October 1982 Report CSG-10

COORDINATED SCIENCE LABORATORY
College of Engineering

A STRUCTURED MEMORY ACCESS ARCHITECTURE

Andrew Richard Pleszkun

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

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) A STRUCTURED MEMORY ACCESS ARCHITECTURE		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER CSG-10
7. AUTHOR(s) Andrew Richard Pleszkun		8. CONTRACT OR GRANT NUMBER(s) N00039-80-C-0556
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 Naval Electronics Systems Command VHSIC Program		12. REPORT DATE 1982 13. NUMBER OF PAGES 116
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) 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)

Computation process
Access process
Structured memory access
Degree of overlap
Address trace analysis

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

When conventional von Neumann architectures reference the memory, addressing information must first be obtained, usually by transfer from the memory to the CPU. The work performed by the CPU can be partitioned into a computation process and an access process. Outside of adding addressing modes to instructions, little has been done to reduce the work performed by the access process or to reduce the demands placed on the memory for access-related activities. This work investigates one method of reducing the von Neumann

DD FORM 1 JAN 73 1473

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

bottleneck and improving the degree of overlap between the computation and access processes.

Program referencing behavior is first studied by analyzing program address traces. With the information gained from the address trace analysis, a Structured Memory Access (SMA) architecture is developed which makes fewer references to memory and permits the access process to be, by and large, decoupled from the computation process, thus providing a maximum degree of overlapped execution and access prediction.

To evaluate the effectiveness of the SMA architecture in reducing addressing overhead, a comparison is made between a hypothetical SMA machine and a VAX-like machine with respect to the number of memory references generated by a set of programs. Depending on the program, the SMA machine reduced the number of memory references to between 1/5 and 2/5 of those required by a conventional VAX.

An estimate is also made of an SMA machines performance relative to that of a VAX. A machine's performance is parameterized by the memory bandwidth and the computational overhead. It was found that performance is very sensitive to these parameters; however, an SMA machine performs significantly better than a conventional machine with the same parameters.

The SMA architecture reduces addressing overhead and provides improved system performance by (1) efficiently generating operand requests, (2) making fewer memory references, and (3) maximizing computation and address generation overlap.

A STRUCTURED MEMORY ACCESS ARCHITECTURE

BY

ANDREW RICHARD PLESZKUN

B.S., Illinois Institute of Technology, 1977
M.S., University of Illinois, 1979

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, 1982

Urbana, Illinois

A Structured Memory Access Architecture

Andrew Richard Pleszkun, Ph.D.
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1982

When conventional von Neumann architectures reference the memory, addressing information must first be obtained, usually by transfer from the memory to the CPU. The work performed by the CPU can be partitioned into a computation process and an access process. Outside of adding addressing modes to instructions, little has been done to reduce the work performed by the access process or to reduce the demands placed on the memory for access-related activities. This work investigates one method of reducing the von Neumann bottleneck and improving the degree of overlap between the computation and access processes.

Program referencing behavior is first studied by analyzing program address traces. With the information gained from the address trace analysis, a Structured Memory Access (SMA) architecture is developed which makes fewer references to memory and permits the access process to be, by and large, decoupled from the computation process, thus providing a maximum degree of overlapped execution and access prediction.

To evaluate the effectiveness of the SMA architecture in reducing addressing overhead, a comparison is made between a hypothetical SMA machine and a VAX-like machine with respect to the number of memory references generated by a set of programs. Depending on the program, the SMA machine reduced the number of memory references to between 1/5 and 2/5 of those required by a conventional VAX.

An estimate is also made of an SMA machines performance relative to that of a VAX. A machine's performance is parameterized by the memory bandwidth and the computational overhead. It was found that performance is very sensitive to these parameters; however, an SMA machine performs significantly better than a conventional machine with the same parameters.

The SMA architecture reduces addressing overhead and provides improved system performance by (1) efficiently generating operand requests, (2) making fewer memory references, and (3) maximizing computation and address generation overlap.

ACKNOWLEDGMENT

The author wishes to express his gratitude and appreciation to his thesis advisor Professor Edward S. Davidson. Professor Davidson's patient guidance and helpful suggestions were invaluable contributions to this work. His insight, encouragement, and concern were indispensable sources of support throughout the author's period of study.

The author would also like to thank his colleagues in the Computer Systems Group at the Coordinated Science Laboratory and Professors B. R. Rau, J. A. Abraham, J. H. Patel, and M. S. Schlansker for their friendship and the stimulating intellectual atmosphere which they provided.

Finally, the author wishes to thank his parents and family for their continual support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1. The von Neumann Bottleneck

In 1946, Burks, Goldstine, and von Neumann authored a paper [Burk46] which established the basic design of general purpose computers. To this day, many general purpose computers found on the market can be classified as von Neumann machines since their organization is basically the same as that proposed in the 1946 paper. These von Neumann machines have common characteristics which affect the referencing of instructions and data.

Since we are interested in the interactions between the central processing unit (CPU) and the memory, we may divide von Neumann machines into a CPU and a memory, ignoring the issue of input-output. The memory is treated as one uniform structure containing both the instructions and the data of a program. In order to reference the memory, addressing information must first be obtained, usually by transfer from the memory to the CPU. Once this information arrives, the CPU performs some operations to generate an operand or instruction address. The number of operations or calculations which the CPU performs to generate addresses depends on the program which is being executed and the basic operations the CPU has in its repertoire.

The interactions between the CPU and the memory can be modeled with respect to address generation, as shown in Figure 1-1 [Hamm77].

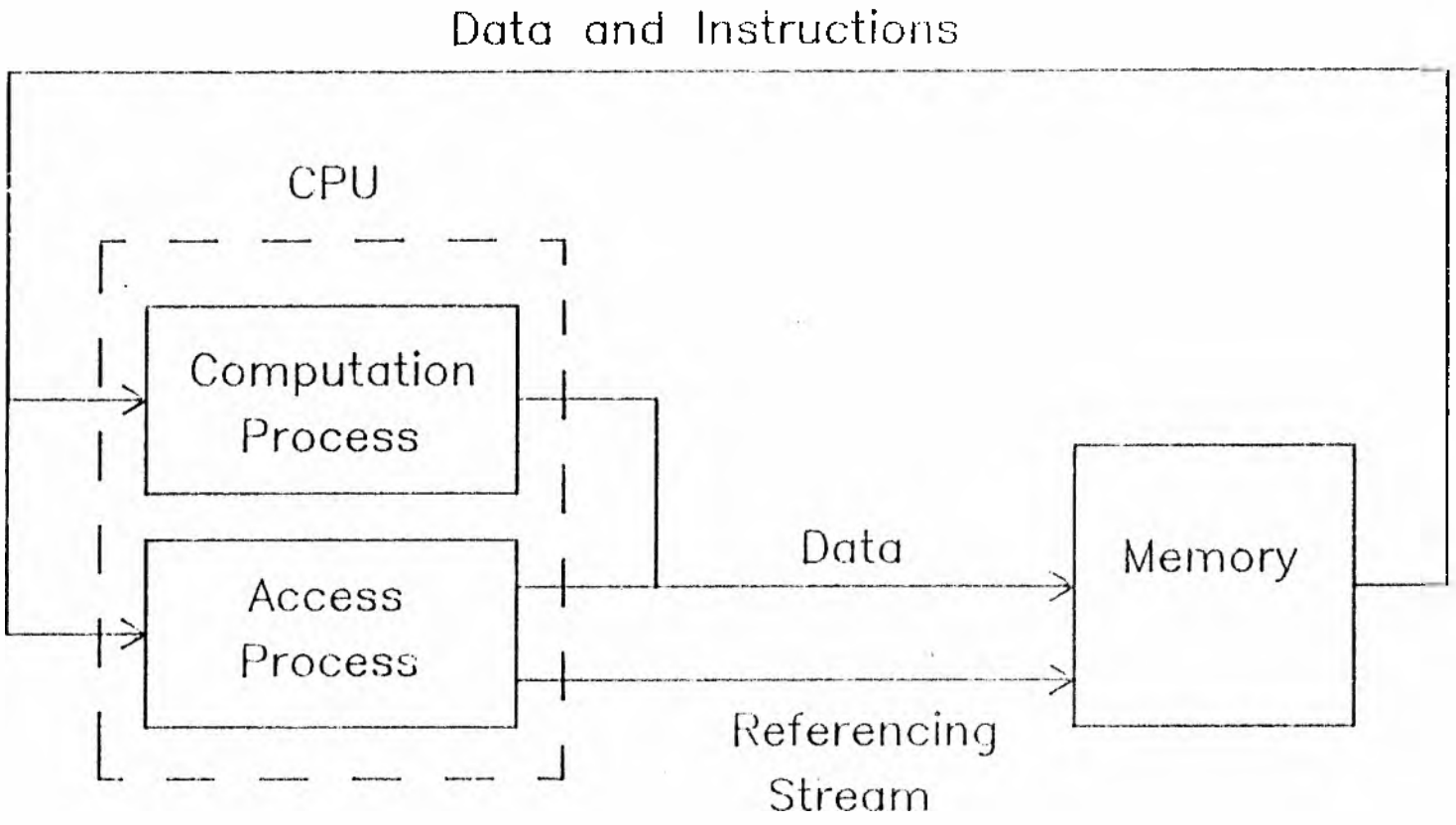


Figure 1-1. CPU-Memory Model.

The work performed by the CPU is partitioned into an access process and a computation process. The access process generates a stream of read and write requests to be serviced by the memory. The memory services write requests by taking data from either the computation or the access process and placing the data in a memory location specified by the access process. The memory responds to read requests by generating a stream of data and instructions which return to the CPU. Some portion of these data and instructions are returned to the accessing process to generate more references, while the remaining portion is received by the computation process. In our view, the computation process performs the useful work of the system, while the work being done by the access process is overhead which should be reduced.

Conventional von Neumann architectures are organized so that the CPU expects to interact with only 1 memory, making a memory request over 1 narrow bus and receiving only 1 word per memory access. Effectively, access to the memory is limited. Furthermore, the data associated with a program is treated as a set of independent items. With such a model of the memory and the data stored in it, the computation and access processes are forced to compete for access to the memory. Access to the memory, therefore, becomes a critical resource and a potential bottleneck of the entire system. This bottleneck is called the von Neumann bottleneck because it results from a von Neumann machine's view of the memory.

The potential for a bottleneck to occur can be reduced if the activities of the access process can be modified to reduce the number of

times the memory is accessed. The overhead due to the access process may also be reduced by overlapping the activities of the access process with those of the computation process. By predicting and prefetching read accesses before the data is actually needed, burst bandwidth requirements are reduced, and memory wait time is reduced. To maximize this overlap, these two processes must be as independent as possible. In a von Neumann machine, while the computation process can be conceptually separated from the access process, in reality it is impossible to distinguish between access-related and computation-related instructions and data. This property of von Neumann machines imposes limitations on the degree of overlap which can be achieved between the computation process and the access process.

A great deal of work has been done to improve the speed with which the CPU can perform its operations and, therefore, the speed with which the computation and access processes perform their tasks. Much has also been done to improve the speed with which memory responds to requests. However, outside of adding addressing modes to instructions, little has been done to reduce the work performed by the access process or to reduce the demands placed on the memory for access-related activities.

This work presents an investigation of one method of reducing the von Neumann bottleneck and improving the degree of overlap between the computation and access processes. Program referencing behavior is first studied by analyzing program address traces. This analysis indicates the types of mechanisms which would aid in reducing addressing overhead. These mechanisms take explicit advantage of a program's structure and of

the regular patterns in which data structures are referenced. Based on these mechanisms, a Structured Memory Access (SMA) machine is proposed and evaluated. The SMA machine has an organization which is somewhat different from conventional von Neumann machines.

1.2. Conventional Answers to the von Neumann Bottleneck

As stated earlier, access to the memory is a very critical resource of a computer system. Most programs place high demands on this resource, so accesses to the memory significantly affect the performance of a system. Computer designers have improved system performance by (1) increasing the speed of the CPU, (2) increasing the speed with which memory responds to requests, and (3) decreasing the number of memory accesses made per program. Of these three approaches, the first two have received the most attention.

The speed with which the CPU performs its operations has been steadily increasing, in part due to the availability of faster hardware. Increases in speed are also due to organizational changes within the CPU such as pipelining [Rama77] [Ande67] and instruction prefetch [Smit75] which overlap the execution of instructions. As noted above, instruction prefetch will reduce the bottleneck somewhat. However, a faster CPU actually aggravates the memory bottleneck since the CPU can make memory requests at a higher rate; yet overall system performance will, nonetheless, improve somewhat.

Concurrent with increases in CPU speed, the speed with which the memory can respond to a request has also been increasing. As with the

CPU, part of this speed-up is due simply to faster hardware; however, speed increases are also achieved by augmenting the memory with a cache and by interleaving memory modules [Kapl73] and [Kuck78].

While concentrating on increasing the speed of the CPU and the memory to improve system performance, computer designers have done very little to decrease the number of memory accesses made by a program. The addition of new addressing modes, combined with the already existing feature of index registers, decreases the number of instructions required to generate a data address [PDP75], [VAX80], [Ston80], and [Amdl64]. The use of an instruction buffer within the CPU can also decrease the number of memory requests for programs with short, frequently executed loops of instructions [Ande67]. By holding the instructions of a loop in such a buffer, the memory is not burdened by retransmitting the loop instructions for each iteration of the loop. We have investigated further ways of reducing the number of memory accesses made per program.

The class of machines known as "super" computers combine the previously mentioned approaches to form machines which are well suited for performing array computations. Super computers achieve their high performance through the use of pipelines, special indexing techniques, and interleaved or skewed memory structures [Rama77] and [Russ78]. Also, in such machines, the cost of high-speed hardware and wide buses is not as critical an issue as in more conventional machine designs. Users of super computers will generally accept reduced performance/cost in order to obtain very high performance.

The indexing mechanisms provided by these machines are of great importance in efficiently using accesses to memory and in reducing the amount of work done by the access process. Indexing mechanisms vary substantially from one machine to the next. For performing matrix-oriented computations, one desires as much flexibility as possible in the way indexing may be used. Most of the super computers permit automatic stepping through vectors of a data structure with a single vector instruction. Generally, if one is accessing a matrix, only a row, column, or diagonal can efficiently be accessed with one instruction. The TI-ASC, however, does provide both inner-loop and outer-loop control for stepping through a matrix [Watst72]. While the indexing facilities combined with the memory structure permit the speedy access of operands, one is faced with rewriting an algorithm to make optimal use of a particular machine. The transformation of the algorithm can, to some extent, be automated by using compilers to vectorize high level language programs. Another approach taken here, however, is to make the computer organization sufficiently flexible so that the algorithm need not be transformed into vector instructions.

1.3. Background for the Structured Memory Access (SMA) Approach

Unlike super computers, conventional computers make very limited explicit use of program structure or data types in the generation of memory requests. Although some of the explicit address calculation has been removed from today's computers, there remains a great deal of computing performed solely for the generation of addresses. By modeling a computation as a computation process and an access process,

Hammerstrom [Hamm77] calculated the addressing overhead and the entropy of the stream of computation references. These statistics were found by analyzing the traces of several programs executed on an IBM 360.

In Hammerstrom's analysis, each program trace is processed in reverse order to permit the tagging of the instructions and data which were used solely for the purpose of address generation. Addressing overhead for a program trace is calculated by summing the number of bits contained in the address generation related instructions and data and dividing the resultant sum by the total number of computation-related memory references. Addressing overhead is measured in bits input to the access process per computation process reference. For a Gaussian elimination program and an eigenvalue-finding program, the addressing overhead was, respectively, 17.2 and 17.0 bits per computation reference. For a floating point benchmark and a symbol manipulation program, the addressing overhead was, respectively, 10.0 and 24.1 bits per computation reference. These results represent a large percentage of the total number of bits input to the CPU from the memory.

The inefficiency of the conventional access process is exposed when the addressing overhead is compared with the entropy of the stream of computation references. The entropy of the computation reference stream is likewise measured in bits per computation reference and is interpreted as the average number of bits needed to select among the possible successor references, i.e. to choose the particular next reference address given the current reference address. If the current and the possible successor reference addresses are known, Hammerstrom

found that for the programs he analyzed, between .845 and 1.86 bits per computation reference are needed to determine the successor reference address. These values can be treated as lower bounds on the number of bits which would be needed to specify a successor reference. Comparing these values to the addressing overhead, we find that they differ by at least an order of magnitude. Thus significantly more bits than necessary are being transferred between the memory and the CPU during the execution of a program.

Addressing overhead represents the number of bits flowing into the access process per computation reference. The access process generates addresses for its own data and instructions and for those of the computation process. These addresses are a type of overhead which can be measured as the number of bits per computation reference flowing out of the access process. Since this overhead is used to fetch information for the access process as well as the computation process, it could be reduced if the access process made fewer references and if memory references were more efficiently specified. In [Hamm77b], two types of second order memories, which reduce the number of bits flowing out of the access process, are proposed and analyzed in detail.

The first of these is the Segmented Random Access Memory (SRAM). A schematic of this memory organization is shown in Figure 1-2. Associated which each RAM of memory is an address register which is divided into k a-bit segments. Such a technique, with k=2, has been used by some memory manufacturers to save pins on large memory chips. For each access of this memory, instead of sending k*a address bits to

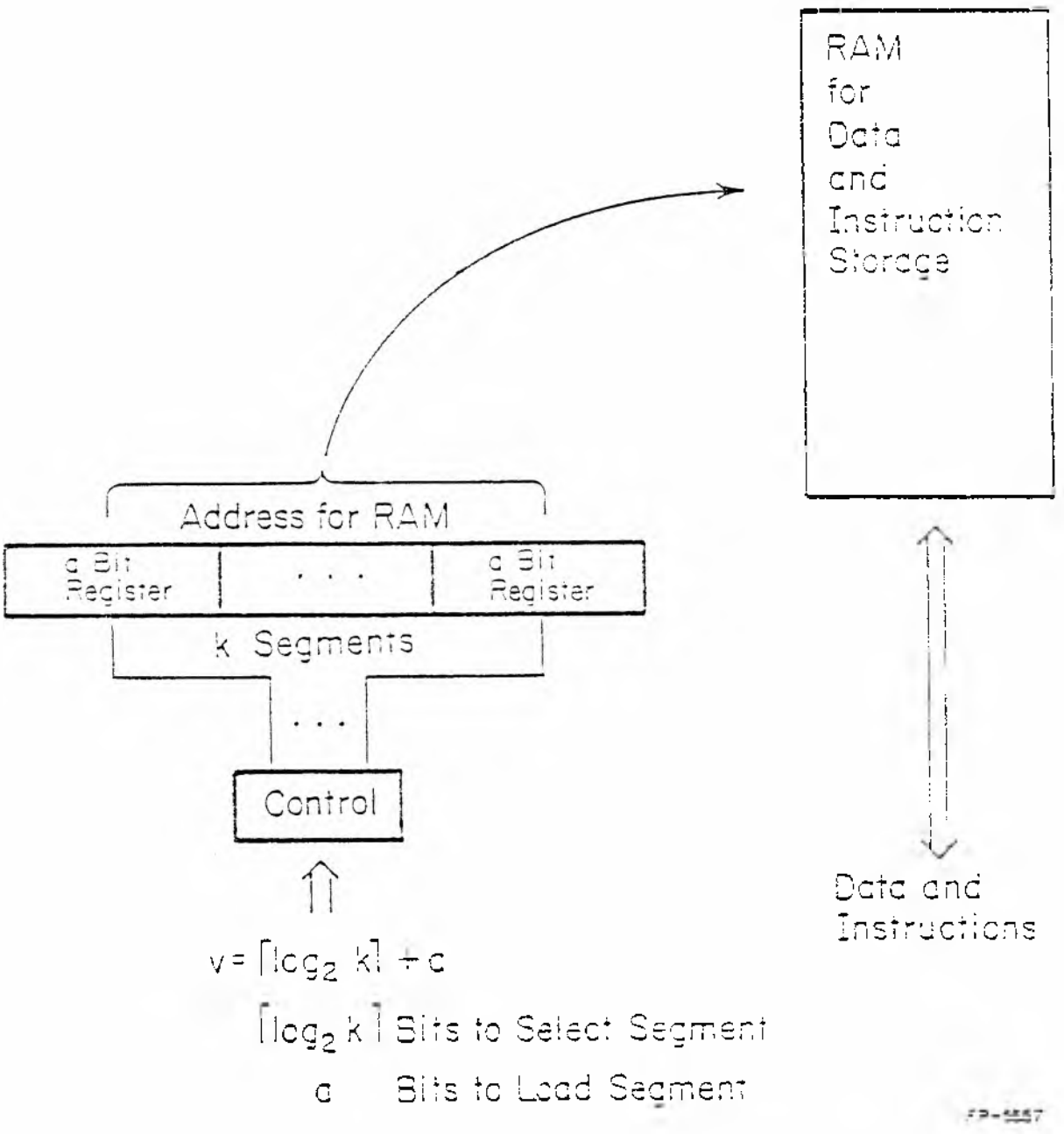


Figure 1-2. Segmented RAM.

the RAM, $v = \log_2 k + a$ bits are needed per transaction. Instead of accessing the RAM directly, these bits access one of the k segments of the address register, replacing the contents of the accessed segment with the value of the a bits. For an access in which the entire contents of the address register must be changed, k transactions are required. Analysis of the SRAM indicates that it increases addressing efficiency by approximately 25%, by reducing the number of bits which must be transferred to the processor per computation reference. The SRAM produces this reduction since it takes explicit advantage of program locality. The SRAM, however, has the disadvantage that it is difficult to allocate memory for a program so as to minimize the number of transactions per access.

