Performance Analysis of Two Competing DADO PE Designs

Daniel P. Miranker
Department of Computer Science
Columbia University
New York City, N. Y. 10027
November 15, 1983

CUCS-83-83

Abstract

In parallel processing, useful computation is performed by having a number of processors computing values and communicating these results to neighboring processors. It is a crucial design issue in any parallel processing architecture to determine the optimal balance of resources for competing requirements of typical problems to be solved by the device, i.e. computation versus communication.

For example, in a highly parallel machine consisting of many individual processing elements (PE's), there is a trade off between the complexity of the constituent PE's and the number of such elements which may be embedded in a fixed silicon area. Part of the PE's circuitry must be dedicated to communication processing. Increasing the ability and the speed at which a PE can perform communication can be done, but only at the expense of the number of processors on a chip.

DADO is a large scale VLSI computer designed for the rapid execution of AI production systems. This paper analyses the nature of the instruction stream expected to be executed on DADO for production system applications, with emphasis placed on the number of processors in the machine and the size of the problem. We describe four different proposed methods of handling I/O and their queueing network models. The models were carefully simulated to determine which I/O scheme and it's resulting circuit complexity is best suited (most efficient) with respect to the DADO instruction stream.

Table of Contents

List of Figures

1 Introduction

DADO is a highly parallel computer comprising a large number of processing elements (PE's) interconnected in a complete binary tree. Communication may occur between processors along the tree edges. Under certain circumstances a processor may broadcast data to all of its descendants in the tree, or a processor may be instructed to report a byte value to all of it ancestors.

Each PE of DADO is a fully capable computer composed of an 8 bit microprocessor, a ROM resident operating system, 16K bytes of RAM, and an I/O section. Under the control of software, a PE may operate in one of two modes: *master* or *slave*. In master mode the PE runs a computer program stored in its local memory. However, embedded within the master's program are blocks of instructions that are broadcast to connected descendant PE's operating in slave mode. The slave PE's execute the broadcast instructions in parallel, in a manner similar to an array processor, or the ILLIAC IV. This type of parallelism is known as single instruction stream multiple data stream (SIMD) execution [1]. Further, the machine can be arbitrarily partitioned into a number of independent subtrees. The root of such a subtree logically disconnects itself from its parent, and becomes the master of the PE's logically connected below. This type of machine has become known as a multiple SIMD (MSIMD) architecture [5].

SIMD parallelism is typically controlled by a conventional processor. The control processor issues a stream of machine level instructions that are executed synchronously in lock step by all the slave processors in the array. DADO is different. Since each PE of DADO is a fully capable computer, and communication between PE's is generally expensive, we wish to make an instruction as "meaningful" as possible. What is communicated as an instruction in DADO is actually a pointer to a procedure, stored locally in each slave PE. Primitive SIMD DADO instructions are in fact parallel procedure calls and may be viewed as macro instructions.

For example, a common instruction that will be executed by a DADO PE is "MATCH(pattern)", where MATCH is a generalized pattern match routine local to each processor.

Transmitting entire procedure calls makes effective use of communications links but introduces a difficult problem. A procedure may behave differently depending on the local data; the same macro instruction may require different amounts of processing time in each PE. In such a device either the PE's must synchronize on every instruction and therefore potentially lay idle while the slowest PE finishes, or the PE's must be able to queue the instruction stream to possibly achieve better performance.

Two design questions need to be answered for DADO: how much circuit complexity should be used in a PE's I/O section and how much of a PE's processing power should be diverted to handle I/O. For a given size (physical) machine, the more silicon devoted to communications the less area there will be for the processors. Thus, faster I/O potentially reduces the number of available processors. The proper balance of the PE's capabilities with respect to the I/O section must therefore be established.

The independence of instruction execution times among PE's suggests that the performance analysis of DADO is analogous in many ways to the performance analysis of open queueing networks. The remainder of this paper describes careful analysis of the DADO instruction stream with respect to the size of the machine, and the size of the problem while considering four different configurations of DADO in terms of open queueing network models.

The next section will describe the nature of the DADO instruction stream. Section 3 will describe the four different proposed configurations of DADO and their respective queueing models. Section 4 will describe the performance simulation method. The results of the simulations is presented in section 5.

2 The Nature of the DADO SIMD Instruction Stream

We are primarily concerned with the behavior of the SIMD subtrees. Resident in the root of a SIMD subtree is a production or set of productions. A production system [4] is defined by a set of rules, (or productions), and a collection of dynamically changing facts, called the *working memory* (WM). A rule in a production system consists of a left hand side (LHS) and a right hand side (RHS). The LHS is a collection of pattern elements to be matched against the contents of the working memory while the RHS contains actions effecting changes in the working memory. A production system repeatedly executes the following cycle of operations:

1. Match: For each rule, compare the LHS against the current WM. Determine if the WM satisfies the LHS.

2. Select: Choose a satisfied rule according to some predefined criteria.

3. Act: Add to or delete elements from the WM as specified by the RHS of the selected rule.

An example rule using the OPS5 production system language syntax [2] is shown in figure . This rule says if there is a WM element in the system representing a message about a new job, and the job's size matches the class definition for medium size jobs, create a new WM element tagging the job with the class name medium.

```
(p categorize-job-sizes                                   ; rule name
 (message ^job <x> ^size <y> ^status new)                 ; pattern element,
                                                          ; <x>, <y>
                                                          ; are pattern variables
 (class-definition ^size <y> ^class-name medium)          ; pattern element
 -->
 (make job ^job-name <x> ^class medium)
 )
```

Figure 1: An Example Production Rule.

A copy of the relevant subset of working memory is stored in a subtree (see [6]). Within a subtree the working memory is fully distributed amongst the PE's. An effort is made to keep the number of working memory elements in each of the PE's of a subtree balanced.

During the match operation, the root PE broadcasts to all it's descendants the first pattern element that is to be matched. The slave processors then attempt to match the pattern element against their local WM. Under the direction of the root PE, those slave PE's that have successfully matched report any values they matched against pattern variables occurring in the pattern element. If there is a variable common to the first pattern element and to subsequent pattern elements the root PE then substitutes each previously reported value in place of the pattern variables in the subsequent pattern elements. The process is repeated recursively with each of the remaining pattern elements until pattern elements have been successfully matched, or until the match in that subtree fails.

