AN OPTIMAL SCHEDULING ALGORITHM FOR INTERLEAVED MEMORIES AND THE EFFECTS ON MEMORY PERFORMANCE DUE TO DEPENDENCIES *

** Benjamin W. Wah
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
(317-493-9251)

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 can be proved to be minimum. Performance of this algorithm for non-random requests is evaluated using simulations. A pipelined processor is used as an example for the generation of non-random requests to the memories. Nonetheless, the source could have been a vector processor or a multi-processor system. The organization investigated has a common set of fixed size buffers to store conflicting requests and an intelligent scheduler that determines the order of initiation of the memory modules. The system is first evaluated with traces under the high request rate assumption, that is, the request rate is very high so that any empty buffers can be filled up immediately. A simple pipeline configuration is then establisned and the effects on the degradation on memory utilization are evaluated using simulations.

Keywords and Phrases

Access dependencies, intelligent buffers, interleaved memories, memory bandwidth, optimal scheduling algorithm, pipelined processor, simulation model.

1. Introduction

The design of large primary memory systems is becoming easier as fast, inexpensive, large scale integrated 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 bandwidth 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 sub-system is still very expensive and can be more than 50% of the total hardware cost [SCH78]. Further, with the development of high speed processors such as the CRAY-1, 1a and multi-processor 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 study the design and evaluate the performance of parallel memories for supporting computers of high access rates.

In the past ten years, there are a large number of analytical models on the performance of interleaved memories for a pipelined processor. Among them are Boland et al. [BOL67], Hellerman [HEL67], Knuth and Rao [KNU75], Burnett and Coffman [BUR70, COF71, BUR73, BUR75], Terman [TER76]. The areas studied include:

(a) Design of conflict resolution buffers to bypass a request which is directed to a busy module;

(b) Modelling of the probability distribution on the module that an access is directed to;

(c) Scheduling of requests to the parallel memories so that the memories can be utilized efficiently.

(d) Analysis of memory performance under the above assumptions.

There are several assumptions made in these previous studies that are not completely valid. First, 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 [BUR75]. 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 sequences 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.

Second, very few studies have been made to minimize the waiting time of a request to the memory. Flores [FLO64] 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 to-

* Research was supported by a summer faculty XL-grant of Purdue Research Foundation

ward the effect of interference from the input/output units and there is no queuing 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.

Third, the effect of dependencies on the memory performance is not completely clear in these studies. Two extremes of this question have been studied. In a conventional uni-processor system with no lookahead, an access cannot be issued until its predecessors have finished. In this case, there is no parallelism involved and the performance of the memory is uniquely determined. On the other hand, when there is complete lookahead so that a memory request can be issued irrespective of its predecessors, the performance of a parallel memory system is limited by the degree of conflict in the accesses and the number of conflict resolution buffers available. These two cases have been studied extensively. However, the performance of a parallel memory system when there is partial lookahead, that is, when some of the dependencies cannot be bypassed, is not studied. In this paper, we provide a simulation model to find out the degradation in memory utilization under the effects of partial lookahead.

2. Characteristics of the Access Sequence of a Pipelined Processor

In this paper, a pipelined organization in the most general sense, instead of specially structured pipeline computers with different arithmetic units (e.g. CRAY1), applications (e.g. vector processing) additional memory support (e.g. cache) and interconnections (e.g. ILLIAC IV), is assumed. The processor is assumed to be directly executing from the main memory. 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 performance of the memory, with or without a cache, even though the results are drawn for a cacheless system.

A memory access sequence generated by a pipelined processor has Class D dependencies as classified by Chang et al. [CHA77] where a dependency is a logical relationship between two addresses such that the second address cannot be accessed (written or read) until the first has been accessed. In a pipelined computer, the computational process (say an instruction) is segmented into several sub-processes 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. The pipeline segments are able to generate memory requests independent of the others and therefore the dependencies in an access stream are bypassed.

