v(i) into the linear memory space where $|V| \leq 2^{ka}$, and let $d(\Gamma,i,j)$ be a distance function which is the number of interactions in an SRAM required under mapping Γ to access location j when one has just accessed location i.

Theorem 4.4.3:

$$\lambda_\Gamma = \sum_{i,j} d(\Gamma,i,j)p_{ij}q_i.$$

Proof: Given that we are in state i (having just accessed node i), the expected number of memory interactions (faults) encountered when leaving i is just

$$\lambda_\Gamma(i) = \sum_j d(\Gamma,i,j)p_{ij}.$$

The expected number of interactions at any instance is just

$$\lambda_\Gamma = \sum_i q_i\lambda_\Gamma(i) = \sum_{i,j} d(\Gamma,i,j)p_{ij}q_i. \qquad \square$$

The optimal allocation problem then is to find an allocation Γ^* such that

$$\forall\Gamma \quad \lambda_{\Gamma^*} \leq \lambda_\Gamma.$$

This, however, is a difficult problem, since there are $(2^{H_0(S)})$ possible mappings. One can compute a lower bound, λ_{LB}, by assuming each location can allocate its successors optimally. In other words, each node i places its N(1) most probable successors such that it can access them in one interaction, its next N(2) most probable successors 2 interactions away, etc. Unfortunately it was discovered that this lower bound is

much too loose to be useful, since for all of our traces, $\lambda_{LB} \approx 1$ and for any mapping Γ, $\lambda_{\Gamma} \geq 1$.

Before describing the actual simulation program, each of the four allocations simulated will be discussed.

Original (ORIG): This is the original allocation of the program and data, using the addresses provided by the trace. To create a more realistic model, this allocation was compressed to create a contiguous array of words, removing the gaps created by filtering out some referenced instructions and data. Also the lowest numbered word of the allocation was assigned location 0, the second, location 1, etc.

H1: This is the H_1 allocation as defined in Chapter 3. This allocation was chosen because it is simple and quick, and though requiring approximate frequency measurements, the H_1 allocation does not need knowledge of the set of successors and their probabilities. This allocation technique should work quite well if there is only one tight grouping of frequently occurring nodes. In that case, absolute frequency would also tend to predict second order behavior. Conversely, if one had several groupings of words that were equally likely, but had very little intercommunication, the H_1 allocation would not work as well. One might then use a separate H_1 allocation for each group (locality), although shared instructions and data between groups would then pose a problem. We chose the single H_1 allocation since we feel that most of our traces represented a small number of such groupings.

Modified H1 (MH1): The modified H_1 allocation starts with an H_1 allocation and using the H_2 graph attempts by use of an iteration algorithm to

improve the allocation iteratively by swapping individual locations. The main objective is to see if multiple groupings of words exist, and if significant improvement can be wrung from the original H_1 allocation.

The iteration algorithm operates on the modified H_2 graph.

Definition 4.4.2: Define a matrix C such that each entry $c_{ij} = d(\Gamma,i,j)q_i p_{ij}$ for each directed arc in the H_2 graph. The modified H_2 graph is an undirected graph such that the arcs are those entries in matrix C' where

$$C' = C + C^T. \qquad \square$$

Theorem 4.4.4: Using the matrix C' defined above

$$\lambda = \frac{1}{2} \sum_{i,j} c'_{ij}$$

where $c'_{ij} \in C'$.

Proof: From Theorem 4.4.3 we have

$$\lambda = \sum_{i,j} d(\Gamma,i,j)q_i p_{ij}.$$

The probability of arc (i,j) occurring, is just $q_i p_{ij}$, and also $d(\Gamma,i,j) = d(\Gamma,j,i)$. Removing the directedness of the arc by matrix addition, all arcs are counted twice, also self loops are doubled. Therefore

$$\lambda' = \frac{1}{2} \sum_{i,j} d(\Gamma,i,j)(q_i p_{ij} + q_j p_{ji}) = \frac{1}{2} \sum_{i,j} c'_{ij} = \lambda. \qquad \square$$

Beginning with an H_1 allocation the iteration algorithm modifies Γ successively by looking at node pairs, i_1 and i_2, and swapping their allocation if the following is true

$$\sum_j c'_{i_1 j} + \sum_j c'_{i_2 j} > \sum_j \frac{d(\Gamma, i_2, j)}{d(\Gamma, i_1, j)} c'_{i_1 j} + \sum_j \frac{d(\Gamma, i_1, j)}{d(\Gamma, i_2, j)} c'_{i_2 j}.$$

In actual operation not all pairs are examined, just the most frequently accessed, since generally a small group of words were accessed more frequently than the rest. It should be mentioned that by using the modified H_2 graph and considering the fact that most words have a small number of successors, the iteration algorithm works very quickly.

H_2 allocation (H2): To achieve an allocation with performance close to λ_{Γ^*}, one must use as much second order information as possible. The H_2 allocation algorithm used here is a heuristic which attempts to group most probable successors closely together. A heuristic was chosen over an optimal algorithm, because of the difficulty involved in constructing such an optimal algorithm. The heuristic allocates fairly well and does so with a reasonable complexity, $\mathcal{O}(n_e \cdot n_v \cdot \log n_v)$, where $n_v = |V|$ and $n_e = |E|$. Of course it does need the H_2 graph with its associated probabilities.

Optimal allocation is essentially a k level clustering problem (all clusters at a level being of equal size). Most algorithms presented for clustering graphs are strictly for $k = 2$ (Ferrari [24]). Optimal H_2 allocation is a difficult problem for large values of k, and would probably not be a cost effective procedure in a general purpose computing environment. Hence we have chosen a reasonably quick heuristic instead.