A particular rule and variable bindings is selected. The details of this selection are unimportant here. The interested reader may see [2]. The RHS of the rule is then evaluated. Two types of actions are possible, adding a new working memory element, or deleting an old one. If the action is a delete from WM, the slave PE's must determine if they have a copy of the WM element to be deleted. The WM element in question is broadcast to all the slave PE's who compare the broadcast element to there own local store. If the match is successful the slave PE frees the storage. If the action is an add, one PE is selected in a subtree and the WM element is stored in that PE.

It should be apparent from the above description that a considerable amount of time is spent in communicating small amounts of data and in the storage management of the messages and WM elements. Data dependent operations are in fact infrequent, but lengthy in comparison to the others.

The DADO SIMD instruction stream can be broken into three types of instructions. The first and most common type (I) are those that handle primitive communication steps and storage management. These tend to have short service times and arrive in bursts. They also have the property that they have identical service times in all PE's.

The second type (E for exponential) of DADO SIMD instructions are the data dependent operations where intensive processing occurs, operations such as Match, and Delete. These tend to be rather lengthy and infrequent compared to the I type. The primary component of these operations is an associative probe where a PE must match a pattern element against it's local WM. The match must succeed through a number of terms within a pattern element. The likelihood of a successful match on a particular term is independent of the success of the match on the previous terms. Therefore an exponential service time distribution is a fair assumption for the E type of instructions.

The third type (S for Synchronize) of DADO SIMD instructions are those that cause the control processor to block and wait for all the PE's to finish all previously issued instructions. Examples of this type of instruction are "Report", where a byte value is communicated from a PE to it's ancestors. Also the "Resolve" operation, which is an instruction integral to the communication section of the PE's that will select a single PE from multiple responders [7].

To determine the arrival rates and service times of the three types of SIMD instructions the current implementation of the OPS5 monitor was examined. The OPS5 monitor is written in Parallel PL/M (PPL/M) [7] which is an extension of the PL/M language that Intel provides for all of its microprocessors. The extensions to PL/M are rather simple. The primary one being a new block delimiter, "DO SIMD". All code within a SIMD block is executed in parallel by the slave processors. Ten pages of PPL/M code were compiled and the resulting assembly language output was analyzed. An example of this code is illustrated in figure 1.

The original source code was examined to determine which instructions corresponded with each of the three types defined above. The types were then mapped onto the assembly language output. The lengths of each of the instructions blocks were tallied as well as the interarrival time of the blocks. The results of the tallies were used to generate basic statistics of the instruction stream. The units of time are normalized to the time to execute a typical machine level instruction in a PE, which is two microseconds in the current implementation. The statistics are summarized in figure 3.

Close examination of the code exposed an important property. The size of WM affected only the length of the E type instructions but not any of the other characteristics in the instruction stream. If twice as many WM elements are packed into a PE then the E type instructions would on average take twice as long.

3 Queuing Models of Proposed DADO Hardware

Within the currently available technology for building DADO are two orthogonal implementation issues. One is whether the PE's processor chip should directly handle low level I/O steps, or whether a distinct semicustom chip be incorporated in the PE design to handle the low level I/O.

The selected processor for DADO is the Intel 8751 single chip computer. The DADO 1 design calls for the I/O ports of each computer chip to be directly tied to the I/O ports of the computer chips of its three tree neighbors. Data is transferred along the tree edges by using a standard four cycle handshake protocol. This simple design has greatly expedited our prototyping. However, to move one data byte from one PE

Figure 2: Example PPL/M Code for Sequentially Loading *DADO*

```
 We will assume that this program is executed within
DADO's CP. The system function READSTR loads string
data into a buffer from some external source.

SEQLOAD: PROCEDURE:
  DECLARE Intelligent-record(64) BYTE SLICE EXTERNAL;
  DECLARE Not _done BYTE SLICE;
  DECLARE (Index,Length) BYTE SLICE;
  DECLARE I BYTE;
  DECLARE Buffer(64) BYTE;

  DO SIMD;
    CALL SENABLE;   ALL PE'S ARE ENABLED
    NOT _DONE = 1;   ALL SLICES INITIALIZED
    INDEX = 0;
  END;

LOADLOOP:
   pick a pe to load the next record into
  DO SIMD;
    CALL Enable;
    A1 = BOOLEAN(Not _Done);
    CALL Resolve;   Only one A1 is now set
    EN1 = A1;       Selectively disable all but one pe
    Not _Done = 0;
  END;

  IF Cprr=0 THEN      If tree is full
  DO;
    Call Writestr(.Mfull);
    RETURN;
  END;
  CALL Readstr(.Buffer,.Length);   Data provided by external source
  IF Buffer(0) =(') THEN RETURN;

  DO I= 0 TO LENGTH-1;
    CALL Broadcast(Buffer(I));
    DO SIMD;
      Intelligent _Record(Index) = A8;
      Index = Index + 1;
    END;
  END;
  DO SIMD;
    Intelligent _Record(Index)=0;
  END;
  GOTO LOADLOOP;
END SEQLOAD;
```

to the next requires twelve machine instructions. A byte broadcast to a subtree takes twelve machine instructions for each level of the tree. The simplicity of the design has allowed us to rapidly implement a 15 node DADO, but it is expected a considerable amount of the available processing power will be consumed in I/O computation.

The alternative is to build an I/O coprocessor for the DADO PE's. The coprocessor will be specifically designed for the I/O task and as a result will be very efficient. Current designs indicate that a byte may be broadcast to all PE's in a tree in less than one 8751 instruction cycle. The efficiency does not come free. The I/O coprocessor will expand the number of chips per PE from two to three. If we are constrained to build a DADO at a fixed complexity then a DADO incorporating the I/O coprocessor may contain only half as many PE's. A DADO with the I/O coprocessor and the same number of PE's will require twice as many boards and fifty percent more chips.