The second memory proposed by Hammerstrom is the Successor Access Memory (SAM). As shown in Figure 1-3, this memory stores pointers to possible successors along with each data and instruction word. Whenever a word is accessed, these pointers are loaded into a set of 2^v successor registers. To access the next word, only v bits need to be sent to the memory. Evaluation of this type of memory indicates that the optimal value of v is 2 for typical computer programs. Thus only 1 bit per transaction is required. On the average between 1.4 and 4.13 bits (transactions) are needed per computation reference. Of course, these values depend on the program and the value of v. Although the SAM is attractive, it is difficult to use and requires complex mapping hardware within the memory. Additionally, when the number of successors is greater than 2^v, multiple transactions per access are needed and some type of indirection mechanism must be provided.

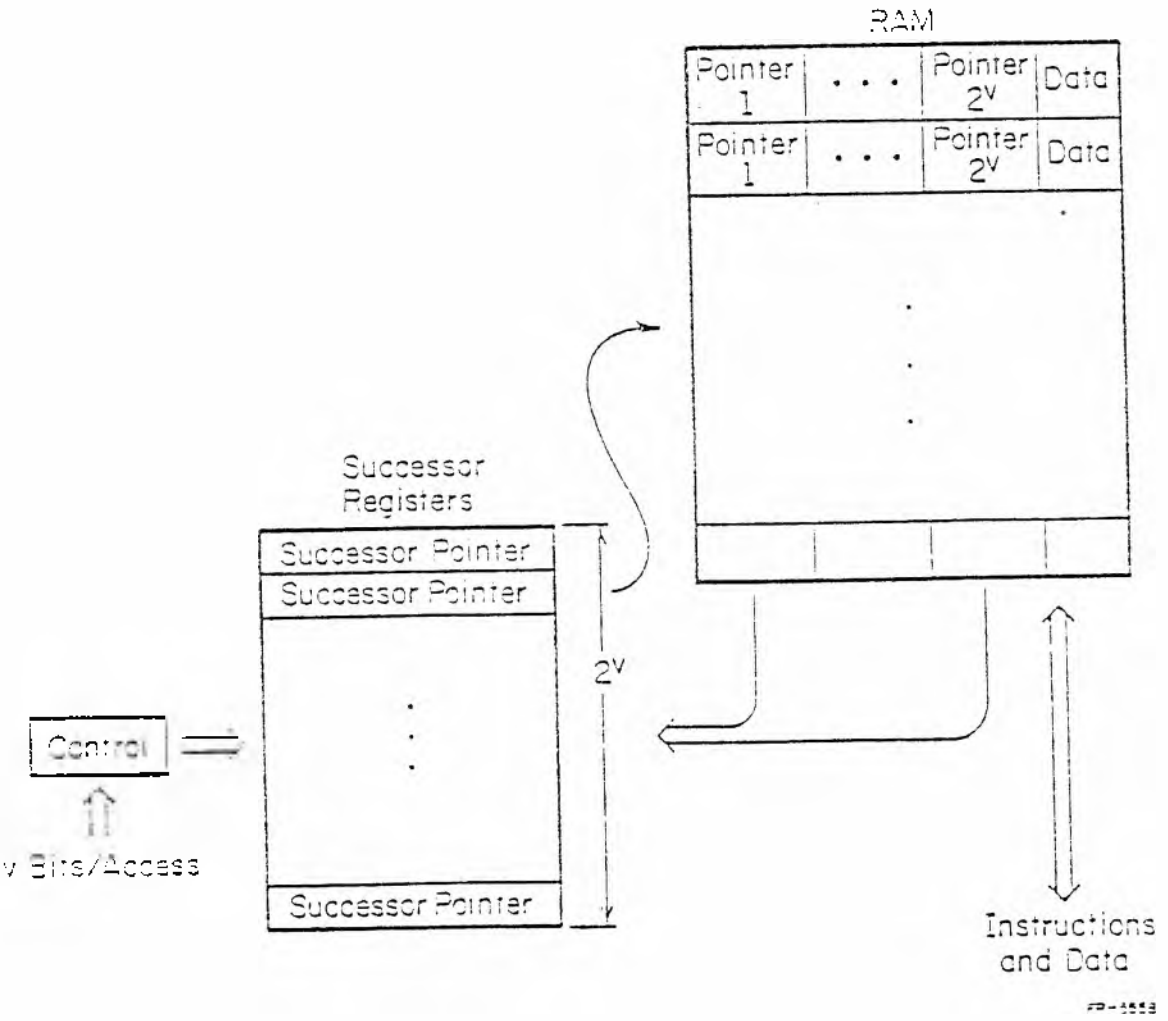


Figure 1-3. Successor Accessed Memory.

The Address Prediction Stack (APS) [Ples81] is a scheme which, when added to a conventional computer system, can reduce the required processor-memory address bandwidth. The analysis of an APS reveals some interesting results about the structure of a program. The APS is a least recently used (LRU) stack which has been extended into a second dimension. Thus, what normally is a single entry in an LRU stack, has been replaced by a line of entries. Each address points to a block of memory and each line of entries is a string of sequential block addresses. Depending on the implementation, a block may be a byte, a word, or a small page of memory. The APS is associatively searched for an address match whenever a memory request is made. If the address is found in the APS, then an identifier for that stack location which contains the address is sent to the memory instead of the entire address. An identical stack is required in the memory which also tracks the memory references. A full memory address may thus be generated within the memory from a given stack location identifier. If an address is not found in the processor's APS, a miss occurs and an entire address must be sent to the memory. Depending on the type of update policies used for the stack and the line parameters of the stack, an APS can reduce the average number of bits needed to specify a memory address to as few as 8 bits per memory reference.

Evaluation of the APS, based on trace driven simulations, shows a very high hit rate for the first two lines of the stack. For example, for a 5 deep APS, with a 5 block line and addresses which point to

blocks of 64 words, an overall hit rate of 98.7% was achieved with the first line of the stack representing 34% of the total references and the second line representing 49%. The high hit rates in the first two lines of the stack demonstrate the phenomenon of interleaved sequential streams. They also demonstrate that programs usually alternate loosely between two streams, the data stream and the instruction stream. Due to the LRU policy, stream alternation causes a high hit rate in the second line.

From the analysis of these three schemes for addressing memory, we find that the access process roughly alternates between instruction and data referencing. Also, data reference sequences are less regular than instruction reference sequences. Knowing, or at least accurately predicting, possible successor memory references is very important in achieving an efficient access process and can significantly reduce the addressing overhead of a program. Additionally, exploiting this predictability leads to a more nearly autonomous operation of the the access process and the computation process, thus permitting an overlapped execution of the two processes.

In the next chapter, the stream of instruction and data references for a set of programs is analyzed. From this analysis we discover the patterns of memory referencing which occur during a program execution. This program referencing behavior indicates the types of address generation mechanisms which should exist to improve the efficiency of

the access process. Chapter 3 presents a description of the Structured Memory Access (SMA) architecture which includes such mechanisms. Chapter 4 describes an SMA architecture implementation. The SMA architecture is evaluated in Chapter 5.

CHAPTER 2

PROGRAM TRACE ANALYSIS

Successful architectural techniques for reducing the von Neumann bottleneck of conventional computers capitalize on the highly structured nature of most computer programs. Caches work well if their update policies accurately predict future memory requests. Index registers work well when programs step through structures such as arrays. But while these methods work because of the structured nature of programs, they make very little explicit use of program structure. Although the SRAM and SAM do make more explicit use of a program's structure, these schemes have serious implementation problems and inefficiencies.

A more detailed look at the structure of memory references is provided by analyzing the structure of instruction and data references. The sequential patterns of instruction references are quite different from those of data references. Analyzing a combined stream of instruction and data references obscures the sequential nature of instruction execution and, at the same time, makes it difficult to find patterns in the accessing of data. Therefore, we found it more useful to separate the subtrace of instruction references and the subtrace of data references and apply distinct analysis techniques to these subtraces.

2.1. Instruction Analysis

In our analysis, the instructions of a program trace are divided into instruction blocks, based on Hammerstrom's definition of ramps and blocks. For our purposes, we do not use ramps, but prefer instead to use a slightly different definition of blocks. A block is a maximal-length ordered set of one or more sequentially stored and executed instructions, where all entry points to the block are only into the first instruction and all exit points from the block only leave the last instruction in the block. Thus a new block always begins with the target instruction of some conditional or unconditional branch instruction. Each block has an associated set of one or more successor blocks which may immediately follow that block in execution. In our trace analysis, while instruction references are formed into instruction blocks, the number of times each successor block is referenced and the order in which successor blocks are referenced are also tabulated. Thus, a control flow graph for the program can be made automatically from this trace analysis.

Four IBM 360 program address traces were analyzed in such a manner. Two of these programs, GAUSS and EIGEN, are floating point programs written in FORTRAN. GAUSS contains 94,273 memory references and performs a Gaussian elimination on a 20-by-20 matrix. EIGEN contains 77,563 memory references and finds the eigenvalues of a 14-by-14 matrix. Of the remaining two programs, CCOBOL is the compilation of a COBOL program containing 120,055 memory references and ECOBOL is the execution of a COBOL program containing 120,068 memory references.

Figure 2-1 shows the number of instruction blocks which have a particular length. The distributions for each of the programs individually were very similar. The figure has the combined results for all four programs. Most of the instruction blocks contain very few instructions. Figure 2-2 shows the total number of times all instruction blocks of a particular length are executed. From this figure, we see that relatively short blocks are executed most frequently. For both figures, the average instruction block length is 5 instructions, while the median instruction block length is 2 instructions.

Not only are block lengths and frequency of use important, but the flow of control, or the order in which blocks are executed, is also of interest. An efficient means of predicting successor blocks is needed since individual blocks contain so few instructions. Table 2-1 is a listing of the percentage of blocks which have one, two, and more than two successors. One-successor blocks are those blocks which are always followed by the same block and which are created when that block ends with an unconditional branch or when some block branches into a set of sequentially executed instructions. Most blocks which occur before a DO loop fall into this category. Blocks with two successors have a data dependent branch occurring as the last instruction of the block. Quite often the branch reflects the end of some nesting level of a DO loop. In such a block, the final operation increments the loop index and tests the index for completion of the loop. In our traces, the blocks which have more than two successors always end with the return from a subroutine which is called from more than two places in the program.

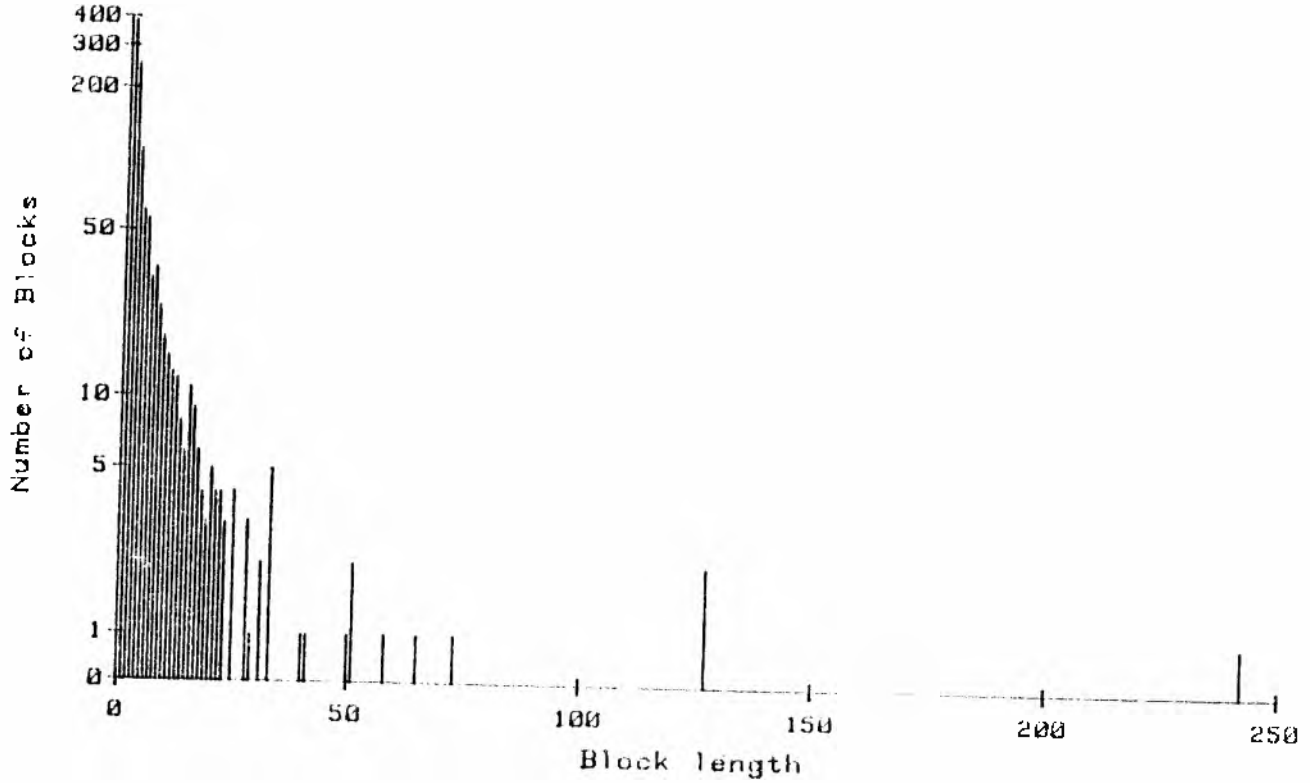


Figure 2-1. Number of blocks with a particular length.

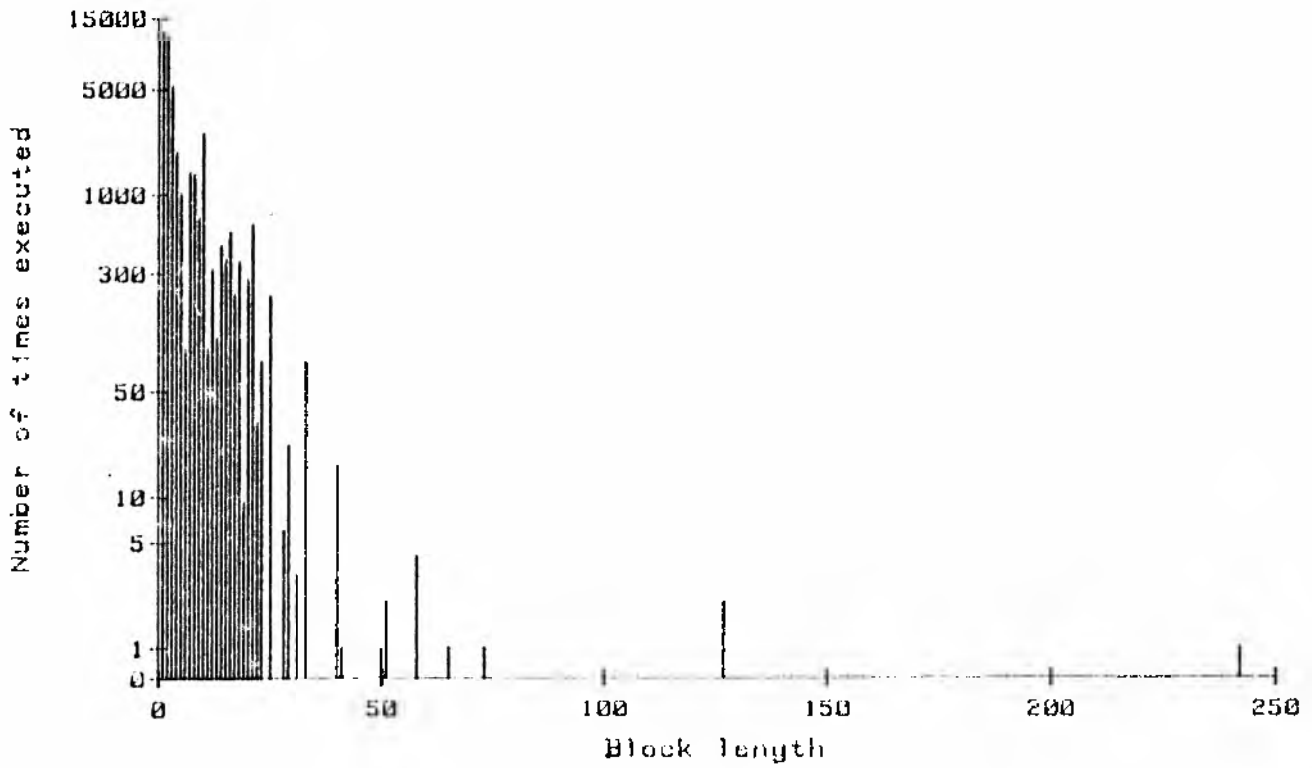


Figure 2-2. Number of times a block of a particular length is executed.

Table 2-1. Percentage of instruction blocks with 1, 2, or more successors.

Program	Number of Successor Blocks		
	1	2	>2
GAUSS			
static	52.9	47.1	0.0
dynamic	5.5	94.5	0.0
EIGEN			
static	55.1	43.5	1.4
dynamic	24.7	69.7	5.6
CCCBCL			
static	56.1	38.2	5.7
dynamic	37.2	49.2	13.6
ECOBCL			
static	60.5	35.0	4.5
dynamic	38.1	55.9	6.0

For each of the measured phenomena, we gathered static and dynamic statistics. A static count refers to the number of times a particular phenomenon occurs in a program listing. A dynamic count is the number of times a phenomenon occurs during an execution of the program. Thus, while a particular loop in a program occurs only once in a static count, the loop may be executed many times. The number of times the loop is executed is reflected in the dynamic count. As can be seen from Table 2-1, in a static as well as a dynamic count, very few of a program's instruction blocks have more that two successors. Among the one-successor and two-successor blocks, the one-successor blocks occur more frequently in a static count, while the two-successor blocks

(particularly loops in GAUSS and EIGEN) are more frequently executed. Subroutines are more common in the COBOL programs, while the GAUSS program has no subroutines at all.

The referencing of instructions is a relatively well-behaved process. Sequential execution of instructions is normally interrupted by a branch to one of two successors. In many cases, blocks branch back on themselves, or a few-block cycle is repeatedly executed, to form loops. Often only one successor block follows an exit from the loop. To perform well, a machine must be able to handle this type of branching efficiently. For the traces we analyzed, the number of subroutine calls was minimal.

2.2. Data Analysis

As with the referencing of instructions, we would like to see what order can be discerned from the more complicated process of referencing data. Program address traces are again used as a basis for finding sequence patterns of data references.

Initially, the trace of data references was analyzed in a manner similar to that of the instruction reference traces. This analysis was performed with the expectation of finding data references grouped into sets of repeatedly accessed sequences; with each sequence having a limited number of successor sequences. When the analysis was performed, we found that most data references formed groups of their own and that the number of possible, distinct successor data references for each data reference was large. This result occurred for two reasons. First, the

data used by the access process was mixed with the data being referenced by the computation process. Thus, if a row of a matrix is being referenced, the data references for index values appear interleaved with references to the matrix itself. The data reference to an index value, therefore, has a large set of successor data references, perhaps including all of the elements of the matrix. The second reason that a large number of successors occurs is that one data structure may be accessed in many ways. For example, in a single program, a matrix may be accessed column by column, row by row, across a diagonal, etc. An element in the matrix can thus have a large number of successor data references in the matrix itself.

When analyzing the data reference stream like an instruction stream, difficulties arise in developing a coherent model of data-referencing behavior because the data references are analyzed without considering the instructions which generate the data references. We therefore changed our approach to analyzing data-referencing patterns associated with the data referenced by a single instruction. If an instruction references more than one data item, the stream of references generated by each item is treated separately.

In this model, an instruction can reference either no operand from memory, scalars from memory, data structure elements from memory, or both scalars and data structure elements from memory. Scalars and data structures can actually be determined by the way in which instructions reference memory, rather than by given information such as in declaration statements. For our purposes, declaration statements can be

misleading and we prefer the following somewhat unusual definitions. If an instruction, or set of instructions, always references one particular memory location, the contents of that location is said to be a scalar. On the other hand, if an instruction references several locations among all executions of that instruction, the contents of the set of locations which it references is called a tentative data structure. The set of data structures are then formed by repetitively taking the set union of pairs of tentative data structures with common elements until no such pairs can be found.

Table 2-2 shows the percentage of instructions which either make no memory references, reference scalars, reference data structures, or

Table 2-2. Percentage of instructions which reference a type of data.

Program	no data reference to memory	scalars	data structures	scalars and data structures
GAUSS				
static	38.0	56.8	5.2	0.0
dynamic	48.1	29.1	22.8	0.0
EIGEN				
static	34.3	60.6	5.1	0.0
dynamic	47.9	37.9	14.2	0.0
CCOBOL				
static	38.8	42.7	12.5	6.0
dynamic	39.2	35.9	11.6	13.3
ECOBOL				
static	45.1	41.1	10.0	3.8
dynamic	55.2	29.3	9.4	6.1

reference both scalars and data structures. A single instruction which references more than one item is only counted once. For example, some instructions reference three scalars each time they are executed. In a static count such an instruction is counted only once. It is interesting to note that for programs such as GAUSS and EIGEN, which are matrix oriented, a high percentage of the instructions do not reference a data structure. Even for the dynamic count, the instructions reference scalars or make no memory reference. This result is somewhat surprising since one might expect most of the executed instructions to reference data structures. The results for CCOBOL and ECOBOL may also be somewhat surprising since one might not expect a heavy reliance on data structures in these two programs; yet, between 15% and 25% of the instruction executed made a data structure reference. These results are somewhat encouraging since scalar addresses should be predictable and data structure reference addresses may be predictable if effective structured access mechanisms can be found.

