An Optimal Algorithm for Scheduling Requests on Interleaved Memories for a Pipelined Processor

C. V. RAMAMOORTHY, FELLOW, IEEE, AND BENJAMIN W. WAH, MEMBER, IEEE

Abstract—**In this paper an optimal algorithm for scheduling requests on interleaved memories is presented. With this algorithm the average completion time for servicing a finite set of randomly generated requests is proved to be minimum. Performance of this algorithm for nonrandom requests has not been proved. However, it is compared with alternate algorithms using simulations. A pipelined processor is used as an example for the generation of nonrandom requests to the memories. Nonetheless, the source could have been a vector processor or a multiprocessor system. Two alternative organizations are investigated, one with a common set of fixed size buffers to store conflicting requests and one with individual fixed size buffers for each module. These two organizations are shown to be equivalent as far as the average utilization and waiting cycles are concerned. An intelligent scheduler determines the order of initiation of the memory modules. An alternative design with separate instruction and data modules is investigated. It is found that separation gains very little in performance because of the unequal rates of access to the instruction and the data modules. The basic assumptions for the analysis are that the dependency effects are ignored and the request rate is very high so that any empty buffers can be filled immediately. The degradation in memory utilization due to dependency effects is studied in a separate paper in this issue.**

Index Terms—**Data modules, instruction modules, intelligent buffers, interleaved memories, memory bandwidth, optimal scheduling algorithm, pipelined processor.**

I. INTRODUCTION

THE DESIGN of large primary memory systems is becoming easier as fast, inexpensive, large scale integrated (LSI) memory chips are made available. In the past because of the lack of these memory elements, intelligent designs like buffering and interleaving have been developed to enhance the bandwidths of memories. This problem has been alleviated to some extent in smaller systems where the memories are built directly using fast, LSI chips. However, in large computers, the storage subsystem is still very expensive and can be more than 50 percent of the total hardware system cost [21]. Further, with the development of high-speed processors such as the Cray-1, 1a, and multiprocessor systems such as the C.mmp, there is an increasing speed mismatch between the CPU and the LSI memory elements that can be used to construct a large and fast primary memory. It is therefore necessary to have intelligent architectural designs and efficient access algorithms for supporting retrieval operations in addition to the utilization of faster memory elements.

In this paper we have evaluated two organizations of an interleaved primary memory system. We have designed a scheduling algorithm and have shown its optimality under the assumption of random independent requests. The random request assumption is tenable for multiprocessor systems with a large number of processors sharing a common memory and each processor making requests independent of others. For a lookahead processor such as a pipelined computer which executes directly from the primary memory (e.g., IBM 360/91), we have simulated the effects of the nonrandomness of the accesses using execution traces, and have found that the algorithm performs well. Although our design does not consider the use of a cache which exists in some computer systems, it is still useful in modeling the performance of a system when a cache is present. When a cache is used, the overall effective memory bandwidth is a function of the main memory bandwidth and the cache bandwidth and is governed by the hit ratio of the cache. Our studies pertain to the overall improvement of the memory bandwidth with or without a cache even though the results are drawn for a cacheless system. Our model is useful in demonstrating that the scheduling algorithm is optimal and therefore the cost of the system is minimal when compared with an alternate design of identical bandwidth and technology.

This paper is divided into seven sections. Section II presents the requirements in the design of an interleaved memory system. Section III describes the characteristics of the access sequence of pipelined processors. Section IV presents a summary of the previous work in parallel memories. The deficiencies in these designs are pinpointed. Section V describes two organizations of the memory and shows that the second organization is equivalent to the first organization as far as average behavior is concerned. The optimal scheduling algorithm is also presented. Section VI evaluates the performance of the scheduling algorithm by proving that the algorithm has optimal average behavior for random requests, and by simulating with execution traces from actual programs and randomly generated requests. Last, Section VII compares the two organizations in their abilities to satisfy the requirements and draws conclusions for future investigations in improving memory utilization.

Manuscript received December 15, 1977; revised March 8, 1978, June 19, 1980, and June 10, 1981. This work was supported in part by Ballistic Missile Defense Contract DASG-60-77-C-0138 and the National Science Foundation under Grants MCS72-03734-A02 and MCS77-27293.

C. V. Ramamoorthy is with the Department of Electrical Engineering and Computer Science and the Electronics Research Laboratory, University of California, Berkeley, CA 94720.

B. W. Wah is with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

0018-9340/81/1000-0787$00.75

II. Requirements for the Design of a Primary Memory

The requirements for the design of a primary memory are as follows.

1) Bandwidth: The bandwidth represents the average throughput of the memory system and is given in terms of bits returned per unit time. In a parallel memory system the bandwidth is the sum of the bandwidths of all the modules

$$\text{Bandwidth} = \sum_{\text{module } k} \frac{\begin{bmatrix}\text{word length}\\ \text{of module } k\end{bmatrix} * \begin{bmatrix}\text{average utilization}\\ \text{of module } k\end{bmatrix}}{(\text{cycle time of module } k)}$$

where the average utilization of a module is the average fraction of time the module is busy. For the case of identical modules, the bandwidth can be written as

$$\text{Bandwidth} = \frac{\begin{bmatrix}\text{number of}\\ \text{module}\end{bmatrix} * \begin{bmatrix}\text{word}\\ \text{length}\end{bmatrix} * \begin{bmatrix}\text{average}\\ \text{utilization}\end{bmatrix}}{(\text{memory cycle time})}$$

$$\text{Bandwidth} = \frac{\text{constant} * \begin{bmatrix}\text{average number}\\ \text{of busy modules}\end{bmatrix}}{(\text{memory cycle time})} \quad (1)$$

where the constant in (1) has a unit of bits. The organization of interleaved memories presented here assumes that all the modules are identical, that is, with the same memory cycle time, and that the word length of each module is kept constant. The objective of maximizing the bandwidth is therefore equivalent to maximizing the average utilization of the modules.

2) Response Time: The response time is the delay between the time a request is issued to the primary memory and the time the request is serviced, assuming that the datum resides in the primary memory. This is also called the waiting time of the requests.

3) Size: This is the required memory size or capacity.

4) Cost: This is the maximum allowable cost of the resultant design that satisfies the above requirements.

The design of the memory must satisfy the above requirements. Moreover, the performance of the final system can be evaluated by using these parameters as evaluation criteria.

III. Characteristics of the Access Sequence of a Pipelined Processor

In order to utilize the parallel memories, there must be parallelism in the access sequence so that multiple accesses can be overlapped. In a conventional uniprocessor system where there is a single access stream, an access cannot be issued until its predecessors have completed. There is no parallelism involved and at most one of the parallel memories is used at any time. The memory utilization is increased when multiple accesses can be issued and serviced in parallel. Architectures such as a multiprocessor system sharing a common primary memory or a pipelined computer would have the capability to issue multiple requests simultaneously.

In this paper a pipelined organization in the most general sense, instead of specially structured pipelined computers with multiple arithmetic units (e.g., Cray I), different applications (e.g., vector processing), additional memory support (e.g., cache), and interconnections (e.g., Illiac IV), is assumed. The processor is further assumed to be executing directly from the main memory. The scheduling algorithm developed in this paper is general enough to be applicable to the interleaved memories of all the specially structured pipelined computers. However, the exact performance is not found for each type of machine.

In a pipelined computer the computational process (say an instruction) is segmented into several subprocesses which are executed by dedicated autonomous units (pipeline segments). Successive processes (instructions) can be carried out in an overlapped mode analogous to an industrial assembly line [28]. The pipeline segments are able to generate memory requests independent of the others and, therefore, the dependencies in an access stream are bypassed.

Occasionally, there exists dependencies among the accesses. A *dependency* is a logical relationship between two accesses such that the second cannot be accessed until the first has been accessed. A memory access sequence generated by a pipelined processor has Class *D* dependencies as classified by Chang *et al.* [9]. Anderson *et al.* have identified three main sources of dependencies: 1) register interlock, 2) branching, and 3) interrupts [1]. In this paper we concentrate on analyzing the memory performance without taking into account the effects of dependencies. A subsequent paper in this issue estimates the performance degradation due to dependencies.

In addition to the effects of address dependencies, the order in which instructions and data are requested also affects the memory performance. For a pipelined processor, the request stream is a sequence of instruction-operand fetch pairs. However, not every instruction involves an operand fetch and if the bus is wide enough, two or more instructions can be fetched in one access. A notable characteristic in this access pattern is that instruction fetches are made in a sequence interlaced with operand accesses. The performance of the memory system may be improved by separating the memory modules into two sets, one for instructions and one for data. In Section VI the effects on memory performance due to separation of instruction and data modules are studied.

IV. Previous Work on the Study of Interleaved Memories

One of the early successful implementations of interleaved memories is in the IBM 360/91 [4]. In this computer the storage system is made up of an interleaved set of memory modules and the degree of interleaving equals the number of memory modules. The memory can service a string of sequential requests by starting, or selecting, a storage unit every cycle until all are busy. In effect, the storage cycles are all staggered (Fig. 1). By using a set of buffers called the request stack, conflicting requests which access the same module can

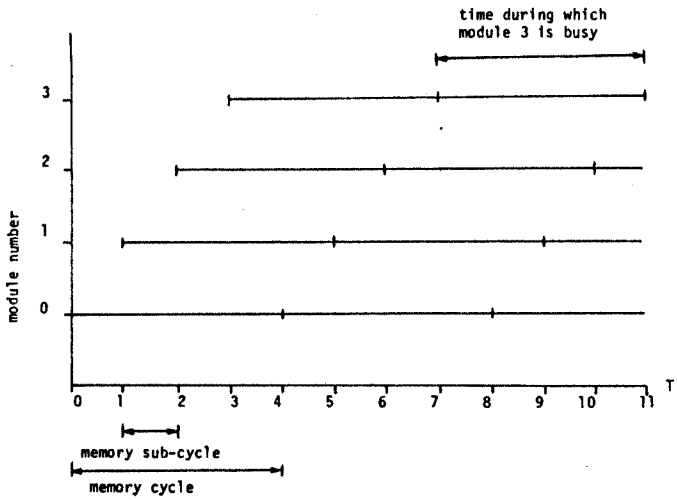


Fig. 1. A Gantt chart to illustrate the operations of the interleaved memories in staggered cycles ($m = 4$).

be resolved by allowing only one of these requests to access the module and storing the rest in the request stack to be issued in later cycles. Simulation results were shown for the average access time and the bandwidth with various degree of interleaving.