Assume that each node in the H_2 graph is labeled with its steady-state probability, q_i, and that each arc is labeled with the arc

probability, $w_{ij} = q_i p_{ij}$. Further assume that arcs and nodes are indexed so that $w_{ij} \geq w_{i(j+1)}$, $\forall j$ and $q_i \geq q_{i+1}$, $\forall i$.

H2 algorithm: (To make this discussion simpler, it is assumed here that the length of the original list of nodes to be allocated, n_v, is an integer power of two. Of course the lgorithm as implemented actually operates on any integer n_v.) The algorithm basically takes the original list of nodes and splits it in two. It then splits each half recursively until all lists are of length one. When a split is made, the first half of the list has a "0" appended to an address string that exists for each node (word). The second half of the list has a "1" appended to node's string. When all lists are of length one, each address string will contain a binary number that is that word's address. Essentially we are allocating for an SRAM with $k = \lceil \log_2 n_v \rceil$ and $a = 1$ under the assumption that such an allocation would also be valid for any other value of k and a (such that $ka \geq H_0(S)$). The lists are actually split as follows:

Step 1: Take the first available node of the original list, the most probable, remove it from the original list, and place it on the split list. Copies of its successor pointers are then placed in an ordered list of successors, most probable first.

Step 2: Take the first element on the successor list, the most probable arc, and attach the node pointed to to the split list (removing that node from the original list) and merge copies of its successor pointers onto the successor list, such that the successor list remains an ordered list

(most probable first) and that only those pointers are included that point to nodes remaining in the original list. If no successors are left on the successor list, start a new node as in Step 1 and go to Step 2.

Step 3: If the length of the split list is less than the length of the updated original list then go to Step 2 else terminate the split process. □

The above algorithm does have two weaknesses. First, it does not combine arcs which emanate from the split list and point to a common node. This would have required extensive searching of the successor list for each node split. Therefore it was not implemented. The second problem concerns the manner in which the split list is grown. The technique used by the algorithm will certainly capture a single tightly bound grouping of nodes. It does not, though, appear to handle other lesser groupings adequately. This again was done for the sake of lower complexity. A better algorithm would probably try to grow from multiple seeds simultaneously.

To perform the allocations mentioned above, a procedure was added to the H012 program, since H012 uses an internal representation of the H_2 graph. The various allocations are placed in a file, to be read by the SRAM simulator. This file contains the actual address or name of each word, as referenced by the trace, and a location in the SRAM given to it by each of the four allocation routines. These allocations are read into an associative table within the simulator, using the original address as key. The simulator then reads the computation reference trace, keeps a

register for each allocation, indicating the last location referenced, and calculates $\tilde{\lambda}$ as the average number of interactions required to access the next reference. The simulator was run for each set of k and a.

In addition to computing $\tilde{\lambda}$ for each allocation, the simulator calculated the $\tilde{\lambda}$ required by the SRAM if 1, 2, or 4 separate address registers existed in the memory (for each allocation). With more registers in the memory, the CPU can have registers pointing to various competing word groupings. For all configurations then

$$\tilde{B} = \tilde{\lambda}[r + a + \lceil \log_a k \rceil]$$

bits are sent each time, where r = 0,1,2 (register select bits). For multiple register SRAMs the simulator always updates the register that requires the least number of interactions to change it to the new address. In case of a tie, the choice is just the first minimum change register encountered.

Table 12 shows SRAM simulation results for GAUSS and LIST. The bits/interaction column is just $v = \lceil \log_2 k \rceil + a + r$. For the H_1 allocation, the trace was split into an instruction and a data reference stream. These streams were also run through the simulator, giving performance equivalent to that of placing instructions and data in separate memories. The H_1 iteration algorithm was only run for a SRAM with one register.

The following conclusions can be derived from the simulation results:

i) For most configurations, $\tilde{B}$ was generally close to the lower bound, $v = r + a + \lceil \log_2 k \rceil$, and for all configurations $\tilde{B}$ was much lower than the original value calculated for the 360 architecture.

Table 12

SRAM Simulation Results

GAUSS:	# of Registers	Bits/ Interaction	MH1	H2	ORIG	H1	H1 Inst.	H1 Data
k=2, a=6;	1	7	8.7	10.3	11.9	10.5	7.2	10.0
	2	8	-	10.8	10.4	9.9	8.2	10.2
	4	9	-	10.6	10.1	9.8	9.1	10.2
k=4; a=3;	1	5	9.2	10.6	14.1	10.5	6.1	9.6
	2	6	-	10.8	10.3	9.8	6.9	10.0
	4	7	-	10.5	9.9	9.6	8.2	10.1

LIST:	# of Registers	Bits/ Interaction	MH1	H2	ORIG	H1	H1 Inst.	H1 Data
k=2; a=7;	1	8	9.6	9.5	12.9	11.5	8.0	10.0
	2	9	-	10.2	11.9	11.5	8.9	9.8
	4	10	-	10.8	12.0	11.6	9.6	10.0
k=4; a=4;	1	6	10.8	9.5	13.1	11.9	7.4	10.5
	2	7	-	10.3	11.6	11.7	8.5	10.5
	4	8	-	10.9	11.5	11.9	9.2	10.5

B (bits/computation reference)

	$\tilde{B}$ for Original Trace	$H_0(S)$	$\tilde{H}_1(S)$	$\tilde{H}_2(S)$
GAUSS	47.3	11.27	9.58	.845
LIST	39.9	10.36	7.88	1.86

ii) As a heuristic, the H_2 allocation did much better than the H_1 or the original (ORIG) for the LIST trace. For GAUSS, however, H2 was only better when one register was used with k = 2.