Separating the data references into separate lists for each instruction allows analysis of each ordered list for data reference patterns. An example of such a data address list is shown in Figure 2-3a. The instruction at location 1 accesses the memory twice each time it is executed. One operand's memory location is always eight memory locations from the previous one, while the other operand is always obtained from the same memory location. From the definitions of data structures and scalars, the instruction at location 1 accesses both a data structure and a scalar. If this instruction's statistics were

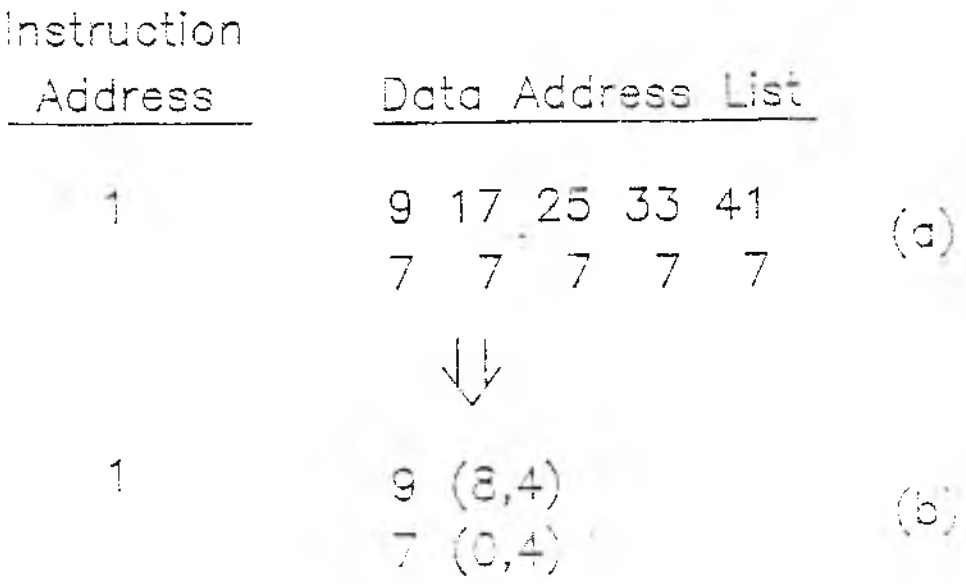


Figure 2-3. Sample data address list.

tabulated in Table 2-2, the statistics would appear in the final column under scalars and data structures. The instruction would be counted once for a static count and five times for a dynamic count. The ordered list of data references may be written in a more compact form by calculating the displacement from one reference to the next. In the case of array references, the same displacement often occurs several times in succession. Thus, to achieve a more compact representation, those references which caused the same displacement to occur several times in succession are replaced by a displacement and a count of the number of times that that displacement occurs. This transformation on the data address list of Figure 2-3a produces the data reference list of Figure 2-3b. The first number is the address of the first data reference. Following that number is a list of pairs of numbers; the first being the displacement and the second the number of times that that displacement occurs. With such a notation, the entire list of data addresses for an instruction can be generated.

From this initial analysis, the frequency of access for scalars and data structures can be found. The total number of unique scalars and unique data structure elements may also be found. The number of unique data structure elements in a program is found by performing a pairwise comparison of data address lists and checking for common addresses. Data references with common addresses are then merged to form a list of the addresses of all the items in a data structure. Table 2-3 is a summary of this information. The first line of entries for a program shows the number of unique items, static references, and dynamic references for scalars and data structures as a percentage of the total number of the respective references. The entries for the unique scalars

Table 2-3. Data analysis results.

Program	Scalars			Data Structures			
	Unique	Static	Dynamic	Unique	Static	Dynamic	Number
GAUSS	32.5 212	64.5 403	51.3 17590	67.5 441	35.5 222	48.7 16679	3
EIGEN	52.4 257	91.7 752	72.9 19422	47.6 233	8.3 68	27.1 7233	7
CCOBOL	20.7 636	46.4 1108	64.4 50503	79.3 2433	53.6 1278	35.6 27894	168
ECOBOL	14.3 1043	77.6 2225	83.9 65877	85.7 6242	22.4 642	16.1 12632	239

and unique data structures should sum to 100.0, since their sum represents the total number of distinct data items referenced by the program. The second row of entries for each program is the actual number of occurrences of each type of reference. For the GAUSS program, scalars represent 32.5% of the data locations referenced by the program. The scalars comprise 64.5% of the static data references and 51.3% of the dynamic data references. The remaining references in each category are data structure references. While 67.5% of the data locations referenced by GAUSS are part of data structures, these locations are partitioned into only 3 data structures. These data structures in fact correspond to the 20x20 A matrix, and the x and B vectors for solving A*x = B.

The number of unique scalars and the number of data structures in GAUSS and EIGEN is modest. While the number of unique scalars for CCOBOL and ECOBOL is higher than for GAUSS and EIGEN, the more significant difference is in the much larger number of data structures. In CCOBOL and ECOBOL, the percentage of unique scalars is much smaller than the number of unique data structure items. For all the programs, the percentage of dynamic scalar references is relatively high. In the case of GAUSS and EIGEN, which are matrix-oriented programs for which one might expect a high percentage of dynamic data structure references, scalars surprisingly comprise more than half of the dynamic references. For CCOBOL and ECOBOL, the dynamic scalar references are an even higher percentage of the dynamic references. While this may not be surprising, it is interesting to note that the dynamic scalar references are high

even though the number of data structures is large and comprise a large portion of the unique data locations referenced.

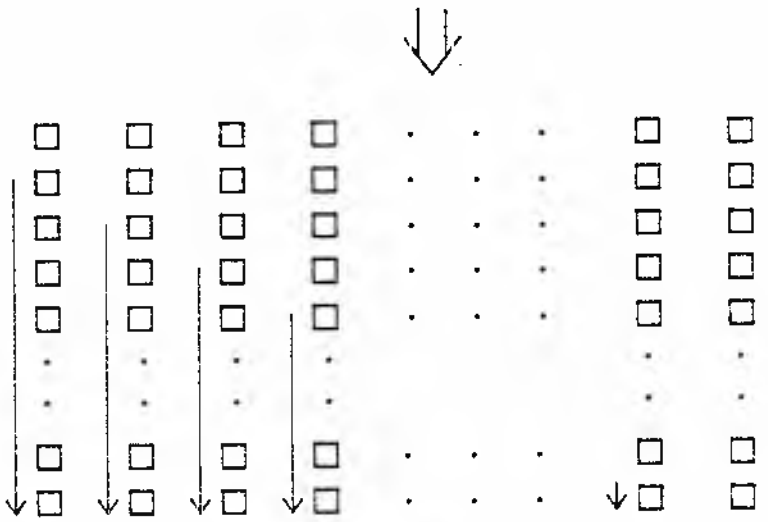
In addition to reconstructing the data structures from the address trace and producing frequency of use information, one can also reconstruct, by studying the address lists, the indexing loops which exist in the program. From the pattern of data structure references, specific loops must exist in the program to generate those patterns. Figure 2-4, for example, shows the data address list for the instruction at location 744054 in GAUSS. From the address list, one can deduce that the lower triangle of a matrix, assumed to be stored in column major order, is being referenced column by column. For such a reference pattern to occur, a loop equivalent to:

```
for i := 1 to n-1 do
  for j := i+1 to n do
    reference matrix element[i,j];
```

where n=20, must occur in the program. Each data address list which referenced a data structure was studied and the loop structure to generate the address list was deduced. Each address list is not necessarily associated with a unique loop structure. Several static data structure references may be combined in the same loop. As we will see in the next paragraph, the number of distinct access patterns is less than the number of static data structure references.

Table 2-4 lists all the unique access patterns for data structures which exist in GAUSS, as deduced from the trace analysis. We refer to these patterns as <u>access mechanisms</u> because each sequencing structure for nested loop indices may be treated as an independent mechanism for

Instruction Address	Data Address List				
744054	738504	(8,18)	(24,1)	(8,17)	(32,1)
		(8,16)	(40,1)	(8,15)	(48,1)
		(8,14)	(56,1)	(8,13)	(64,1)
		(8,12)	(72,1)	(8,11)	(80,1)
		(8,10)	(88,1)	(8, 9)	(96,1)
		(8, 8)	(104,1)	(8, 7)	(112,1)
		(8, 6)	(120,1)	(8, 5)	(128,1)
		(8, 4)	(136,1)	(8, 3)	(144,1)
		(8, 2)	(152,1)	(8, 1)	



```
for i := 1 to n-1 do
  for j := i+1 to n do
    reference matrix element [i,j]
```

Figure 2-4. Data address list analysis.

Table 2-4. Access mechanisms for GAUSS (n=20).

Structure	Index Level								
	i			j			k		
	Initial	End	Step Size	Initial	End	Step Size	Initial	End	Step Size
A[n,n]	1	n	1	1	n	1			
	1	n-1	1	i+1	n	1			
	1	n-2	1	1	n	1	i+1	n	1
	2	n-1	1	n	i	-1	i	n	1
B[n]	1	n-2	1	1	n-i	1			
	2	n-1	1	1	n-1	1			
	1	n	1						
C[n]	1	n	1						

the accessing of data structures. The index levels labeled i, j, and k represent, respectively, the outer, inner, and next inner levels of nesting for a loop. None of the loops in our program traces are nested more than 3 deep. The column headings labeled init, end, and step size represent, respectively, the initial value, the final value, and the step-size for an index. As may be seen, the GAUSS program has relatively few access mechanisms. While the step size for the indices is always a constant here, the initial and end values of the indices can either be a constant, be dependent on the array size, or be dependent on the current value of some higher level index. The distinction between

these types of values is important since the time at which these values can be bound differs substantially. A constant value is, of course, known when the program is compiled and thus the value may be incorporated in the code. A value which is a function of the size of the array may not be known until the program data is loaded. Once the program is called or the portion of the program which accesses the matrix is executed, the size of the matrix can be made readily available and this size remains constant while the matrix is being accessed. In contrast, a higher level index value is not set until the appropriate outer loop variable has been set. Furthermore, this value changes during execution every time the inner-loop is reentered from the outer-loop. Outer-loop index-dependent values for inner-loop index limits must therefore be bound and rebound during execution. In those cases where the initial or the final value is a function of the array size or an outer index value, the functions turn out to be very simple. Such a function is known at compile time; however, the value of the function is different for each execution of the program or actually changes during execution. A fourth possibility, not present in GAUSS, is that a loop is terminated when some data value condition is calculated and tested during computation.

Table 2-5 for EIGEN is similar to Table 2-4. While the number of distinct access mechanisms is greater, they exhibit the same features as those found in the access mechanisms for GAUSS. There are only three data structures shown in Table 2-5; however, Table 2-3 has 7 data structures listed for EIGEN. The four extra data structures listed in

Table 2-5. Access mechanisms for EIGEN (n=14).

Structure	Index Level								
	i			j			k		
	Initial	End	Step Size	Initial	End	Step Size	Initial	End	Step Size
A[n,n]	1	n	1						
	n	2	-1						
	n	1	-1						
	n	2	-1	1	i-1	1			
	n	3	-1	1	i-1	1			
	n-1	2	-1	1	i	1	1	j	1
	n	2	-1	1	n-i	1			
	n-1	2	-1	1	i-1	1	j+1	i	1
	n	2	-1	2	n-1	1	j	n-1	1
B[n]	1	n	1						
	n	1	-1						
	1	n-1	1						
	2	n	1						
C[n]	n	1	-1	1	i	1			
	n	2	-1						
	n	2	-1	1	i	1			

Table 2-3 are due to data structures which are referenced by block move instructions found in the initialization part of the program. The references to these four data structures are not shown in Table 2-5 because these references are not generated within program loops.

In an instruction block, several access mechanisms may be active at the same time and a single access mechanism may be used in more than

once place in the block. Also, a large number of scalar references occur in an instruction block which references data structures. In our experience, most of these scalar references are used to control access mechanisms. A machine which minimizes the number of references made by the access process for controlling itself, must provide the ability to execute access mechanisms with far fewer such memory references.

CHAPTER 3

STRUCTURED MEMORY ACCESS MACHINE (SMA) ARCHITECTURE

From the trace analysis described in the preceding sections, it is possible to determine the control and data structures of a program and the mechanisms by which data structures are accessed. Realizing that these structures remain intact in the program's transformation from a high-level language to machine level, we believe it is possible to design a machine which reduces addressing overhead by taking advantage of a program's structural information. In this section such a machine, the Structured Memory Access (SMA) architecture, is proposed. By carefully organizing the machine's architectural features, one can develop an access process which makes fewer references to memory. In addition, the access process may, by and large, be decoupled from the computation process, thus providing a maximum degree of overlapped execution.

Access process overhead exists in two forms. Address specification overhead refers to the increasing number of address bits needed to address a memory location as the address space becomes large. Most of these bits are redundant, given sequence information about address sequences. This type of overhead translates directly into wider address fields wherever an entire address is specified. One such case is the addressing of scalars and branch targets. The second and more costly form of overhead is address calculation overhead, which refers to address calculations explicitly performed by the CPU. Address

calculation overhead involves some combination of extra instructions, parts of instructions, registers, memory accesses, and computation time. These types of overhead can be greatly reduced if machines were designed differently.

As in most machines, we assume instructions are normally executed sequentially. At some point in the execution of a sequentiality, a branch occurs due to a decision made by the program. To generate requests for instructions, the starting address of the program and also all the target addresses of all branches which occur during the running of the program must be specified. In some conventional processors, the target address of a branch is stored with the branch instruction. In many cases, however, since programmers are permitted to use the entire address space for storing programs, the target address usually contains as many bits as an entire word or even more. If instructions are, at most, one word long, an entire target address cannot be stored in the branch instruction itself. To remedy this problem, conventional machines are designed so that the target address information in the branch instruction is either an indirect pointer to the true target address, an offset to be added to a value in some base register, or an offset to be added to the current value of the program counter.

Since the number of instruction blocks or, equivalently, the number of branch targets is not as large as the number of memory locations, these methods for specifying the target address cause an address specification overhead. One way to reduce this overhead is to specify an instruction block name as the target of the branch instead of

specifying a target address. Since the number of instruction blocks is small with respect to the total address space, the number of bits needed to specify a target block is small. Such a reduction in address specification overhead is bought at the cost of special tables, implemented in hardware, which store information for directly translating block numbers into actual addresses.

The SMA machine uses a different approach to reduce the address specification overhead when accessing the target of a branch. The complete branch target address is specified in the branch instruction. However, since the SMA machine provides instruction buffers to capture repeatedly executed instruction blocks, the number of times the branch instruction and the target address are accessed is reduced. The instruction buffer effectively limits the number of bits fetched from memory to specify a branch target address.

The addressing of scalars is another source of address specification overhead and similar to the overhead of specifying a branch target address. To reference scalars, the SMA machine provides a base register. A scalar reference specification is an offset to be summed with the contents of the base register to form an entire scalar address. Entire scalar addresses are not specified with the instruction as for branch target addresses because one can expect a scalar reference to occur several times in an instruction block. To reduce scalar specification overhead, the entire address for the scalar references in a block would have to be held in an instruction buffer. Such a scheme would therefore require a large instruction buffer.

The referencing of data structures is the prime cause of address calculation overhead and poses a much more serious addressing problem. Address calculation overhead causes the larger than expected number of scalars in the inner loops of the Gaussian elimination program. To eliminate this overhead, we propose that special hardware be provided to generate data structure references. Properly organized, this hardware can reduce the number of memory accesses which must be made to generate a data structure address.

Conventional machines have index registers to aid in the generation of data structure reference addresses. Although these are intended to reduce overhead, significant inefficiency is apparent in the analysis results for GAUSS. If one considers a high-level description of the Gaussian elimination algorithm, there is little explicit use of scalars. The analysis of GAUSS, however, reveals a great number of scalars used in the program. One could argue that these are used for initialization or for I/O, but this does not appear to be the case since the dynamic count for the number of scalar references is extremely high. As will be seen in Section 5.1, many of these scalars can be eliminated, since they are being used either for tracking loop indices or for the generation of data structure references.

The SMA machine implements the function of index registers by using a hardware stack. This stack tracks all the indices used by a program, and all data structure references are made by using a set of these index values. To reduce the number of bits which need to be transferred from the memory when generating an entire address for accessing a data

structure element, tables, located in the SMA machine, are used to store the base address of a data structure and other information necessary to generate an entire address from indices. These tables must be loaded before any instruction which uses them is executed. Depending on the amount of space allocated for the tables, the number of data structures, and access mechanisms, the tables may only have to be loaded once, at the beginning of program execution. If a program has too many data structures and access mechanisms for a single loading, the tables may also be loaded at other times during the execution of the program. A data structure reference specification is a set of pointers to table entries. Section 4.1.3 provides a detailed explanation of these tables, the information in them, and how they are used to generate a data structure reference. Such a scheme provides the flexibility of generating an access mechanism while maximizing the speed of address generation through the use of pipelining techniques.

Generally, the value of an index only needs to be associated with the access process. Thus, the stack containing indices, the tables for generating data structure references, and the address generation portion of the CPU may be separated from the computation-oriented portions of the CPU. This partition divides the computer system into two processors: a computation processor (CP) and a memory access processor (MAP). The CP is used strictly for the computation process, i.e. the useful computations of the system, while the MAP is responsible for the access process, i.e. generating all addresses for data and instructions. The index stack and the associated access tables mentioned above are

kept in the MAP. Since only the MAP generates addresses, it controls all transactions with the memory.

This SMA is shown schematically in Figure 3-1. There is no address bus between the CP and the memory since all memory requests are generated and controlled by the MAP. Also, since the CP is not responsible for addressing, the instructions sent to the CP contain no addressing information. Thus, the instructions are short and contain little more than opcodes and register tags. The CP is strictly devoted to performing computations and contains the ALU of the system; instructions and data are streamed into the CP by the MAP. The CP may receive entire blocks of instructions which it then holds in an internal instruction buffer. If a block happens to loop upon itself, the CP may execute in a loop mode similar to the loop mode of the IBM 370/91 [Ande67]. In addition, the CP also has a set of registers for holding the scalars used by an instruction block. The internal instruction buffer and the registers are provided to eliminate some repeated memory accessing and its associated time and load on the MAP.

The MAP, which is responsible for providing instructions and data to the CP, "knows" that the content of memory is composed of instruction blocks, scalars, and data structures. Each of these presents the MAP with a unique set of access problems which the MAP handles through its special hardware. Some of this hardware is activated by a set of special instructions which are intended for the initialization and control of the access mechanisms used for address generation. An SMA program is somewhat different from a program for a conventional machine

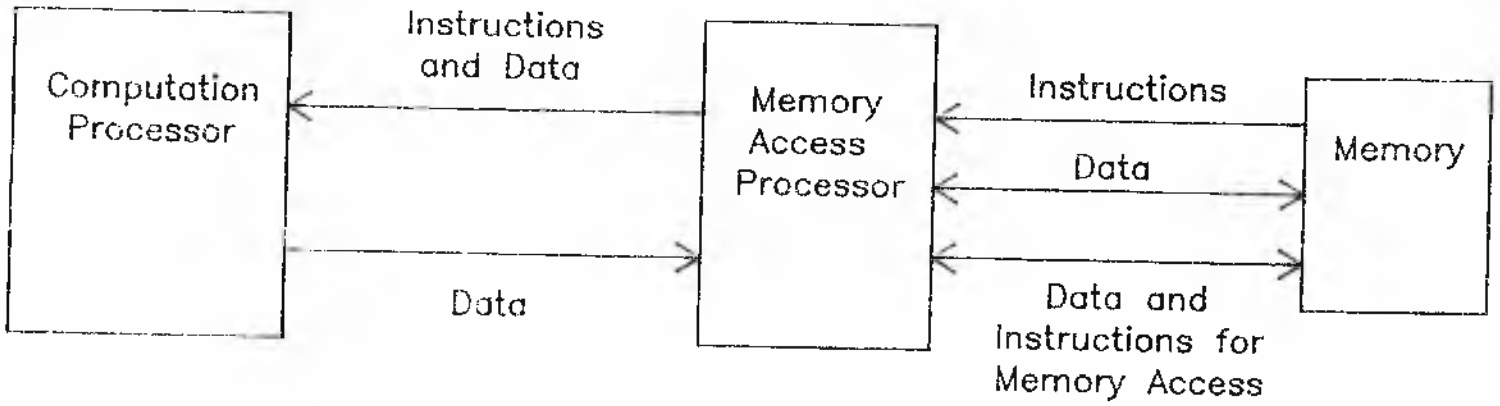


Figure 3-1. SMA structure.

since it can contains two types of instructions, MAP and CP instructions and the data type of an operand is explicitly specified in an instruction. This extra information found in SMA instructions requires that, at compile time, the compiler be capable of distinquishing loop control branching from data dependent branching and scalars from data structures.

Just as with the CP, the MAP has an internal buffer to hold its instructions and the operand specifications of CP instructions. The MAP can, therefore, operate in a loop mode fashion. Operation of the MAP is, to a great extent, independent of the CP. When the MAP begins receiving instructions it forwards the instructions to the CP. The MAP saves the operand specification portions and begins generation of operand addresses. The operand addresses are placed on a queue of outstanding memory requests: one queue for read requests and one queue for write requests. As soon as a read request is serviced, the operand returned by that request is forwarded to the CP. With such a scheme, the CP concentrates on the useful calculations of a program, while the MAP is left with the important, but overhead-related, generation of operand addresses.