The earliest attempt to model the performance of interleaved memories was done by Hellerman [13]. A saturated request queue (a queue in which requests are never exhausted) with random requests is assumed, and no provision is made for the queueing of these requests on busy modules. The request queue is scanned until a repeated request is found. This constitutes a collision. Hellerman's results show that with m memory modules, the average number of requests scanned before a collision is approximately $m^{0.56}$ for m between 1 and 45. This is taken to be an indication of bandwidth. Knuth and Rao [15] show an alternate, exact way to calculate the bandwidth. However, both of these results are pessimistic because they do not allow the queueing of conflicting requests to the same module.

Burnett *et al.* have developed a number of models on parallel memories. In two of these models [6], [7] they assume that the modules operate synchronously (all modules start and end their cycles simultaneously) and a scanner scans a saturated request queue and admits new requests to be serviced until it attempts to assign a request to a busy module. In two other models [10], [8] they further assume that a set of blockage buffers is present so that requests made to a busy module can be stored and issued in later cycles. The scanner continues to scan the request queue until all the modules have been allocated or all the buffers are occupied. In effect, the maximum size of the request queue inspected by the scanner never exceeds $b + m$, where b is the number of buffers and m is the number of memory modules. They have also studied a request model similar to Strecker's model [24] by assuming a probability α for the succeeding request to request the next module in sequence and a probability of $(1 - \alpha)/(m - 1)$ to request any other module. They have developed two algorithms that modified the request pattern in order to increase the bandwidth. The first one is called the Instruction-Data Cycle Structure, which divides the request queues into two subqueues, the instruction queue and the data queue. These two subqueues are inspected in alternate memory cycles. They found that there are improvements from −4–12 percent in bandwidth (the number of modules varies from 8 to 16) over a model with four blockage buffers and a single queue [8]. The second algorithm, the Group Request Structure, separates a memory cycle into two subcycles; the first subcycle is used for servicing the instruction queue, and the second subcycle is used for servicing the data queue. They found that there are 8–16 percent improvements over the same Instruction-Data Cycle Structure algorithm. Terman [25] has made a trace driven simulation on the Instruction-Data Cycle Structure algorithm and found that the theoretical predictions of Burnett and Coffman fit well with the simulation results for the fetching of instructions, but their predictions do not fit well with the simulation results for data requests, which are more random than instruction requests and are difficult to model accurately.

Many other researchers have studied models of parallel memories. These include Flores [11], Skinner and Asher [22], Ravi [19], Bhandarkar [3], Sastry and Kain [20], Baskett and Smith [2], Briggs and Davidson [5], Chang, Kuck, and Lawrie [9], Smith [23], and Hoogendoorn [14], etc. These studies are directed toward multiprocessor systems and we will not describe them here.

In the remainder of this section the deficiencies found in the previous models are summarized.

1) All the previous models assume that the memories operate synchronously. As Burnett and Coffman pointed out, simultaneous memory operations offer more opportunity to take advantage of program behavior in a particular memory system [8]. However, with synchronous operations, there is the problem of returning the results of the accesses from the memory. Since the results from each module are available simultaneously, extra buses or queues are needed to return these data to the processor. Further, a pipelined processor usually makes requests in sequence rather than in batches. Therefore, it is desirable to study a model in which the memory modules operate out of phase. By out of phase, we mean either: a) the initiations of the modules are asynchronous, or b) the initiations of the modules are timed by a clock and during a clock interval, at most one module can be initiated. Because the operations of asynchronous modules are much more difficult to control, only case b) is considered in this paper.

2) Very few studies have been made to minimize the waiting time of a request to the memory. Flores [11] has made a quantitative study relating the waiting time factor to the memory cycle time, the input/output time, and the worst case execution time for different numbers of memory banks. However, his study is directed toward the effect of interference from the input/output units and there is no queueing of requests. In other models a saturated request queue is assumed, and the effects of waiting time are not considered. When the

queue size is finite, it is possible to develop optimal algorithms which minimize the average waiting time of requests in the queue. In this paper the number of queued requests is assumed to be finite so that the effects on waiting time can be studied.

V. The Organizations of Primary Memory for a Pipelined Processor

We present in this section two different implementation alternatives of interleaved memories (Organizations I and II). The two organizations differ in the configurations of the memory request buffers. In Organization I, a single set of request buffers is assumed to be shared by all the modules, while in Organization II, individual request buffers exist for each module. The general assumptions made are as follows.

1) The request rate from the processor is assumed to be high enough so that any empty buffer in the memory system is filled up by an incoming request immediately. Buffers are assumed to exist at the processor end so that any additional requests generated by the processor can be queued there. The requests that can be served by the modules are those that exist in the memory buffers only. The high request rate assumption, in essence, means that the request queue is always saturated and is made because we want to get an upper bound on the performance of the memory. This assumption will be relaxed when we consider the effects of dependencies on the memory performance in another paper in this issue.

2) Each request is assumed to be an integer from 0–$m-1$, which is the module it requests, and is obtained as the residue of dividing the address by m.

3) The service time of each module (the read time or the write time) for a request is assumed to be constant. This is a good model for semiconductor memories. We also assume that a memory module, once initiated to start a memory cycle, is not available until the end of the cycle.

4) A memory cycle time is the time it takes for a memory module to service a request. Each memory cycle is assumed to consist of m equally spaced memory subcycles. It is further assumed that exactly one module can be initiated to service a request at the beginning of a memory subcycle and it takes m subcycles (1 memory cycle) to service any request, i.e., homogeneous service times. With this assumption the problem of multiple data paths is resolved because at most one module finishes in each subcycle and the system is never confronted with returning results from more than one module simultaneously. The modules are therefore clocked by the memory subcycles.

5) The waiting time of a request is the time between the initiation of a request and the completion of service. A *waiting cycle* is defined similar to Flores [11] as the ratio of the waiting time and the memory cycle time.

In Organization I (Fig. 2) there are m memory modules, a single set of $b+1$ associative buffers, $B_T, B_1, B_2, \cdots, B_b$, and an intelligent scheduler which schedules a memory module to start a memory cycle. The modules operate out of phase in a fashion called staggered cycles. One example of a staggered cycle is shown in Fig. 1. The set of $b+1$ associative buffers are

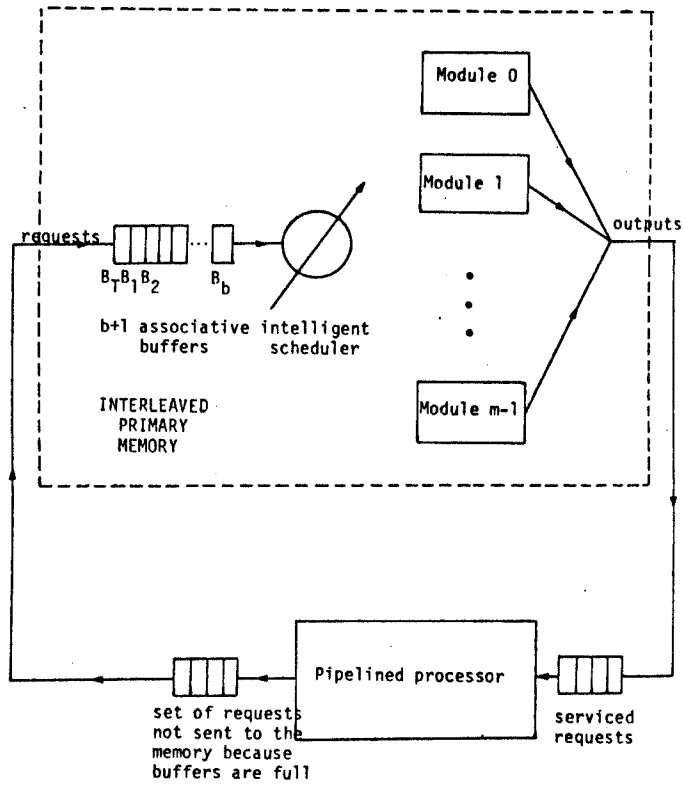


Fig. 2. Organization of I–An interleaved memory system with a single request queue.

used to store incoming requests. A request queued on a specific module can be retrieved in one associative search operation. Whenever a request is taken out from a buffer, all the requests behind it are pushed one location up so that B_T is empty. The buffer B_T has an additional function, namely, to receive requests from the bus. Due to our assumption of high request rate, B_T is filled immediately whenever it is empty. The queueing discipline for those requests in the buffers which address the same module is essentially first-in–first-out (FIFO). Other queueing disciplines are not studied because only uniprocessor systems are considered in this design.

The heart of the control in the memory system is the intelligent scheduler. The scheduler, using a scheduling algorithm, decides at the beginning of each memory subcycle whether to initiate a memory module and if so which module to initiate. The selection of a module to initiate is determined by the information about the requests in the associative buffers and by the knowledge about the status of the modules (free or busy). The optimal scheduling algorithm investigated in this paper is as follows.