Although the dependencies are bypassed due to the use of pipelining, the accesses are not independent and the probability that a module is accessed is governed by a complex function of the word size, instruction format and the nature of the program. Furthermore, there exist cases where the effects of dependencies cannot be totally eliminated. Anderson et al. have identified three main sources of concurrency limitations which tend to reduce the performance of the pipe [AND67]. These are: (a) register interlock, (b) branching and (c) interrupts. Various methods have been introduced to solve these dependency problems [TOM67]. For example, register interlocks can be solved by using forwarding; the sequentialism due to interrupts can be improved by using imprecise interrupts as in IBM 360/91. The most predominant effect on the performance of the memory is due to branching. When a conditional or unconditional jump instruction is encountered, request supply to the memory can either be discontinued until the condition code is set and the target instruction is returned from the memory, or some future (guessed) instructions can be prepared for execution. During the time interval when no request is sent to the memory, the memory is not fully utilized.

In this paper, we use randomly generated requests and execution traces to evaluate the memory performance. We present a scheduling algorithm which can be proved to be optimal and minimizes the average completion time of a finite set of random, independent requests. We have also assumed a simple pipe configuration in order to study the effects of dependencies on the memory performance. Since the pipeline configuration assumed is simple, the performance results estimated will be an upper bound to the performance results of general pipeline computers.

3. The Organization of Primary Memory for a Pipelined Processor

We present in this section an organization of an interleaved memory system. 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 also 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 serviced by the memory are those that exist in the memory buffers only. In this assumption, the dependency effects are assumed to be totally ignored. The memory performance obtained is therefore an upper bound for the actual performance under dependencies.

(2) Each request is assumed to be an integer from 0 to $m-1$, (m is the number of memory modules), 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 assumption for semi-conductor 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) 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:

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

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

where the constant in eq. (1) has a unit of bits. Since all the modules are assumed to be identical and the word length of each module is kept constant, the objective of maximizing the bandwidth is equivalent to maximizing the average utilization of the modules.

(5) 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 sub-cycles. It is further assumed that exactly one module can be initiated to service a request at the beginning of a memory sub-cycle and it takes m sub-cycles (1 memory cycle) to service any request, that is, homogeneous service times. With this assumption, the problem of multiple buses is resolved because at most one module finishes in each sub-cycle and the system is never confronted with returning results from more than one module simultaneously. The modules are therefore clocked by the memory sub-cycles.

(6) The waiting time that a request spends in the memory is defined in terms of a waiting cycle. A <u>waiting cycle</u> is defined similar to Flores [FLO64] as the ratio of the waiting time and the memory cycle time.

In the interleaved memory organization (Figure 1), there are m memory modules; a single set of b associative buffers, B_1, B_2, ..., 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

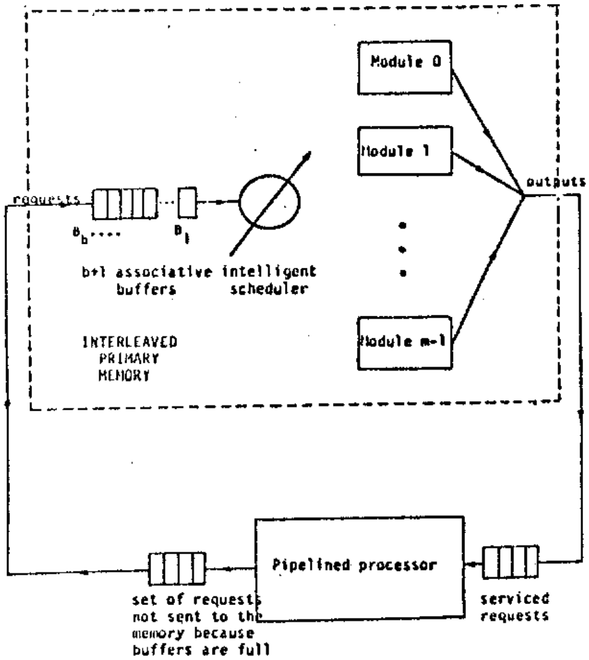

Figure 1. Organization I - An Interleaved Memory System with a Single Request Queue

cycles. One example of a staggered cycle is shown in Figure 2. The set of b associative buffers are 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_b is empty. The buffer B_b has an additional function, namely, to receive requests from the bus. Since the request rate is very high, B_b is filled immediately