A key question is will the increase in communication speed overcome the loss of the half of the processors? The answer revealed in section 5 is yes.

The second major issue facing the architects of DADO, is whether or not to buffer incoming instructions. The 8751 computer has a sophisticated interrupt mechanism. All incoming communication control lines are in fact connected to the interrupt mechanism. Since all the necessary hardware support is available for buffering, it is only a modest addition to the PE's operating system code to change either of the above

Arrival Times

Instruction Type	Mean	Variance	Sample Size
I	3.34	30.3	6.7
E	41.8	107.4	5
S	18.5	97.69	10

Service Times

Instruction Type	Mean	Variance	Sample Size
I	14.28	435.7	25
E	530	5	;grows linearly with the ;number of WM elements
S:DADO1	22	0	10
S:DADO2	4	0	10

Figure 3: Statistics Characterizing the DADO Instruction Steam.

DADO configurations from polled, busy wait I/O, to asynchronous interrupt driven buffered I/O. But this apparent improvement does not come for free. To buffer the instructions requires an additional 25 instructions when both performing the I/O and when processing the instructions. (see appendix II).

There are four DADO models to be analyzed. DADO 1 refers to the configuration without the I/O coprocessor, while DADO 2 refers to that configuration with the I/O coprocessor. Figure 3 illustrates the queuing model of DADO 1 without buffering (DADO 1a). The basic execution loop for a PE is to accept an instruction, pass it to its children and then execute the instruction. At each stage of the tree there is a communication delay, after which the job is given to the PE's two children to continue with the I/O operation and to subsequently execute. The queue length for the processor is zero. If a PE is busy it's communication path is blocked. If the previous stage wishes to pass a new instruction to a blocked PE it must also block.

It has been stated that 90 percent of the time spent in interpreting a production system is spent in the match phase. On DADO this corresponds to the computations performed by a working memory subtree. For all four models it is that instructions enter from above the root of the subtree. To simplify the simulation it was assumed and that the three instruction types arrive with independent Poisson arrival rates. However, the instruction stream generator must stop while an S type of instruction is outstanding. The S type of instructions have constant service time in all PE's. The E type instructions were simulated assuming exponential service time whose mean was based on the size of WM. In reality different I type of instructions may have different service times, though an individual I instruction has the same service time in all PE's. To capture this aspect of the I type instructions required assuming that all I type instructions had constant service time equal to there means.

Figure 5 illustrates the queueing model of DADO 1 with buffering (DADO1b). The model is the same, except there is now an infinite queue feeding instructions to the processors. We can assume the queue is infinite because the synchronizing instructions (S type) from the control processor will prevent the queues from ever getting very large and the PE's will never block. However we must now add 25 units to the I/O delay time and also 25 units to the service time of the instructions.

Figure 6 illustrates the queueing model of DADO 2 without buffering (DADO 2a). The I/O propagation time through all of the I/O processors is less than a single 8751 machine cycle, so the propagation delay

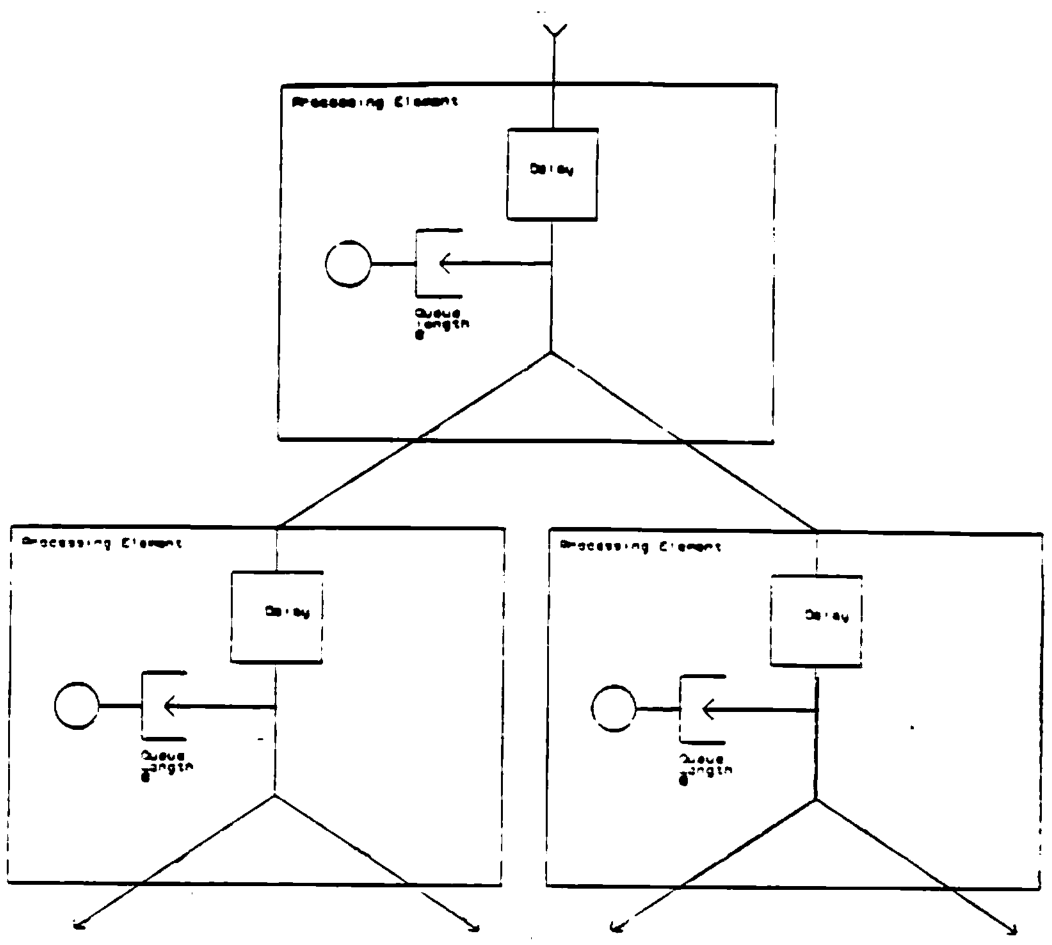

Figure 4: Queuing Model of DADO1a (no buffering).

may be ignored. An instruction is passed to all PE's in the tree at once. Not only does the additional hardware increase the speed of an I/O operation but it also has the affect of flattening the queuing model of the tree into a linear array. However since there is no buffering, the I/O section must wait for all PE's to finish executing the instruction before communicating the next instruction. Figure 7 illustrates the queueing model of DADO 2 with buffering (DADO 2b) and is analogous to DADO 1b. Instructions may queue up at the individual processors, but 25 units of time must be added to the I/O propagation time and 25 time units to the service times.