Maximum-Work-Free-Module-First (MWFMF) Algorithm

In this algorithm a dynamic list of free modules is kept. Conceptually, at the beginning of a memory subcycle the buffers are checked associatively to see if any requests are queued on the free modules. If there is none, no module is initiated. If there is at least one free module with nonzero number of queued requests, an associative search is made on the buffers and the free module with the maximum number of requests queued on it is initiated. In case of ties, only the first one is

initiated [Fig. 3(a)].[1] The implementation of this algorithm can be done by using an additional associative memory of size m in the scheduler [Fig. 3(b)]. Each word in this associative memory can function as a counter and is used to indicate the number of requests queued on the corresponding module. The corresponding word is incremented/decremented when a request enters/leaves the request buffers. The free module with the maximum number of requests can be obtained by performing a maximum search on those words in this associative memory corresponding to the free modules, e.g., [18]. The maximum search algorithm shown in [18] is parallel by word and serial by bit and the time to perform a maximum search is proportional to the number of bits in the memory. The speed of this algorithm is therefore proportional to $\lceil \log_2 (b + 2) \rceil$, where $\lceil x \rceil$ is the smallest integer larger than or equal to x.

In addition to the overhead related to the execution of the scheduling algorithm, there is also the overhead of selecting the request from the associative buffers and sending it to the memory module. This overhead consists of matching the selected module number against all the requests in the buffers and selecting the first request if multiple responses occur in the match. Using a bit-serial word-parallel equality matching algorithm, e.g., [18], and a binary tree type multiple match resolution circuit, e.g., [12], this overhead is proportional to $\lceil \log_2 m \rceil + \lceil \log_2 (b + 1) \rceil$. In general, the overhead associated with the MWFMF scheduling algorithms is very small, and the selection of a module and the corresponding request to be initiated in the next subcycle can be overlapped with the current subcycle.

Two alternative algorithms have been investigated. They are as follows.

Round-Robin (RR)

All the modules are initiated in a round-robin fashion regardless of whether a request is queued on the module. The scheduler does not make use of any information about the status of the system. The implementation of this algorithm is very simple and the scheduler only has to know the current module initiated. In Fig. 1 the Gantt Chart for the operation of a 4-way interleaved memory with RR scheduling algorithm is shown. This is the scheduling algorithm that is implemented in most interleaved memory systems today.

First-Free-First (FFF)

In this algorithm only the information about the status of the modules (free or busy) is utilized by the scheduler. There is a FIFO list of free modules. At the beginning of a memory subcycle, the scheduler puts a busy module to the end of the free list if this module finishes its cycle. It then initiates the module at the head of the free list if there are any requests queued on it, otherwise the module at the head is appended to the tail of the free list and no other modules are checked in this subcycle. The scheduler may also check all the subsequent

[1] Multiple requests can be initiated in a subcycle. But since the return bus can return at most one piece of datum in any subcycle, only one read (which generates return data) and multiple writes (which do not generate return data) can be initiated simultaneously. The effect due to this improvement, as seen later, is very small.

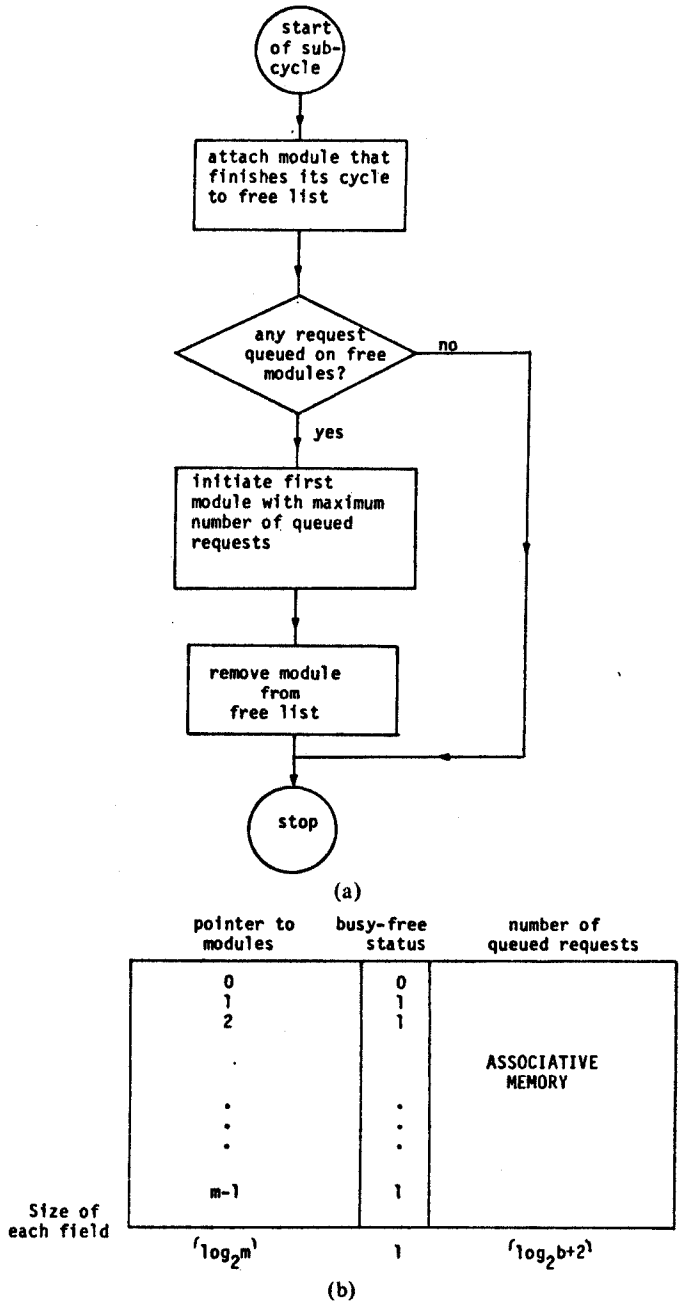


Fig. 3. (a) MWFMF scheduling algorithm. (b) Implementation of the MWFMF scheduling algorithm using associative memory ($\lceil x \rceil$ is the smallest integer larger than or equal to x).

modules in the free list, but the time for this is proportional to the number of modules and is not feasible when this number is large.

In contrast, the MWFMF algorithm utilizes both the information about the status of the modules and the requests in the buffers and therefore is expected to perform better. The RR and FFF algorithms are inferior to the MWFMF algorithm under the assumption of random, independent requests because the MWFMF algorithm is shown to be optimal. The MWFMF algorithm is also seen to perform better under trace driven simulations. These are shown in Section VI.

At the end of each memory subcycle, at most one request is serviced. The result is sent back to the processor. The necessary queue for storing these results is excluded from the memory organization.

The requests of the system come into the memory in a specific pattern. Two types of access patterns are considered in this design.

1) Random Accesses with No Address Dependency: The addresses are random and are independent of each other. Although this access pattern is not characteristic of pipeline computers, the random request assumption is useful as a lower bound for the performance when the accesses are correlated. It is also useful to model multiprocessor systems where the number of processors is much larger than the number of modules.

2) Accesses from the Execution Traces of a Monoprogrammed Pipelined Computer: The addresses in the execution traces are correlated and they represent a similar addressing behavior when the actual program is executed on a pipelined processor. We have used execution traces from a pipelined processor representing large scientific applications, the CDC 7600, in this study.

Organization II is similar to Organization I except that separate sets of buffers exist for each module (Fig. 4). Requests from the processor are continuously moved into the buffers of each module via B_T until a request in B_T is directed toward a module whose buffers are already full. The request in B_T is blocked, and as a result, further requests are blocked from entering the memory. When the module responsible for this blocking has finished servicing its current request, one request from its buffers is serviced which results in an empty buffer. The blocking request in B_T is moved into this empty buffer. Because buffers for other modules may not be full when the blocking situation occurs, one or more requests can then be accepted to the memory system until the previous blocking situation occurs with one of the modules. When $b = 0$ there is only one buffer B_T in the system and this is exactly the same as Organization I with $b = 0$. Therefore, Organization II degenerates into Organization I when $b = 0$.

The buffers used in Organization II are simpler than that of Organization I. Associative search capabilties are not necessary for these buffers. The implementation of the scheduler is similar to that of Organization I. The advantage with this system is that the request buffers are simple shift registers and therefore are cheaper. However, in order for this organization to operate at full capacity, more than one request may have to be moved across the bus into the memory in a memory subcycle. Since we assume that a pipelined processor generates on the order of one request every memory subcycle, the blocking situation may not always occur and the memory is underutilized. Further, it is necessary to build a faster bus so that multiple requests can be moved across the bus in a memory subcycle. Due to these reasons, the buffers in Organization II may be underutilized and it is difficult to estimate its performance. However, if it is assumed that the bus is wide enough to transfer parallel requests into the memory in a subcycle and enough requests are generated so that blocking always occurs, the following shows that Organization II is equivalent to Organization I as far as the average behavior is concerned.