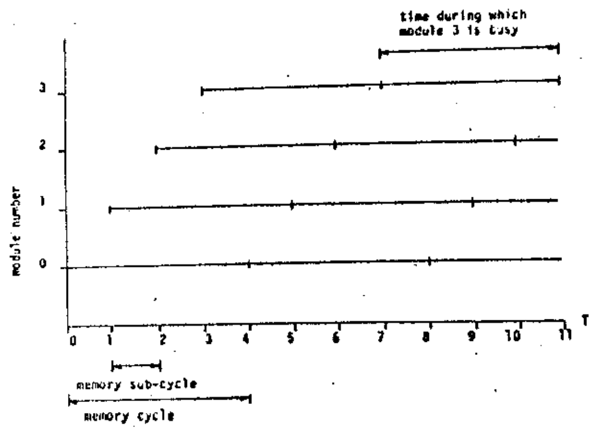

Figure 2. A Gantt Chart to illustrate the Operations of the Interleaved Memories in Staggered Cycles (m=4)

whenever it is empty. The queuing discipline for those requests in the buffers directed towards the same module is essentially First-In-First-Out (FIFO). Other queuing disciplines are not studied because only uni-processor systems are considered in this design.

The center of the control in the memory system is the intelligent scheduler. The scheduler, using a scheduling algorithm, decides at the beginning of each memory sub-cycle 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). An optimal algorithm for scheduling random, independent requests in this organization has been shown in [WAH79]. This is the Maximum-Work-Free-Module-First (MWFMF) Algorithm.

This algorithm utilizes both the information about the status of the modules and the requests in the buffers. A dynamic list of free modules is kept in the system. Conceptually, at the beginning of a memory sub-cycle, 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 at least one exists, an associative search is made on the buffers and the module with the maximum number of requests queued on it is initiated. In case of ties, only the first one is initiated,[1] (Figure 3a). The implementation of this algorithm can be done by using an additional associative memory of size m in the scheduler (Fig. 3b). 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. [RAM78]. The maximum search algorithm shown in [RAM78] 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 $\lceil \log_2(b+1) \rceil$ where $\lceil x \rceil$ is the smallest integer larger than or equal to x.

In scheduling the requests, the scheduler can examine only requests in the associative buffers. However, there is a request queue which contains requests to be serviced by the memory and this represents requests to be serviced by the memory whether they reside in the associative buffers or not. The size of the associative buffers may be greater than, equal to, or less than the number of requests in the request queue. If it is greater than the size of the associative buffers, extra requests are queued in the processor end. In a pipelined processor, memory requests can be generated continuously until a dependency occurs. At this point, the request stream is discontinued until the 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. In our assumed organization, 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. However, in a practical implementation, the pipelined processor is only able to look ahead a fixed amount of instructions and this is modelled by a fixed and finite number of associative buffers in the system (which may be greater than or less than

[1] Multiple requests can be initiated in a sub-cycle. But since the return bus can return at most one piece of datum in any sub-cycle, 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 is extremely small because of the small fraction of data writes in a program and its applicability is also limited by memory interference.