Having a two-processor organization presents several options for locating the circuitry which makes branch decisions. In conventional systems, the ALU makes branch decisions. In our case, however, all branch decisions need not be routed through the CP. Two types of branch decisions occur: decisions based on program data and those based on indices used for referencing data. We propose that branching decisions

can originate from either the CP or the MAP. The branches resolved in the CP are based on scalar or data structure item values. Those branches which are based on index values are resolved in the MAP since the MAP tracks all index values.

Such an organization not only reduces the addressing overhead for a system but also reduces the serial dependence which exists between the access process and the computation process. Since the MAP makes branch decisions based on index values during the execution of a loop, the MAP can generate memory requests for operands before the CP is ready to execute the instructions requiring those operands. In fact, the MAP should normally stay ahead of the CP so as to minimize the amount of time that the CP waits for data requests from memory. Of course, there are occasions when the MAP must wait for the CP and vice-versa. For example, the MAP must wait on the CP when the MAP's data read queue is full or when the CP must resolve a computation-dependent branch. On the other hand, the CP must wait for the MAP when the MAP's data write queue is full or when the MAP's data read queue is empty.

Super computers and vector machines contain special hardware for array referencing; however, the programming of these machines quite often requires rearranging of an algorithm to suit the hardware. Furthermore, their structured data access mechanisms are usually limited to a single vector of the structure at a time, i.e. "a constant stride," or constant step size access mechanism with one index. Also, the same operation must be executed on each element of the vector. The TI-ASC offers somewhat more flexibility by providing both an inner and outer

loop control for stepping though a matrix, i.e. two active indices. The SMA machine provides more flexibility in the accessing of matrices since it offers more index levels by providing a stack on which to store indices. For example, Tables 2-4 and 2-5 show that when accessing a 2-dimensional structure, occasionally three levels of nesting are used. These up extra levels of nesting could also prove useful for providing non-constant strides.

In vector machines, the vector access mechanisms are explicitly coded into instructions and then recognized and setup during execution time. The SMA architecture is designed so that data structure access mechanisms are recognized as early as possible. Depending on the access mechanisms, accesses mechanisms can be set as early as compile time or load time. This early recognition can lead to reduced run-time overhead.

The SMA organization described above is used to reduce the addressing overhead, primarily by improving the accessing of data structures through efficient access mechanisms and prefetching. The process of accessing instructions can likewise be improved if information concerning the instruction block structure of a program, which is apparent in high level source code, is kept with the program as it is translated down to machine level. Retaining the block structure of a program can be used advantageously to cause the CP to enter and leave loops.

As mentioned in Chapter 2, the IBM 360/91 has a built-in loop mode [Ande67]. Loop mode control is generated dynamically during execution.

Upon recognizing a short backwards branch, it is assumed that the second iteration of a loop is about to begin. The instructions of the loop are refetched and trapped in the loop buffer where they remain for repeated execution until the loop ending branch is unsuccessful.

The loop buffers in the CP and the MAP also trap loop instructions. Unlike the IBM 360/91, however, the loop mode control is set up at compile time. Loop structures are quite explicit and obvious in the high level language source code available at compile time. If the instruction blocks which form the body of a loop are sufficiently short, they may all be stored in the instruction buffer at the same time. The processors thus are able to trap the body of a loop the first time the loop is executed. The loop buffers eliminate the needed for repeated memory accesses for the same instructions during the execution of a loop. In any case, repetition requires no data dependent branch and no wait time. Execution continuation after the loop is also efficient when the successor block is known, since it can be prefetched during loop execution.

Many machines today do not have an explicit loop mode. Instead, caches are used. While caches are not only used to replace loop mode, a cache does, in fact, perform functions similar to a loop buffer. When a cache miss occurs, in addition to accessing the word of memory which caused the miss, several adjacent words are also placed in the cache. If a program does not have a great deal of spatial locality, it is easy for the extra memory words which were brought into the cache to go unaccessed until they are replaced in the cache. Since access to the

memory is a critical resource, information which is brought into the cache and not used is a waste; furthermore, instructions compete with data for cache space because of the way in which caches are structured. There is no guarantee that needed information will stay in the cache because stack update policies don't "know" which information is useful; at best, these policies are only heuristic. By keeping block information with the program (and explicitly controlling data structure accesses), one knows exactly which instructions to save, and when prefetching, exactly how many instructions to prefetch. Hopefully, one can also predict data references sufficiently far ahead of the need for the data values at the CP. Thus a few small buffers and a simple memory access processor can potentially eliminate the need for a fast cache memory, and a single slow memory is sufficient.

This chapter has presented an overview description of the SMA architecture. The main feature of this architecture is the decoupling of the computation process from the access process. A memory access processor (MAP) handles all transactions with the memory and is responsible for the generation of memory requests. The MAP forwards instructions and data to a computation processor (CP). The CP performs all the useful computation of the system and is not burdened with the overhead of address generation. The decoupling of the computation and access processes maximizes the execution overlap which can be achieved. The SMA architecture also provides instruction buffers and address generation mechanisms which reduce the number of memory references which are made.

CHAPTER 4

AN SMA IMPLEMENTATION

In the previous section, the general organization and features of the SMA architecture were presented, as were the reasons for their inclusion. In this section, we describe how such a machine could be implemented. This description outlines one possible implementation of an SMA machine and is used to demonstrate that such a machine is feasible. An attempt is made to keep the discussion sufficiently general so as not to be burdened by such issues as word size, buffer size, etc. While these issues are important if such a machine is actually constructed, a discussion of them does not add to an understanding of the machine's operation. When specific numbers are given for certain machine features, they are given only for illustrative purposes.

Our discussion is primarily concerned with two issues: the way in which data accesses are made and the aspects of program control. Since the SMA machine is primarily proposed to improve the efficiency of data referencing, we first concentrate on the manner in which data references are generated. The mechanisms which perform data structure accesses influence the way in which program control is accomplished, thus, program control is discussed subsequently.

4.1. Data Referencing

4.1.1. Data Types

One feature of the SMA machine is the capability of the MAP to generate addresses for all data which is referenced during the execution of a program. The SMA machine's performance is expected to be better than that of conventional machines since the SMA machine has special hardware mechanisms for the generation of operand addresses. In this implementation, the SMA distinguishes among four types of operands: (1) immediate operands, (2) scalar operands, (3) data structure operands, and (4) index operands. Immediate operands are data whose values are embedded in an instruction. Scalar and data structure operands are defined as they were previously when the analysis of program address traces was discussed. Data structure operands are items from structures such as vectors and matrices. The final type of operand, an index operand, is the value of one of the current indices found on the index stack. The index operand is used only to read a current index value from the index stack and transfer its value to the CP. An index operand differs from a scalar operand primarily in that the index operand originates from the MAP while the scalar operand originates from the memory. To generate operand addresses efficiently, the MAP must know which of the operand types an instruction is referencing. The operand type may be specified in a subfield of an instruction's operand field or it may be implicitly associated with a particular instruction. Additionally, an indirect addressing mode is provided specifically for use in the calling of subroutines and in the

accessing of data items from structures such as linked lists. As with operand type specification, indirect addressing may be specified in a variety of ways.

While we only discuss the referencing of four operand types, this discussion does not preclude the addition of other operand types in other versions of an SMA machine. For instance, at some time it may be desirable to distinguish explicitly among several types of data structures. Instead of having a data structure operand, one may wish to have an array operand, a linked list operand, a binary tree operand, etc. For each operand type, some special accessing mechanisms are provided to improve the speed with which an operand address is generated.

4.1.2. Immediate and Scalar Operands

The value of an immediate operand is found in an instruction's operand field. If indirect addressing mode is specified with an immediate operand, the immediate data found in the instruction's operand field is treated as the address of the actual operand. Thus, indirect addressing with immediate operands causes a memory access. This immediate mode of operand referencing is provided to reduce the need to access memory for repeatedly used constants or constants which are to be loaded into registers.

Scalar data is treated in the manner of a vector rather than as a set of disassociated items. The MAP provides base registers which contain the beginning address of a scalar data area. The specification

for a scalar operand includes a specification of a base register and a displacement into a scalar data area. The MAP can have more than one scalar base register to aid in the accessing of local and global variables, such as during subroutine calls. Such a base register can be used as the argument pointer set during a subroutine call.

Such a scheme is used to minimize the number of bits which are transferred between the MAP and the memory when scalars are referenced. For the programs we studied, the number of simultaneously active scalars is relatively small, particularly in an SMA program (see section 5.1). Those scalars which are active at the same time are known at compile time, thus they can be elements of the same scalar data vector. Specifying only a displacement into a scalar data area can reduce the size of instructions since the number of bits needed to specify the displacement is small. For example, if the operand field of an instruction is 8 bits long, 2 bits could be used to specify which of the 4 data types is being referenced and 1 bit could be used to specify indirect addressing. The remaining 5 bits in scalar mode could then be used as a scalar displacement into a scalar data area from which one of 32 scalar items could be selected. If for some reason, more than 32 scalars are needed, the scalar displacement register can be loaded with a new base address. With a reasonably intelligent compiler, the number of times the scalar displacement register is reloaded can be kept to a minimum thus minimizing the number of bits transferred between the memory and the MAP.

4.1.3. Data Structure and Index Operands

The SMA's memory access processor has special mechanisms to track indices for data structure computations. These indices are used to generate the addresses for specific items of the data structure to be referenced. We describe below how these indices are tracked, how a data structure reference is specified, and how indices are used to generate an address.

An index is specified by its current value, final value, step-size and indexing level. When the index is first established, the current value is equal to its initial value. At any time, several indices may be active, and the level, or nesting, of these indices is dictated by the time at which they were instantiated or setup. So, if one has a simple loop structure such as:

```
for I := 1 to n by 1 do
  for J := 1 to n by 1 do ..... ;
```

the index I is at a higher level than the index J. In the SMA, the current value, final value, and step-size of an index are kept on a LIFO stack structure known as the index stack (IS). Each stack position is numbered sequentially, with the bottom of stack numbered level 1. This convention provides a convenient way of referring to the current value of any active index because the bottom-most stack entry corresponds to the outer-most level of nesting, i.e. level 1. When a "setup index" instruction is executed by the MAP, the initial value, final value, and step-size are pushed onto the stack. To change the current value of an index, an "increment index"" instruction is used. This instruction must

specify three items, the first gives a number for the level of the index which is being incremented. The MAP uses this number to select an index on its stack. When the instruction is executed, the current value of the index is incremented by its associated step-size. The second two items are initial addresses of blocks which are the targets of a branch outcome. If the current value of the index is less than the final value, control is transferred to the first block which is specified. If, on the other hand, the current value equals or exceeds the final value, control is transferred to the second specified block.

This approach of operating on indices in the MAP has several advantages over conventional methods. The operations for controlling the indices are relatively simple and regular and can, therefore, be efficiently performed in special hardware, thereby reducing the amount of code needed to accomplish them. By checking the index value after each increment and by having the branching information available, the next instruction can start being accessed while the CP is still performing the final computation of a loop. Furthermore, no guess is made about which direction an index-based branch will take, thus no time is wasted in fetching blocks of instructions from the main memory which will not be used.

When the current value of the index has reached its final value, that index is removed from the stack. Two other methods of removing indices from the stack are (1) the "remove index" instruction which removes the highest level current index from the top of the stack and (2) a "clear all indices" instruction which removes all indices from the stack.

As mentioned earlier, the setup instruction places the initial value, final value, and step-size for an index on the stack. These values could be fetched from the memory when they are placed on the index stack; however, if they can only be fetched from memory one at a time, the memory fetches would involve substantial overhead. The method used here, which reduces this overhead, loads the MAP with a set of templates for these values at the start of program execution. A template is a specification of the values needed to initialize an index on the index stack. Templates are loaded into an index template table. When an index is setup in the IS, the IS is loaded directly from the index template table. For a particular program, the number of distinct templates could be fairly small. For example, in Tables 2-4 and 2-5 notice the number of times an index has an initial specification of 1,n,1. In all, GAUSS requires 995 dynamic index setups and 16 static index setups, but only 8 templates. The situation for EIGEN is similar since 655 dynamic and 27 static setups are needed but only 14 templates are used. Each index activated with a particular initial specification can use the same entry from the index template table. In addition to the time saved by having templates stored in the MAP, using templates reduces the number of accesses which are made to the memory. Even if the number of templates exceeds the table size, judicious reloading would limit additional overhead.

To access data structures in the SMA, one must combine index values to form a data address. In the SMA, information for forming proper

combinations is stored in two data tables within the MAP. As with some of the other repeatedly used information, the contents of the tables may be loaded when the program begins execution. The two tables are the access pattern table (APT), which indicates the index levels to use, and the access information table (AIT), which contains information about data structures.

Each line of the APT is divided into several dimension fields. The number of dimension fields is an implementation issue and limits the maximum number of dimensions a structured data access mechanism may have. Thus, each dimension field is associated with a dimension of the data structure. Each dimension field is divided into 2 subfields. The index level subfield (ILF) indicates which level of the index stack (IS) is associated with that dimension field. The number of bits allocated to the ILF must be large enough to specify one more than the depth of the IS. The extra bit combination is used when a data structure has fewer dimensions than the maximum permissible. When found in an ILF field, this extra bit combination indicates that the corresponding dimension is not used for the data structure being accessed.

The second subfield (IOF) contains the value of a small positive or negative offset to be added to the index before the index is used. This feature is useful since quite often the index of a data structure access is an existing presently active index plus or minus a small constant. From the APT and the index stack, the values of indices for a data structure reference are determined. An important aspect of using an APT is that an entry in the APT may be used by more than one data structure

since the information is not altered during execution and does not depend on accessing a specific data structure. Such sharing aids in minimizing the number of lines found in the APT.

The values of the indices pointed to by the APT entries are used with information in the AIT to generate a data structure address. For each data structure currently being used by the program, there is an entry in the AIT. If the number of data structures in a program is sufficiently small, the AIT need only be loaded at the beginning of program execution. If the AIT needs to be reloaded, a compiler using static information from the source program can keep the reloading of the AIT at a minimum. Each entry of the AIT is composed of three types of values: (1) the base address of the data structure (DSBA), (2) a displacement for each dimension of the data structure (DISP), and (3) an upper bound for each dimension of the data structure (UPB).

With this background, once it is known that a data structure is to be referenced and whether direct or indirect addressing is to be used, a data structure reference may be made by specifying an entry in the APT and an entry in the AIT. A data structure address is generated by the following calculation:

```
Address =

    (structure base address) +
    ((first index value + offset) * first dimension displacement) +
    ((second index value + offset) * second dimension displacement) +
                                .
                                .
                                .
    ((final index value + offset) * final dimension displacement)
```

The terms in this address are obtained as shown below, where SA (specification for APT entry) and SN (specification for AIT entry) are pointers to entries in the APT and the AIT, respectively. The index stack (IS), the APT, and AIT are treated as vectors whose elements are levels of the stack or lines in the table where each line may have several fields. The notation X[Y].Z refers to the Z field of line Y in table X. The address is then given by:

Address =

```
AIT[SN].DSBA +
(IS[APT[SA].ILF1] + APT[SA].IOF1) * AIT[SN].DISP1 +
(IS[APT[SA].ILF2] + APT[SA].IOF2) * AIT[SN].DISP2 +
                        .
                        .
                        .
(IS[APT[SA].ILFn] + APT[SA].IOFn) * AIT[SN].DISPn
```

The last n terms of this equation represent the successive displacements from the base address for each dimension of the data structure. Before each of these terms is added in, a bounds check can be made on the value of an index by using the upper bounds (UPB) in the AIT. Namely, after an index value for a dimension is obtained from the IS and the offset in the APT is added to that index, the resultant value is compared to the upper bound for that dimension, if the index value exceeds the upper bound, a referencing out-of-bounds error has occurred.

Occasionally, an algorithm must know the current value of an index. For example, in a Gaussian elimination algorithm, before pivoting is performed, a row by row search is performed for the largest valued element in a column. The value of the index associated with the row

which contains the largest valued element is saved as a scalar in memory (possibly temporarily in a CP register) to be used later when rows are exchanged. Thus, the current value of the index must be transmitted to the CP from the MAP. In this example, before the rows are exchanged, the saved index value is read from memory into the MAP and placed on the index stack via the setup index instruction. Since indices are stored in and tracked by the MAP, referencing an index, while straightforward, is a slightly different process than referencing scalars or data structures. For these reasons, index operands are a special type of operand. As with using indices to access data structures, an index is referenced by indicating which level of the index stack is to be accessed.

Although the MAP performs many functions, a decoupled organization permits a great deal of overlap with the "useful" computations occurring in the CP. Data addresses are generated in the MAP at the same time the CP is performing its calculations; thus, address calculation overhead is limited. In conventional architectures, a great deal of access overhead information must be repeatedly brought into the CPU from the memory. Since the MAP repeatedly performs very regular calculations, many of these calculations can be pipelined. The address generation for a data structure, in particular, is a very good candidate for pipelining. Addresses can then be streamed to memory at near maximum bandwidth and the rate of address generation is extremely high.

4.2. Control Issues

While the preceding discussion indicates how operand addresses are formed, nothing has been said concerning the control structures of the SMA or how the CP and the MAP are coupled. As shown in Figure 4-1, interior to the MAP are several functional units and data storage areas which control the flow of instructions and data to the CP. We describe these functional units and data areas briefly before presenting a more detailed individual description.

As its name implies, the instruction fetcher is responsible for generating instruction requests. The instruction fetcher sends the instructions it receives from the memory to the instruction preprocessor. Among other things, the instruction preprocessor determines whether to save an instruction in the MAP or to forward it to the CP. The instruction preprocessor places MAP instructions and all operand specifications associated with MAP and CP instructions into an operand and instruction buffer in the MAP. This operand and instruction buffer is analogous to an instruction buffer in a conventional CPU. An address generation unit steps through the MAP instructions and operand specifications found in the operand and instruction buffer and generates operand addresses. These operand addresses are placed on a read queue or a write queue. When the memory returns the data associated with addresses in the read queue, that data is sent to a FIFO buffer in the CP. The CP sends data to the MAP for the write queue. Addresses in the write queue which have received their associated data are serviced by the memory.

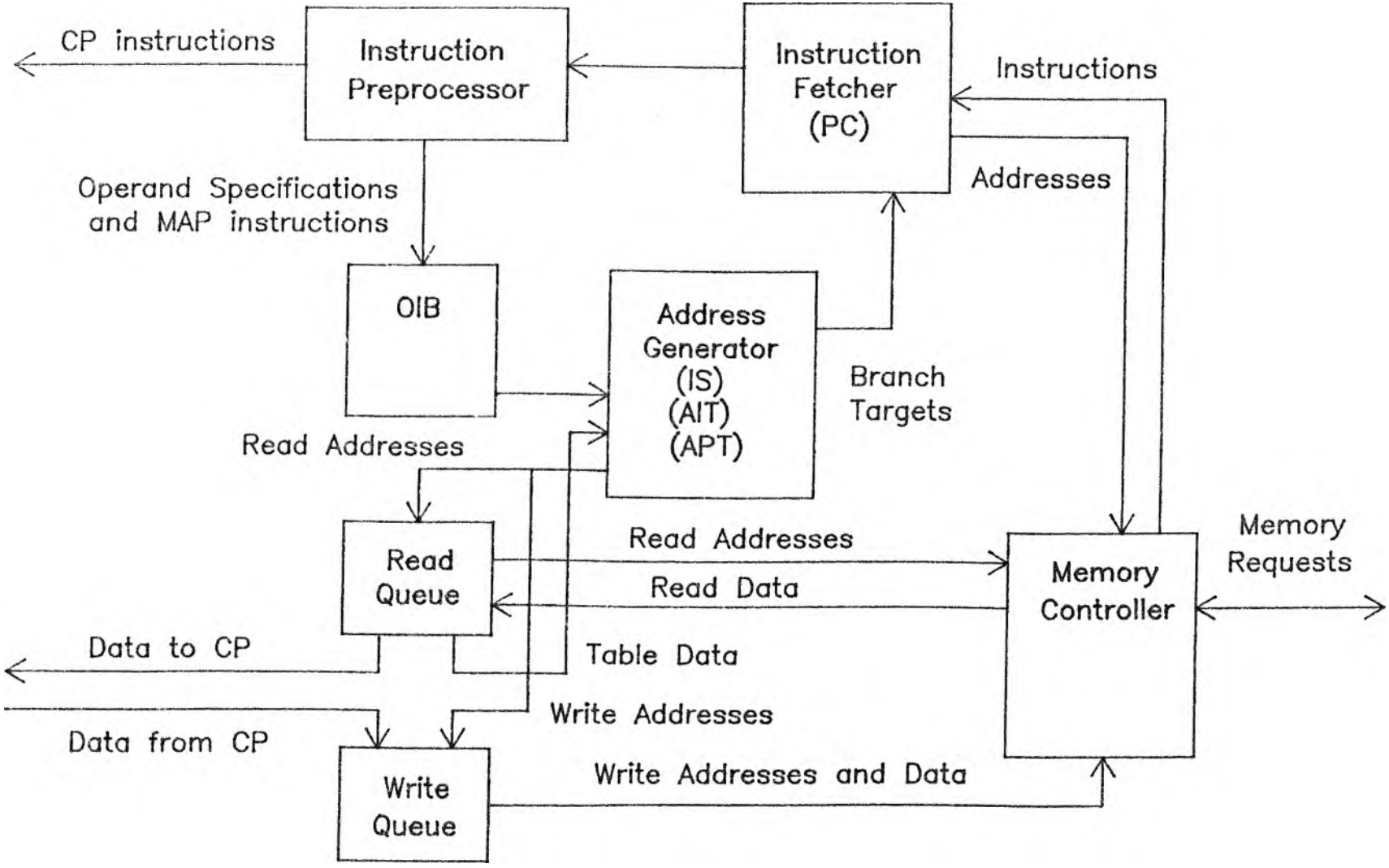


Figure 4-1. MAP internal organization.