iii) The best heuristic for GAUSS was the modified H1 (MH1) which uses the iterative improvement algorithm. This result though was obtained by allowing the algorithm to run for a considerable length of time. For 2 or 4 registers the H_1 iteration algorithm would have taken excessively long to run. The results for MH1 demonstrate that the H_2 allocation is far from optimal for GAUSS. However, since the H_2 results are still fairly close to $v = a + r + \lceil \log_2 k \rceil$, the MH1 heuristic appears to be much too expensive for general purpose use. (For LIST, however, the H_2 allocation did perform better than MH1.)

iv) For the split H_1 allocation, the instruction stream, when using the largest k, always gave the lowest $\tilde{B}$. Also the addition of multiple registers tended to decrease performance further. For the data reference, however, the best performance varied unpredictably, but seems relatively constant. Splitting the reference stream and using two separate memories for instructions and data should give the best performance, especially if one were allowed to "tune" each SRAM to the characteristics of its stream.

v) For the combined instruction and data stream, the memory generally did slightly better with r = 1,2. This is intuitively consistent, since with more registers, pointers to commonly occurring

tightly bounded groups of words can be efficiently kept, thereby lowering the fault rate.

vi) It was found with the most highly segmented memory, a = 1, that the fault rate, $\tilde{\lambda}$, and $\tilde{B}$ increased drastically. Therefore this case was not considered further.

In conclusion, the SRAM is difficult to allocate optimally, though with simple heuristics like our H allocation algorithm, $\tilde{\lambda}$ is close to one. The main problem with the SRAM, however, is that v itself is fairly large. Since v alone is greater than $\tilde{H}_2(S)$, it would be senseless to try to incorporate more second order information into the allocation as an optimal allocation would do. Basically then, the mapping mechanism itself requires too much overhead to allow the SRAM to approach $\tilde{H}_2(S)$.

Before concluding this section a brief word is necessary concerning indexing by a CPU into a SRAM. The problem occurs when accessing a complete array requires several segment changes. Consequently, when an index instruction is executed, the number of interactions with memory will depend on the value in an index register. This problem, however, can be solved fairly easily by adapting the CPU so that, depending on the value in the index register, it takes the appropriate action to move the SRAM from the current state to the desired state independently of the instruction stream. This same approach can be used for multi-level indirection.

In conclusion, SRAMs yield a marked improvement over an M_1 with v = w. However, SRAMs do not approach the ideal performance for an M_2. Actually their required bandwidth is fairly close to that of a Huffman encoded M_1 with v = 1. This capability should not be understated since

it yields a significant improvement over the original value of $\tilde{B}$. Consequently, we feel this memory could be applicable to a general purpose computing environment. It increases addressing efficiency (e.g. no base registers need be maintained) and it is relatively easy to use.

In the next section we examine a memory which is almost an ideal M_2 and therefore has a much smaller addressing overhead.

4.5. Analysis of the Successor Accessed Memory (SAM)

The SAM is a random access memory with a specialized mapping mechanism. Essentially, the CPU addressing mechanism has been completely moved into the memory. In addition, we give this mechanism program specific knowledge of second order reference stream behavior. This knowledge has the distinct advantage of not only allowing reduction of the address computation requirements of the CPU, but also of the communication requirements between the CPU and memory. The mechanism discussed here then is just a particular kind of mapping mechanism which takes advantage of second order efficiency discussed earlier.

A very important difference between this memory and the SRAM, is that the SAM does not have a global allocation problem. However, there is a coding problem associated with each reference's successors. If we have v address bits per transaction, we choose from 2^v possible successors for each reference. Since most references have at most one or two successors, we would not want v to be too large. The problem is, what do we do when

the number of successors for a particular reference is greater than 2^v? In this case, some indirection method must be used by the mapping mechanism. This can be accomplished by having a bit in each data word, which when set indicates that the word contains no data and is only a dummy word providing another level of indirection to enable a greater number of successors to be accessed (see Figure 15). This technique does require the CPU to be aware of the path it must take to access the desired successor. Possible solutions to this and the indexing problem will be discussed in more detail at the end of this section.

Since various successors to a particular reference may have different probabilities of occurrence, the best way to allocate indirect pointers to minimize the expected number of levels, is to use Huffman encoding. In fact from coding theory we have the following theorem:

Theorem 4.5.1: A Successor Accessed Memory can be allocated (using indirect multi-level addressing) such that

$$H_2(S) \leq B_2 \leq H_2(S) + v$$

where 2^v is the number of immediate successors available for each referenced word, and v is the number of address pins used by the memory.