4 Analysis Method, RESQ2 queuing network simulation package

The four models above were simulated using the IBM Research Queuing Network Simulation package [3]. The package has a number of very powerful simulation primitives, including generation of job streams according to a variety of distributions, and active queues with a variety of queueing service disciplines. In addition RESQ2 provides a new construct called a passive queue.

The passive queue provides the means to introduce flow control and finite buffer lengths into a queueing model. A passive queue consists of a pool of tokens and nodes that may allocate, deallocate, create or destroy tokens. A job may only pass an allocate node if there are tokens in the pool that may be assigned to the job. Subsequently the job may release the tokens back in to the pool by passing through a release node. Tokens may also be created and destroyed when a job passes the corresponding node type. Figure 8 illustrates the use of a passive queue to model an active server with queue length of zero. By initializing

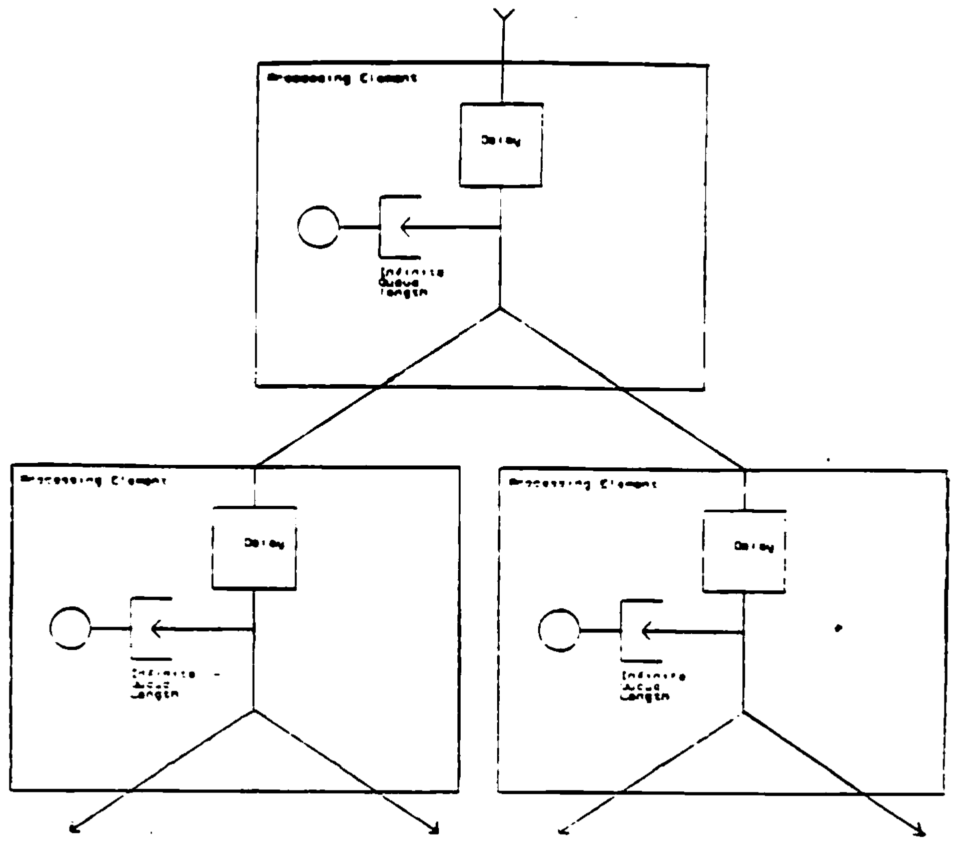

Figure 5: Queuing Model of DADO1b (buffering).

the pool with a single token, a single job causes the passive queue's allocate node to block, and jobs must naturally queue up behind the allocate node.

Another important feature of the RESQ2 package is the ability to associate variables with each individual job, and assign values to them. Subsequently these variables and the states of the passive queues may be tested to dynamically route the jobs. Extensive use of this feature was made to model the DADO instruction stream (figure 9). A generator was used to simulate Poisson arrivals of each of the three instruction types. A job variable was associated with the job identifying the instruction type. However, when a Sync instruction is issued, the control processor must block and continue only when all PE's have finished. A global variable was created (s_cnt) to represent the number of processors remaining to execute the sync job. Whenever a sync job is created s_cnt is initialized to the number of PE's. If s_cnt is zero, instructions may proceed to the PE models. If s_cnt is nonzero then all three instruction sources are directed into sinks.

The RESQ2 submodel representing a DADO 1 PE is illustrated in figure 10. The key to flow control in DADO 1 is to introduce a feedback loop where a PE creates a dummy job and routes it back to its parent. A passive queue only allows a job into the PE submodel when it contains three tokens, representing the case that each child has fed back a token indicating it has received the next instruction and the third token representing that the PE has completed the previous instruction. The first node in the submodel sends the dummy feedback jobs to a create node, adding tokens in the passive queue. Otherwise the jobs must wait to pick up the three tokens before passing through the I/O_in_delay. After the