The two organizations discussed are operating in steady-state. This means that the systems have been operating for a long time so that initial start-up effects have diminished. Further, since the buffer size is limited and fixed and the request rate from the processor is assumed to be very high, the average arrival rate to the memory system must equal to the average service rate. The average arrival rate and the average waiting time are finite and satisfy Little's Formula [16]. Let

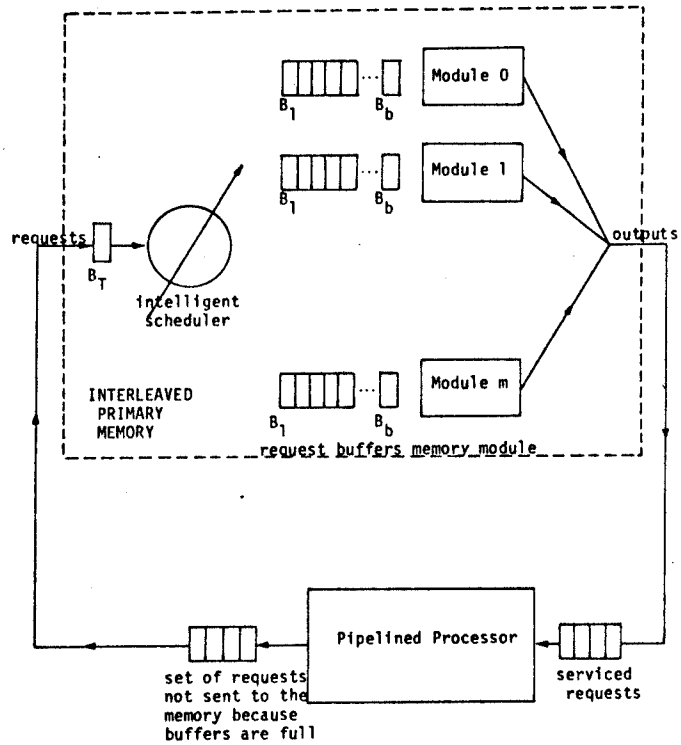


Fig. 4. Organization II–An interleaved memory system with multiple request queues.

m = number of memory modules,
b_I = number of buffers for the system in Organization I (not including B_T),
b_{II} = number of buffers for each module in Organization II (not including B_T),
e_{Bi} = utilization of the buffers $B_1, \cdots, B_b$ in Organization i (i = I, II; $e_{B,I} = 1$),
e_T = utilization of buffer B_T ($e_T = 1$ for both organizations),
B_I = number of buffers $B_1, \cdots, B_{b_I}$ in Organization I ($B_I = b_I$),
B_{II} = number of buffers $B_1, \cdots, B_{b_{II}}$ in Organization II ($B_{II} = m*b_{II}$),
$u_{m,b,i}$ = expected utilization of the modules in Organization i (i = I, II),
$w_{m,b,i}$ = expected waiting memory cycles of the requests in Organization i (i = I, II),
M_i = expected number of requests in Organization i (i = I, II),
λ_i = expected arrival rate per memory cycle in Organization i (i = I, II).

Then

$$M_i = (e_{B,i}*B_i + e_T) + u_{m,b,i}*m \tag{2}$$

$$\lambda_i = m*u_{m,b,i} \tag{3}$$

and they satisfy Little's Formula

$$M_i = \lambda_i * w_{m,b,i}. \tag{4}$$

The importance of Little's Formula lies in the fact that the average module utilization, the average number of waiting cycles, and the average buffer utilization are related. Once two of them are obtained, the other can be calculated easily. Further, if $\lambda_I = \lambda_{II}$ and b_{II} is chosen to be $b_I/(e_{B,II} * m)$, then $M_I = M_{II}$ and $W_I = W_{II}$. That is, Organizations I and II are equivalent as far as the average behavior is concerned. The only difference lies in the buffer utilization which is less than 1 in Organization II, whereas the buffers are fully utilized in Organization I. In the next section we present our proofs for Organization I only because the two Organizations are equivalent and the results are directly applicable. It is also implicit from now on that the subscript i (i = I, II) is omitted and Organization I is assumed unless otherwise stated.

VI. Evaluation of the Organizations

In this section we evaluate the performance of the memory organizations proposed. We first prove that the MWFMF scheduling algorithm has optimal average behavior for random requests. The simulation results using randomly generated requests and execution traces from actual programs are also shown.

A. Optimality of the MWFMF Scheduling Algorithm

So far, we have assumed that the request rate from the processor is very fast so that any empty buffers in the memory can be filled. If all the requests are available before the memory starts servicing them, this means that the request queue is infinitely long which is an unreasonable assumption. The *request queue* used in here means the queue of requests to be serviced by the memory whether they reside in the associative buffers or not.

In proving the optimality of the MWFMF algorithm, we assume that the requests in the request queue are *independent*, *randomly generated*, and of a *finite size*. The randomness assumption is made for mathematical tractability and is able to yield a lower bound in the memory utilization when the accesses are correlated. The size of the associative buffers may be greater than, equal to, or less than the number of requests in the request queue.

In a pipelined processor memory requests can be generated continuously until a dependency has been resolved. The number of requests generated between two dependencies is finite and we can regard that they are available in the request queue after the first dependency has been resolved. Due to our high request rate assumption, the empty associative buffers in the memory are filled up immediately after the first dependency is resolved. Other requests in the request queue are also available, but they cannot be moved into the memory because there are no available buffers.

In a practical implementation the pipelined processor is able to look ahead a fixed number of instructions and this is modeled by a fixed and finite number of associative buffers in the system (which may be greater than, less than, or equal to the size of the request queue). The intelligent scheduler is allowed to examine the associative buffers in making the scheduling decision. The objective of the scheduling algorithm is to complete the service of the requests in the request queue as fast as possible so that the throughput of the memory is maximized. The symbols used in the following theorems are

b = number of associative buffers − 1,
m = number of memory modules,
N = total number of requests that have to be serviced between two dependencies,
$\{(l_0, i_0), (l_1, i_1), \cdots, (l_{m-1}, i_{m-1})\}_k$ = state of the memory system

where

(l_j, i_j) = state of module j,
l_j = number of requests queued on module j in the buffers,

$$\sum_{j=0}^{m-1} l_j = b + 1 \text{ and } l_j \geqq 0 \qquad j = 0, 1, \cdots, m-1$$

$$i_j = \begin{cases} 0 & \text{if module } j \text{ is free} \\ n & 0 < n \leqq m \text{ if module } j \text{ is busy.} \end{cases}$$

In the case that module j is busy servicing a request, n is the number of subcycles that module j has serviced for this request. The number of subcycles remaining before the completion of service for the current request is $(m - n) \bmod m$.

k = number of requests in the request queue not including those in the associative buffers, that have to be serviced; k is also used in the induction proof.

C_j = completion time for the jth request (the completion time is counted from the time that the system initially starts with an unserviced request queue to the time when the jth request has completed its service).

$C_{\max}\{(l_0, i_0), (l_1, i_1), \cdots, (l_{m-1}, i_{m-1})\}_k$ = maximum completion time for processing all the requests in the associative buffers and the request queue given that the state is $\{(l_0, i_0), \cdots, (l_{m-1}, i_{m-1})\}_k$.

$EC_{\max}\{(l_0, i_0), (l_1, i_1), \cdots, (l_{m-1}, i_{m-1})\}_k$ = expected maximum completion time for processing all the requests in the associative buffers and the request queue given that the state is $\{(l_0, i_0), \cdots, (l_{m-1}, i_{m-1})\}_k$.

Before the main theorem can be stated, the following three lemmas must first be proved. Lemma 1 establishes the need for executing the MWFMF scheduling algorithm at the beginning of each subcycle. Lemma 2 establishes a basis for the induction proof of the main theorem and it also shows the optimality of the MWFMF algorithm when the buffer size is very large so that all the requests in the request queue reside in the associative buffers. Lemma 3 augments Lemma 2 by further showing that algorithm MWFMF minimizes the sum of completion times of all the requests.

Lemma 1:

a) In a period of m subcycles, every module can be initiated at most once.

b) At the beginning of each subcycle, there exists at least one free module available for scheduling.

The proof of this is obvious and will not be shown here [27].

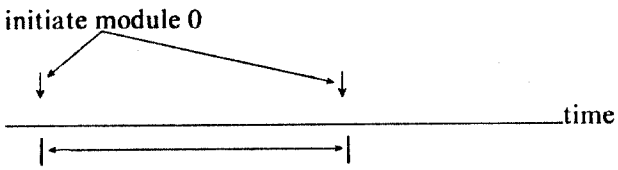
Lemma 2: If all the requests in the request queue reside in the associative buffers (that is, the buffers are large enough to accommodate all these requests), then algorithm MWFMF minimizes the maximum completion time in Organization I.

Proof: The maximum completion time is determined by the module with the largest number of queued requests. Assume without loss of generality that

$$l_0 > l_1 > \cdots > l_{m-1}.$$

Case 1: $i_0 = 0$

MWFMF schedules module 0 first.



All modules will be initiated at most once in here due to Lemma 1 (if number queued on it is nonzero) and all requests queued on every module except 0 can be initiated before the last request queued on module 0 is initiated.

$C_{\max} = l_0{*}m$ subcycles (initiate module 0 first).

For another algorithm, if module $j \neq 0$ is initiated, then module 0 can only be initiated in the next subcycle after module j has been initiated.

$\min C_{\max} = l_0{*}m + 1$ subcycles (initiate module $j \neq 0$ first).

In this case the minimum $C_{\max}$ is greater than the $C_{\max}$ of using the MWFMF scheduling algorithm.

Case 2: $i_0 > 0$. Let module j be the module such that

$$i_j = 0 \text{ and } i_0 > 0, i_1 > 0, \cdots, i_{j-1} > 0.$$

Since $l_0 > l_1 > \cdots > l_{j-1} > l_j$, then module j is the free module with the largest number of queued requests. This will be the module scheduled by the algorithm MWFMF. In fact, the module scheduled at this point is unimportant because the maximum completion time is governed by module 0.

$$C_{\max} = l_0{*}m + (m - i_0) \text{ subcycles.}$$

In summary

$$\min C_{\max} = l_0{*}m + (m - i_0) \bmod m \text{ subcycles.}$$

Optimum algorithm: MWFMF.

On the other hand, if $l_0 = l_1 > \cdots > l_{m-1}$ and $i_0, i_1 = 0$, then the $C_{\max}$'s are identical whether module 0 or 1 is scheduled first. A similar proof holds for the case $l_0 \geq l_1 \geq \cdots \geq l_{m-1}$. Q.E.D.