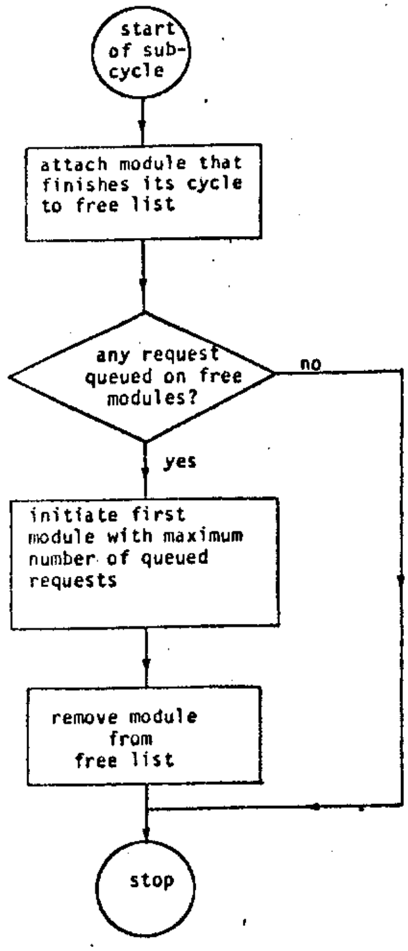

Figure 3a. MWFMF Scheduling Algorithm

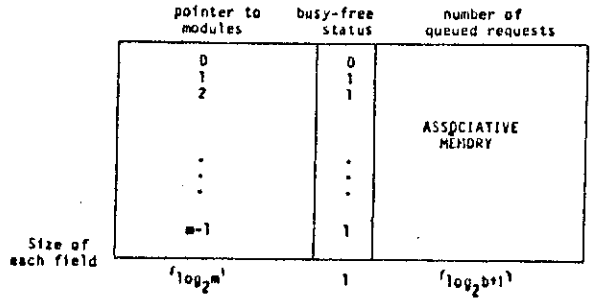

Figure 3b. Implementation of the MWFMF Scheduling Algorithm Using Associative Memory (x is the smallest integer larger than or equal to x)

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 therefore 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 organization discussed is operating in steady state. This means that the system has been operating for a long time and the initial start up effects have diminished. Further, since the queue size is limited and fixed, and the request rate from the processor is assumed to be very high. the average arrival rate must equal to the average service rate. The average arrival rate and the average waiting time are finite and satisfy Little's Formula [LIT61].

Let

- m = number of memory modules;
- b = number of buffers in organization;
- e_b = utilization of buffers B_1, B_2 ... B_b (e_b = 1 under high request rate assumption; $e_b < 1$ when dependencies are present);
- $u_{m,b}$ = expected utilization of each memory module;
- $w_{m,b}$ = expected waiting memory cycles of the request;
- M = expected number of requests in system;
- λ = expected arrival rate per memory cycle;

Then

$$M = e_b * b + u_{m,b} * m \qquad (2)$$

$$\lambda = u_{m,b} \qquad (3)$$

and they satisfy Little's Formula [LIT61],

$$M = \lambda * w_{m,b} \qquad (4)$$

The importance of Little's Formulae 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.

The organization presented in this section is used as a basis for the evaluation of the memory performance. In the next section, we present the evaluation results without the effects of dependencies. In Section 5, the performance of the memory under dependencies are also shown.

4. Evaluation of the Memory Organization

The MWFMF algorithm presented in the last section is optimal in an average sense because it minimizes the average completion time for a finite set of random, independent requests. This is shown by the following theorem.

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.

Proof

The proof of this theorem is very long and only a sketch of it is presented here [WAH79]. The theorem can be proved by induction on k, the number of requests in the request queue not including those in the associative buffers. The induction basis starts by k=0, which means that the buffers are large enough to accommodate all the requests in the request queue, and prove that the algorithm minimizes the maximum completion time for random, independent requests. This can be proved by evaluating the completion times for scheduling free modules with different number of queued requests. It is found that by scheduling a free module with a larger number of queued requests, the completion time for all the requests in the request queue is always equal to or better than scheduling a free module with a smaller number of queued requests. By adjacent pairwise interchange, it is therefore better to schedule the free module with the maximum number of queued requests. The induction hypothesis assumes that the theorem is true for a positive integer k, and the induction step proves the theorem for k+1. At k+1, the scheduler schedules a request which results in k requests in the request queue. It can be shown by using the induction hypothesis that by scheduling a free module with the maximum number of queued requests, the expected completion time for the remaining requests in the request queue is minimized.

Q.E.D.

Although Theorem 1 establishes the optimality of the MWFMF algorithm, no throughput values are obtained analytically. In fact, it is very difficult to obtain a closed form solution using queuing theory and the solution using embedded Markov Chains results in a large number of states [WAH79]. Our evaluation are therefore based on simulations using randomly generated requests and execution traces. The simulation program is written in FORTRAN and the simulations were run on a CDC6400 computer. 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 traces 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 CDC7600 and they personify program characteristics of scientific applications.

The detailed simulation results are not presented here [WAH79]. We have selected a few sets and have plotted them in Figures 4 and 5. Some observations that can be made from these figures are:

(a) The memory utilization asymptotically approaches 1 as the buffer size is increased (Figure 4).

(b) The trace driven simulation results show a higher memory utilization 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 (Figures 4, 5).

(c) The memory utilization is higher when the buffer size is increased but is smaller when the degree of interleaving increases (Figure 4). 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.

(d) Similarly, the number of waiting cycles is larger when the buffer size is increased but is smaller when the degree of interleaving increases (Figure 5).