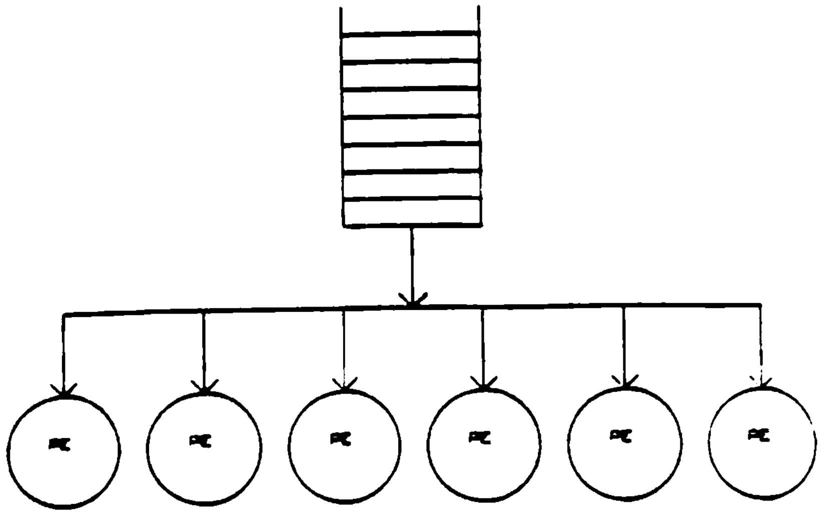

Figure 6: Queuing Model of DADO2a (no buffering).

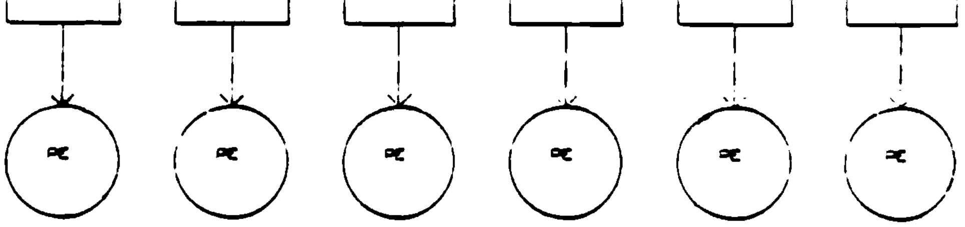

Figure 7: Queuing Model of DADO2b (buffering).

I/O_in_delay, the job is split into two copies. One copy is sent back to the PE's parent to indicate the end of the I/O while the second copy feeds the I/O_out_delay. From there three copies are made. These are directed to the two children, and the PE's processing unit. After the job is serviced in the PE, a flow token is created and added to its own passive queue. If it is a sync job, it decrements the s_cnt as well.

It should be noted that in the case of DADO 1a there is a natural pipelining effect. The upper PE's see an

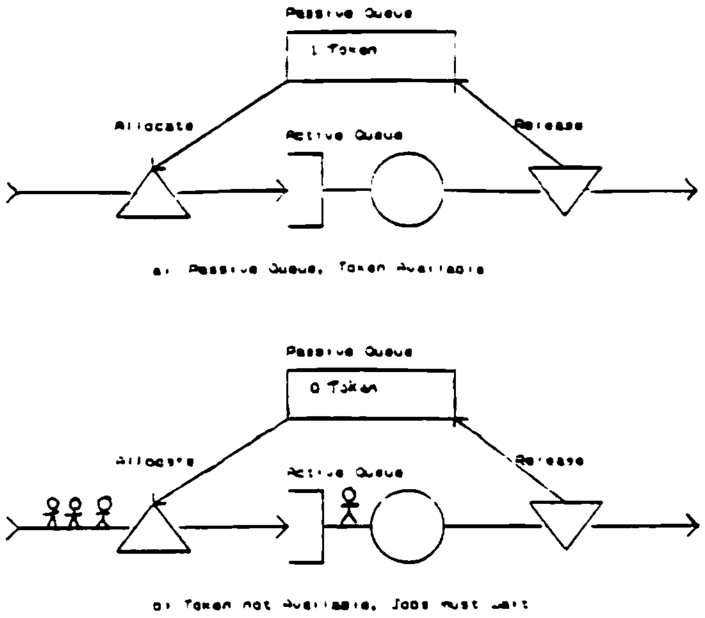

Figure 8: Passive Queue Construct.

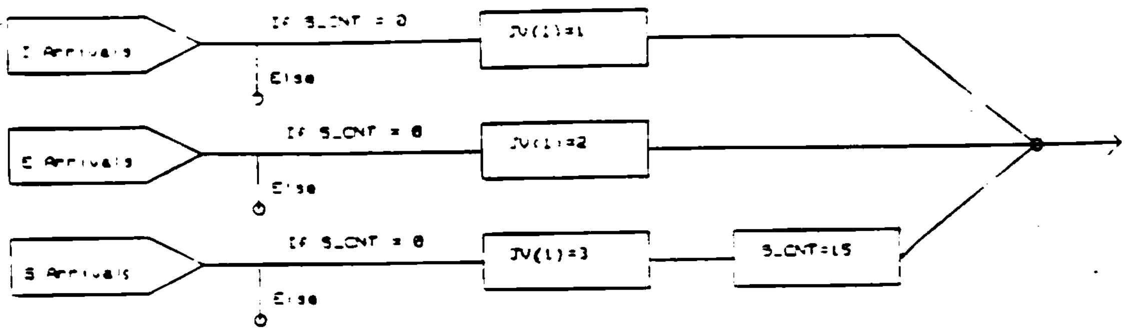

Figure 9: DADO SIMD Instruction Stream Generator Model.

instruction earlier than the lower ones, and are able to start the next instruction before the lower ones have finished. The pipelining effect improves the performance of the DADO 1 design but is a great liability whenever data must be communicated up the tree. Before data can be communicated up the tree the entire pipe of instructions down the tree must first be terminated and emptied. Only after the reported byte has reached the top of the tree will the next instruction be allowed down. Since the root of the tree is waiting to receive the reported byte, those instructions that communicate information back up that are the sync instructions. To properly model the delay required to reverse the pipe, the last sync job