The CP has an instruction buffer to hold the instructions it receives from the MAP. An execution unit in the CP steps through the instruction buffer, executing instructions one by one. If an instruction needs a data item from memory, that data is found at the head of the FIFO buffer. If the buffer is empty execution is suspended until a data item is received from the MAP. Along with each data item, the CP receives an additional bit from the MAP which is used as an end-of-data signal. Assertion of the end-of-data signal in loop mode indicates that execution of the current instruction loop is to terminate and that the CP should begin execution of another block found in its instruction buffer, or wait until a new instruction block arrives from the MAP. The CP generates data, which the MAP must write to memory, and signals the success or failure of a test for data-dependent branches.

In the following two sections we discuss in more detail the flow of instructions into the MAP, how operand requests are serviced, and how the MAP and CP communicate to resolve branches. Following that discussion is a section which presents the operation of the CP in more detail. Finally, section 4.2.4 discusses subroutine handling in the SMA.

4.2.1. Instruction Fetching and Operand Request Servicing

A program begins execution by having the monitor or operating system jump to the beginning of the program. That is, the operating system sets the program counter (PC) to the starting address of the program. In the SMA machine, the PC is located in the instruction fetcher of the MAP. When the PC is set to the beginning of the program,

the instruction fetcher generates requests for instructions from the memory. Instruction requests are generated until the end of a block is encountered. If the instruction at the end of a block is a branch instruction, be it for the CP or the MAP, the instruction fetcher suspends operation until the branch is resolved.

While the instruction fetcher is fetching the next instruction, or waiting for a branch resolution, the instruction preprocessor checks the opcode of an instruction to determine the number of operands the instruction has and whether the instruction is for the MAP. While the CP instructions are passed to the CP, without complete operand specifications, some additional bits are concatenated on the opcode for each operand to indicate if the value of an operand is to be found in a register or at the head of the buffer holding data forwarded by the MAP. Additionally, an end-of-block bit is sent with the last instruction of the block. Those instructions which are for the MAP and operand information for all instructions are routed to the MAP operand and instruction buffer (OIB). The instruction preprocessor places the proper number of operand specifications on the OIB by interpreting the number of operands for the instruction from the instruction's opcode field. The OIB saves the MAP instructions and operand specification information which the MAP needs to generate the memory requests for the instruction block currently being sent to the CP.

The OIB is a fixed length FIFO stack structure; the important fields of the OIB are labeled as shown in Figure 4-2. The data/instruction bit of a line indicates whether the information stored

Data or Instruction (1)	Read (1)	Write (1)	Operand Field or Instruction Opcode	End of Block (1)	Address

Figure 4-1. The operand and instruction buffer (OIB).

in that line of the OIB is a MAP instruction or an operand specification. The end-of-block field indicates that the line entry is the last OIB entry of a block. This information is necessary if information for more than one block is to be simultaneously stored in the OIB. The read and write bits indicate whether an operand is to be read from or written to memory. The address field holds the address of the first instruction of an instruction block. The address is saved to be used later to check whether a block loops upon itself.

The instruction preprocessor is responsible for setting the fields of the OIB. The data/instruction field, end-of-block field, address field, and the operand specification can be determined by the instruction preprocessor when it receives an instruction from the instruction fetcher. To determine if an operand is to be read from or written to memory, a convention similar to that used in conventional machines can be used for the SMA machine. For instructions with one operand, the instruction opcode determines whether the operand access will cause a read or a write. If an instruction has two operands, the first causes a read from memory and the second causes a read followed by a write. If three operands are specified, the first two cause reads from memory while the third causes a write to memory. Thus, depending on the number of operands, the instruction preprocessor sets either the read bit, write bit, or both the read and write bits for each operand. The address field holds the address of the first instruction of the block as saved from the PC.

The instruction fetcher and instruction preprocessor can operate simultaneously. The instruction fetcher can generate a request for another instruction as soon as the fetcher passes the current instruction to the instruction preprocessor, provided the current instruction is not the last instruction of the block. While the fetcher is obtaining another instruction, the instruction preprocessor can check the instruction just received. Once the preprocessor places entries into the OIB, the address generator can begin its work. The address generator need not wait until the operand specifications for an entire block are placed in the OIB.

With the information stored in the OIB, the address generation unit can generate all the data requests required by a program. As the OIB is loaded, the address generator can begin executing MAP instructions and generating operand addresses by stepping through the entries of the OIB with its own internal program counter. Using the one-bit data/instruction entry in the OIB, the MAP knows whether the entry is an operand, for which it must generate an address, or a MAP instruction, which it must execute. Operand addresses are formed as described in the previous chapter and are placed in a read queue or write queue depending on the value of the read and write bits. If both bits are set, an address is placed in both queues.

For every address that the address generator places on a queue, it also sets a one-bit indicator if indirect addressing is being used and one bit to indicate whether the operand is destined for the MAP or the CP. The address generator determines whether indirect addressing is being used by examining the operand specification.

The read and write queues are organized as shown in Figure 4-3. Each entry in the queue is divided into 6 fields: 4 one-bit status fields, an address field, and a data field. The read queue has an extra (end-of-data) status bit which is explained below when branch resolution is discussed. Since the addresses for both CP and MAP operands share the same queue to preserve order, one bit is used to differentiate between the two types of addresses. The second status bit indicates whether direct or indirect addressing is being used. These two bits are set by the address generator when an address is placed in a queue. A received bit is used for every queue entry to indicate that read data has arrived from the memory or that write data has been received from the CP. The received bit is necessary since operands may arrive from or be written to memory out of order, depending on the memory organization and the use of indirection. Also, since immediate operands are placed on the queues to preserve order and are "ready" when placed, their received bits are set immediately. Finally, a fourth status bit is used because the queues are used as buffers for operands going from memory to the CP and vice-versa. Since the wait time in the queue may vary, a fourth status bit, a done bit, is added for each entry to indicate that the associated entry is no longer needed.

Thus, in the case of read operands, the read address is placed in the address field while the CP/MAP and indirect bits are set appropriately and the received and done bits are reset. (Of course the received bit is set if the operand is an immediate operand). A memory

CP/MAP (1)	Indirect (1)	Received (1)	Done (1)	Address	Data	End of data (read only)

Figure 4-3. The read and write queues.

request controller, which has pointers to the read and write queues, selects the next entry in one of the queues to be serviced by memory. With each memory request, the controller sends a tag unique to that request. These tags are needed to place a serviced memory request in the proper queue position when the memory services requests out of order. When the data associated with an address in the queue arrives, the indirect bit is checked by special hardware. If the indirect bit is set, the data item is placed in the address field and the indirect bit is reset. The indirect read queue entry is now treated as a direct address and generates a new memory request. If a data item arrives and the indirect bit is not set, the received bit is set and the data item is placed in the data field. The data item may be removed from the queue by the MAP when it is ready to use the data itself or transfer it to the CP as appropriate. When the data item is removed, the done bit is set, indicating that that queue location is ready for re-use.

The addresses in the write queue wait for data from the CP or the MAP. When addresses are placed in the write queue, any entries which have the indirect bit may be serviced immediately. An indirect write address, in effect, forces a read to be performed to obtain the "real" write address. Once the direct address for an indirect write request arrives, it is placed in the address field of the entry and the indirect bit is reset. When the actual data to be written arrives from the CP or MAP, it is placed in the data field and the received bit is set. As soon as the indirect bit is reset and the received bit is set, the memory request controller may make a memory write request for the entry.

The done bit is set when the memory request is serviced, indicating that the data item has been written to the memory and the queue location is ready for reuse.

The addresses in the read and write queues are kept in the order that they were generated so that the CP receives read operands in the expected order and the MAP receives write data in the proper order. If a data item is to written to memory, and then in the near future, is to be read from memory, it is possible for the address to that data item to appear in both queues at the same time. In such a case, the read must not be permitted to occur before the write. Thus, each time a read address is placed on the read queue, the write queue must be checked for an outstanding write to that address in order to prevent the reading of invalid data. Furthermore, if an operand is used repeatedly, such a comparison of read and write addresses could eliminate some memory traffic. This feature, the so-called "multi-access feature" is not used in this SMA organization.

4.2.2. Branching

The previous section described how instructions are brought into the MAP and the CP, how operand addresses are generated, and how operands are fetched from the memory and written back to the memory. In this section, we consider what occurs when a branch instruction is encountered.

As stated in Section 4.2.1, when the instruction fetcher of the MAP reaches the end of a block, it checks whether the instruction is a

branch instruction. If the instruction is a branch, the instruction fetcher suspends further sequential instruction requests. As with any other instruction, the instruction fetcher forwards the branch instruction to the instruction preprocessor where it is handled like any other instruction. Thus, the branch instruction makes its way to the OIB and the address generation unit of the MAP eventually encounters the branch instruction during its normal execution of items from the OIB. If the branch depends on a condition in the CP, a signal must be received from the CP before the instruction fetcher and the address generator can resume operation. This signal indicates the success or failure of the branch. If the branch, however, depends on the value of an index in the index stack, the branch is resolved immediately since the MAP requires no interaction with the CP. Thus, if the result of an index-dependent branch requires executing a new block of instructions, the instruction fetcher can begin fetching the instructions of the new block while the CP is performing calculations on the data for a previous block. The address generator can even begin making data requests for the new block while the previous block is still executing in the CP. This feature represents a significant improvement over conventional branch resolution and prefetch mechanisms in the higher degree of computation overlap, elimination of most branch wait time, and the reduction of memory accesses.

At any one time, the CP's instruction buffer may contain the CP instructions for more than one instruction block. The OIB in the MAP must, at the same time, be capable of holding the accessing information

and MAP instructions corresponding to the instruction blocks in the CP. The CP's instruction buffer and the MAP's OIB, while they hold information for the same number of blocks, are not necessarily the same size. Depending on the program which is running, sometimes the OIB will be filled to capacity leaving the instruction buffer partially empty and at other times, the instruction buffer will be filled while the OIB will have some unused capacity. Monitoring the amount of information held by both the buffers is the responsibility of the instruction preprocessor since the instruction preprocessor fills the OIB and forwards CP instructions to the CP.

When a branch is resolved, there is a chance that the target block of the branch is already resident in the OIB and the CP's instruction buffer. The address generator checks for this situation by comparing the branch target address against the first address of each block currently found in the OIB. (Recall that the first address was saved in the OIB by the instruction preprocessor when the block was fetched.) If there is a match, the operand specifications and instructions associated with that block do not have to be refetched since they are already in the OIB and in the instruction buffer of the CP from the previous time the block was executed. The address generator can therefore immediately begin generation of data addresses for the new block. If, on the other hand, the information for the block is not in the OIB, the instruction fetcher is signaled by the address generator that a new block must be fetched. In such a case, the address generator must wait until instructions and operand specifications for the new block begin to be loaded into the OIB before it can begin generating operand addresses.

Given that the instruction preprocessor has filled the OIB and the CP instruction buffer with information for the same blocks, when the address generator of the MAP determines whether the OIB contains a particular block, the address generator can assume that the same holds true for CP 's instruction buffer. Since the CP contains no information about branch targets, the CP requires help from the MAP to determine whether the CP contains a block which is the target of a branch. Therefore, when a branch is resolved, the MAP must signal the CP which one of the following three branch options the CP should take: (1) continue repeated execution of the currently executing block, (2) execute some other block found in the CP's instruction buffer, or (3) expect to receive a new instruction block from the MAP. If the second option is followed, the MAP must additionally specify which of the CP's resident blocks the CP should execute. The MAP accomplishes this signaling by using a one-bit end-of-data signal associated with the data bus. For each data item the CP receives, the CP checks whether the end-of-data signal is set.

Normally, the CP is in a loop mode type of operation and expects a stream of data from the MAP. That is, if the end of the currently executing block is not a branch which depends on data in the CP, the execution unit of the CP will re-execute the currently executing block as long as the CP receives data from the MAP and the end-of-data signal is not set. Thus, the first branch option is the default mode of operation and is signaled by the absence of an end-of-data signal. This

mode of operation is especially well suited for executing an instruction block which operates on an array. Since the number of times such a loop is executed depends on the size of the array and the value of indices in the IS, branches will occur in the MAP based on the value in the IS. The only effect of these branches have on the CP is that data continues to be supplied to the CP.

If the MAP determines that branch options 2 or 3 are to be followed, the end-of-data signal is sent to the CP after the last data item associated with the currently executing block. An end-of-data signal only informs the CP that a new instruction block is to be executed; the CP must also be informed if the new block is in the CP's instruction buffer or if the CP should expect to receive a new block from the MAP. This information is conveyed by sending data over the data bus with the end-of-data signal. The value of this data determines whether option 2 or option 3 is followed. One data value is reserved to indicate that the CP should expect a new block from the MAP, i.e. follow option 3. Any other data value is a pointer to a block in the CP's instruction buffer, i.e. option 2.

The MAP signals the CP by placing the end-of-data signal and the pointer on the read queue with the receive bit set. The read queue thus differs from the write queue since an extra bit is needed for the end-of-data signal. Every time the flow of program execution is switched to a different instruction block, the switch is accomplished through use of the end-of-data signal; thus, program execution in the CP is controlled through the data stream. Using an end-of-data signal in

the data stream is a significant departure from the way program flow is controlled in conventional architectures.

When the CP performs the test for a data dependent branch, such as searching a list for a key, the MAP ceases prefetching data until the branch is resolved. This wait time incurred by the MAP is undesirable when such a test is executed frequently. Instead, the wait time could be used to prefetch the data for one of the branch targets. If the MAP prefetches data for a branch target, but the target is not taken, the CP must be signaled to purge the prefetched data from its buffer. The end-of-data signal provides a convenient way of disposing of data wrongly prefetched by the MAP. A reserved data value, sent with the end-of-data signal, can signal the CP to purge all buffered and incoming data until the next end-of-data signal. Such a reserved value would be written by the MAP into its read buffer whenever the MAP continued prefetching data and received a wrong-way branch indication from the CP. This signaling capability would be allowed only by special CP branch instructions whose opcodes would instruct the CP to purge data upon a wrong-way branch. All data in the CP read buffer is then purged up to the "purge" end-of-data signal and all following data is purged up to the next end-of-data signal. Prefetching instructions in such a case has no purge problem since the next end-of-data signal after the "purge" end-of-data signal indicates which instruction block to execute next.

The methods for communication between the MAP and the CP are designed to limit the number of interruptions in execution due to branching. Branches which depend on data in the MAP may occur many

times without interrupting the operation of the CP; therefore, once the CP has a block of instructions in its buffer, the MAP can keep a stream of data flowing into the CP.

4.2.3. The Computation Processor

We are primarily concerned with the CP as it interacts with the MAP unit. That is, while the CP contains the ALU and performs the "useful" computations for a program, designing an ALU is not our goal. We are more concerned with the way in which the CP deals with the information it receives from the MAP.

Basically, the CP receives a stream of CP instructions and data from the MAP and produces data. As pointed out in the preceding section, the CP's instruction buffer and the OIB must store at all times information for the same number of instruction blocks. Since instruction blocks do not normally have the same length in both buffers, this requirement may cause empty space in one buffer or the other at some times. The instruction buffer is a fixed-length FIFO stack similar to the MAP's OIB. As the CP receives instructions from the MAP, it stores them in its instruction buffer, checks for the one-bit, end-of-block marking, and also makes a beginning-of-block mark. The CP executes the incoming instructions as soon as their associated data arrive. As stated in Section 4.2.2, the last data item for a block of instructions is followed by an end-of-data signal and a value which either points to another block in the instruction buffer or indicates that a new block is to be received from the MAP. With such a design, for the bodies of single block inner loops which fit in the CP buffer,

loop mode occurs automatically. If there are a number of short blocks in a program which are repeatedly used, the CP instruction buffer can hold these. Switching between execution of these blocks is accomplished with the end-of-data signal and the value of the pointer sent with the signal. If the CP has processed the last instruction of a block but has not yet used any data with the end-of-data signal, it simply re-starts execution from the beginning of the block. The CP then continues to execute the buffered instructions in the block repetitively until the data with the end-of-data signal is received.

The CP only generates two types of items for the MAP. The first is data which is to be written back to the memory. Since the MAP generates write addresses for the data and saves them in order in a queue, and the write data is generated in the same order by the CP, there is no problem in sending only data without identifiers back to the MAP. The second item which the CP generates for the MAP is a branch resolution signal for a branch which is dependent on data in the CP. Since the test for such a branch is performed in the CP, the MAP may have to wait for a signal from the CP to indicate the success or failure of the branch. Although the CP sends this signal, it does not actually know the target address of the branch. It is thus not really executing a branch instruction; it is merely reporting the outcome of a test.

The CP also includes a number of data registers. While the SMA reduces the number of registers needed, it does not eliminate their usefulness in the CP. The SMA could be designed with no registers in the CP; however, temporary data values and repeatedly used scalars would

frequently need to be written to and reread from the memory. A CP instruction opcode explicitly indicates whether registers are to be used. The register names used by an instruction are sent to the CP from the MAP as immediate operands with the instruction.

4.2.4. Subroutine Calls

The preceding sections presented the features of the SMA machines which deal with the execution of the main body of a program. These features can also handle most types of subroutine calls, since a jump to a subroutine is equivalent to a branch to a block. The use of recursive calls creates special problems which can be alleviated by providing special architectural features.

Consider some of the common ways in which subroutines are called in programs. First, assume that all of the subroutines which are to be used with a program are compiled with the main program and that none of the subroutines are recursive. Furthermore, assume that each subroutine is called from only one place. In such a case, the code for the subroutine could be inserted in line and no special jumps would need to be made to execute the subroutine. Such a case is, in essence, implemented as a macro expansion.

Now, consider the previous case extended to permit a subroutine to be called from more than one place. Since copies of the subroutine could exist in several places, one could plant code inline with each call at the price of wasted space. On the other hand, one could use less space by compiling a subroutine with the main program, but

including an extra value with the subroutine call which indicates the block from which the subroutine was called. Upon completion, the subroutine could test this value to select which in a series of return jumps should be performed. The selected jump would return control to the proper block. These schemes can each be implemented without an explicit subroutine call or return instruction. Such is the case if the exact sequence of blocks to be executed in a program is known at compile time.

In many cases, however, subroutines are not compiled at the same time as the main program. Simple jumps cannot then be used for subroutine calls or returns. Also, general purpose subroutines may be called from many programs in many different ways; thus at compile time it is not known to where control is to return. These are some of the reasons for the introduction of the control stack for subroutine returns found in many machines. When a subroutine call is made, the place to which control is to return is automatically pushed on a control stack. When a subroutine return instruction is executed, a value is popped from the top of the stack and placed in the program counter. In this way, calls and returns from subroutines are handled very neatly. The control stack may be stored either in the central processor or in the memory. While having the control stack implemented in the memory is expensive in terms of execution time, such an implementation does permit the luxury of a larger stack, thereby reducing control stack overflow problems. To access a control stack stored in memory, some stack access information, such as a top of stack pointer is kept in the processor.

A further reason for implementing the control stack in the main memory becomes apparent when one considers the problems associated with recursive subroutine calls. Recursive subroutines occur in an important class of algorithms, and many high-level programming languages permit recursion. Because of the popularity of recursive algorithms, computer architects have introduced hardware mechanisms for their support. These mechanisms could be duplicated in software; however, this involves substantially more overhead in terms of time than having the hardware mechanisms available. Having the control stack in the memory permits using the stack as a convenient place to store passed parameters, and local variables, as well as return pointers.

The SMA uses a control stack for handling subroutine calls. This is done by providing a stack pointer (SP), frame pointer (FP), and an argument pointer (AP) similar to the VAX system. These pointers are maintained in the MAP and MAP instructions are provided to access the pointers along with push and pop operations for the SP.

4.3. A Sample SMA Program

To clarify the operation of the SMA, consider the following small section of code:

```
V:  for i := 1 to n do
W:    for j := 1 to n do
X:      C[i,j] := 0;
Y:      for k := 1 to n do
Z:        C[i,j] := C[i,j] + A[i,k] * B[k,j];
```

This program segment performs the matrix operation C = AxB, where A, B, and C are each nxn matrices. This program segment demonstrates how the

SMA machine handles several levels of nesting and a repeated inner loop reference of a data structure item. For this program to run on the SMA machine, consider the information which must be resident in the MAP. The index template table, loaded by a MAP instruction, only needs one entry of 1,n,1 since all the setup index instructions in this program can use the same template.

The AIT and APT entries for our sample program are shown in Figure 4-4. As stated in Section 4.1.3, the AIT contains the information needed by the MAP to generate addresses for the three data structures used in this program. There is a base address entry for each data structure and a displacement for each data structure's second dimension. Since all the data structures are only 2 dimensional, the third dimension displacements are zero. The last three values of each entry are the upper bound limits for the corresponding dimension. The matrices in the sample program are each nxn, thus the first and second dimension upper bound entries are both n. Since the matrices are only 2 dimensional, the third dimension upper bound is set to 0.

The information in the APT indicates how indices are to be used to generate operand addresses. To generate APT entries, the compiler must keep track of the indexing level of each program index. In our example, the index i is at level 1, the index j is at level 2, and the index k is at level 3. Furthermore, only three different pairs of indices, (i,j), (i,k), and (k,j), are used in the program. To use these indices, the APT entries are as shown in Figure 4-4.