Proof: From Gallager [25] we have

$$(1)\ \frac{H(S)}{\log_2 D} \leq \bar{n} \leq \frac{H(S)}{\log_2 D} + 1$$

where $\bar{n}$ is a coding rate and D is the alphabet size being used. Now let

$$B_2 = \sum_j p(m_j) \cdot B(m_j)$$

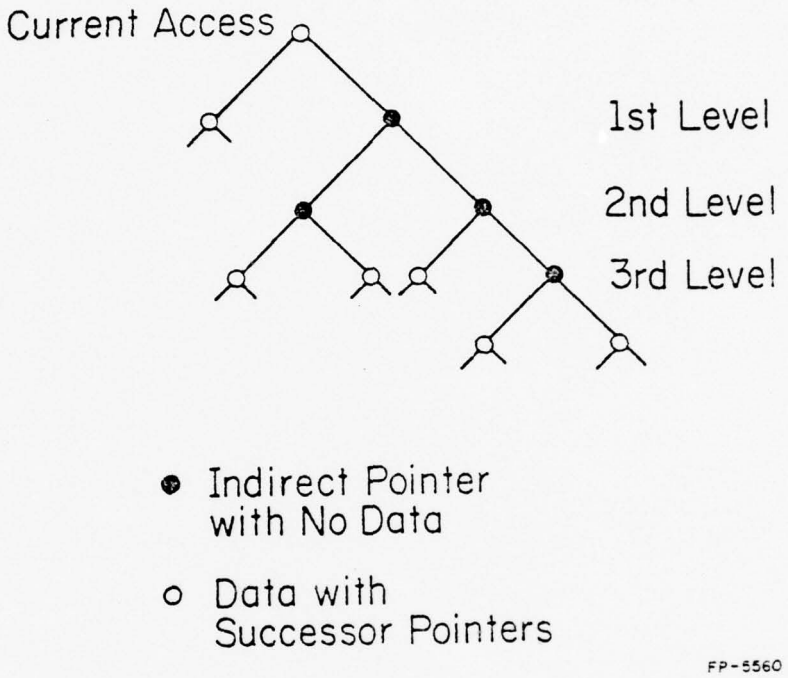


Figure 15. SAM allocation tree (v = 1).

where $$B(m_j) = \sum_i p(m_i|m_j)\cdot n_{i|j}$$

$n_{i|j}$ being the length of the code word (in bits) needed to code the reference to location m_i, given that m_j has just been accessed, and

$$H_2(S) = \sum_j p(m_j)\cdot H(S|m_j)$$

where $$H(S|m_j) = \sum_i p(m_i|m_j)\log_2 p(m_i|m_j).$$

Our actual code alphabet size is $D = 2^v$, i.e., we can choose from 2^v successors with each reference, so by applying (1)

$$\frac{H(S|m_j)}{v} \leq \bar{n}(m_j) \leq \frac{H(S|m_j)}{v} + 1$$

where $\bar{n}(m_j) = \sum_i p(m_i|m_j)\cdot n^*_{i|j}$, $n^*_{i|j}$ being the length of the code (in words of v bits each) required to code m_i given that m_j has just been referenced. Since $B(m_j) = v\cdot\bar{n}(m_j)$ we have (multiplying by v)

$$H(S|m_j) \leq B(m_j) \leq H(S|m_j) + v.$$

Multiplying all inequalities by $p(m_j)$ and summing over j we have

$$\sum_j p(m_j)H(S|m_j) \leq \sum_j p(m_j)B(m_j) \leq \left(\sum_j p(m_j)H(S|m_j)\right) + v\cdot\sum_j p(m_j)$$

or

$$H_2(S) \leq B_2 \leq H_2(S) + v. \qquad \square$$

The above theorem tells us two things: first, that we can bound B_2 from below and above, and second, that the smaller v we use the better (though with smaller v the SAM would become more program structure sensitive.) This scheme, of course, assumes complete knowledge of the conditional probabilities $p(m_i|m_j)$. What we would like is a heuristic

algorithm which only has knowledge of successors and possibly some partial knowledge of probability distributions. Such a heuristic does, in fact, exist and still gives smallest $\tilde{B}$ with $v = 1$.

The heuristic we used is as follows. If all successors are equally likely, which is often true for instructions fetching data from an indexed array, then the optimal allocation is a balanced tree (see Figure 16a). Now if one location dominates all the rest (each next probability is greater than the sum of all those remaining), then a sequential allocation is optimal (see Figure 16b). Our heuristic uses only sequential or balanced tree allocation. The only decisions it must make are when to use sequential and when balanced tree allocation, and if using sequential allocation, in what order should the successors be allocated. For our analysis the estimated probabilities $\tilde{p}(m_i|m_j)$ are known, the decisions being made were therefore based on those $\tilde{p}(m_i|m_j)$. Furthermore, it is not unreasonable to expect a compiler to be able to estimate whether sequential or balanced tree allocation should be used for a particular reference based on the type of statements being compiled at that instant. For example, array references would almost always be allocated as a balanced tree, instruction references sequentially. It should be noted that our results are slightly better than would be expected in an actual environment as a consequence of the above assumption of knowledge of the $\tilde{p}(m_i|m_j)$. Of course if a program were to be run many times, post factum optimization could be used very effectively.

Our analysis is of the H_2 graph (as defined in Section 4.4) rather than an actual simulation using the program traces. As with the