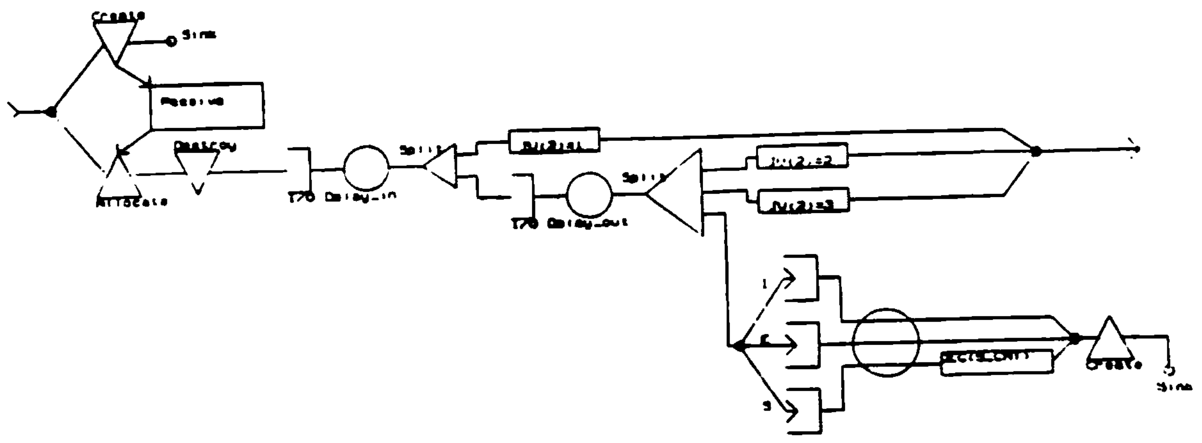

Figure 10: RESQ Queueing model of a DADO 1 PE.

is routed through an additional queue called sync_delay before the instruction stream is allowed to proceed.

In DADO 2, since the tree has been flattened to an array, the RESQ model representing a DADO 2 PE is considerably simpler (figure 11). It is only the server portion of the DADO 1 PE. A passive queue is introduced outside the PE submodel to permit only a single job into the system at a time.

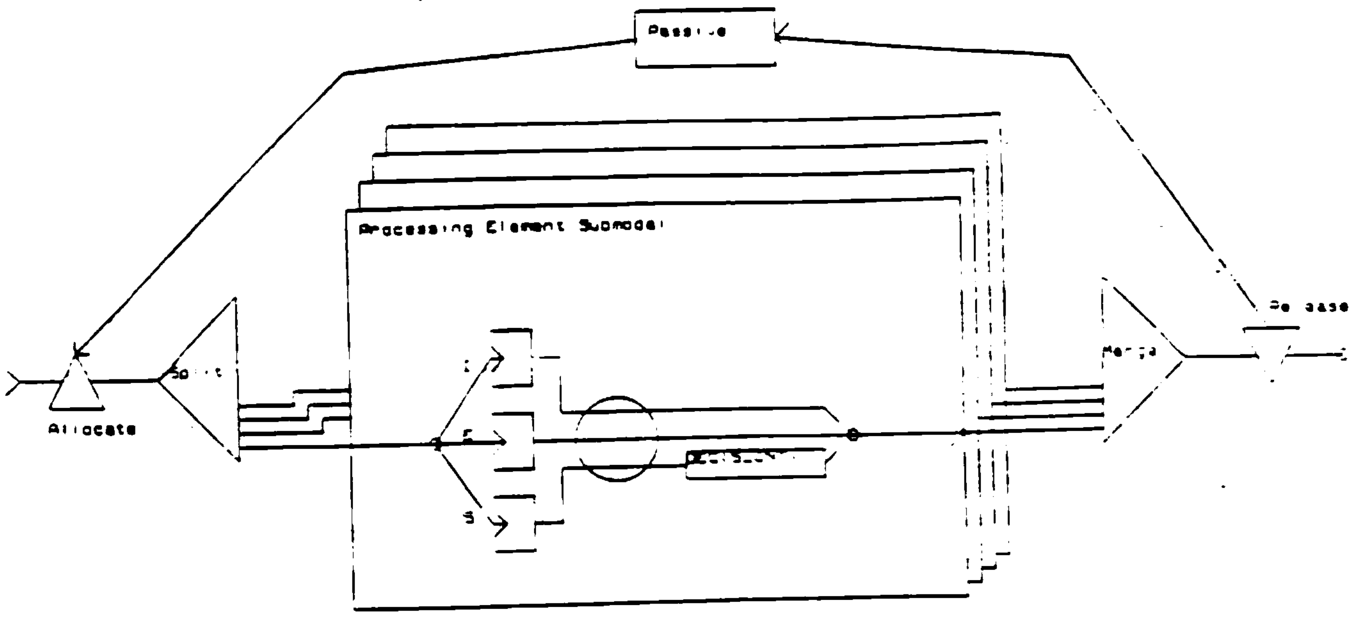

Figure 11: RESQ Queueing model of a DADO 2 PE.

Changing either of these models from the nonbuffered case to the buffered case requires the passive queues be initialized to a large number of tokens. The extra tokens will allow the simulator to queue up jobs within the PE's. The instruction stream generator has been designed to block after a sync instruction. The queues will not grow very large. The change further requires that 50 instructions for the ring buffer overhead be added to service times for the individual instructions.

5 Results

Figure 11 contains the most basic numbers derived from the simulation of a four level DADO tree. We can see that in the four level tree the I/O chip results in a 55 and 68 percent increase in speed in the cases with and without buffering, respectively. Results of adding a ring buffer are negative. In DADO 1 and DADO 2 the addition of the ring buffer reduced the the throughput by 20 and 27 percent, respectively.

The extra overhead of the ring buffer reduces from the overall speed of the machine. A closer look at the input data makes these results understandable. The majority of the instructions seen by the PE's are the I type instructions whose average service time is 14 instruction cycles. The 50 instruction cycle overhead for the ring buffer causes these I type instructions to require about 4 times as many CPU cycles. A decrease in performance of only 20 to 27 percent instead of 400 percent indicates that buffering does increase the processor utilization considerably.