Access Pattern Table

Entry	1st Dimension		2nd Dimension		3rd Dimension	
	Index Level	Offset	Index Level	Offset	Index Level	Offset
1	1	+0	2	+0	0	+0
2	1	+0	3	+0	0	+0
3	3	+0	2	+0	0	+0

Access Information Table

	Entry		
	1	2	3
Base Address	A–base	B–base	C–base
2nd Dim. Displ.	n	n	n
3rd Dim. Displ.	0	0	0
1st Dim. UB	n	n	n
2nd Dim. UB	n	n	n
3rd Dim. UB	0	0	0

Figure 4-4. The access pattern table (APT) and the access information table (AIT).

The AIT is loaded by the MAP "load AIT" instruction. The AIT entries are loaded one at a time and one load AIT instruction must be executed for each entry. The load AIT instruction has two operands and operates in two modes. For both modes, the first operand is the number of the AIT entry which is to be changed. The second operand is the base address for a data area which contains the AIT information. For each entry, a load AIT instruction will generate more than one memory access since an AIT entry is more than one word long. Since it is assumed that an AIT entry is composed of a fixed number of sequential words and the second operand of the instruction is the base address for the entry information, the MAP can generate the proper number of sequential addresses to fetch an AIT entry from the memory. The two modes of operation treat the second operand as a direct base address for the AIT entry information or as an indirect address which points to a memory location whose contents are the base address for the AIT entry information. The direct mode is useful for the loading of information about global data structures, while the indirect mode is useful when call by reference of data structures is desired for passing parameters to subroutines. The indirect mode, of course, involves substantial overhead in time, since one indirect reference requires two memory accesses.

The APT is loaded by the MAP "load APT" instruction which is exactly the same as the load AIT instruction, however it loads the APT instead of the AIT. The index template table is loaded in the same way as the AIT and APT, using its own "load TMP" instruction. To load the

APT, the AIT, and the index template table in the sample program, indirect addressing mode is not needed since the information in the tables is local to the currently executing code segment.

Let us assume that the APT and AIT entries are stored contiguously in memory and that each APT entry requires 3 words of space and each AIT entry requires 6 words. If the first entry of the APT is at location 100, entries 2 and 3 are at locations 103 and 106 respectively. Entries 1, 2, and 3 of the AIT are then in locations 109, 115, and 121, respectively. Also, we assume that the template is initially stored in memory location 127.

An assembly language listing for this example is found in Figure 4-5. The listing gives the SMA instructions and their memory locations. To the left of the column labeled "Instruction Location" are labels which correspond to statement labels in the source program. The entries in the "Comments" column indicate whether an instruction is for the CP or the MAP and whether the instruction is the end of a block (EOB). Where operands are required, each operand is represented by a set of numbers in parentheses. The first number is the operand type. Operand type 1 is immediate, while a data structure is of type 2. The second field for immediate operands is the value of the operand. For the data structure operands, the second number is the AIT entry while the third number is the APT entry.

The instructions at locations 1 through 3 load the APT while the instructions at locations 4 through 6 load the AIT. The index template table is loaded with the instruction at location 7. For each of these

Instruction Location		Instructions	Comments
	1	LDAPT (1,1) (1,100)	MAP
	2	LDAPT (1,2) (1,103)	MAP
	3	LDAPT (1,3) (1,106)	MAP
	4	LDAIT (1,1) (1,109)	MAP
	5	LDAIT (1,2) (1,115)	MAP
	6	LDAIT (1,3) (1,121)	MAP
	7	LDTMP (1,127)	MAP
V:	8	SET-UP (1,1)	EOB, MAP
W:	9	SET-UP (1,1)	EOB, MAP
X:	10	CLR (2,3,1)	CP
Y:	11	SET-UP (1,1)	EOB, MAP
Z:	12	MUL3 (2,1,2) (2,2,3) (1,1)	CP
	13	ADD2 (1,1) (2,3,1)	CP
	14	INCR (1,3) (1,12) (1,15)	EOB, MAP
	15	INCR (1,2) (1,10) (1,16)	EOB, MAP
	16	INCR (1,1) (1,9) (1,17)	EOB, MAP
	17	STOP	EOB, MAP

Figure 4-5. Sample SMA program listing.

instructions, immediate operands are used. Thus, the instruction at location 1 loads entry number 1 of the APT with the information found at memory locations 100, 101, and 102. The instructions at locations 8 and 9, respectively, setup the i and j indices on the index stack. Since the setup command references the index template table, its operand is an immediate operand and simply points to an entry in the table. For our program, entry one is always used.

The instruction at location 10 sends a zero to C[i,j]. As indicated by the 2 in the operand specification, this is the first instance a data structure reference is made. This operand specification indicates that a data structure is to be accessed and that entries 3 and 1 of the AIT and the APT are to be used, respectively. Instruction 11 sets up index k, again using the first entry of the index template table. The multiply instruction at location 12 multiplies two operands and places the product in a register. The first two operand specifications indicate that data structures are to be accessed while the final operand specification is an immediate value of one. An immediate operand may be used either as a register tag pointing to the register from which an operand value is to be obtained, or it may be the value of the operand itself. The distinction between these two uses of an immediate operand (register tag or immediate value) is coded in the opcode. In this case, the multiply instruction is coded so that the CP treats the immediate operand as a register tag pointing to register 1. The following add instruction forms the sum of a register and a second operand and places the result in the second operand. The first operand

is an immediate operand specifying register 1, while the second operand specifies the data structure. Instructions 14 through 16 are increment instructions. The increment index instruction takes three operands; the first operand is the index level to be incremented. The increment index instruction not only increments the index, but also checks whether the final index value has been exceeded and performs a branch. If the final value is not exceeded, the second operand is used as the target of the branch. If the final value is exceeded, the third operand is used as the target of the branch.

In the sample program, there are 7 instruction blocks, as seen from the number of EOBs. Since the MAP's OIB and the CP instruction buffer hold information for the same blocks, when the OIB contains the MAP instructions and operand specifications for instructions 9-16, the CP contains instructions 10, 12, and 13.

When executing a block which does not require input data, the CP does not go into loop mode. The CLR instruction, at location 10, is an example of such a block. In such a case, the CP always waits for an end-of-data signal from the MAP to indicate whether to repeat execution of the block or to execute a new block.

The MAP only sends an end-of-data signal to the CP if there is at least one CP instruction in the destination block. Since the MAP contains all operand specifications and since every CP instruction has at least one operand specification, the MAP can determine the existence of a CP instruction in a destination block by checking its OIB. Whenever the MAP encounters an end-of-block, the MAP constructs an

end-of-data signal. Unless there is at least 1 CP instruction in the destination block, the signal is not sent to the CP. If a CP instruction is not found, the MAP instructions of the destination block are executed until the next end-of-block is encountered. A new end-of-data signal then replace the previously constructed end-of-data signal. Thus, several end-of-blocks may be found before an end-of-data signal is actually sent to the CP.

When our sample program is executed, instructions 1 through 9 are executed with no CP-MAP interactions. During this time, the CP is idle, waiting for an end-of-data signal. That signal is setup when instruction 9 is executed since instruction 9 is the end of a block. After 9 is executed, the MAP determines that the block containing instructions 10 and 11 contains a CP instruction. Thus, the end-of-data signal is sent to the CP indicating that the block beginning at instruction 10 is to be performed. Notice that although instruction 8 is the end of a block, no end-of-data signal is sent to the CP, since its target block (i.e. instruction 9) does not contain a CP instruction.

After the CP executes instruction 10, it will try to enter loop mode, i.e. try to re-execute the block containing instruction 10. However, since this a block does not require input data, the CP enters in an idle state, waiting for an end-of-data signal. This end-of-data signal is generated in the MAP at instruction 11 and instructs the CP to execute instructions 12 and 13. Since the first instruction of the block (instruction 12) takes an input operand, the CP will execute these two instruction repeatedly until it encounters an end-of-data signal.

On the n^{th} iteration, instruction 14 constructs an end-of-data signal. However, the target of the branch is now the block containing instruction 15. Since that block does not contain a CP instruction, this end-of-data signal is not sent to the CP. Instruction 15 also constructs an end-of-data signal. Unlike instruction 14, the first n-1 times instruction 15 is executed, an end-of-data signal is sent to the CP, instructing it to execute the block containing instruction 10. Execution from instruction 10 proceeds as previously described.

On its n^{th} execution, instruction 15 does not send an end-of-data signal since its successor block only contains instruction 16, a MAP instruction. The end-of-data signal constructed by instruction 16 is not sent to the CP since both of the instruction 16's successors are blocks which contain a no CP instructions. The first n-1 times instruction 16 is executed control is transferred to instruction 9. From instruction 9, execution proceeds as described above. On the n^{th} iteration of instruction 16, control is transferred to instruction 17, and the MAP halts. At this point, the CP is in an idle state, awaiting an end-of-data signal to indicate which block it should execute next. The next end-of-data signal will instruct the CP to begin a new code segment.

From Figure 4-5, one can see that most of the instructions are for the MAP and not quite half of the program is the setup of the MAP tables. While the use of these tables requires some extra initial time, they reduce the number of instructions in the inner loop of the program. As the matrix size increases, the proportion of execution time spent in

the inner-most loop also increases. Thus, the time to perform instructions 1 through 7 tends to become insignificant.

Instructions 12 through 14 correspond to the inner-loop of the program. These represent 3 instructions, 2 CP instructions and 1 MAP instruction, and 4 operand accesses to memory. If one unit of time is required for the execution of each instruction, for the generation of each operand addresses, and if for each memory access and the CP and the MAP achieve overlapped execution, then the MAP utilization would be 1 (4 operands and instruction 14) the CP utilization would be 2/5, and the memory bus utilization would be 4/5. This situation seems to indicate a poor utilization of the CP. One must keep in mind, however, that this poor utilization only exists if the CP, the MAP, and the memory operate under the time constraints given above. Under such constraints the MAP is the bottleneck of the system for this program. Due to the repetitive, simple, and regular nature of the operations performed by the MAP, we believe that the MAP can be designed, using pipelining techniques, so that the time taken to generate an operand address or execute a MAP instruction is significantly less than the time needed to execute a CP instruction and that a proper balance for executing real programs can be achieved.

CHAPTER 5

SMA EVALUATION

This chapter evaluates the effectiveness of the SMA machine in reducing addressing overhead by comparing an SMA machine's performance to that of a VAX with respect to three algorithms. Two of the algorithms are written in FORTRAN while the third is written in PASCAL. From the high-level program source, each program is compiled into assembly language for a VAX running the UNIX operating system and for a hypothetical SMA machine. To compile a FORTRAN or PASCAL program into SMA assembly language, the VAX assembly listing is modified only with respect to the way data referencing occurs. That is, when a matrix is being accessed, SMA instructions are added to setup the indices for the matrix and to increment these indices. These SMA instructions, however, eliminate the need for some of the variables used and calculations performed by the VAX. Care is taken not to give either machine any special advantages. Thus, the code produced for the SMA by this transformation of VAX machine code is not hand optimized to any extent.

The algorithms used for comparison are a Gaussian elimination (GAUSS), an eigenvalue-finding algorithm (EIGEN), and a quicksort algorithm (QSORT). GAUSS is a FORTRAN program from IBM's SSPS package of subroutines [SSPP68], while the eigenvalue-finding program is from the Eispack subroutine package [Smit74]. Specifically, the GAUSS program is the SIMQM routine. For the EIGEN program, the kernel routine, HQR was selected. QSORT uses a recursive algorithm taken from Horowitz and Sahni [Horo76].

For each of the programs, the instruction blocks are identified from the high-level source. Figure 5-1 is a diagram of the control flow for GAUSS in terms of instruction blocks. In the case of GAUSS, only two of the branches are probabilistic in the sense that they are truly data dependent. Each of the other branches in the program are determined by the value of an index. These and the unconditional branches are handled very well by the MAP of the SMA machine.

The control flow for EIGEN is more complicated than that for GAUSS and involves 61 instruction blocks. Although a diagram of EIGEN's control flow is not shown, it has 7 inner loops and contains 24 probabilistic branches. In any case, the control flow for GAUSS, as well as EIGEN, is identical on both the VAX and the SMA machine.

As seen from Figure 5-2, this is not the case for QSORT. The SMA version of QSORT has more blocks than the VAX version because indices and access tables must be setup before the loop of blocks 4,4a and 5,5a can be executed. Unlike the GAUSS program, most of the branches are data dependent. Blocks 4 and 5 of the VAX version, which perform similar functions, are each split into two blocks for the SMA version. The data dependent branches which cause possible loop back in blocks 4 and 5 of the VAX version correspond to the branches at the end of blocks 4 and 5 of the SMA version. Blocks 4a and 5a of the SMA version each consist of a single increment index instruction. This instruction causes a branch back to make the data dependent branch.

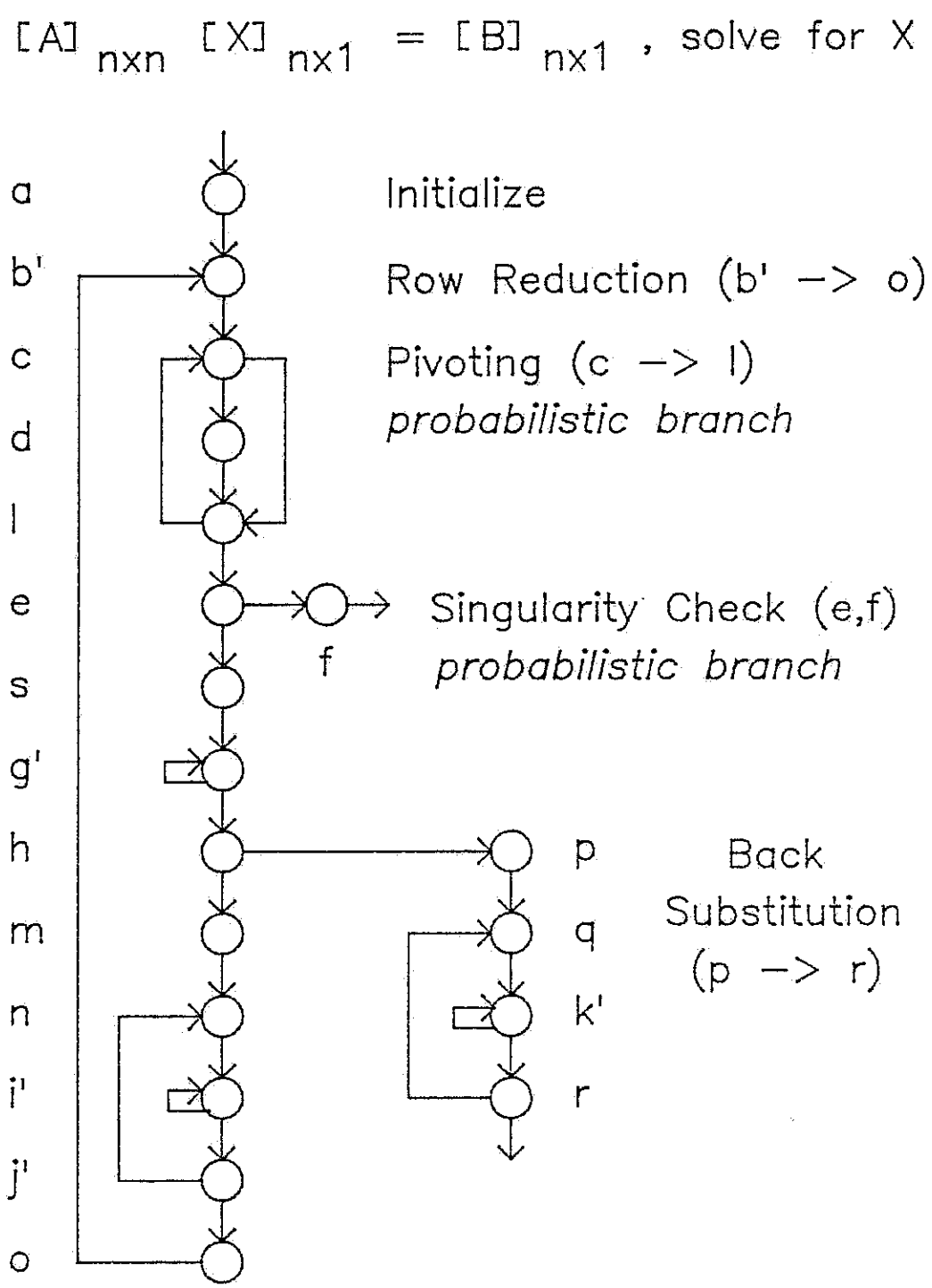


Figure 5-1. Instruction blocks for Gaussian elimination

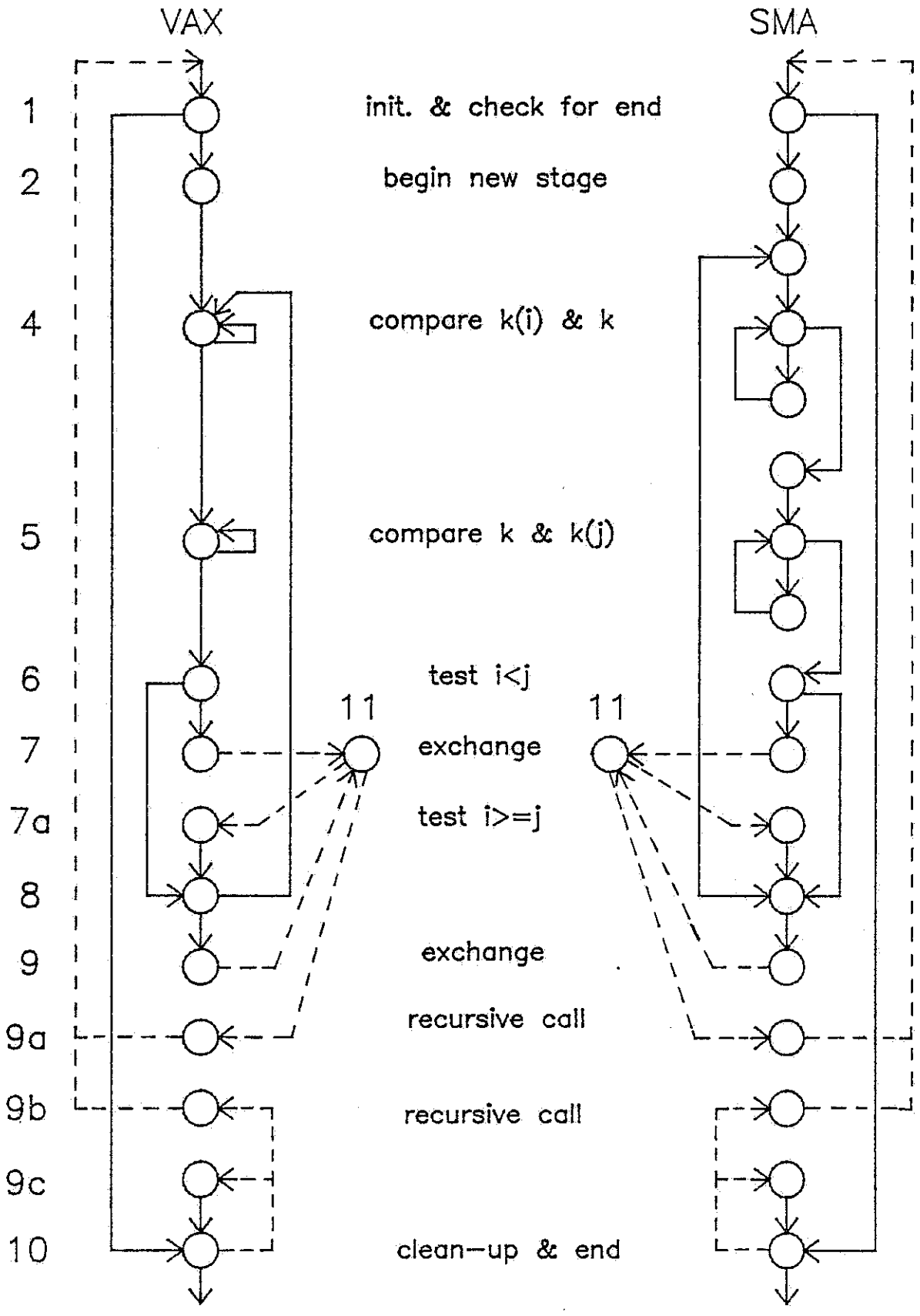


Figure 5-2. Instruction blocks for quicksort.

In the VAX version, the increment of the index is done within the body of blocks 4 and 5.

Another way in which QSORT differs from the other two algorithms is in the use of recursion in the algorithm and the use of a subroutine call. The subroutine calls to block 11 are indicated by the dashed line from blocks 7 and 9. The subroutine returns are represented by the dashed lines from block 11 to blocks 7a and 9a. The two recursive calls of QSORT are indicated by the dashed lines leaving blocks 9a and 9b and entering block 1. The return from a recursive call occurs at block 10. Depending on the origin of the call to the current iteration of this recursive procedure, control returns either to block 9b or to block 9c. If the recursive call occurred from block 9a control returns to block 9b; if from 9b control returns to 9c. The deterministic nature of selecting successor blocks for block 10 is not explicitly shown in Figure 5-2. The overhead for recursion in both machines is roughly the same.