Lemma 3: If all the requests in the request queue reside in the associative buffers, then algorithm MWFMF minimizes ΣC_j in Organization I, where C_j is the completion time for the jth request.

Proof: Assume without loss of generality that

$$l_0 > l_1 > \cdots > l_{m-1}.$$

Consider two modules a, b, $(0 \leq a, b \leq m-1)$ such that $i_a = 0, i_b = 0$, and $l_a > l_b$. Let $C_{a,b}(C_{b,a})$ be the sum of completion times of scheduling a before b (b before a) for modules a and b only. If b is scheduled before a, then

$$C_{b,a} = \sum_{i=1}^{l_b} im + \sum_{i=1}^{l_a} (im + 1)$$

$$= \frac{m}{2}[(l_a + 1)l_a + (l_b + 1)l_b] + l_a.$$

Comparing this with the case of scheduling a first, it is found that

$$C_{a,b} = \sum_{i=1}^{l_a} im + \sum_{i=1}^{l_b} (im + 1)$$

$$= \frac{m}{2}[(l_a + 1)l_a + (l_b + 1)l_b] + l_b.$$

Since $l_a > l_b \Rightarrow C_{a,b} < C_{b,a}$, this implies that scheduling the module with a larger number of queued requests can reduce ΣC_j. By adjacent pairwise interchange,[2] it is therefore better to schedule the module with the maximum number of queued requests if it is free. If this module is not available, scheduling the free module with the maximum number of queued requests is also optimum. Q.E.D.

From the proofs of Lemmas 2 and 3, it is seen by using the MWFMF algorithm that:

1) The throughput of the memory is at a maximum because the maximum time to complete a set of requests is minimized (Lemma 2).

2) The average waiting time is minimized. This is because C_j, the completion time for the jth request, equals the waiting time for the jth request, $W_j = C_j$ (all the requests are available at $t = 0$). As a result, average waiting time $= \Sigma W_j/M$ is also minimized (Lemma 3).

Lemmas 2 and 3 have also been proved with no assumption on the distribution and dependence of requests in the buffers. In proving Theorem 1, the assumption of independent, random requests must be made in order to calculate the expected maximum completion time.

Theorem 1: If all the requests in the request queue do not reside in the associative buffers (that is, the buffers are not large enough to accommodate all the requests in the request queue), then algorithm MWFMF minimizes the expected maximum completion time for independent, random requests in Organization I.

Proof: In order to prove this theorem, the following two parts must be proven and the theorem follows from the result of part a).

a) Algorithm MWFMF minimizes the expected maximum completion time for independent, random requests.

[2] Since a and b are any two modules such that $l_a > l_b$ and we discover that it is better to schedule module a than b, the same reasoning can be applied so that it is better to schedule module a' where $l_{a'} > l_a$. By propagating the same reasoning for every pair of modules, we see that the free module with the maximum number of queued requests should be scheduled.

b) Let states

$$S_1 = \{\cdots, (l_a^1, i_a), (l_b^1, i_b), \cdots\}_k$$

$$S_2 = \{\cdots, (l_a^2, i_a), (l_b^2, i_b), \cdots\}_k$$

where "$\cdots$" indicates that the remaining states are identical for S_1 and S_2.

We assume that

$$l_a^2 > l_a^1$$

$$l_b^1 > l_b^2.$$

Since the states of other modules are identical

$$l_a^1 + l_b^1 = l_a^2 + l_b^2.$$

and

$m > i_a > i_b > 0$ or $m > i_b > i_a > 0$ with equal probability.

If $l_a^2 > l_b^1$, then $EC_{\max}(S_1)_k \leqq EC_{\max}(S_2)_k$,

If $l_a^2 = l_b^1$, then $EC_{\max}(S_1)_k = EC_{\max}(S_2)_k$.

These two parts can be proved by induction. The truth is first established for $k = 0$, i.e., when all the requests reside in the buffers. These parts are then assumed to be true for any positive integer k and the proof will be complete by proving the case of $k + 1$. The proof of part a) is shown here. The proof of part b) is similar, but very long and is omitted [27].

i) $k = 0$. MWFMF is optimal. This is established by Lemma 2.

ii) Induction hypothesis: Assume that part b) of the theorem is true for a positive integer k and MWFMF algorithm minimizes the expected maximum completion time for independent, random requests when the number of remaining requests in the request queue is k.

iii) When the number of remaining requests is $k + 1$, without loss of generality, let modules $0, 1, \cdots, j$ be the set of free modules. Choose any two modules, say 0 and 1, so that $l_0 > l_1$ and there does not exist $p \in \{2, \cdots, j\}$ such that $l_0 > l_p > l_1$. We want to compare the difference between scheduling modules 0 and 1.

1) Schedule module 0 at the beginning of this subcycle:

$$\{(l_0, 0), (l_1, 0), \cdots, (l_{m-1}, i_{m-1})\}_{k+1} \Rightarrow \{(l_0 - 1, 0), (l_1, 0), \cdots, (l_{m-1}, i_{m-1})\}_{k+1}.$$

A new input now enters the buffers; this input can be a request directed to any module in the set with equal probability $1/m$ (due to the assumption of independent, random requests). New states after scheduling module 0 and the current subcycle has ended:

0 enters: $S^0 = \{(l_0, 1), (l_1, 0), \cdots, (l_{m-1}, (i_{m-1} + 1) \bmod m)\}_k$

1 enters: $S^1 = \{(l_0 - 1, 1), (l_1 + 1, 0), \cdots, (l_{m-1}, (i_{m-1} + 1) \bmod m)\}_k$

$\cdots$

$m - 1$ enters: $S^{m-1} = \{(l_0 - 1, 1), (l_1, 0), \cdots, (l_{m-1} + 1, (i_{m-1} + 1) \bmod m)\}_k$.

2) Schedule module 1 at the beginning of this subcycle:

$$\{(l_0, 0), (l_1, 0), \cdots, (l_{m-1}, i_{m-1})\}_{k+1} \Rightarrow \{(l_0, 0), (l_1 - 1, 0), \cdots, (l_{m-1}, i_{m-1}\}_{k+1}.$$

New states after scheduling module 1 and the current subcycle has ended:

0 enters $\bar{S}^0 = \{(l_0 + 1, 0), (l_1 - 1, 1), \cdots, (l_{m-1}, (i_{m-1} + 1) \bmod m)\}_k$

1 enters: $\bar{S}^1 = \{(l_0, 0), (l_1, 1), \cdots, (l_{m-1}, (i_{m-1} + 1) \bmod m)\}_k$

$\cdots$

$m - 1$ enters: $\bar{S}^{m-1} = \{(l_0, 0), (l_1 - 1, 1), \cdots, (l_{m-1} + 1, (i_{m-1} + 1) \bmod m\}_k$.

It is seen that $EC_{\max}(S^0) = EC_{\max}(\bar{S}^1)$, $EC_{\max}(S^1) < EC_{\max}(\bar{S}^0)$, and $EC_{\max}(S^j) < EC_{\max}(\bar{S}^j)$ for $j \neq 0, 1$. In proving $EC_{\max}(S^0) = EC_{\max}(\bar{S}^1)$, we can rename module 0 to be module 1 and vice versa in state S^0 and let $i_a = 0$, $i_b = 1$, $l_a^1 = l_1$, $l_b^1 = l_0$, $l_a^2 = l_0$, $l_b^2 = l_1$ in the induction hypothesis of part b). The other parts can similarly be proved. Since the expected $C_{\max}$ is a weighted sum of the expected $C_{\max}$ of all the corresponding states, it is, therefore, better to schedule module 0, the module with a longer queue first. By using the adjacent pairwise interchange argument, the free module with the maximum number of queued requests should be scheduled first. With the induction argument, the theorem is therefore proved. Q.E.D.

The above theorem has demonstrated that algorithm MWFMF is optimal in the sense that it minimizes the average completion time for a fixed set of random requests. Intuitively, algorithm MWFMF is better because it tries to keep all the modules as busy as possible. Suppose that some of the modules are requested more often than others. The requests to these more frequently requested modules become a bottleneck to the system whatever scheduling algorithms are used. However, a better scheduling algorithm should make use of the free cycles to schedule some requests for the less popular modules so that these requests would not accumulate behind the processing of the more popular requests. This is the deficiency that occurs in other algorithms and is overcome by the MWFMF algorithm.

In addition to proving that the MWFMF algorithm has the best average case behavior, it may be necessary to show that the algorithm also possesses the best best case behavior and the best worst case behavior. However, in this case, the best case and the worst case behavior are identical for all algorithms. The best case behavior occurs when all the requests are made in a sequential order, that is, $0, 1, \cdots, m - 1, 0, 1, \cdots, m - 1$, etc. No contention would occur and the throughput of the memory is maximized, that is, 1 request serviced every subcycle. On the other hand, the worst case behavior occurs when all the requests are directed to a single module. In this case the bottleneck is at this module and the throughput of the memory is 1 request serviced every m subcycles. Algorithm MWFMF is better than other algorithms because it has a better average case behavior even though its best and worst case behavior are identical to the other algorithms.

Although the expected maximum completion time of the algorithm is minimized, it is not possible to make a similar

conclusion as in Lemma 2 that the expected throughput of the memory is maximized because in this case there is no relationship between the expected maximum completion time and the expected throughput of the system. It is also difficult to prove a similar theorem for the ΣC_j case as in Lemma 3.

Although Theorem 1 establishes the fact that the MWFMF algorithm is optimal, no throughput values are obtained analytically. In fact, it is very difficult to obtain a closed form solution using queueing theory and the solution using embedded Markov chains results in a large number of states [27]. Our evaluations are therefore based on simulations using randomly generated requests and execution traces.