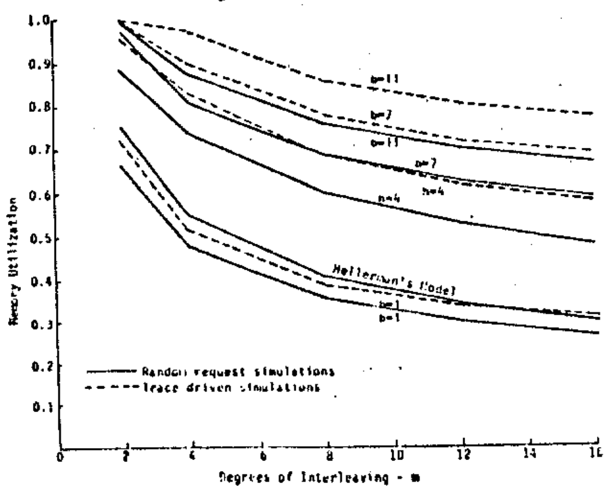

Figure 4. The Decrease of Average Memory Utilization with respect to Degrees of Interleaving for Organization I with MWFMF Algorithm

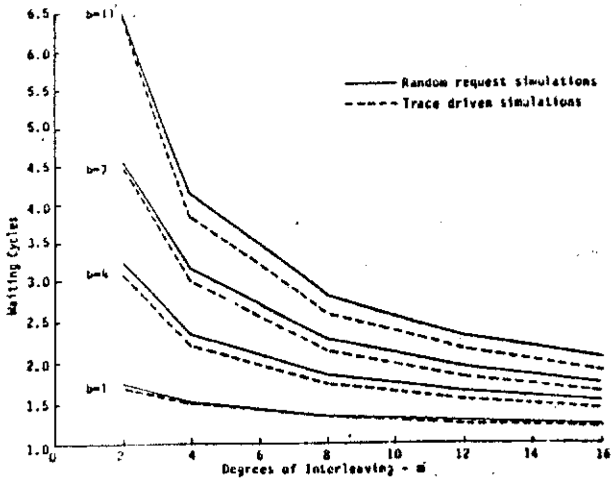

Figure 5. The Decrease of Waiting Cycles with respect to the Degrees of Interleaving for Organization I with MWFMF Scheduling Algorithm

5. Degradation in Performance Due to Dependencies

In this section, we present a simulation model to evaluate the effects of dependencies on memory utilization. Before the simulation results are presented, the instruction pipe is first characterized.

Dependent instructions issued by the instruction pipe have the following characteristics. When a conditional jump occurs, the condition code is set earlier by an instruction that may still be in the pipe. Until that instruction finishes and sets the condition code, the jump instruction cannot proceed and the memory is idle. It is assumed that the pipe prefetches from both the successful and the unsuccessful branches of the conditional jump, but does not decode the target instruction. If it is an unsuccessful jump, the pipe can proceed after the condition code is set. If it is a successful jump, the memory is idle until both the condition code is set and the prefetched target instruction is returned from the memory. An unconditional jump can be modeled as a successful conditional jump in which the condition code is available immediately. The effect of register interlocks on the memory performance is very small because they can be solved by other methods [TOM67]. Lastly, an interrupt is the same as a successful conditional jump in which the entire pipe has to be emptied. Therefore, without loss of generality, all dependencies can be represented as a successful (the jump is taken) or an unsuccessful conditional jump.

A parameter of the pipe that varies with the memory configuration is the degree of prefetch. The number of prefetched instructions should be kept as small as possible because when a conditional jump is encountered, one of the two branches is not traversed and therefore the prefetched instructions for the non-traversed branch are wasted. On the other hand, the degree of prefetch should be high enough so that the pipe can be kept busy all the time. Let

r = average number of requests generated per instruction executed;

g = number of instructions per instruction word;

f = number of prefetched memory words.

In the traces we have used, $r=0.6$ and $g=2.787$. We assume that the pipe is executing at an average speed that is the same as the memory, that is, at a rate of $\frac{u_{m,b}}{r}$ instructions per unit time. Since it takes an average time $W(= w_{m,b} * m)$ to fetch an instruction, the pipe would have executed $g * f$ instructions in this time interval at a rate of $\frac{u_{m,b}}{r}$ if no dependency occurs. Therefore

$$\frac{g * f}{u_{m,b/r}} > w_{m,b} * m$$

We set

$$f = \left\lceil \frac{w_{m,b} * u_{m,b} * m}{g * r} \right\rceil \tag{5a}$$

The value of f established here only uses the average behavior of the memory system. A high value of f would be necessary if the worst case memory parameters are used.