5.1. Number of Memory References Generated

The results of a static analysis of the programs are shown in Table 5-1. For each program on both machines, the number of access patterns, the number of distinct data structures, and the number of data structure references are the same. As pointed out above, the number of instruction blocks differed only for the QSORT program. The number of distinct data structures found in GAUSS and EIGEN differ from the number shown in Table 2-3 because the source programs used for the evaluation in this chapter are different from the programs analyzed in Chapter 2.

Table 5-1. Statistics from a static analysis of GAUSS, EIGEN, and QSORT.

Number of	GAUSS		EIGEN		QSORT	
	VAX	SMA	VAX	SMA	VAX	SMA
instruction blocks	19	19	61	61	14	18
distinct scalars	16	6	36	23	8	8
distinct data structures	2	2	3	3	1	1
access patterns	11	11	19	19	1	1
instructions	123	50	534	251	68	59
data references	84	40	446	319	61	62
scalar references	62	13	386	251	54	53
data structure references	22	22	60	60	7	7
index references	0	3	0	8	0	2

Except for QSORT, the greatest difference among the programs is the number of instructions and the number of scalars and scalar references. In the SMA version, GAUSS and EIGEN require fewer than half the instructions needed in a VAX version. When counting the SMA instruction MAP instructions are also included in the total number of instructions. That is, a MAP instruction like setup is counted as the total number of instructions for a program. The difference in the number of data references between the VAX and the SMA versions for GAUSS is as dramatic as the difference in the number of instructions. The difference between the number of static data references to memory for the VAX and the SMA versions of EIGEN is not as large; nevertheless, the SMA makes only approximately 75% of the data references of the VAX version. Since the

VAX and the SMA versions of GAUSS and EIGEN each make the same number of data structure references, the difference in data referencing is due to the scalar references. The SMA programs have fewer distinct scalars than the VAX programs; thus, for GAUSS and EIGEN, the VAX programs not only have more scalars but also need more instructions to operate on these scalars.

The static analysis of QSORT reveals a less pronounced difference between the VAX and SMA versions. While the SMA version uses 9 fewer instructions than the VAX version, the number of data references actually increases by 1. Also, the number of distinct scalars is the same for both versions.

The differences found in the static analysis translate directly into substantial differences in the dynamic count of the number of memory references for each program. To obtain this dynamic count for GAUSS and EIGEN, the number of memory references generated by each block is calculated as a function of n, the matrix size. For data dependent branches, successors are chosen to produce a path with the largest number of instructions and data references. Therefore, this a worst case dynamic memory reference analysis. In this analysis, it is also desirable to see what effects loop mode has on the number of data references. Thus for each machines there are two cases: one with loop mode and one without loop mode. Instead of giving the VAX and the SMA machine the same size loop buffer and comparing them with respect to that buffer size, it was deemed fairer to compare the two machines given that they had a loop buffer large enough to hold the same number of

inner loop blocks, regardless of the difference in size between these blocks. Thus, in the case of the GAUSS program, blocks (n, i', j'), blocks (q, k', r), and blocks (c, d, 1) were considered inner loops for loop mode execution. To hold each set of these blocks in a loop buffer, the VAX would need to provide a buffer of 24 instructions, while the SMA would need a buffer of only 8 instructions.

For QSORT, a dynamic analysis is not as straightforward as for GAUSS and EIGEN, since (1) the order of elements in the list to be sorted critically effects all calculations and (2) the routine is written recursively. Fortunately, Knuth [Knut73] gives an analysis of the number of times sections of a QSORT algorithm are executed as a function of n, the length of the list to be sorted. The algorithm presented by Knuth differs from the algorithm we use in that a linear insertion sort is not used here when the size of the partitions becomes small. To reflect this, the parameter M of the execution time parameters was set to 1; thus, the results given for QSORT are only valid for lists of length greater than 3.

For loop mode on the VAX version of QSORT, the loop buffer was assumed to be large enough to hold either block 4 or block 5. Such a buffer would require 7 locations for instructions. The corresponding blocks in the SMA version are (4,4a) and (5,5a). The loop buffer to hold one or the other of these sets of blocks would have to be only 3 instructions long.

Table 5-2 shows the equations used for calculating the number of times instructions, scalars, and data structures are referenced when the

Table 5-2. Dynamic counts of instructions, scalars, and data structures as a function of n for GAUSS, EIGEN, and QSORT. (h_n is the sum from i=1 to n of 1/i)

	VAX	SMA
GAUSS		
inst.	$\frac{13}{3}n^3 + 20n^2 + \frac{122}{3}n + 4$	$\frac{13}{3}n^3 + 15n^2 + \frac{107}{3}n + 4$
scl.	$\frac{4}{3}n^3 + 11n^2 + \frac{77}{3}n + 10$	$\frac{5}{2}n^2 + \frac{21}{2}n + 1$
ds.	$\frac{4}{3}n^3 + 7n^2 + \frac{11}{3}n$	$\frac{4}{3}n^3 + 7n^2 + \frac{11}{3}n$
EIGEN		
inst.	$715n^3 - 3014n^2 + 3695n + 1991$	$270n^3 - 1135n^2 + 1597n + 1056$
scl.	$365n^3 - 1510n^2 + 2742n + 1722$	$205n^3 - 96n^2 + 1781n + 1176$
ds.	$115n^3 + 74n^2 + 257n + 90$	$115n^3 + 74n^2 + 257n + 90$
QSORT		
inst.	$\frac{112}{3}(n + 1)(h_n) - \frac{565}{9}n - \frac{1535}{18}$	$\frac{58}{3}(n + 1)(h_n) - \frac{151}{9}n - \frac{869}{18}$
scl.	$\frac{71}{3}(n + 1)(h_n) - \frac{305}{9}n - \frac{979}{18}$	$\frac{26}{3}(n + 1)(h_n) - \frac{28}{9}n - \frac{394}{18}$
ds.	$\frac{16}{3}(n + 1)(h_n) - \frac{106}{9}n - \frac{194}{18}$	$\frac{16}{3}(n + 1)(h_n) - \frac{106}{9}n - \frac{194}{18}$

each program is executed. The equations for the number of instructions are generated by first multiplying the number of instructions in a block by the number of times a block is executed as a function of n. The products formed for each block are then added together to obtain the total number of dynamic instruction references. The equations for scalars and data structures are found in a similar manner. As one might expect, the number of data structure references made by the VAX and the SMA are identical for each program. Comparing the dynamic counts for scalars, one can see that significant differences occur, especially for GAUSS. These differences can be seen more clearly when the total number memory references is plotted versus n.

Figure 5-3 gives a dynamic count of the number of memory references required by the GAUSS program for a VAX and SMA machine with and without loop mode as a function of n. For the GAUSS program, the SMA machine always makes fewer memory references than the VAX, even if the VAX has a loop mode. The number of memory references needed by an SMA machine running the GAUSS program on a 100 x 100 matrix is only 20% of the number of memory references made by the VAX without loop mode.

The results for a dynamic analysis of EIGEN are shown in Figure 5-4 and are similar to the results for GAUSS. For EIGEN, the buffers needed to hold the important loops are substantially larger than in the case of GAUSS. The VAX requires a loop buffer of 50 instructions, while the SMA needs only a loop buffer of 26 instructions. The SMA machine without loop mode generates approximately the same number of memory references as the VAX with loop mode. For the EIGEN program operating on a 100 x

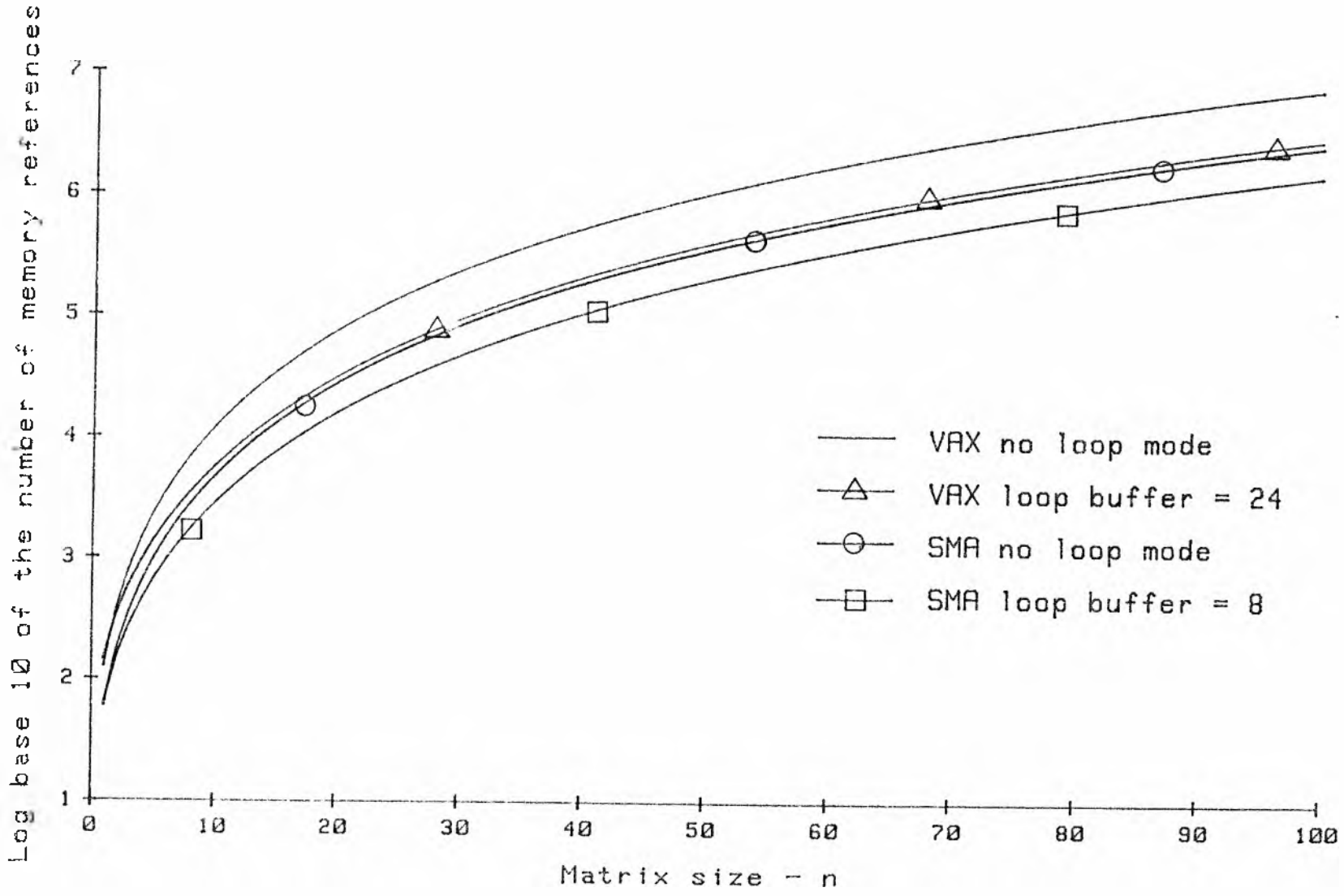


Figure 5-3. Log of the number of memory references for GAUSS for an nxn matrix.

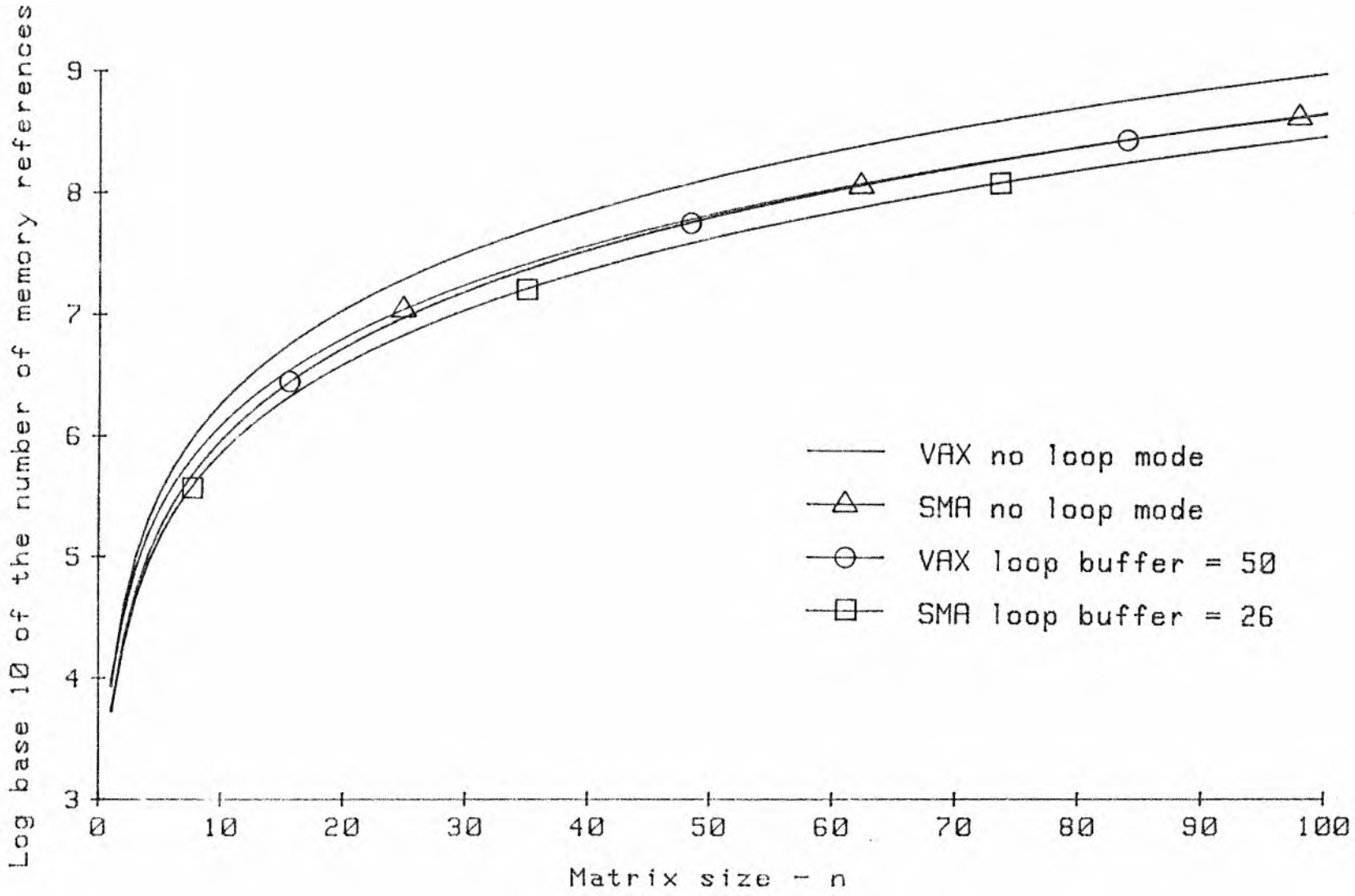


Figure 5-4. Log of the number of memory references for EIGEN for an nxn matrix.

100 matrix, the SMA with and without loop mode makes only 30.5% and 47.2%, respectively, of the references made by the VAX without loop mode.

As shown in Figure 5-5, the results of a dynamic count of the number of references made by QSORT are quite similar to those of EIGEN. This is a little surprising given the similarity of the static counts for VAX and SMA on QSORT. While the SMA QSORT has slightly fewer instructions and a few more instruction blocks than the VAX QSORT, the number of scalar references is almost identical. The reason the SMA version performs better lies in the way its instructions are distributed among the instruction blocks. Since the SMA QSORT has more instruction blocks, some of the instruction blocks must have fewer instructions than the corresponding blocks in the VAX QSORT. The blocks which are executed most frequently, that is the inner loops, are the ones which are most reduced in instruction count. Also the number of memory accesses for operands is reduced in these blocks. So, while an SMA program may generate extra blocks for the initialization of access mechanisms, this overhead is counter-balanced by the reduced amount of time spent in the inner loops.

5.2. An Estimate of Relative Performance

Part of a program's execution time is spent fetching instructions and data from the memory, while the remaining portion of the execution time is spent computing. The amount of time spent in obtaining information from the memory is determined by the effective memory cycle time. The amount of time required to perform computations is also fixed

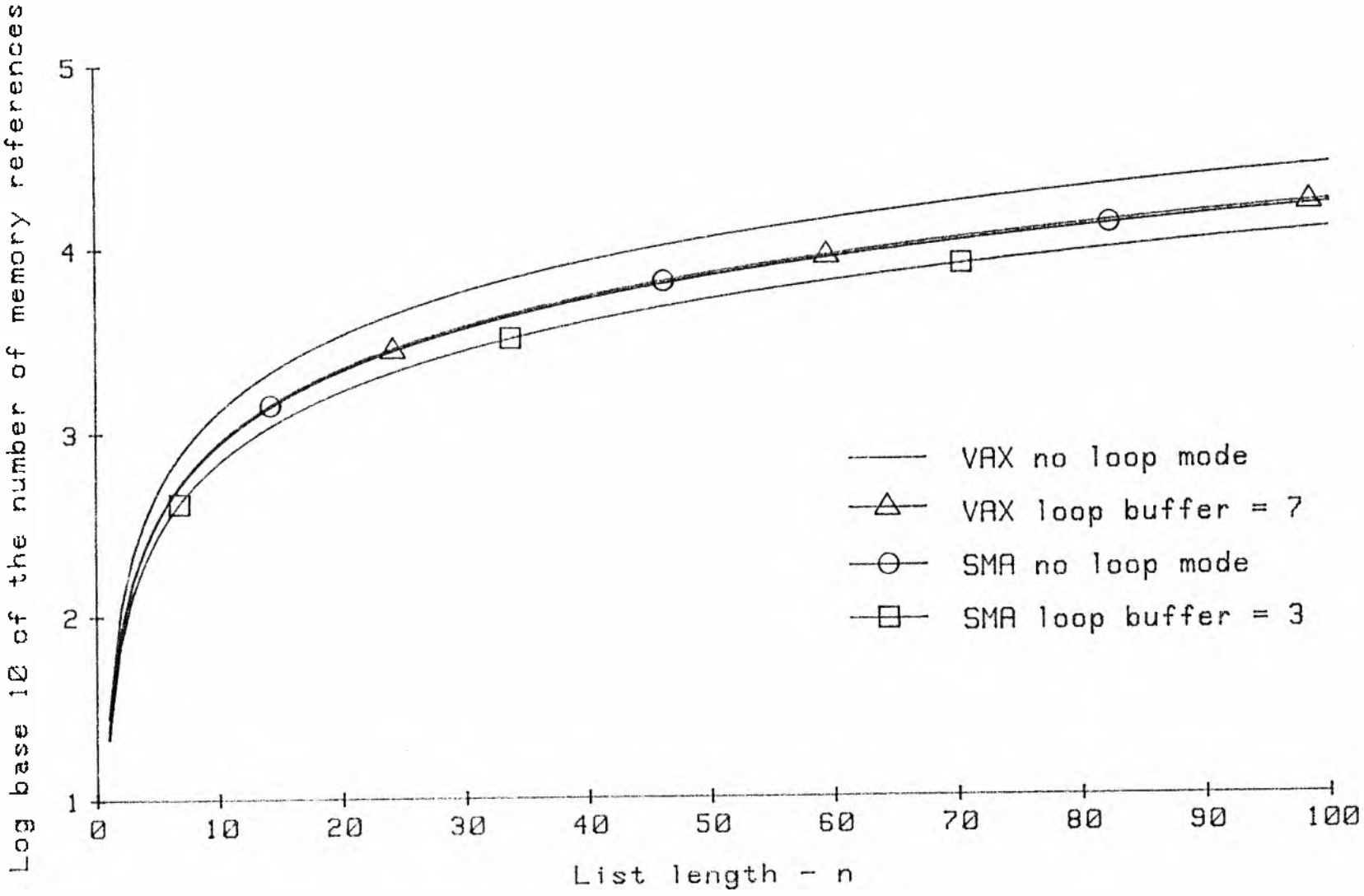


Figure 5-5. Log of the number of memory references for QSORT for a list of length n.

for a particular system. Since both of these are fixed, execution time can only be reduced by overlapping memory accesses with computations. If memory accesses completely overlap computations, the execution time for a program would be the number of memory references multiplied by the memory cycle time. This quantity is the fastest a program could execute.

Due to complex CP instructions, resource conflicts, data dependence, conditional branches, and other synchronization problems, it is not generally possible to overlap all of the computation time with memory referencing activity. Thus, the execution time of a program is the total amount of time spent referencing memory plus some amount of unoverlapped computation time. We call this unoverlapped computation time the computational overhead. A portion of this computational overhead can be allocated to each memory reference, permitting the execution time of a program to be expressed as:

$$T = M\,(1/v + c)$$

where M is the number of memory references, c is computational overhead, and the term 1/v is the amount of time needed per memory access. Thus, the total execution time of a program is the number of memory references multiplied by the sum of the amount of time needed per reference and the amount of computational overhead per memory reference. The variable v is the memory bandwidth and is included as a parameter so that comparisons can be made between machines whose memory speeds differ. A larger v represents a faster memory and, therefore, a reduced memory

access time. If the memory is interleaved, v takes the interleaving factor into account. As the degree of interleaving increases, so does the value of v. The same algorithm executed on different machines will yield a different execution time because the term M will vary from machine to machine, as will the term c. To compare two machines that have the same memory bandwidth, v is set to 1 and the terms c and M must be given.