C. Simulation Results

The simulations were run on a CDC 6400 computer. The simulation program is written in Fortran and the total amount of time to generate all the results took over 12 h on the CDC 6400. Two types of request sequences are considered, one in which the requests are generated randomly and one in which the requests are derived directly from the execution trace of a program. The traces used have a size of 500 000 and were obtained by running a scientific Fortran program derived from Ballistic Missile Defense applications on a CDC 7600 and they personify program characteristics of scientific applications. They have the following characteristics.

fraction of instruction word fetches		0.597
fraction of data word fetches		0.336
fraction of data word stores		0.067
average number of operand accesses per inst. executed		0.600
number of instructions per instruction word		2.787
fraction of instructions that need data		0.242
fraction of instructions that are		
unconditional jumps		0.044
successful conditional jumps		0.030
unsuccessful conditional jumps		0.015
Average distance in terms of the number of instructions between the instruction setting the condition code and the jump instruction (unconditional or conditional) at the decode segment		0.890
Average distance in terms of the number of instructions between the instruction setting the condition code and the jump instruction (conditional) at the decode segment		1.693
number of instructions executed between		
conditional jumps	mean	22.3
	standard deviation	10.3
unconditional jumps	mean	22.8
	standard deviation	24.7
successful conditional jumps	mean	33.9
	standard deviation	19.2
all dependent events (conditional + unconditional jumps)	mean	11.4
	standard deviation	10.1

The simulation results are obtained without taking into account the effects of dependencies. These effects are studied in a different paper in this issue. The detailed simulation results are not presented here [27]. We have selected a few sets and have plotted them in Figs. 5-10.

In Figs. 5-10 the performance of Organizations I and II are shown. The actual simulation results are used for $b \leq 6$ while estimations are made for $b > 6$. The values of memory utilization are interpolated because the asymptotic values of memory utilization when b is large are known (1.0). For Organization I, $e_{B,I} = 1$ and therefore the waiting cycles can be calculated directly using Little's Formula. For Organization II the asymptotic values of waiting cycles and buffer utilization are not known and therefore both of them must be extrapolated.

Some observations that can be made from these figures are as follows.

1) The memory utilization asymptotically approaches 1 as the buffer size is increased (Fig. 5).

2) The MWFMF algorithm gives the highest memory utilization and the smallest number of waiting cycles for both random request and trace driven simulations (Figs. 5 and 6).

3) The trace driven simulation results show a higher memory and buffer utilizations and a smaller number of waiting cycles than the random request simulation results due to a higher correlation between consecutive requests. As a result, the requests are likely to be made in a consecutive order and there is less contention in the system (Figs. 5-10).

4) The memory ultilization is higher when the buffer size is increased, but is smaller when the degree of interleaving increases (Figs. 5 and 7). A larger number of buffers results in a larger variety of requests and this accounts for the increased utilization. On the other hand, when the number of memory modules increases, there is a smaller probability that a request in the buffers can be serviced and this accounts for the decreased utilization.

5) Similarly, the number of waiting cycles is larger when the buffer size is increased, but is smaller when the degree of interleaving increases (Figs. 6 and 8).

6) In Organization II the rate of increase of the buffer utilization is smaller as the buffer size is increased (Fig. 9). This also accounts for the diminishing increase in memory utilization as the buffer size is increased. The reason for this is because an increase in b by 1 in Organization II results in 8 buffers being added to the system ($m = 8$) and the memory utilization quickly approaches 1. The difference between the different algorithms is also small for large values of b because the memory utilization for all these algorithms quickly approaches 1.

7) The buffer utilization in Organization II is smaller as the degree of interleaving increases because there is a higher probability that B_T is blocked when the number of modules is increased (Fig. 10).

D. Effects of Separating the Instruction and the Data Area

The previous results have been obtained from simulations using a merged instruction and data area. Since an instruction access results in some data accesses, it is desirable to place the data accessed in modules not conflicting with the next instruction accessed. This motivates us to investigate the sepa-

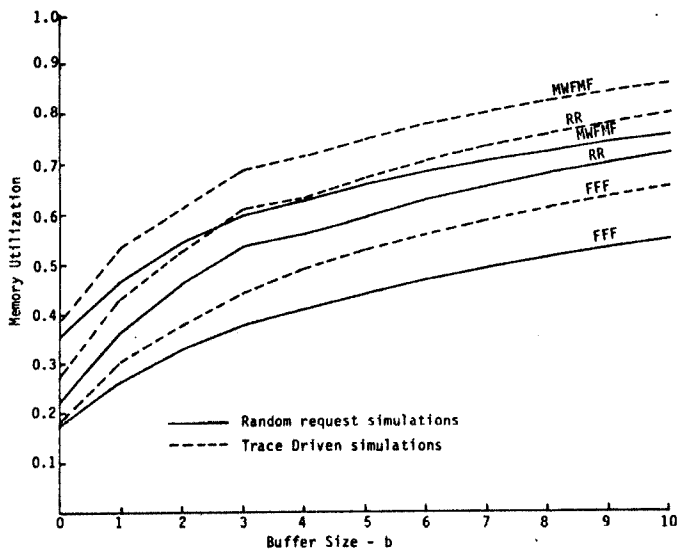


Fig. 5. Improvement of average memory utilization with buffer size for Organization I ($m = 8$).

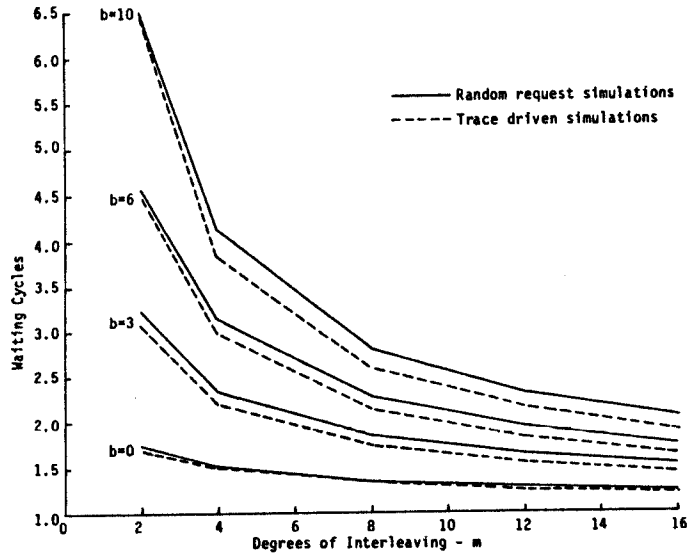


Fig. 8. Decrease of waiting cycles with respect to the degrees of interleaving for Organization I with MWFMF scheduling algorithm.

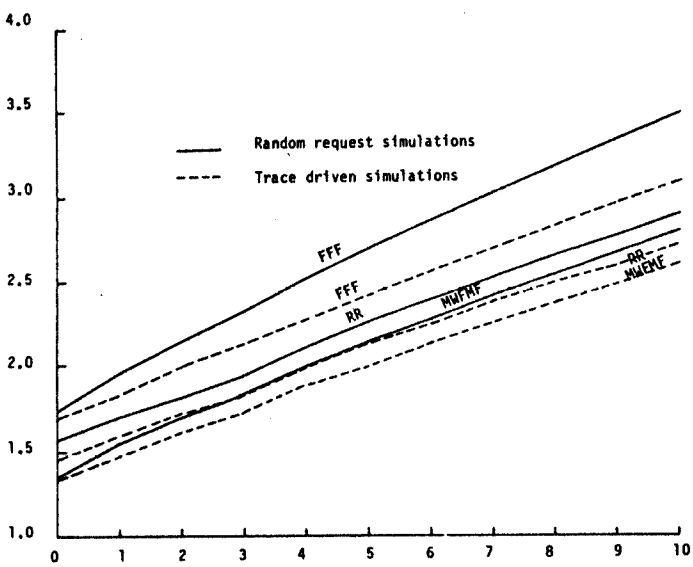


Fig. 6. Increase of average waiting cycles with buffer size for Organization I ($m = 8$).

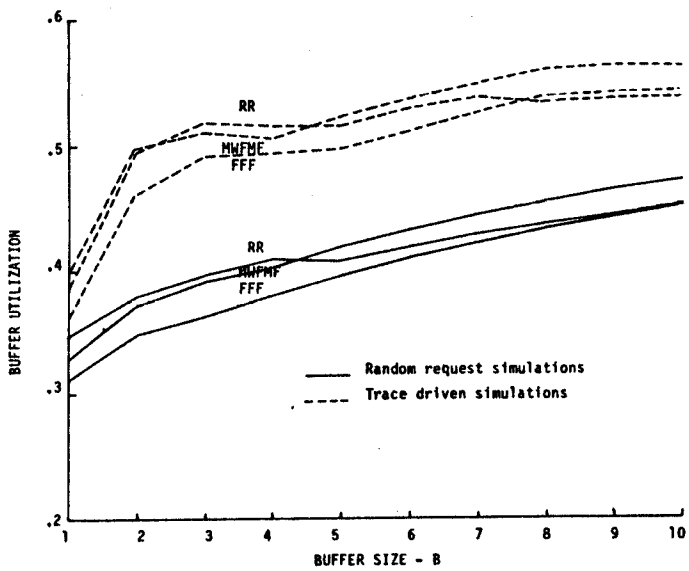


Fig. 9. Average buffer utilization for Organization II ($m = 8$). (These do not include the utilization of B_T.)