A parameter related to f is the pipe length. If f memory words are prefetched, it means $g * f$ instructions are prefetched and the pipe length should be $g * f$. A more accurate measure of f is to use the value before we take the ceiling. The required pipe length is:

$$L = \left\lceil \frac{w_{m,b} * u_{m,b} * m}{r} \right\rceil \tag{5b}$$

Two alternatives of the simulator are compared. The first alternative uses a pipe of length L (Eq. 5b) and assumes that instructions are fetched ahead of their corresponding operand. In this case, a priority must be associated with an instruction prefetch and an operand access to determine the one that should be sent to the memory when they are both available. A second alternative considers that the traces are made up of instruction-operand fetch pairs, that is, the corresponding operands for an instruction are always fetched after the instruction. At the beginning of a jump, since we know from the traces whether a jump is taken or not, we can generate instruction prefetches for the branch that is not taken. At the end of the dependency, the accesses for instruction-operand pairs resume. This case does not represent faithfully the access characteristic of the accesses made to the memory. However, the memory utilization results differ very little from the first alternative and the indeterminism of whether an instruction prefetch or an operand access should be generated in a sub-cycle is removed. The reason why the utilization results differ so little is because the correlation between instruction and data accesses is very small. For these reasons, we assume in our simulations that the second alternative is taken.

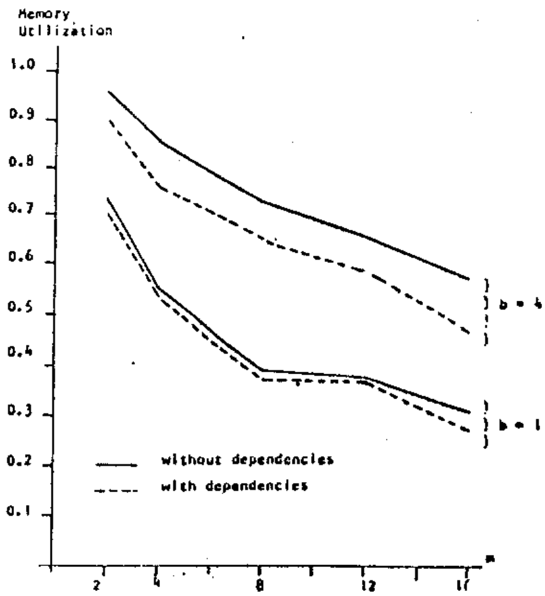

Figure 6. A Plot of the Simulation Results showing the Decrease in Memory Utilization With and Without the Effects of Dependencies (Prefetches for both successful and unsuccessful branches are included)

A simulation program is written in ASPOL with a pipe which prefetches f memory words ahead of time. The degradation in memory utilization due to dependencies is illustrated in Figure 6. It is seen that dependencies cause a degradation in memory utilization, and is more pronounced when the buffer size is large. Furthermore, the curves shown are not smooth because of the different degrees of prefetch in each case. Notice that the memory utilization, as compared with Figure 4, is higher when prefetches for both the successful and unsuccessful branches are included. This is because the addresses for the prefetches are sequential and this increases the sequentiality in the access stream.

The above evaluations only give an average value for the performance. In fact, if the memory can be utilized in some other way (e.g. for peripheral processing) when a dependency occurs, the degradation may not be so significant. The above analysis also reveals the fact that when the occurrences of dependent requests are frequent, it is not beneficial to use a pipelined computer in a batch mode. High degree of program interleaving using multiprogramming would help in reducing the degradation due to dependencies.

7. Conclusion

In this paper, we have presented an organization 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 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. 4, we have shown the performance of Hellerman's model [HEL67] together with our simulation results. Although Hellerman's model is a simple model and allows no queuing of requests, it is useful as a lower bound for the performance of other systems. It is seen that with a random request queue, Hellerman's model is better than our organization with b = 0, but is worse for b > 0. Note that the performance curves all have the same shape. The comparison with other models in the current literature is not meaningful because they differ significantly.