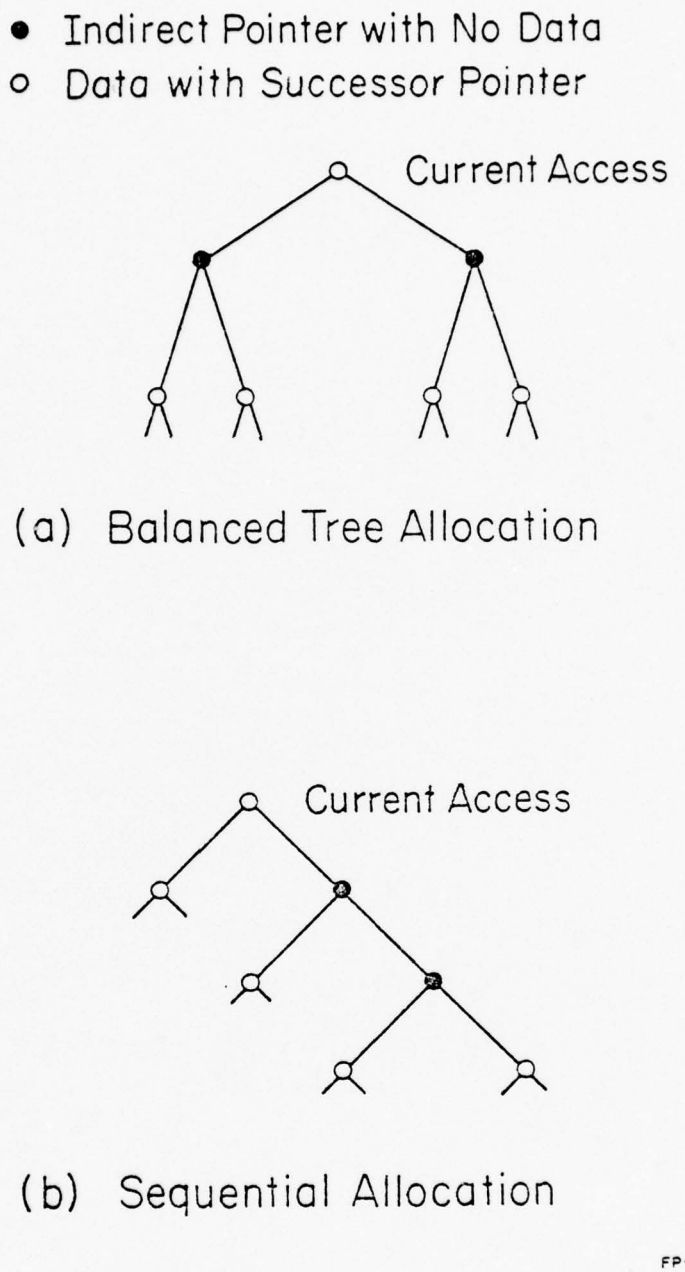


Figure 16. SAM heuristic allocation (v = 1).

SRAM, it is not necessary to have greater than second order knowledge of the process to simulate an M_2. With our analysis we find $\tilde{\lambda}$ (λ estimate) as defined in Definition 4.2.2 and then $\tilde{B}$ where

$$\tilde{B} = v \cdot \tilde{\lambda}.$$

For balanced tree allocation, the expected number of levels (accesses) required to fetch reference i's successor is

$$\lambda_{TREE}(i) = \log_v S(i) \tag{1}$$

where S(i) is the number of successors to location i. The expected number of levels required to access i's successor with sequential allocation is

$$\lambda_{SEQ}(i) = \sum_{\ell=1}^{\left\lceil \frac{S(i)}{(2^v-1)} \right\rceil} \left(\ell \cdot \sum_{j=(\ell-1)(2^v-1)}^{\ell(2^v-1)} p_{ij} \right) \tag{2}$$

where the p_{ij} are indexed such that $p_{ij} \geq p_{i(j+1)}$ for all j. Equation (2) is derived from the fact that (2^v-1) data elements are allocated at each level with $\left\lceil \frac{S(i)}{(2^v-1)} \right\rceil$ levels required.

The expected number of levels averaged over the stationary probability, q_i, of each reference is

$$\lambda_{TREE} = \sum_i q_i \cdot \lambda_{TREE}(i) \tag{3}$$

$$\lambda_{SEQ} = \sum_i q_i \cdot \lambda_{SEQ}(i) \tag{4}$$

$$\lambda_{BOTH} = \sum_i q_i \cdot \min[\lambda_{TREE}(i), \lambda_{SEQ}(i)]. \tag{5}$$

The $\lambda_{SEQ}(i)$ calculation is inaccurate in one minor respect, if $S(i) = k(2^v-1)+1$ for some k, formula (2) will allocate one level greater than is necessary, since it will place the last element on a new level

with a pointer from the previous level pointing to it. This approach over estimates $\lambda_{SEQ}(i)$ slightly, but does make the computations easier.

The above calculations were actually done by a procedure added to the H012 program, since H012 has an internal representation of the H_2 graph.

The results of these calculations for GAUSS and LIST (the worst and best programs) are given in Table 13 for $1 \leq v \leq 3$. These results are quite interesting. Over all cases λ_{BOTH} is much less than either λ_{TREE} or λ_{SEQ}, this indicates that generally there is no predominance of one type of distribution over another. Thus the use of a selection mechanism is justified. Furthermore, even with this heuristic allocation technique, the optimal $\tilde{B}$ is with $v = 1$, and for all traces $\tilde{B} \leq \tilde{H}_2(S) + 1$ with $v = 1$.

Of course λ_{BOTH} decreases monotonically with increasing v. This means that although addressing efficiency is greater with the smallest v (with $v = 1$,A would be only a few bits/computation reference), speed would decrease accordingly. In a distributed multi-processor environment, however, this may not be a serious problem, since we have considerably reduced the communication requirements between the CPU and memory modules. If it were a problem, higher values of v should be considered.