The computational overhead, c, is difficult to measure. It varies from one program to another and also from one machine to another. This difficulty occurs because c averages together such machine dependent functions as the degree of overlap, the speed of functional units, the latency in execution pipelines, and the dependencies in the programs. Thus, different models of even the same machine will have different values of c. The value of c is also a function of the memory bandwidth v. An increased memory bandwidth can be equated to a faster memory or reduced effective memory access time. As memory access time decreases, less computation time can be overlapped with memory accesses, causing c to increase. The degree to which c is affected by v depends on the particular machine-memory pair. Even if c could be accurately measured for a particular machine and memory, any comparisons using c would only be valid with respect to that machine coupled with that memory. In our case, we would like to compare the performance of an SMA machine with that of a conventional machine. Since an SMA or conventional machine can be designed many different ways and since each design can have a different value of c, c is treated as a parameter in our comparisons of performance.

The performance of a machine is given by the inverse of the execution time. For comparison, we decided to look at the performance of conventional machines and an SMA machine for c ranging from 0 to 2 and for v taking on values of 1, 2, 4, and 8. The computational overhead is in units of memory cycle times per memory reference, as is the term 1/v. The factor M is given by an analysis of the program to be run. For comparison, we use the number of memory accesses generated by the GAUSS and EIGEN programs run on a 100x100 matrix and the QSORT program run on a list of 100 elements. To aid in comparing one machine with another, performance is normalized to the performance of a conventional machine with no computational overhead (c=0) and a memory bandwidth of one (v=1).

The results of calculating this normalized performance for GAUSS are shown in Figure 5-6. Machines with and without loop mode are treated separately because the presence of loop mode affects the number of memory accesses required by the program. With a loop mode, those blocks which can be stored in the instruction buffer and which loop upon themselves only generate instruction references to load them into the buffer prior to the first loop iteration. Once loop mode is established, memory requests for the instructions of the loop are not needed.

Each vertical line of the graph represents the relative performance of a machine with a particular memory bandwidth and with the computational overhead ranging from 0 to 2. Therefore, a conventional

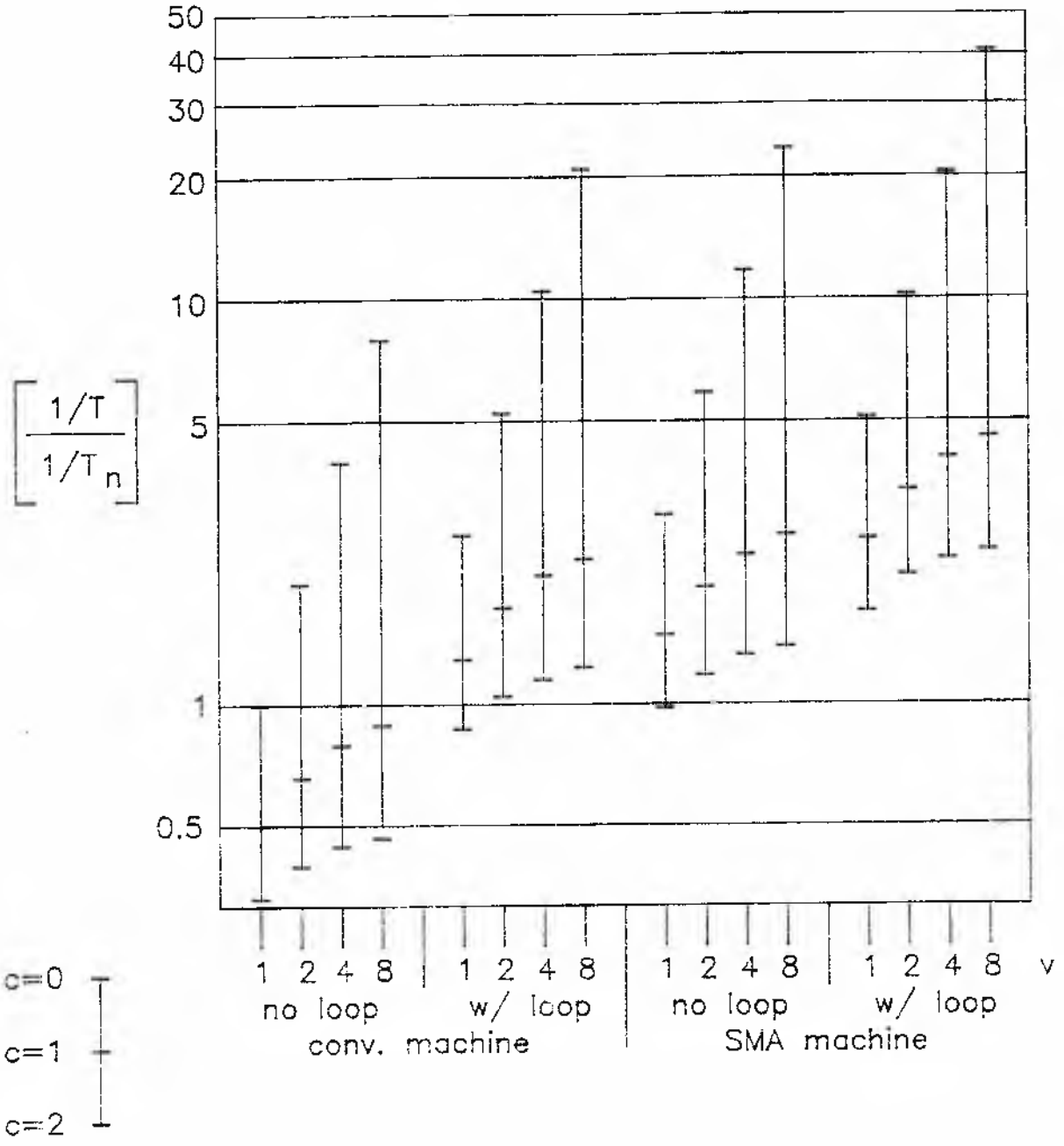


Figure 5-6. Normalized performance for GAUSS. (v = memory bandwidth, c = computational overhead, T = run time)

machine with no loop mode, v=1, and c=1 performs half as well as the same machine with c=0. In other words, the v=1 machine with c=1 would require approximately twice as much time to run a Gaussian elimination program on a 100 x 100 matrix as the machine with c=0. At the other extreme, a c=0 SMA machine with loop mode and a memory bandwidth of 8 would perform approximately 42 times better than the base machine: conventional, no loop, v=1, and c=0.

The normalized performance for EIGEN and QSORT are shown in Figures 5-7 and 5-8, respectively. When one compares the base machine to an SMA machine running EIGEN and QSORT, the performance improves greatly but not as dramatically as for Gaussian elimination. From the figures, one can see that for a given memory bandwidth, a conventional machine with loop mode and an SMA machine without loop mode perform almost equally as well. Furthermore, performance is sensitive to changes in computational overhead, especially when c varies from 0 to 1. Different machines should not simply be compared with the same value for c.

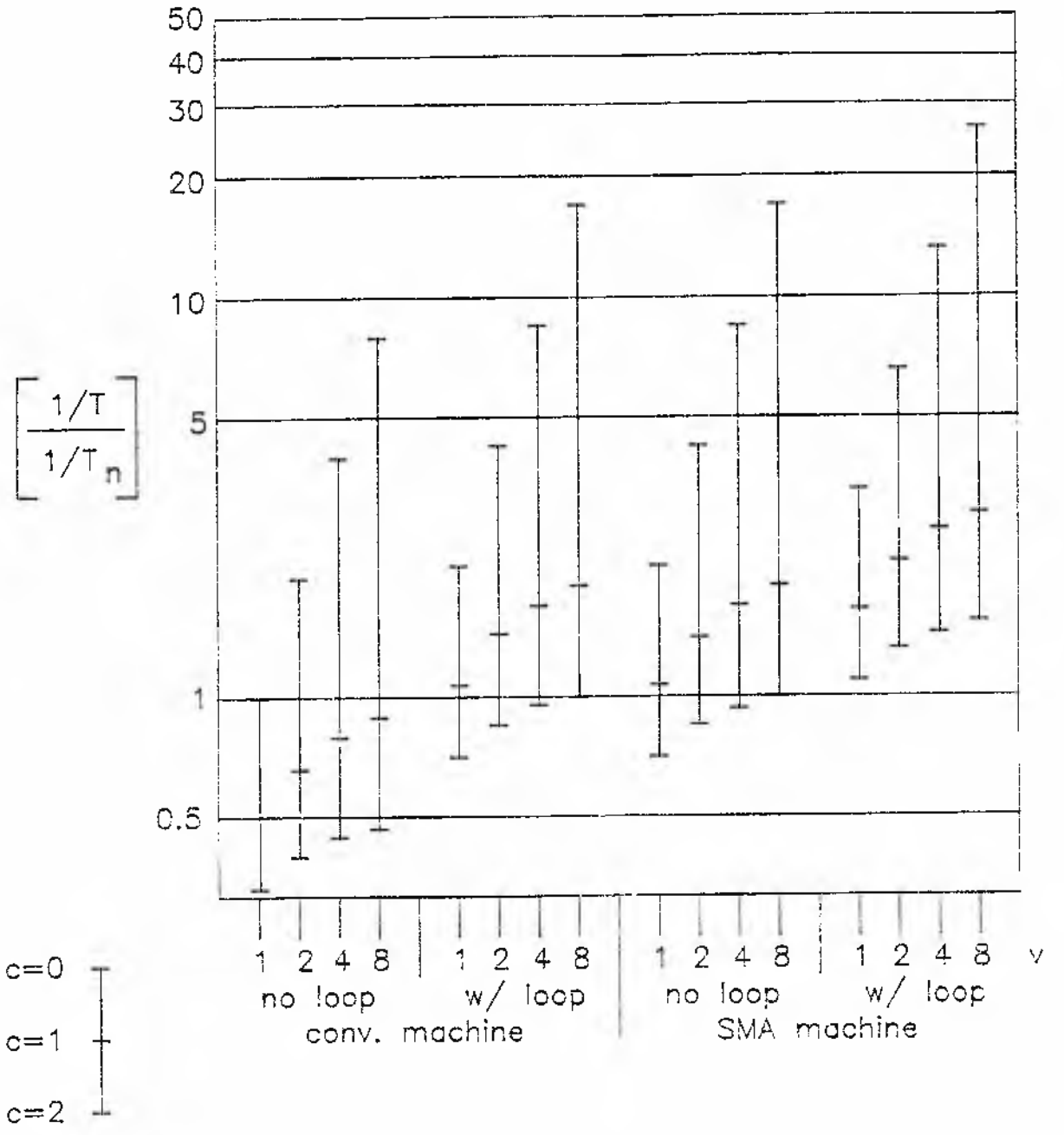


Figure 5-7. Normalized performance for EIGEN. (v = memory bandwidth, c = computational overhead, T = run time)

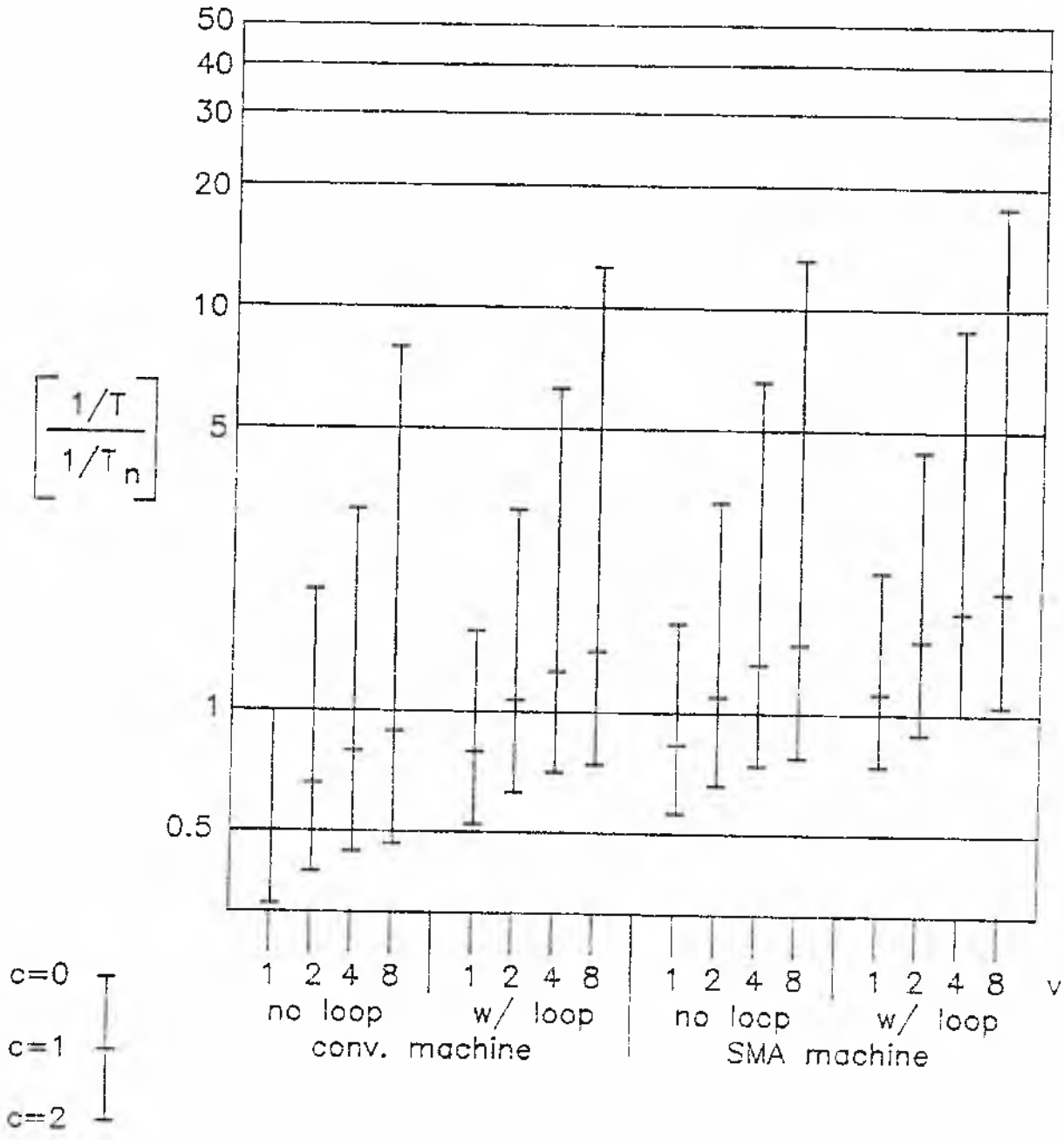


Figure 5-3. Normalized performance for QSORT. (v = memory bandwidth, c = computational overhead, T = run time)

CHAPTER 6

CONCLUSIONS

6.1. Summary of Results

Due to the von Neumann bottleneck, a feature inherent in conventional machine design, inefficiencies exist in the way address generation is performed in most conventional machines. The studies cited in Chapter 2 confirm the existence of this bottleneck. The research presented here has studied the addressing process to discover where the addressing inefficiencies lie and how they can be reduced.

An extensive analysis of some program address traces was performed to quantify the types of phenomena which occur in the address stream. Using this analysis, we can automatically deduce a program's data and control structure from a reference trace. No other information or intuition is required. These structures correspond directly to structures which can be found in the high-level language versions of the programs. The types of control structures found in the traces indicate the types of features which should be included in a machine designed to generate memory references efficiently.

The proposed Structured Memory Access (SMA) machine contains features to reduce substantially the addressing overhead found in the execution of a program. The SMA machine is divided into two different types of processors: a computation processor (CP) and a memory access processor (MAP). As their names imply, the CP is responsible for the

useful computations of the system, while the MAP generates all the memory references for a program. Thus, the MAP performs all transactions with the memory and passes instruction opcodes and data to the CP. The SMA machine reduces addressing overhead by providing special access mechanisms in the MAP to generate references efficiently for blocks of instructions and several data types. The storing of bounds information permits bounds checking to occur automatically in hardware when data structures are accessed. Because of the system's organization, the CP and MAP can operate relatively independently of one another. In particular, prefetching of instructions and data is an inherent feature of the SMA machine.

The operation of the MAP and its interactions with the CP were discussed in some detail as were the types of access mechanisms which reside in the MAP. Some attempt was made to keep the discussion at such a level so as not to be distracted by implementation details which have no fundamental effect on the SMA machine's architecture. The machine's ability to reduce addressing overhead was then evaluated. A comparison was made between a hypothetical SMA machine and a VAX-like machine with respect to the number of memory references generated by a set of programs. Depending on the program, the SMA machine reduced the number of memory references to between 1/5 and 2/5 of those required by a conventional VAX.

The performance of the SMA machine was then evaluated. A machine's performance was parameterized by the memory bandwidth and the computational overhead. It was found that performance is very sensitive

to these parameters; however, an SMA machine performs significantly better than a conventional machine with the same parameters.

6.2. Suggestions for Future Research

This research primarily considered the accessing patterns found in the accessing of arrays. While arrays represent a large and important class of data structures, there are other types of data structures which are used regularly by programmers. Two frequently used data structures are linked lists and binary trees. The addressing overhead involved in accessing these data structures has yet to be investigated.

The SMA implementation which is discussed in Chapter 4 has the instruction fetcher checking every instruction for an end-of-block. Thus, a memory request for the next instruction is not made until the current instruction has been received by the instruction fetcher. If the MAP could be given the starting address and the block length of successor blocks, accessing of instructions could made block-oriented instead of instruction-oriented.

In the performance evaluation which was presented, performance was parameterized by the computational overhead and memory bandwidth. Since performance is very sensitive to computational overhead, detailed simulations should be performed to quantify the computational overhead of a system, as other system parameters vary.

The MAP of an SMA machine is required to perform all address generation. In order to distribute this work, investigations should be made of the types of access mechanisms which could be stored with a data

structure. These mechanisms could then be incorporated within the memory where a particular type of data structure is stored. Moving MAP functions into the memory could lead to intelligent memories and the partitioning of the memory into several specialized memory units. Each specialized memory unit would be an expert at referencing a particular type of data structure. One approach to improving the performance of the MAP effectively would be to use multiple MAP units. Investigations should be made into the feasibility of using not only multiple MAPs but also using multiple CPs.

References

[Amdl64] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks Jr., "Architecture of the IBM System/360," IBM Journal of Research and Development, Vol. 8, No. 2, April 1964, pp. 87-97.

[Ande67] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling.," IBM Journal of Research and Development, Vol. 11, No. 1, January 1967, pp. 8-24.

[Burk46] A. W. Burks, H. H. Goldstine, and J. von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," U.S. Army Ordanance Department Report, 1946. Reprinted in Bell and Newell (1971), pp. 92-119.

[Cohl81] E. U. Cohler and J. E. Storer, "Functionally Parallel Architecture for Array Processors," Computer, September, 1981, pp. 28-36.

[Hamm77a] D. W. Hammerstrom and E. S. Davidson, "Information Content of CP Memory Referencing Behavior," Fourth Annual Symposium on Computer Architecture, March 1977, pp. 184-192.

[Hamm77b] D. W. Hammerstrom, "Analysis of Memory Addressing Architecture," Tech. Report R-777, Coordinated Science Lab., Univ. of Illinois, Urbana, IL. July 1977.

[Horo76] E. Horowitz and S. Sahni, The Fundamentals of Data Structures, Computer Science Press, Inc., 1976, pp. 347.

[Kapl73] K. R. Kaplan and R. O. Winder, "Cache-based Computer Systems" Computer, Vol. 6, No. 3, March 1973, pp.30-36.

[Knut73] D. E. Knuth, The Art of Computer Programming, Volume 3:Sorting and Searching, Reading, Mass.: Addison-Wesley, 1973, pp. 114-123.

[Kuck78] D. J. Kuck, The Structure of Computers and Computations, Vol. 1, John Wiley & Sons, 1978.

[Ples81] A. R. Pleszkun, B. R. Rau, and E. S. Davidson, "An Address Prediction Mechanism for Reducing Processor-Memory Address Bandwidth," Proc. 1981 IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management, Nov. 11, 1981, pp. 141-148.

[PDP75] PDP11 Processor Handbook, Digital Equipment Corporation, 1975.

[Rama77] C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," Computing Surveys, Vol. 9, No. 1, March 1977, pp. 61-102.

[Russ78] R. M. Russell, "The CRAY-1 Computer System," Communications of the ACM, Vol. 21, No. 1, January 1978, pp. 63-72.

[Smit74] B. J. Smith, J. M. Boyle, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, Lecture Notes in Computer Science, Volume 6: Matrix Eigensystem Routines - EISPACK Guide, Springer-Verlag, 1974.

[Smit78] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," Computer, December 1978, pp. 7-21.

[Smit82] J. E. Smith, "Decoupled Access/Execute Computer Architectures," Ninth Annual Symposium on Computer Architecture, April 1982, pp. 112-119.

[SSPP68] 1130 Scientific Subroutine Package Programmer's Manual, International Business Machines Corp., 1968, p. 115.

[Ston80] H. S. Stone, Introduction to Computer Architecture, Science Research Associates, Inc., 1980.

[VAX80] VAX11/780 Architecture Manual, Digital Equipment Corporation, 1980.

[Wats72] W. J. Wastson, "The TI ASC - A Highly Modular and Flexible Super Computer Architecture," Proc. AFIPS Fall Joint Computer Conference, 1972, pp. 221-228.

VITA

Andrew Richard Pleszkun was born in Chicago, Illinois on December 7, 1955. In 1977, he received his B.S. degree in Electrical Engineering from the Illinois Institute of Technology where he was a member of Tau Beta Pi. He obtained his M.S. degree in Electrical Engineering from the University of Illinois in 1979. At the University of Illinois he was employed as a research assistant with the Energy Research Group from 1977 to 1979, a teaching assistant with the Department of Electrical Engineering from 1979 to 1980, and a research assistant with the Computer Systems Group at the Coordinated Science Laboratory from 1980 to 1982.