We have also presented a simulation model to find the degradation in memory utilization due to jumps. When a jump occurs, the request stream to the memory stops after all the prefetches have been made, until the condition code is set or the target instruction of the jump is fetched from the memory. During this time interval, the memory remains idle most of the time. The degradation in memory utilization depends very much on the configuration of the pipe and the characteristics of the access stream. A lot of techniques are available which reduce the access rate to the memory even though the reduction is not due to dependencies. It would be too restrictive to evaluate the performance for a specific pipeline computer. We have therefore developed a simple pipe organization. Since the pipeline configuration assumed is very simple, the performance results obtained will be an upper bound to the performance results of general pipeline computers.

References

[AND67] Anderson, D.W., Sparacio, F.J. and Tomasulo, R.M., "The IBM System 360 Model 91: Machine Philosophy and Instruction Handling", IBM J. of Research and Develop., Jan. 1967, pp. 8-24.

[BOL67] Boland, L.J., Granito, G.D., Marcotte, A.V., Messina, B.V., and Smith, J.W., "The IBM System/360 Model 91: Storage Systems", IBM J. of Res. and Dev., Jan. 1967, pp. 54-68.

[BUR70] Burnett, G.J. and Coffman, Jr. C.G., "A Study of Interleaved Memory Systems", Proc. AFIPS 1970 SJCC, Vol. 36, pp. 467-474, AFIPS Press, Montvale, N.J.

[BUR73] Burnett, G.J., and Coffman, Jr. E.G., "A Combinational Problem Related to Interleaved Memory Systems", JACM, 20, 1, Jan. 1973, pp. 39-45.

[BUR75] Burnett, G.J., and Coffman, Jr. E.G., "Analysis of Interleaved Memory Systems Using Blockage Buffers", CACM, Vol. 18, No. 2, Feb. 1975, pp. 91-95.

[CHA77] Chang, D.Y. et al., "On the Effective Bandwidth of Parallel Memories", IEEE Trans. on Comp., May 1977, pp. 480-490.

[COF71] Coffman, Jr., E.G., Burnett, G.J., and Snowdon, R.A., "On the Performance of Interleaved Memories with Multiple Word Bandwidths", IEEE Trans. Comp., C-20, 12, Dec. 1971, pp. 1570-1573.

[FLO64] Flores, I., "Derivation of a Waiting-Time Factor for a Multiple Bank Memory", JACM, Vol. 11, No. 3, July 1964, pp. 265-282.

[HEL67] Hellerman, H., Digital System Principles, McGraw Hill, New York, 1967, pp. 228-229.

[KNU75] Knuth, D.E., and Rao, G.S., "Activity in an Interleaved Memory", IEEE Trans. on Comp., Vol. C-24, No. 9, Sept. 1975, pp. 943-944.

[LIT61] Little, J.D.C., "A proof of the Queuing Formula $L = \lambda W$," Operations Research, Vol. 9, pp. 383-387, 1961.

[RAM78] Ramamoorthy, C.V., Turner, J.L., and Wah, B.W., "A Design of a Cellular Associative Memory for Ordered Retrieval", IEEE Trans. on Comp., Vol. C-27, No. 9, September 1978, pp. 800-815.

[SCH78] Schunemann, C. and Spruth, W.G., "Storage Hierarchy Technology and Organization", Digital Memory and Storage, W.E. Proebster, ed., Braunschweig: Wieweg, 1978.

[TER76] Terman, F.W., A Study of Interleaved Memory Systems by Trace Driven Simulation, Technical Note No. 94, Digital Systems Lab., Stanford Electronics Lab., Stanford University, Stanford, CA, 94305, Sept. 1976.

[TOM67] Tomasulo, R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM J. of Research and Develop., Jan. 1967, pp. 25-33.

[WAH79] Wah, B.W., A Systematic Approach to the Management of Data on Distributed Data Bases, Ph.D. Dissertation, University of California, Berkeley, Dec. 1979.

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

He is currently an assistant professor of electrical engineering at Purdue University, West Lafayette, Indiana. His current research interests include database machines, high performance parallel architecture, distributed computer systems and computer system performance evaluation.