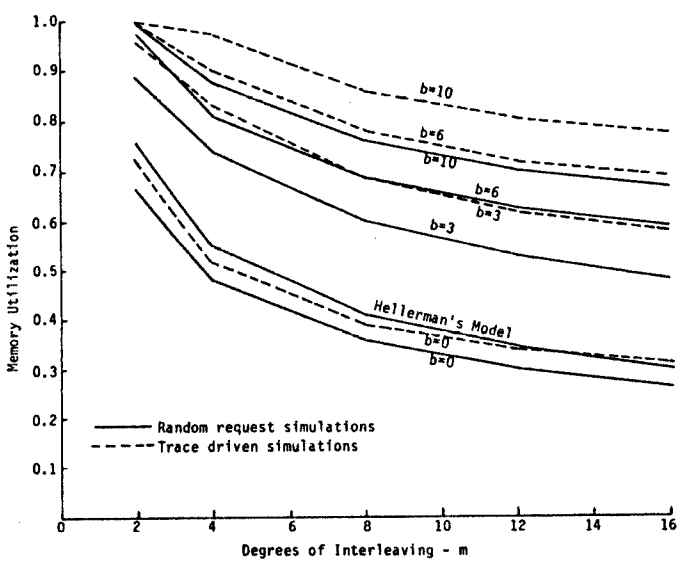


Fig. 7. Decrease of average memory utilization with respect to degreees of interleaving for Organization I with MWFMF algorithm.

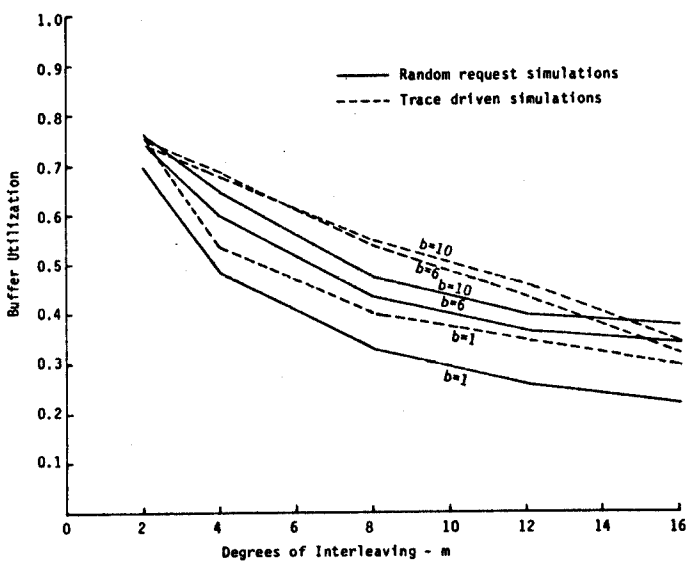


Fig. 10. Average buffer utilization versus the degrees of interleaving for Organization II with MWFMF scheduling algorithm ($m = 8$). (These do not include the utilization of B_T.)

TABLE I
COMPARISON BETWEEN MERGED AND SEPARATED INSTRUCTION-DATA MODULES FOR ORGANIZATION I-TRACE DRIVEN SIMULATIONS

			RR		MWFMF	
	m	b	Memory util.	Waiting cycles	Memory util.	Waiting cycles
Merged Inst.-Data Areas (m=6)	6	0	0.338	1.49	0.459	1.36
		1	0.501	1.05	0.624	1.53
		2	0.657	1.76	0.695	1.72
		3	0.726	1.92	0.752	1.89
Separate Inst.-Data Areas (4-2 ways)	6	0	0.336	1.50	0.486	1.34
		1	0.517	1.64	0.619	1.54
		2	0.618	1.81	0.692	1.72
		3	0.696	1.96	0.730	1.91

TABLE II
COMPARISON BETWEEN MERGED AND SEPARATED INSTRUCTION-DATA MODULES FOR ORGANIZATION II-TRACE DRIVEN SIMULATIONS

			RR			MWFMF		
	m	b	Mem. Util.	Wait. Cycle	Buf. Util.	Mem. Util.	Wait. Cycle	Buf. Util.
Merged Inst.-Data Areas (m=6)	6	0	0.34	1.49	-	0.46	1.36	-
		1	0.69	1.95	0.49	0.69	1.94	0.48
		2	0.80	2.48	0.50	0.79	2.49	0.50
		3	0.81	2.66	0.45	0.81	2.67	0.45
Sep. Inst.-Data Areas (4-2 Ways)	6	0	0.34	1.50	-	0.49	1.34	-
		1	0.65	1.98	0.47	0.67	1.90	0.43
		2	0.75	2.42	0.45	0.76	2.41	0.45
		3	0.77	2.96	0.45	0.77	2.95	0.45

ration of instruction and data areas into different modules in the main memory. Sastry *et al.* [20] and Nutt [17] have made some pioneering studies on the separation of instruction and data areas, but they have assumed a nonpipelined multiprocessor system. Burnett and Coffman have also studied the problem of separating the instruction and the data queue [8], but the effects of separating the instruction and the data modules are not studied.

In this section we present a comparison between an organization with merged instruction and data modules and an organization with separate instruction and data modules using the traces available. In Organizations I and II discussed earlier, consecutive instruction words are allocated in consecutive instruction memories and consecutive data addresses are allocated in consecutive data memories. The characteristics of the traces reveal that 60 percent of the accesses are instructions and the rest are data accesses; therefore, the modules should be divided according to this ratio approximately. It is also desirable to have the number of instruction modules and the number of data modules an integral power of 2 for ease of address decoding. It is not possible to satisfy the above two requirements simultaneously. We have chosen to divide the modules into a 4–2 partition so that four of the modules are instruction modules and two are data modules. A 6–4 partition can be chosen, but the number of instruction modules is not an integral power of 2. An 8–4 partition behaves in the same way as a 4–2 partition.

Since there are 6 modules in the 4–2 partition, it is necessary to compare the performance of the 4–2 partition against a hypothetical 6 way interleaved system with merged instruction and data modules. The results are shown in Tables I and II. It is seen that the differences between the two alternatives are minimal. In fact, in some cases the merged organization seems to perform a little better. This is due to the unequal utilization and waiting cycles of the modules in the separated case. From the simulation results on the utilization of the individual modules (not shown), the instruction modules are found to have a higher utilization than the data modules. The unequal utilization is caused by the fact that the ratio of instruction access to data access is not exactly 4 to 2. Had it been exactly 4 to 2, the utilization of the instruction and data modules would be approximately equal and the separated case would perform better.

One way to improve the performance of the system is to design the system with a good instruction-data access ratio so that the utilization of the instruction and data modules are approximately equal and the number of instructions and data modules are integral powers of 2. However, this ratio is highly program dependent and is impossible to fix at the design stage. We conclude that the improvement due to separations is minimal for this architecture, the CDC 7600, and the specific class of programs.

VII. CONCLUSION

We have presented in this paper two organizations of an interleaved memory system which utilizes a finite buffer space for the storage of requests. We have designed a scheduling algorithm which allows a finite set of requests to be processed in the minimum expected completion time. However, the performance of our system is obviously less than the performance of systems with an infinite saturated request queue which is an unrealistic assumption. In Fig. 7 we have shown the performance of Hellerman's model [13] together with our simulation results. Although Hellerman's model is a simple model and allows no queueing of requests, it is useful as a lower bound for comparing the performance with other systems. It is seen that with a random request queue Hellerman's model is better than our Organization I with $b = 0$, but is worse for $b > 0$. Note that the performance curves all have the same shape. Since Organization II degenerates into Organization I for $b = 0$, it is worse than Hellerman's model for $b = 0$, but better for $b > 0$. The comparison with other models in the current literature is not meaningful because they differ significantly.

We can improve our model slightly by considering the following. The rationale behind the constraint that only one module may be initiated in any subcycle is because the return bus can return at most one piece of datum in any subcycle. But since reads generate return data while writes do not, we can initiate two or more modules in a subcycle provided that exactly one of the requests is a read. The improvement in utilization due to this is only about 2 percent. The improvement is not significant because the fraction of writes in our trace is less than 7 percent of all the accesses and its applicability is also limited by memory interference.

The questions that still remain to be resolved are how can one select between Organization I and Organization II and how does one choose the parameters of the system in order to satisfy all the requirements. In the hardware requirements

Organization I needs associative search capabilities in the buffers, while Organization II does not. However, the availability of fast associative memory can help in this regard. The actual performance of Organization II may be worse because it may require the transfer of more than one request into the memory system during a memory subcycle and sometimes this is not possible in a pipelined system. Organization II gives a slightly better performance than Organization I when a maximum of one request is allowed to be transferred into the memory in each subcycle and the effective buffer sizes in both organizations are identical. Tradeoff in cost and performance must be made in the selection of the organization. In order to answer the second question we have raised, we need to design a cost model of the system. The cost of individual components is highly technology dependent and will not be discussed here. However, the designer can find a configuration with the minimum cost based on the bandwidth and the response time requirements. Assuming that the bus width is determined and fixed, the designer can use the average utilization (a function of the degrees of interleaving and the number of buffers) as an alternate measure of bandwidth. The response time can also be normalized with respect to the speed of the memory to give the waiting cycle. Using Little's Formula, the average number of requests in the memory or the average number in the request buffers can be obtained. The designer can then substitute the values for the average utilization and the buffer size into the formula obtained by regression [27] and obtain a set of values of m and b satisfying the requirements. By evaluating the costs of the different configurations, the cost of the memory can be estimated. The final configuration selected will be the one with the minimum cost.

The MWFMF scheduling algorithm we have studied in this paper is optimal in the sense that it minimizes the expected finish time for a finite sequence of random, independent requests. Although there exist restrictions and the performance of specially structured computers, e.g., Cray I, TIASC, ILLIAC IV, etc., are not evaluated, our scheduling algorithm is still applicable to these machines.