As with the SRAM, the reference stream was split into an instruction reference stream and a data reference stream. Each substream was tested with its own memory. The instruction stream did very well with faulting occurring rarely even with $v = 1$. This was expected because of the highly sequential nature of the instruction reference stream. For the data stream, the results were not nearly as good. This was also expected,

Table 13

SAM Analysis Results

		v = 1	v = 2	v = 3
LIST	B_{BOTH}	1.40	2.23	3.13
Combined Stream	B_{SEQ}	1.75	2.40	3.19
	B_{TREE}	1.75	2.55	3.13
Instruction Stream -	B_{BOTH}	1.04	2.01	3.00
Data Stream -	B_{BOTH}	2.00	2.57	3.33
GAUSS	B_{BOTH}	2.75	3.50	4.13
Combined Stream	B_{SEQ}	19.39	14.04	10.52
	B_{TREE}	3.23	4.00	4.63
Instruction Stream -	B_{BOTH}	1.01	2.00	3.00
Data Stream -	B_{BOTH}	3.88	4.13	4.33

(bits/computation reference)

	$\tilde{B}$ for Original Trace	$H_0(S)$	$\tilde{H}_1(S)$	$\tilde{H}_2(S)$
LIST	47.3	11.27	9.58	.845
GAUSS	39.9	10.30	7.88	1.86

since data is rarely fetched sequentially and shows much less structure than the instruction stream. By using two separate memories we would achieve some speed-up; i.e., $\lambda_{AVERAGE}$ for all references would be much smaller. However, we then would need another bit to determine which memory we are accessing, thereby increasing the bits sent per interaction by one. The net result would be a slightly faster configuration than a single memory, with a slightly greater addressing bandwidth than a single memory.

Thus the main objective of this chapter, to find a second order memory structure with $\tilde{B}$ close to $\tilde{H}_2(S)$ is realized with the SAM. The SAM should therefore be considered seriously as a system component, and a careful study of its other attributes in a system context is justified.

One important question that should be addressed concerns the time and space trade-offs between a CPU with this type of memory architecture and the standard 360 architecture with random access memory. Table 14 gives the space (in bits) and time (in references) requirements for each implementation for the LIST and GAUSS programs. The time requirements were calculated directly from the trace statistics collected by our analysis programs. The bit requirements for the unfiltered programs are also computed from program statistics. The bit requirements for the filtered programs running on a SAM are the number of bits required to store the filtered program plus the bits required to store all successor pointers. The number of pointers was calculated while H012 calculated $\tilde{\lambda}$ and $\tilde{B}$ for the SAM.

Table 14

SAM Space-Time Trade-offs

		Original Program	Required with SAM Architecture
LIST	Time in References	97,244	≈ 92,160
	Space in Bits	98,016	≈ 161,000
GAUSS	Time in References	94,272	≈ 207,750
	Space in Bits	41,216	≈ 149,300

It should be noted that the successor pointers could be coded much more efficiently to achieve considerable space savings over those requirements given in Table 14. For example, by using default values and relative displacement instead of absolute pointer values, the total bit requirements could be reduced significantly.

The results (for LIST and GAUSS) in Table 14 are quite interesting. For LIST, there is roughly a 65% increase in space requirements, but no increase in execution time. In fact, execution time is shortened slightly. For GAUSS, however, there is a tremendous increase in space requirements (roughly three times) and execution time (roughly doubled). This is the worst case for all our traces, and appears to stem from the increased overhead resulting from indexing into large arrays. It is our feeling that if some simple indexing capabilities existed in the SAM, much of the above overhead would disappear, perhaps approaching the requirements for LIST.

The indexing problem and the related area of indirect addressing are indeed nontrivial problems with the SAM. Solutions do, however, exist. The indexing problem can best be described with the following example; if two distinct portions of the instruction stream access the same array, then each referencing instruction must not only work its way down the balanced tree to the array element that is needed, but must also, upon leaving the array, make a choice that leads it back to the proper place in its respective instruction stream or "ramp." This problem is further complicated by the fact that the particular array element accessed depends on the value in the index register. One solution would be for the program

to give data to the CPU which tells the CPU the approximate size of the array. From this data and the contents of the index register, it would not be difficult for the CPU to automatically generate the interactions required to fetch the desired array element. Array size data could also be stored in the dummy (empty data) fields of the indirect pointers. This approach could also apply, with some modification, to indirect addressing. However, computed addresses, so it seems, would have to be excluded from systems using SAMs. These ideas are not meant to be taken as specific solutions to the SAM indexing problems. Rather they merely demonstrate that the indexing problem need not be a major obstacle in the use of this type of memory.

It is quite possible that the form that a second order memory should take would use the ramp as a basic unit. The CPU would receive ramps from the memory as the primary form of instruction communication. The currently executing ramp within memory would then automatically set up its own addressing structure <u>including</u> indexing. The ramp's individual index rate could be added automatically when a ramp is entered allowing immediate array access without CPU intervention. Since each ramp would have its own index storage, accessing arrays common to many ramps would be trivial. Also since the ramp structure would carry knowledge of previous executions of a ramp, enough information should be available to lower $\tilde{B}$ below $\tilde{H}_2(S)$, possibly even $\tilde{H}(S)$! Lowering $\tilde{B}$ below $\tilde{H}(S)$ would be possible, since the memory suggested above would actually keep more information on data referencing behavior than we allowed in the $\tilde{H}(S)$ model of Section 2.4. Also, a ramp structured memory would require considerably

less pointer storage, since only pointers to successor ramps would be necessary. Of course, other addressing information would be required for data accessing within the ramp, but the tremendous pointer storage requirements of the SAM would simply not exist.

In fact, with the ramp memory, many accesses may be effected with only one interaction. This fact may make it attractive to share the memory among several processes. Such sharing would allow the memory unit to be used in a multiprocessing environment.

In concluding this section, it must be noted that not only are SAM-like memories attractive for general purpose environments, but they could also be useful in special purpose environments. Already the Fairchild F8 microprocessor has a limited form of on-chip addressing incorporated into the memory chips [26]. This consists of an auto-incrementing register which is generally used as a program counter.

Essentially, with the SAM we have traded off storage space for decreased bandwidth, i.e. we can increase the functional capabilities of the chip and reduce its address pin requirements to one pin. For the LSI system designer, this trade-off would appear to be quite inviting.