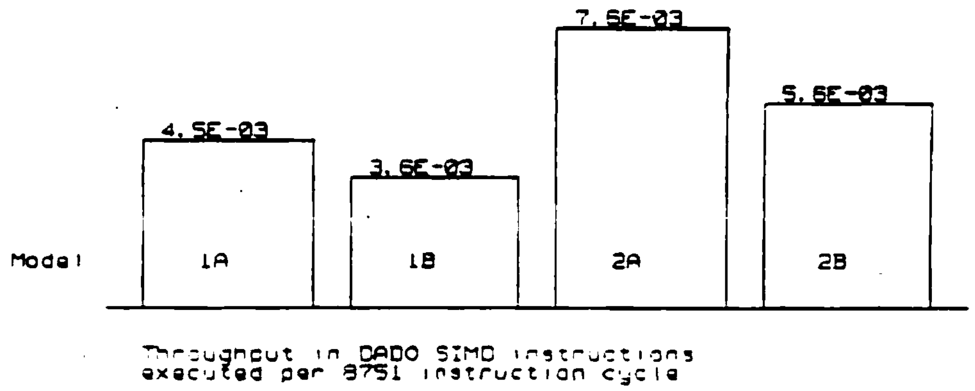

Figure 12: Throughput Results For The Four DADO Models.

Figure 13 shows the effects of changing the average size of the data dependent operations. Six plots are shown. The plots labeled 2A4 and 2A5 are throughput for a DADO 2 four and five levels deep (15 and 31 PE's), respectively. Similarly, the plots labeled 1A4 and 1A5 correspond to a DADO 1 four and five levels deep. Both configurations show a moderate decrease in performance when more PE's are added.

The important plot in figure 13 is labeled 2A4(x/2). This plot corresponds to placing twice as much working memory in the PE's of a four deep DADO as is in plot 2A4, or the same amount of working memory as is found in DADO 1A5. Assuming we are to build a DADO of fixed complexity, since a DADO 2 would incorporate an I/O chip, it would necessarily contain half as many PE's as a DADO 1. The comparison of the plots 2A4(x/2) and 1A5 indicates the relative throughput of the two machines on the same size problems. The simulations indicate that if the data dependent instructions are in fact an average of 500 cycles in length, then DADO 2 still has a 16 percent improvement in speed over DADO 1. However, the 2A4(x/2) is extrapolated in the graph and we can see that we are not far from the crossover point where it in fact becomes favorable to build a DADO 1.

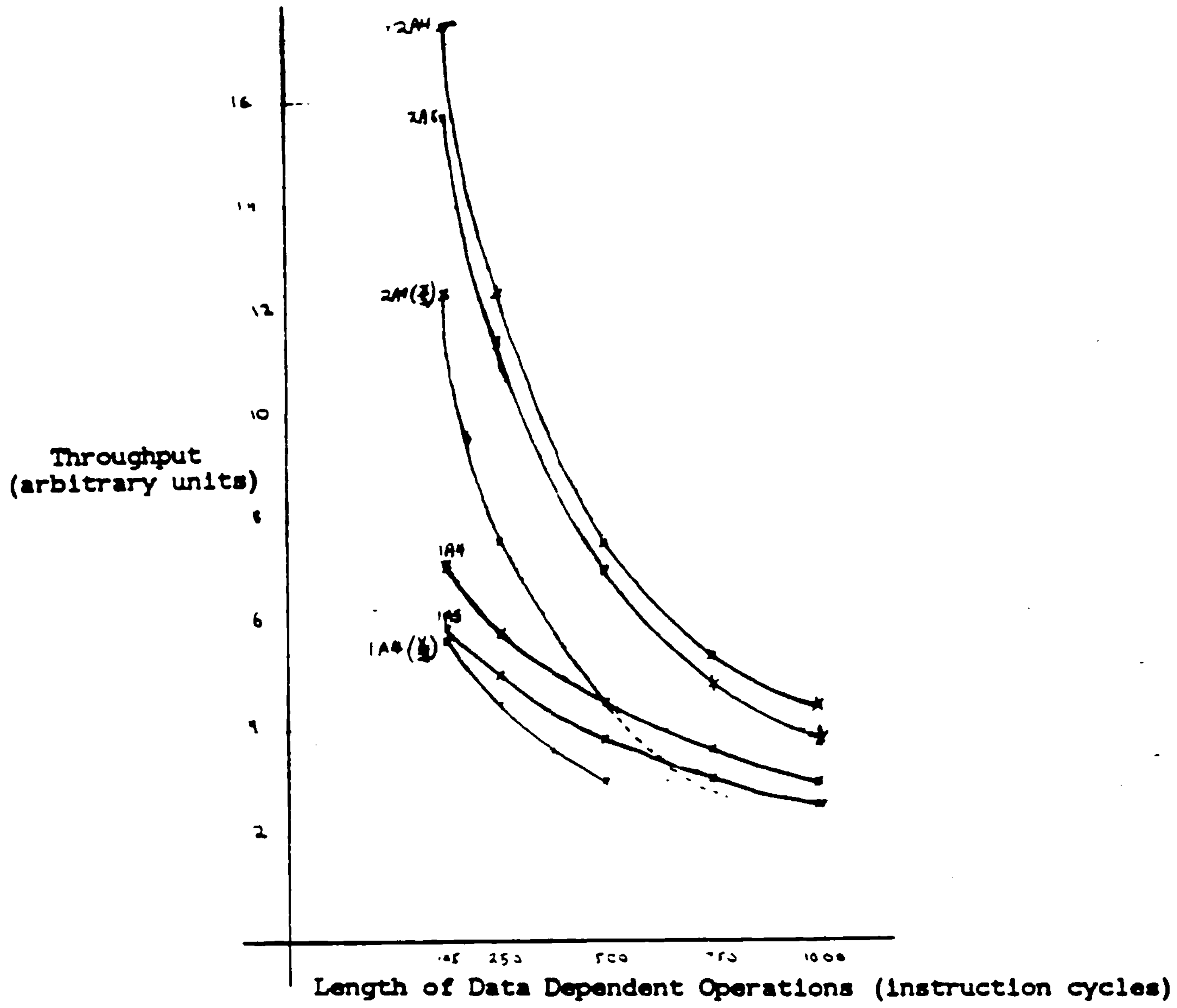

Figure 13: Throughput vs. Length of Data Dependent Operations.

The plot of 1A4(x/2) is included for comparison of the performance degradation.