Although we have done a large number of simulations in this paper, our results only bring us one step closer to reality. However, we can conclude that the two organizations are useful for the design and implementation of the main memory systems. The cost of control may be high in some cases, but it is still small as compared to the cost of the main memory and it is cost-effective to utilize complicated control algorithms. The effects of dependencies on memory and buffer utilizations are not considered in this paper.This will be presented in a separate paper in this issue.

ACKNOWLEDGMENT

The authors would like to thank Dr. C. R. Vick and Dr. C. G. Davis of the Ballistic Missile Defense Advanced Technology Center and Mr. B. Covelli of the System Development Corporation for supplying them with the necessary traces from the CDC 7600-TIASC-PEPE computer network. Dr. Y. W. Ma is also thanked for a careful reading of the manuscript and providing many valuable suggestions.

REFERENCES

[1] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, " "The IBM system 360 model 91: Machine philosophy and instruction handling," *IBM J.Res. Develop.*, pp. 8-24, Jan. 1967.

[2] F. Baskett and A. J. Smith, "Interference in multiprocessor computer systems with interleaved memory," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 327-334, June 1976.

[3] D. P. Bhandarkar, "Analysis of interference in multiprocessors," *IEEE Trans. Comput.*, vol. C-24, pp. 897-908, Sept. 1975.

[4] L. J. Boland, G. D. Granito, A. V. Marcotte, B. V. Messina, and J. W. Smith, "The IBM system 360 model 91: Storage systems," *IBM J. Res. Develop.*, pp. 54-68, Jan. 1967.

[5] F. A. Briggs and E. S. Davidson, "Organization of semiconductor memories for parallel—pipelined processors," *IEEE Trans. Comput.*, vol. C-26, pp. 162-169, Feb. 1977.

[6] G. J. Burnett and C. G. Coffman, Jr., "A study of interleaved memory systems," in *Proc. AFIPS 1970 SJCC*, vol. 36, pp. 467-474.

[7] ——, "A combinational problem related to interleaved memory systems," *J. Ass. Comput. Mach.*, vol. 20, pp. 39-45, Jan. 1973.

[8] ——, "Analysis of interleaved memory systems using blockage buffers," *Commun. Ass. Comput. Mach.*, vol. 18, pp. 91-95, Feb. 1975.

[9] D. Y. Chang, D. J. Kuck, and D. H. Lawrie, "On the effective bandwidth of parallel memories," *IEEE Trans. Comput.*, vol. C-26, pp. 480-490, May 1977.

[10] E. G. Coffman, Jr., G. J. Burnett, and R. A. Snowdon, "On the performance of interleaved memories with multiple word bandwidths," *IEEE Trans Comput.*, vol. C-20, pp. 1570-1573, Dec. 1971.

[11] I. Flores, "Derivation of a waiting-time factor for a multiple bank memory," *J. Ass. Comput. Mach.*, vol. 11, pp. 265-282, July 1964.

[12] C. C. Foster, "Determination of priority in associative memories," *IEEE Trans. Electron. Comput.*, vol. EC-17, pp. 788-789, Aug. 1968.

[13] H. Hellerman, *Digital System Principle*. New York: McGraw-Hill, 1967, pp. 228-229.

[14] C. H. Hoogendoorn, "A general model for memory interference in multiprocessors," *IEEE Trans. Comput.*, vol. C-26, pp. 998-1005, Oct. 1977.

[15] D. E. Knuth and G. S. Rao, "Activity in an interleaved memory," *IEEE Trans. Comput.*, vol. C-24, pp.943-944, Sept. 1975.

[16] J. D. C. Little, "A proof of the queuing formula $L = \lambda W$," *Oper. Res.*, vol. 9, pp. 383-387, 1961.

[17] G. J. Nutt, "Memory and bus conflict in an array processor," *IEEE Trans. Comput.*, vol. C-26, pp. 514-521, June 1977.

[18] C. V. Ramamoorthy, J. L. Turner, and B. W. Wah, "A design of a cellular associative memory for ordered retrieval," *IEEE Trans. Comput.*, vol. C-27, pp. 800-815, Sept. 1978.

[19] C. V. Ravi, "On the bandwidth and interference in interleaved memory systems," *IEEE Trans. Comput.*, vol. C-21, pp. 899-901, Aug. 1972.

[20] K. V. Sastry and R. Y. Kain, "On the performance of certain multiprocessor computer organizations," *IEEE Trans. Comput.*, vol. C-24, pp. 1066-107, Nov. 1975.

[21] C. Schunemann and W. G. Spruth, "Storage hierarchy technology and organization," in *Digital Memory and Storage*, W. E. Proebster, Ed. Braunschweig, W. Germany: Vieweg, 1978.

[22] C. E. Skinner and J. R. Asher, "Effects of storage contention on system performance," *IBM Syst. J.*, vol. 4, pp. 319-333, 1969.

[23] A. J. Smith, "Multi-processor memory organization and memory interference," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 754-761, Oct. 1977.

[24] W. D. Strecker, "Analysis of the instruction execution rate in certain computer structures," Ph.D. dissertation, Carnegie-Mellon Univ. Pittsburgh, PA, 1970.

[25] F. W. Terman, "A study of interleaved memory systems by trace driven simulation," Digital Syst. Lab., Stanford Electron. Lab., Stanford Univ., Stanford, CA, Tech. Note 94, Sept. 1976.

[26] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Develop.*, pp. 25-33, Jan. 1967.

[27] B. W. Wah, "A systematic approach to the management of data on distributed databases," Ph.D. dissertation, Univ. of California, Berkeley, Dec. 1979.

[28] C. V. Ramamoorthy and H. F. Li, "Pipeline architecture," *ACM Comput. Surveys*, vol. 9, pp. 61-102, Mar. 1977.

C. V. Ramamoorthy (M'57-SM'76-F'78) received the undergraduate degrees in physics and technology from the University of Madras, Madras, India, the M.S. degree and the professional degree of Mechanical Engineer, both from the University of California, Berkeley, and the M.A. and Ph.D. degrees in applied mathematics and computer theory from Harvard University, Cambridge, MA.

He was associated with Honeywell's Electronic Data Processing Division from 1956 to 1971, last as Senior Staff Scientist. He was a Professor in the Department of Electrical Engineering and Computer Sciences at the University of Texas, Austin. Currently, he is a Professor in the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

Dr. Ramamoorthy was Chairman of the Education Committee of the IEEE Computer Society and Chairman of the Committee to develop E.C.P.D. Accreditation Guidelines for Computer Science and Engineering Degree Programs. He also was the Chairman of the AFIPS Education Committee, a member of the Science and Technology Advisory Group of the U.S. Air Force, and a member of the Technology Advisory Panel of Ballistic Missile Defense (U.S. Army). Currently, he is Vice President of the IEEE Computer Society for Educational Activities.

Benjamin W. Wah (S'78 M'79) was born in Hong Kong, on September 7, 1952. He received the B.S. and M.S. degrees in electrical engineering and computer sciences from Columbia University, New York, NY, in 1974 and 1975, respectively, the M.S. degree in computer science, and the Ph.D. degree in engineering from the University of California, Berkeley, in 1976 and 1979, respectively.

Currently, he is an Assistant Professor with the School of Electrical Engineering, Purdue University, West Lafayette, IN. He is Vice Chairman for distributed databases of the IEEE Computer Society Technical Committee on Distributed Processing. His current research interests include database machines, high-performance parallel architecture, distributed computer systems, and computer system performance evaluation.

A Hard Programmable Control Unit Design Using VLSI Technology

BULENT I. DERVISOGLU, MEMBER, IEEE, AND DONALD J. CRISCIONE

***Abstract*—Microprogramming has become a widely used technique which brings versatility to the control unit of a digital system. However, since all microcommands contained in a microinstruction are changed simultaneously, this form of control requires the coding of all possible combinations of parallel commands as separate microinstructions. This causes the resulting microprograms to become space- and timewise inefficient and makes it very difficult to control and/or synchronize independent processes. A new approach to modeling and implementing the flow of control in hardware systems is presented in this paper. Here a control flow is realized by a set of independent control operators that pass control to each other according to the precedence relationships defined by the control function which is implemented. The detailed architecture of a VLSI chip to implement a control unit as a hard-programmable control unit (HPCU) is also given. An HPCU contains many independent control operators and allows mask-programmable connections to be made among the operators to implement any given control function. Examples included in the paper demonstrate the effectiveness of an HPCU in implementing complex parallel control flows.**

***Index Terms*—Control flow expression, control graphs, control operator, mask-programming, microprogrammable control unit, microprogramming, parallel control flows, precedence and reset equations.**

Manuscript received June 30, 1980; revised February 17, 1981 and May 12, 1981.

B. I. Dervisoglu is with the Sperry Research Center, Sudbury, MA 01776.

D. J. Criscione is with Computer Vision, Bedford, MA 01730.

I. INTRODUCTION

AS DIGITAL systems become more and more complex, it is becoming increasingly difficult to design their control units. There are two contributing factors to the problem. First, advances in integrated circuit (IC) technology has made it feasible to use multiple copies of data subfunction units (registers, adders, etc.) and/or use complex data modules in designing a digital system. This requires the generation of a large number of control signals by the control unit which must also respond to the many demands/conditions that may be created by the data units. Second, the degree of sophistication of control functions has increased greatly due to theoretical advancements, especially in the area of parallel processing. As a result, control units are required to orchestrate the execution of instructions in a digital system not according to a fixed schedule, but rather as independent executions which are synchronized with each other according to some boundary conditions. For example, $Z = A * B + C * D$ requires that the summation of the two product terms follow the evaluation of these terms, but the products may be evaluated independent of each other and in any order.

Developments in LSI and VLSI technology have greatly impacted the design of digital systems. However, there has been very little change in the design and implementation of control units. Thus, designers are still limited to two choices:

0018-9340/81/1000-0800$00.75