4.6. Summary and Conclusions

In Chapter 3 we showed that extremely efficient addressing cannot be achieved unless at least second order reference stream information is used.

In this chapter we have examined higher order memory architectures which attempt to decrease addressing overhead by taking advantage of this second order structure. Several examples of second order memories were presented, two of which, the Successor Accessed Memory (SAM) and the Segmented Random Access Memory (SRAM) were analyzed and their operation evaluated by simulation.

The main conclusion we can derive from the results is that more efficient addressing can indeed be achieved by using a memory system which is designed to exploit second order reference stream behavior. The SRAM, which is easily implementable and uses second order information to a limited extent, achieves a significant improvement over the current state of the art. Furthermore, the SAM, which uses more reference stream information performs many times better than the SRAM, but requires us to extract the required accessing information during compile time and store it within the memory.

The main result of this chapter is that through increased memory complexity, addressing bandwidth and hence addressing overhead can be reduced significantly (more than an order of magnitude for our traces). It is our belief that this type of memory does have a place in the distributed intelligence, LSI computer systems of the future. In a sense, the major portion of the addressing function can now be placed within the memory where it belongs and the oft casually referred to "intelligent memory" may become a reality.

CHAPTER 5

CONCLUSION

5.1. Summary and Conclusions

In Chapter 2, the addressing and computation processes within an executing CPU were defined. From this followed the definitions of the addressing overhead, A, and the information content of the computation process memory reference stream, H(S). It was shown that $A \geq H(S)$, making A vs. H(S) a good measure of addressing process efficiency. Techniques were then developed for obtaining $\tilde{A}$ (estimated A) and $\tilde{H}(S)$ (estimated H(S)) from actual program execution on an IBM 360. A system of programs was presented which calculated $\tilde{A}$, $\tilde{H}(S)$, as well as the low order entropies; $H_0(S)$, $\tilde{H}_1(S)$, and $\tilde{H}_2(S)$. The results of these calculations were then examined. They showed that $\tilde{A}$ was considerably larger (usually an order of magnitude larger) than $\tilde{H}(S)$ for all traces. $\tilde{A}$ was, in fact, even larger than $H_0(S)$ and $\tilde{H}_1(S)$, demonstrating considerable coding inefficiency. Perhaps the most interesting result concerned the second order entropy $\tilde{H}_2(S)$ which was very close to $\tilde{H}(S)$ for most programs and much smaller than $\tilde{H}_1(S)$ for all programs. This indicates that second order knowledge of a program's behavior fairly completely identifies the program's structure and is therefore quite useful. In concluding Chapter 2 it was demonstrated that for one typical program, addressing overhead accounted for some 58.5% of the total bit stream entering the CPU.

Chapter 3 discussed some methods for reducing $\tilde{A}$. A model of the address register load process for an R+D architecture was developed which

demonstrated that addressing could be done more efficiently (for our traces) if the number of address registers and/or the displacement were reduced in the IBM 360 architecture. It was shown that R+D addressing is reasonably efficient, but sensitive to proper address register allocation. The problem of such allocation was examined briefly and was demonstrated to be a difficult problem to do well. In concluding the chapter the possible improvements to GAUSS's addressing overhead when using the techniques presented in the chapter were analyzed. It was discovered that for GAUSS, $\tilde{A}$ could be reduced by approximately 59% (to about $H_0(S)$) by a variety of techniques. It was further concluded that without major architectural changes this improvement, while significant, is about the best that one could hope for.

Chapter 4 then looked at some more radical architectural changes. In particular the concept of second order memory was introduced. This memory architecture takes advantage of the second order behavior discussed in Chapter 2, to reduce addressing overhead. Two examples were analyzed in detail. One was a moderate approach, the Segmented Random Access Memory (SRAM), which is not too difficult to use and approaches $\tilde{H}_1(S)$ in performance. The second example, the Successor Accessed Memory (SAM), is a true second order memory which approaches $\tilde{H}_2(S)$ in performance. This memory is unfortunately more difficult to use, but does represent a considerable further gain in addressing efficiency.

The main contributions made by this research include:

i) Addressing architecture can be more closely matched to program behavior reducing much unnecessary overhead.

ii) Memories can be built which, by using knowledge of program behavior, require only a small fraction of the CPU/Memory bandwidth currently being used.

iii) More concise models of a memory reference stream are available, which incorporate knowledge of higher order reference stream behavior.

iv) Program size, through increasing address efficiency, can be reduced. This reduction could be even more significant when coupled with Hehner's work [2] on opcode and data size optimization.

In conclusion, to obtain a very large improvement in CPU/Memory bandwidth and CPU addressing efficiency, higher order memories and accompanying radical changes in CPU architecture and system software are necessary. The potential payoff looms ever larger as computer memory and addresses continue to grow in size, LSI chips become larger, and interconnection costs increase relative to logic costs.