A possible deficiency in the above analysis is that we are only simulating relatively small DADO machines. For this reason DADO's of size 6 deep (63 PE's) were also simulated on the expected instruction stream. Both DADO 1 and DADO 2 degrade gracefully when expanded (see figure 14).

6 Conclusions

It appears that the custom I/O coprocessor will give us substantial performance increases. The I/O technique in DADO 1 results in a slow percolation of instructions thorough the DADO tree. One might expect the flattening effect of the I/O processor to create an appreciable performance increase. However, the increase is only 65 percent. The indication is that the PE's are spending an appreciable amount of time computing, and that in DADO 1 the I/O accounts for about half of the processing time. Also the

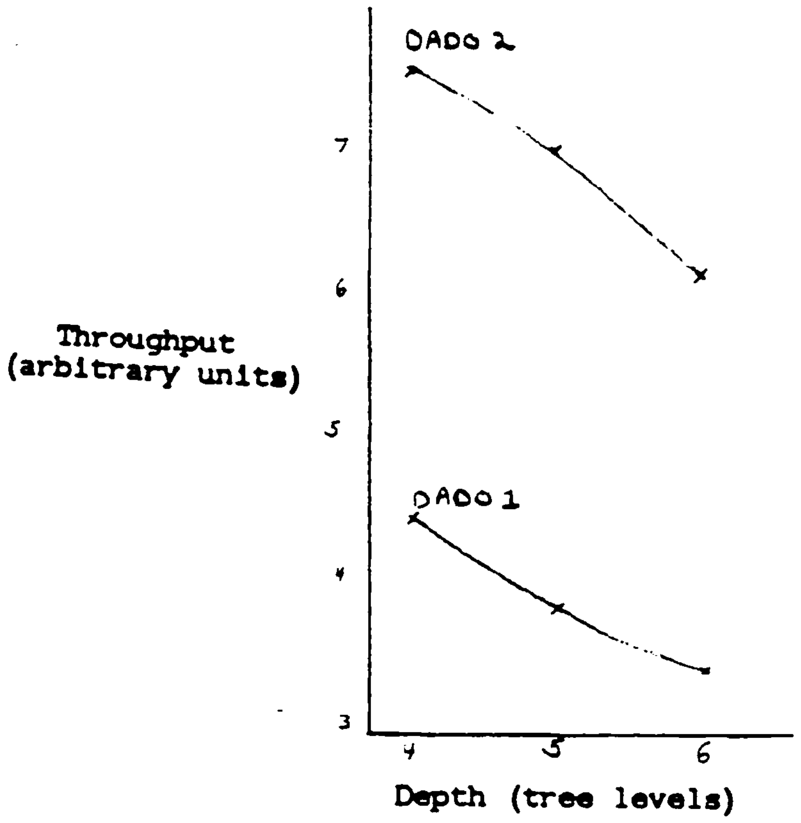

Figure 14: Performance of Deeper DADO trees.

percolation of instructions creates a natural pipelining of the DADO instruction stream producing an appreciable performance improvement. When we simulated the loading of a DADO 2 with as much working memory as would be conveniently held by a DADO 1 of twice the size we still have a performance improvement of 16 percent.

The effects of buffering the instruction stream are negative. The data dependent operations simply appear too infrequently with respect to the storage management type. The storage management instructions are burdened with the overhead of the ring buffer.

7 Acknowledgments

I would like to thank Jim Kurose whose private tutorials saved me from hours of additional toil and aggravation. Also Doug DeGroot, Ed Macnair and the IBM Corporation for the extensive use of both hardware and software facilities, and the financial support while doing this work at the Watson Research Laboratory.

8 Appendix I: Simulation Results

Model	Length of data dependent instructions(E)				
	125	250	500	750	1000
1A	6.93	5.81	4.46	3.61	3.00
1B			3.37		
2A	17.3	12.3	7.59	5.45	4.45
2B	8.56	7.03	5.59	4.24	3.43
1A5	5.92	5.06	3.83	3.16	2.60
1A6			3.38		
2A5	15.7	11.4	7.03	4.86	3.81
2A6			6.10		

9 Appendix II: Code to Determine Ring Buffer Overhead

```
/* This is a sample of code for handling a ring buffer */

/* This is the input routine, called as an interrupt  */

Byte_in:  Procedure Interrupt;
          call disable;                                  /* disable interrupts *
          if inptr = high_limit then inptr= lowlimit;   /* wrap around? */
          else inptr = inptr + 1;

          if inptr = outptr + 1 then call block;        /* buffer full?  */
          atinptr = inport;                              /* get byte */
          call enable;                                   /* enable interrupts */
          end;


/* This is the read buffer routine  */

Get_byte: procedure byte;

          if outptr = inptr - 1 the call block;     /* buffer empty? */
          if outptr = highlimit then outptr = lowlimit;  /* wrap around? */
          temp= atoutptr;
          outptr= outptr + 1;
          return(temp);
          end;
```

10 Appendix III: RESQ Model Definitions

References

[1] Flynn, M. J.
Some Computer Organizations and Their Effectiveness.
IEEE Transactions on Computers :,, 1972.

[2] Forgy,Charles L.
OPS5 User's Manual.
Technical Report Carnegie-Mellon University-CS-81-135, Department of Computer.Science, Carnegie Mellon University, July, 1981.

[3] Sauer, Charles H., MacNair, Edward A., Kurose, James F.
The Reserach Queueing Package, CMS Users Guide.
Technical Report RA 139 #41127, IBM Research Division, 12, 1982.

[4] Chase,E (editor).
Visual Information Processing.
Academic Press, 1973.

[5] Siegel.
PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition.
IEEE Transactions on Computers C-30(12):934-947, December., 1981.

[6] Stolfo, S. J. and Shaw, D. E.
DADO: A Tree-Structured Machine Architecture for Production Systems.
In *Proceedings of the National Conference on Artificial Intelligence.* American Association for Artificial Intelligence, August, 1982.

[7] Stolfo S. J., D. Miranker, and M. Lerner.
PPL/M: The Systems Level Language for Programming the DADO Machine.
Technical Report, Department of Computer Science, Columbia University, 1984.
(submitted to ACM TOPLAS).