5.2. Topics for Further Research

Though some have been briefly discussed previously, this section presents some areas for further research.

i) Application of the information theoretic approach to the computation process. Such research could examine the efficiency of the communication link between the computation process and memory. Probabilistic characterization of the computation process would be required. This approach then would eventually lead to a reduction of the H_2 graph in both size of program and execution time.

ii) Improved allocation algorithms for program compilation. This would include not only heuristics to solve the problem as discussed in Chapter 3, but also work in the area of adaptive restructuring of compiled programs for improved addressing.

iii) Application of the techniques presented in this dissertation to the analysis of multi-level memory hierarchies and paging environments.

iv) Analysis of the performance improvements possible in special purpose computing environments, where programs are compiled once and executed many times. This would also include applications to micro-programmed control structures within general purpose machines.

v) Improvement of second order memories by investigating, among other things, improved pointer coding and accessing, and "ramp" accessed memories in an attempt to reduce the large amount of information needed by the Successor Accessed Memory (SAM) and decrease program execution time. Adaptive or predictive techniques may exist for certain classes of program behavior. For example, indexing mechanisms could be included in the Successor Accessed Memory to avoid the large access trees required for arrays. This is a worthy goal since array accessing contributed the most significant term in our calculations of the overhead function of the SAM.

vi) Further refinement of the estimation of $\tilde{H}(S)$ to capture more information in the model system itself. This could include checking the invariance of $\tilde{H}(S)$ for the same algorithm and data on different architectures, variation of program behavior across different input data,and indexing information as derived in v) above.

vii) Research on the performance potential of an architecture with multiple address and multiple computation processors operating in a distributed "load-sharing" environment.

viii) Adaptation of instruction sets and compilers to take best advantage of a computer system that uses "intelligent" memory.

ix) Extending this research to evaluate architectures with a separate memory addressing processor rather than including this processor with the memory (as in Chapter 4) or with the computation processor (as in Chapter 3 and conventionally).

REFERENCES

1. L. L. Constantine, "Formalization of Computer Power," Modern Data, November 1968, pp. 58-67.

2. E. C. R. Hehner, "Information Content of Program and Operation Encoding," J. ACM, Vol. 24, No. 2, April 1977, pp. 290-297.

3. C. C. Foster and R. Gonter, "Conditional Interpretation of Operation Codes," IEEE Trans. on Computers, Vol. C-20, No. 1, January 1971, pp. 108-111.

4. D. W. Clark, "LIST Structure: Measurement, Algorithms, and Encodings," Ph.D. Dissertation, Computer Science Dept., Carnegie-Mellon University, Pittsburg, Pa., August 1976.

5. C. E. Shannon, "Prediction and Entropy of Printed English," Bell Systems Technical Journal, Vol. 30, No. 1, January 1951, pp. 50-64.

6. R. G. Gallager, Information Theory and Reliable Communication, John Wiley and Sons, New York, 1968, p. 45.

7. R. Ash, Information Theory - Interscience Tracts in Pure and Applied Mathematics, No. 19, John Wiley and Sons, New York, 1965, Lemma 14.1, p. 16.

8. Ibid., Theorem 2.3.1, p. 33 and Theorem 2.4.1, p. 35.

9. Ibid., Theorem 2.5.1, p. 37.

10. G. P. Basharin, "On a Statistical Estimate for the Entropy of a Sequence of Independent Random Variables," Theory of Probability Applications, Vol. 4, No. 3, pp. 333-336.

11. P. G. Hoel, S. C. Port, and C. J. Stone, Introduction to Stochastic Processes, Houghton Mifflin Co., New York, 1972, p. 73.

12. R. G. Gallager, Information Theory and Reliable Communication, John Wiley and Sons, New York, 1968, Theorem 3.6.1, p. 68.

13. W. T. Wilner, "Burroughs B1700 Memory Utilization," FJCC, 1972, pp. 579-586.

14. L. P. Horowitz, et al., "Index Register Allocation," J. ACM, Vol. 13, No. 1, January 1966, pp. 43-61.

15. B. Wichman, ALGOL 60, Compilation and Assessment, Academic Press, London, 1973, pp. 36-37.

16. D Gries, Compiler Construction for Digital Computers, John Wiley and Sons, New York, 1971, pp. 178-179.

17. T. A. Welch, "An Investigation of Descriptor Oriented Architecture," The 3rd Annual Symposium on Computer Architecture, 1976, pp. 141-146.

18. R. N. Ibbett, "The MU5 Instruction Pipeline," The Computer Journal, Vol. 15, No. 1, p. 9.

19. B. Wichman, ALGOL 60, Compilation and Assessment, Academic Press, London, 1973, p. 98.

20. M. J. Flynn, "The Interpretive Interface: Resources and Program Representation in Computer Organization," Proc. Symposium on High Speed Computation and Algorithm Organization, University of Illinois, April 1977.

21. H. A. Sholl, "Direct Transition Memory and its Application to Computer Design," IEEE Trans. on Computers, Vol. C-23, No. 10, October 1974, pp. 1048-1061.

22. N. K. Ouchi, "Orthogonal Storage Ring Organization for Computer Memory," Stanford Electronics Laboratories Technical Report No. 47, November 1972.

23. D. Ferrari, "The Improvement of Program Behavior," Computer, Vol. 9, No. 11, November 1976, pp. 39-47.

24. D. Ferrari, "Improving Locality by Critical Working Sets," Comm. ACM, Vol. 17, No. 11, November 1974, pp. 614-620.

25. R. G. Gallager, Information Theory and Reliable Communication, John Wiley and Sons, New York, 1968, Theorem 3.3.1, p. 50.

26. L. Sullivan, "A Microprocessor Needn't Take Many IC's," Electronic Design, Vol. 24, No. 12, June 7, 1976, pp. 126-132.

VITA

Daniel Wayne Hammerstrom was born in Bozeman, Montana on October 10, 1948. He graduated with distinction from Montana State University in 1971. At Montana State University he was a member of Tau Beta Pi and Phi Kappa Phi. In 1972 he obtained an MSEE from Stanford University. From 1972 to 1974 he served as an officer in the United States Air Force in the capacity of Computer Systems Design Engineer at the Electronic Systems Division, L. G. Hanscom AFB, Bedford, Massachusetts. He was, in 1975, a General Electric Fellow at the University of Illinois, and from 1975 to 1977 was employed as a research assistant at the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign.