Evaluation of the cedar memory system: configuration 16 x 16*

K. Gallivan, W. Jalby, H. Wijshoff

RUU-CS-91-16
June 1991

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Evaluation of the cedar memory system: configuration 16 x 16*

K. Gallivan, W. Jalby, H. Wijshoff

Technical Report RUU-CS-91-16
June 1991

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

EVALUATION OF THE CEDAR MEMORY SYSTEM: CONFIGURATION 16 × 16*

K. Gallivan†, W. Jalby††, H. Wijshoff‡

† Center for Supercomputing Research and Development,
University of Illinois at Urbana-Champaign, USA
†† IRISA, University of Rennes, France
‡ Department of Computer Science, Utrecht University,
the Netherlands

Abstract

In this paper we present some basic results on the performance of the Cedar multiprocessor system. Empirical results are presented on the 16 processor–16 memory bank system configuration, which show the behavior of the Cedar system under different modes of operation.

1 Cedar description

The architecture of the Cedar system is characterized by the use of cluster-based computational and memory hierarchies. The basic Cedar cluster, a modified Alliant FX/8 provides the first two levels of the computational hierarchy – vector processing within each computational element (CE) and concurrency within the cluster. The highest level of parallelism is, of course, across multiple clusters. In the 16 × 16 configuration each of the four clusters has 4 CE's, causing the useful peak performance of this system to be 34 × 4 = 136 Mflops.

The lowest level of the memory hierarchy comprises the storage private to each CE. This includes a 16 KB instruction cache and 2 KB vector register storage (eight vector registers each of which hold thirty two elements). The CE's in each cluster share a 512 KB cache (two boards of 256 KB each). These caches form the second

*This work was supported by the Department of Energy under Grant No. DE-FG02-85ER25001, the National Science Foundation under Grant No. NSF 89-20891, the NASA Ames Research Center under Grant No. NASA NCC 2-559, Cray Research Inc. and Alliant Computer Systems.

level of the hierarchy. Data at these two levels is accessible at a rate necessary to satisfy the demands for data of the pipelined functional units on all of the CE's.

The next level, cluster memory, can contain up to four memory boards per cluster. Note that this is half the number possible on a standard Alliant FX series machine. Four of the slots were required to facilitate communication with global memory. The present standard cluster memory board has a size of 8 MB, however, 32 MB boards are available from Alliant, and these may be used later to upgrade cluster memory.

The highest level of the hierarchy, global memory, is shared by all of the clusters and contains one memory module for each CE in the system. The global memory board contains one 2 MB memory module and a synchronization processor (discussed below). Therefore, the size of global memory of the 16×16 system is 16×2 MBytes = 32 MBytes. Each global memory module is pipelined and can return a 64-bit word every two cycles for a bandwidth of 2.94 MW/s. The latency for a particular memory reference within a module is four cycles.

The processors and the global memory are connected via the Global Interconnection network (see Figure 1), which comprises two unidirectional packet-switched networks. The basic component of the network is an 8-by-8 crossbar with 80-bit data path that includes a 64-bit data word. Notice that the processors have direct access to the global memory; they do not communicate through an intermediate level of memory as with the cluster cache-memory hierarchy. Since only 16 memory modules and processors are used in the four cluster system, not all of the connections possible between the two stages are necessary.

The basic transactions of the network are

- Read: address and control to global memory; address and control and data from global memory.
- Write: address and control and data to global memory; acknowledgment from global memory.
- Synchronization: can require a packet of up to four words.

Each CE communicates with the Global Interconnection network via a private Global Interface board across the cluster crossbar. In addition to providing access to the global memory, the GIB's also contain hardware to perform two crucial global memory functions: the dispatching of synchronization operations and the prefetch of global memory data. In the former case, the GIB monitors the outstanding writes to global memory from the CE issuing the synchronization instruction. When all of the acknowledgments of these writes return to the GIB the synchronization instruction proceeds and information is sent to the appropriate memory module.

The prefetch unit on each GIB provides each CE with a method of offsetting the relatively long latency of a global memory reference. The GIB contains a 4 KB prefetch data buffer where the data is stored until accessed by the CE or some

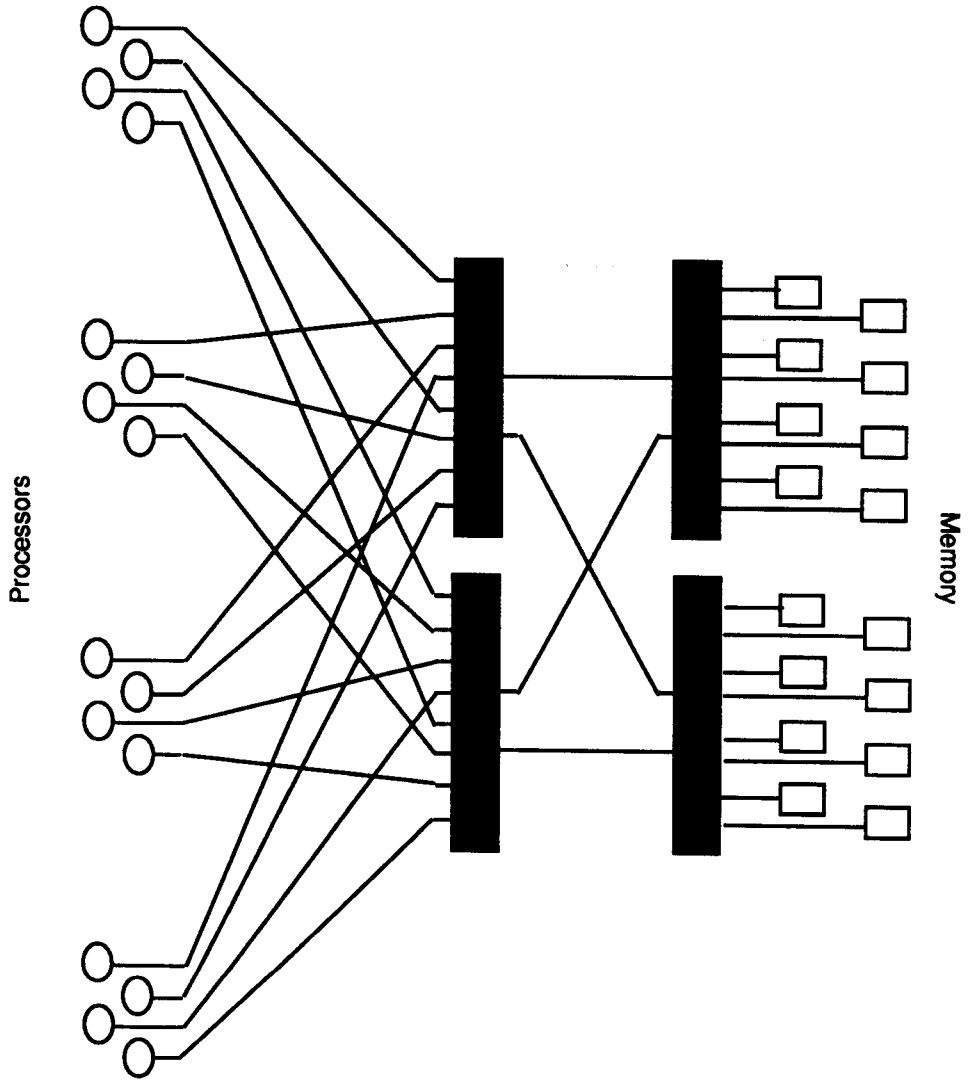

Figure 1: The Cedar interconnection network

other action invalidates its data. This buffer is direct-mapped within a page and has a valid/invalid bit per location. It also contains other memory-mapped registers including: length register, stride register (in bytes), address register and a 32-bit mask register. Two instructions control the operation of the prefetch. The first, *ldstprf*, is a microcoded instruction that loads an address and starts the prefetch. It is assumed, of course, that the other appropriate registers have been loaded. The second is a memory mapped instruction that turns the prefetch unit off. This is done by clearing memory location __CEDAR$PO__.

The prefetching of data can be started in two ways. The *ldstprf* starts it explicitly. It is also possible to start it implicitly by loading all of the registers appropriately and issuing a *vmove* from global memory. Setting the prefetch length register causes the implicit arming of the prefetch unit, so it should be the last register loaded. When the next vector access of global memory occurs, the address is automatically loaded in the prefetch address register and the prefetching begins.

One complicating factor about the prefetch unit is that it works with physical addresses only. This is a result of it being detached from the CE, which performs the virtual to physical translation. As a result, when the prefetch unit crosses a page boundary it must stop until it sees the physical address of the next element in the vector. This address is issued by the CE when the data element is accessed by the CE and therefore it is not really prefetched at all. Subsequent elements in the vector on the same page are then prefetched. When a restart of the prefetch unit occurs, the data which is contained in the prefetch buffer is invalidated in preparation to receive elements from the new page. This is necessary since the buffer is direct-mapped within a page. Therefore, in order for prefetch to be useful when page boundaries are crossed, it is important to copy data in the prefetch buffer to registers or cluster memory.

Another factor that must be considered when assessing the usefulness of the prefetch buffer is that it is not saved at context switches. When your task is suspended due to the end of the time-slice or page faults, the contents of the prefetch buffer is lost.

The improvement in performance realized by using the prefetch unit can be significant. One CE fetching a vector of length 32 with unit stride (assuming no other CE is prefetching and the presence of more than one memory module) requires about 43 cycles – 12 for the first word to return and 1 cycle for each subsequent word. The same operation without prefetch requires about 208 cycles for a speedup of 4.8. Of course, in practice various other considerations such as contention and strides can affect the bandwidth achieved.

At this point it is possible to describe the physical memory space as seen from a particular CE. The 32-bit address allows access to

- 2 GB (31-bits) of cluster memory shared with CE's in the same cluster.
- 2 GB - 8 KB of global memory shared with all CE's.

- 8 KB (two pages) of special memory locations private to each CE and located on the CE's GIB. These locations include:
  - prefetch buffer and prefetch registers,
  - synchronization unit locations, and
  - locations for memory-mapped instructions.

Synchronization is provided in hardware at the cluster and global level. Test-and-set atomic operations are supported in both levels, enabling users to write their own busy-wait synchronization routines. The concurrency control bus provides low-cost synchronization for parallel loops within a cluster. A concurrent loop involving all of the CE's in a particular cluster can be started in less than 10 cycles, and a CE can get its next iteration of the loop in less than 5 cycles. The assignment of iterations allows iteration j to be given to a CE only after all iterations $i < j$ have been assigned but not necessarily completed. There are eight synchronization registers shared by the CE's within a cluster which can be used to pass synchronization from an iteration to an iteration with higher index – *advance* and *await* synchronization. These registers allow the higher iteration to continue within 5 cycles of the posting by the lower iteration.

The synchronization processor on each memory board in cooperation with the GIB's of each CE provide synchronization at the global memory level. Its capabilities are related to the synchronization primitives discussed in [3]. The synchronization primitives operate on a (key, data) pair in global memory. The primitive requires the specification of a logical test and a test-key, if the test requires two operands. If the test succeeds, an operation is performed on the (key, data) pair. Some of these operations require the user to supply another operand here denoted key-2. The operations supported include

- read key field of pair,
- write key-2 into key field,
- add/subtract 19-bit constant to key field,
- add/subtract key-2,
- logical operations on key field and key-2,
- the result field of the operation can be read before or after the operation is performed,
- and a read or write of the data field may be done.

2 Description of experiments

2.1 The LOAD/STORE kernels

The memory system has been explored by a generalization to the Cedar architecture of earlier work done within a cluster for the purposes of characterization and performance prediction, [1, 2]. This approach makes use of a set of parameterized memory access kernels informally referred to as the LOAD/STORE kernels. In order to predict performance on Cedar these memory system kernels must be augmented with a similar set of kernels which isolate the effect of the control constructs available on Cedar.

The kernels were built in order to investigate the behavior of the memory system stressed by various memory request streams. They were parameterized in a manner which respects the following constraints:

- It must be possible to adjust the set of parameters in such a way to emulate memory requests sequences arising from real codes.
- The parameters have to be chosen so they can be varied independently in order to analyze precisely the behavior of the memory system and its interaction with the memory request stream.

Our study is restricted to a steady state analysis, i.e., each CE loops around the same piece of code (which by construction will have exactly the same pattern of memory requests) a large number of times. The main reasons for concentrating on steady state analysis were to limit the number of possible parameters affecting the behavior and to reduce the number of cases which are pathologically difficult to analyze. There are ways to approximate the effect of transient behavior on Cedar for performance prediction but they are beyond the scope of this paper.

The main parameters that were varied during the experiments are:

1. Number of CE's and Clusters
2. Mode of request on each CE: scalar, vector without prefetch, vector block prefetch (implicit and explicit)
3. Type and pattern of requests: various LOAD, STORE combinations such as, LOAD, STORE, LOAD-STORE, LOAD-LOAD-STORE, etc.
4. Temporal distribution of requests
5. Spatial distribution of requests: stride and offset
6. Scheduling

Let us examine in turn each of these parameters and their potential effect on the memory system behavior.

The number of CE's and Clusters issuing requests is the most obvious parameter to vary the workload imposed on the memory. However, it should be noted, that a priori for a same number of CE's requesting, the partitioning of the active CE's across the clusters may have a nonnegligible impact. For example given 4 CE's active, the behavior maybe different if the 4 CE's belong to the same cluster or if they are evenly distributed across the clusters.

The mode of request obviously affects the issue rate of the requests but it also alters the way they are handled. In scalar mode and vector mode without prefetch, the processor can have at most 2 outstanding requests to global memory pending at any time. Since vector mode implies that a series of independent fetches are required 2 outstanding requests are maintained for the duration of the vector instruction. In scalar mode, however, the number of outstanding requests maintained over a series of instructions depends upon resource dependencies. Each request may have to pay the full cost of latency.

When prefetch is used the prefetch unit can issue and handle several outstanding requests to global memory. The number of outstanding requests allowed by the prefetch unit is controlled by length of the prefetch block, *bpl*, which can range from 1 byte to 512 64-bit words. The prefetch unit can be used in two modes: implicit and explicit. In both modes, a burst of *bpl* requests is emitted by the prefetch unit at a rate of 1 request per 1.5 cycle. In implicit mode for a LOAD, the CE then attempts to transfer elements, in order, from the prefetch buffer into vector registers. If the next data element in order has not arrived in the prefetch buffer the CE stalls until the data is available. (Data returning from global memory, however, is loaded into the prefetch buffer any order by the prefetch unit.) The implicit mode, therefore, is effectively an accelerated global memory vector load instruction.

In explicit mode, the CE is allowed to continue executing any instructions following the prefetch start which does not involve the prefetch unit, e.g., operations involving registers or cluster memory. At some point, however, the CE will attempt to access the data that was prefetched and it will then behave in a manner identical to the implicit prefetch.

The type of request, LOAD or STORE, affects the significance of the mode of request. In the case of loads, the discussion above applies. On the other hand, requests for writes are emitted as fast as the processing of the particular instruction allows, e.g., every cycle for a vector write. The CE does not wait for an acknowledgment of a write completion. The GIB/CE pair has an explicit instruction, that is used for synchronization purposes, which stalls the CE until all outstanding write requests for the CE have completed. In this paper, we will will concentrate LOAD requests.

The pattern of memory request is intended to study the effect of interleaving vector LOADS and STORES. The reason for studying the mixing the types of requests is that they result in different traffic patterns on the forward and reverse

networks. In the case of a LOAD (STORE) request, a packet of one word (two words) traverses the forward network and a packet of two words (one word) travels back from memory across the reverse network. This asymmetry may generate a difference in performance between a long sequence of LOADS followed by a sequence of STORES and a sequence interleaving LOADS and STORES. The first sequence heavily loads the reverse network initially, during the LOAD sequence, then the forward network, during the STORE sequence. The second sequence achieves a better temporal balance on both networks.

The temporal distribution mainly refers to the variation in issue request rate. This is achieved in two ways. Inside a block prefetch request (implicit or explicit), the use of different vector mask values allows us to emulate various distributions. For example, a mask set to the value 010101 will in be in fact equivalent to issuing a request every other cycle. More complex patterns allow the generation small bursts of requests.such as 11110000 The insertion of a variable number of null operations, NOPS, between the prefetch blocks allows us to vary the distribution at a higher level. (This level is more useful from a performance prediction point of view).

The spatial distribution essentially covers the way the banks are addressed. The simplest parameter is the stride. It affects the order in which the memory banks are accessed as well as the number of distinct memory banks accessed. For example, assuming that every processor starts in the same bank (0), striding by 2 will concentrate all the requests to the even numbered banks. Another parameter used to affect the spatial distribution is the offset. This parameter selects the bank in which each processor starts its requests and is typically a function of the processor number, i.e., it depends upon which cluster a CE is in and its local CE number $0 \leq p \leq 3$. As is seen below, the careful selection of these parameters can significantly affect the bandwidth from global memory.

Since our experimental templates use loops as the basic control construct, the iteration scheduling also plays a key role. Two types of scheduling have been studied: self scheduling in which the iterations of the loop are dynamically allocated to each processor, and static scheduling where the iterations assigned to a given processor are determined a priori. In this last scheme, the number of iterations is equally distributed among the processors. Therefore any load imbalance with that scheme will allow to detect asymmetries in the behavior of each processor. This is particularly significant in networks where conflict arbitration is based on processor numbers. The results below are all from tests which used self-scheduling.

2.2 Description of a basic LOAD/STORE kernel

The code which implements the vector-concurrent prefetch version of the LOAD kernel is typical. The other forms are simple modifications. (The crucial portions of the kernels are implemented in assembler but are given below in a high-level language.)

The code comprises several nested loops. The outermost loop distribute the work

Table 1: Parameters for experiments.

N_i	512
S_i	512
S_{bf}	32, 64, 128, 256, 512
I_{nop}	0, 128, 256, 384, 512, 640, 768, 896, 1024 1536, 2048, 2560, 3072, 3584, 4096

among the clusters: each cluster will run exactly the same code. (This loop has been implemented to minimize as much as possible the associated overhead.) Inside each cluster, a loop over N_i iterations is executed where each iteration consists of a CE performing S_i memory accesses as a series of prefetches with block size S_{bf}. The temporal distribution of access is controlled by the insertion of NOPS at key points in the iteration. The values of the parameters used in the experiments is given in Table 1.

```
        Loop over clusters
            DO i = 1, N_i Parallel loop over iterations
            (self-scheduled within a cluster)
P1a:            Prologue (executed only once on each CE)
P1b:                Preparation of the iteration
P2:                 Execution of I_nop NOPS
                    DO j = 1, ⌈S_i/S_bf⌉ Loop over block fetches (sequential on a CE)
P3:                     Enabling a block fetch of S_bf words
                        DO k = 1, ⌈S_bf/32⌉
P4:                         Vector LOAD of 32 elements
                        ENDDO
                    ENDDO
            ENDDO
        End loop
```

Figure 2: Basic LOAD template code.

The code for the basic LOAD kernel is shown in Figure 2.2 The loop is decomposed itself into 5 major segments:

1. $P1a$: This code corresponds to the setup of the loop on a cluster. It is executed once by each CE at the beginning of the loop.

2. $P1b$: This phase covers the loading of all the parameters necessary to the execution of an iteration on one CE. It is executed on each iteration.

```
      Loop over clusters
          DO i = 1, N_i Loop over iterations
          (self-scheduled within a cluster)
P1a:          Prologue (executed only once on each CE)
P1b:              Preparation of the iteration
                  NB = ⌈S_i/S_bf⌉
                  DO j = 1, NB Loop over block fetches (sequential on a CE)
P2:                   Enable and start a block prefetch of S_bf words
P3:                   Execution of I_nop/NB NOPS
                      DO k = 1, ⌈S_bf/32⌉
P4                        Vector LOAD of 32 elements
                      ENDDO
                  ENDDO
          ENDDO
      End loop
```

Figure 3: The explicit prefetch template code.

3. $P2$: this section of code consists of a sequence of I_{nop} NOP instructions to control the temporal distribution.

4. $P3$: The block prefetch mode is enabled here, allowing the CE/GIB to perform memory requests in blocks of S_{bf} words.

5. $P4$: This section of code loads $N_{bf}\lceil S_{bf}/32\rceil$ blocks into the vector register from the prefetch buffer. It is needed because of the vector register length of 32.

Note that the above described basic kernel issues prefetches implicitly. In the next section we will describe how this kernel can be modified to issue explicit prefetch instructions together with other modifications to this basic kernel.

2.3 Variants of the LOAD kernels

The basic LOAD kernel is easily altered to generate the variants needed to study Cedar. The modifications needed to perform explicit instead of implicit prefetches can be obtained by distributing the NOP instructions at the iteration block level in the implicit form evenly among the prefetch blocks, see Figure 3. For the purpose of comparison, care must be taken to keep the total amount of NOP instructions to be equal in both the implicit prefetch code and the explicit prefetch code. This kernel allows us to simulate masking latency with CE activity not involving accesses to global memory.

Other modifications are obtained by making the following changes to the basic LOAD kernel.

- The STORE kernels are simply obtained by replacing the LOAD instruction by a STORE instruction. Combinations of LOAD and STORE instructions are obtained likewise.
- The scalar kernels are derived by substituting a sequence of scalar loads in place of the vector LOAD/STORE instruction.
- The vector LOADS with no prefetch were generated by suppressing the instructions which enable the prefetch buffer.
- Variation in the temporal distribution was obtained by varying the I_{nop} parameter.
- Variation in the spatial distribution were generated by assigning to the vector increment registers different values and changing the starting address of the vector request.

2.4 Control and Synchronization Kernels

The synchronization instructions appropriate for large and medium grain parallelism are:

- `event_post/event_wait`,
- `lockon/lockoff`,
- `set_qlock/clear_qlock`.

These instructions all may involve activity on the operating system level. They do not exploit the capabilities of the Cedar synchronization processor except for the global memory `test_and_set` instruction.

The `event_post/event_wait` and the `lockon/lockoff` always involve action by the operating system. When a task wait for an event or attempts to lock a lock variable, the status of the task changes from ready to not–ready. Upon the posting of the required event the synchronization routine causes the operating system to reclassify all of the tasks waiting for the event from not–ready to ready status. In the case of a lockoff call releasing a lock variable, the synchronization routine reclassifies at most one of the waiting tasks. The synchronization routine does not make use of a cross–cluster interrupt. As a result, even after a synchronization routine has reclassified a task to ready–status, the task may still have to wait a considerable period of time before a cluster's scheduler process polls the global ready queue and restarts the task.

At the other extreme, a busy–waiting approach could be used to avoid the involvement of the operating system. However, this has the potential disadvantage of the waiting tasks needlessly consuming many cycles and perhaps preventing other

tasks from performing useful work. The `set_qlock/clear_qlock` routines attempt a compromise approach. The routine `set_qlock` first attempts a busy wait for a short period of time or test other conditions before involving the operating system. The operating system executes a **DAWDLE** operation. The **DAWDLE** operation does not change the status of the waiting task, rather it places it at the end of the ready queue.

The kernels which were used for the synchronization experiments are all variations of the following form:

DO $i = 1, N$ Loop to ensure correct timings
  CALL helper–task (Start up a second Task)
    Synchronize both clusters
    for initial alignment of both tasks
    Get Time-stamp
      DO $j = 1, Konst$
        CALL lockoff(alll)
        CALL lockon (blll)
        Wait for x milliseconds
        (no memory activity involved)
      ENDDO
  ENDDO
End loop

The second task (helper–task) executes the following program:

DO $i = 1, N$ Loop to ensure correct timings
    Synchronize both clusters
    for initial alignment of both tasks
      DO $j = 1, Konst$
        CALL lockon (alll)
        CALL lockoff(blll)
        Wait for x milliseconds
        (no memory activity involved)
      ENDDO
  ENDDO
End loop

3 Experimental Results

In this section, we present selected results of the global memory experiments and give some analysis of the behavior of the system as a function of the various parameters.

3.1 Basic performance limits

Key considerations in interpreting the results below are the underlying performance limits implied by the components of the memory system and the experimental set up. In this section we summarize these limits.

The CE's within each cluster are vector processors which can, in principle, execute two floating point operations per cycle. (A cycle is 170 ns.) This implies a peak execution rate of 11.76 Mflops per CE. If one takes into account the startup of the chained vector instructions which perform the computations the effective peak rate drops to 8.56 Mflops per CE. In practice, a reasonable rule of thumb is that 28 to 35 Mflops per cluster is the most one can expect. Since each CE has one port to the cluster memory system, a cluster cache is designed to have a hardware limiting access rate of 8 64-bit words per cycle or 47 MW/s. For the 16 CE configuration we have only 4 CE's per cluster and therefore the hardware limit on the cluster-cache/4-CE combination is 23.5 MW/s. The cluster memory has a hardware limiting access rate of 4 words per cycle or 23.5 MW/s.

Each global memory bank can deliver 1 word every two cycles for a rate of 2.94 MW/s per bank. A network link can transfer two words per cycle. However, since on a read (write) an address/data two-word pair must be transferred in the reverse (forward) network, a link can be thought of as effectively transferring 1 word per cycle or 5.94 MW/s per link. The 4 interstage network links in each direction therefore are the bottleneck for this configuration. The GIB/CE pair operating in prefetch mode can issue a request to the network and absorb a returning data word from the network in 1.5 cycles for a potential demand of 3.92 MW/s per CE. When prefetch is not used the performance is limited by the number of outstanding requests to memory a CE allows (presently 2). A cost of roundtrip latency of about 13 cycles is paid per pair of words yielding approximately 1 MW/s per CE.

Table 2 summarizes the hardware limits for each of the components of the memory system and the aggregate limit for a 16 CE/memory bank Cedar configuration. Clearly, when all CE's are accessing data only the cluster cache level can satisfy the data demands of the vector functional units. Of course, if a very small number of CE's are accessing data then global memory with prefetch can almost satisfy the data demand since the GIB is designed to issue requests at the CE peak rate.

Table 3 shows the relative performance degradation of each level of the memory hierarchy with the cluster cache level taken as 1. The hardware rates indicate that the most remote level of the hierarchy is 6 times farther than the cache. They also predict that the potential improvement by using the prefetch unit to offset latency does not bring global memory in line with the cluster memory access rate due to the network link bottleneck.

Component	Unit rate	Aggregate rate
Cluster cache/4-CE	23.5	94
Cluster memory/4-CE	23.5	94
Network link	5.94	23.8
Memory bank	2.94	47
GIB/CE		
w/o prefetch	$\approx$1	$\approx$ 16
w/ prefetch	3.92	62.7

Table 2: Hardware limiting access rates in MW/s.

Component	Slowdown
Cluster cache/4-CE	1
Cluster memory/4-CE	1
Global memory	
w/ prefetch	4
w/o prefetch	$\approx$ 6

Table 3: Relative performance degradation of memory hierarchy based on hardware rates.

3.2 Basic memory performance.

The performance of the cluster memory hierarchy has been characterized in detail and an attendant performance prediction strategy developed elsewhere, [1, 2], and will not be repeated in detail here. As expected, the performance varies considerably for each of the parameters listed above. For comparison purposes, Table 4 contains the basic performance of a single cluster memory system for a stride 1 access using an iteration block of 512 (as is used for the global memory experiments) and 0 NOPS. Note that this implies an aggregate cluster memory vector read bandwidth of approximately 37 MW/s which is a factor of 2.5 below the hardware limit.

Table 5 shows the basic global memory vector read bandwidths for various numbers of CEs distributed across different numbers of clusters. No prefetching is used. The results are the averaged values over several observations using 0 nops per iteration block of 512 writes. problem. Note that the 16 CE performance comes close to the hardware limit of 16 MW/s. The basic performance for scalar reads are shown in Table 6. Note for both of these modes of operation there is almost no contention.

Table 7 shows the basic global memory vector read bandwidths, with implicit prefetch turned on, for various numbers of CEs distributed across different numbers of clusters. The results are the averaged values over several observations using 0

Type	Processors	
	1	4
Vector read	2.3	9.3
Vector write	2.0	5.6
Scalar read	1.2	5.3

Table 4: Basic cluster memory bandwidths in MW/s.

Total CEs	Clusters	Bandwidth
1	1	0.98 (1.0)
2	2	1.92 (1.9)
3	3	2.86 (2.9)
4	1	3.95 (4.0)
4	4	3.81 (3.9)
8	2	7.68 (7.8)
12	3	10.8 (11.0)
16	4	13.4 (13.7)

Table 5: Global memory vector read w/o prefetch bandwidths and speedups.

nops and a prefetch block length of 32 and 512 and an implicit prefetch strategy. Here we see that the bandwidth is clearly limited by the maximum throughput of the network links (23.8 MW/s). Actually we see that the performance speedup starts really breaking down at 8 CE's (speedup = 5.4), and going from 8 CE's to 16 CE's the speedup only increases to 6.1. Note that overall the aggregate cluster bandwidth is faster than the global bandwidth by a factor of slightly more than 2.

Table 8 shows the basic global memory vector stores bandwidths for various numbers of CE's distributed across different numbers of clusters. The results are the averaged values over several observations using 0 nops per iteration block of 512 writes. No dummy test and set was performed each iteration to check for completion of outstanding writes. Clearly, the vector stores outperform the vector loads and very quickly approach the hardware limit of the 4 network links. As a result, there is very little increase in speedup when going from 8 to 16 CE's.

3.3 Global memory prefetch performance

Figures 4 and 5 show the implicit prefetch bandwidth with different prefetch block sizes as a function of the sparsity of access implied by the NOP values, for 1 and 16 CE's respectively. The sparsity, ρ, is NOPS/512 and measures the number of extra

Total CEs	Clusters	Bandwidth
1	1	0.71 (1.0)
2	2	1.39 (1.9)
3	3	2.08 (2.9)
4	1	2.85 (4.0)
4	4	2.77 (3.9)
8	2	5.48 (7.7)
12	3	8.07 (11.4)
16	4	10.22 (14.4)

Table 6: Global memory scalar read bandwidths and speedups.

Total CEs	Clusters	pf=32	pf=512
1	1	3.00 (1.0)	3.41 (1.1)
2	2	5.54 (1.8)	6.20 (2.1)
3	3	7.39 (2.5)	8.10 (2.7)
4	1	11.04 (3.7)	12.05 (4.0)
4	4	8.80 (2.9)	9.04 (3.0)
8	2	14.70 (4.9)	16.33 (5.4)
12	3	16.73 (5.6)	17.64 (5.9)
16	4	18.05 (6.0)	18.39 (6.1)

Table 7: Basic global memory vector read bandwidths in MW/s and speedups.

cycles added per data element fetch in the block of 512 elements that a CE fetches on a single iteration using various prefetch block sizes. If $\rho = 0$ then there are no added dead cycles, i.e., the access is as dense as possible. As ρ increases the sparsity of the access increases and the amount of contention is decreased. (Of course, the value of ρ only reflects the sparsity within an iteration block. The other cycles due to loop control and other CE overhead also contribute dead cycles to increase sparsity but those are fixed for all of the experiments.)

If contention is not a significant factor the observed bandwidth should be a hyperbolically decreasing function of ρ as is seen for $p = 1$. When contention is a problem, however, increasing the sparsity may reduce contention and the general shape of the bandwidth curve is decreasing but concave. This is due to the fact that the reduction in contention improves the effective bandwidth more than the additional cluster work, represented by NOPS, decreases the effective bandwidth. The change in the shape of the curves for $p = 16$ compared to $p = 1$ illustrates the effect of contention. The more intense the contention for the configuration the

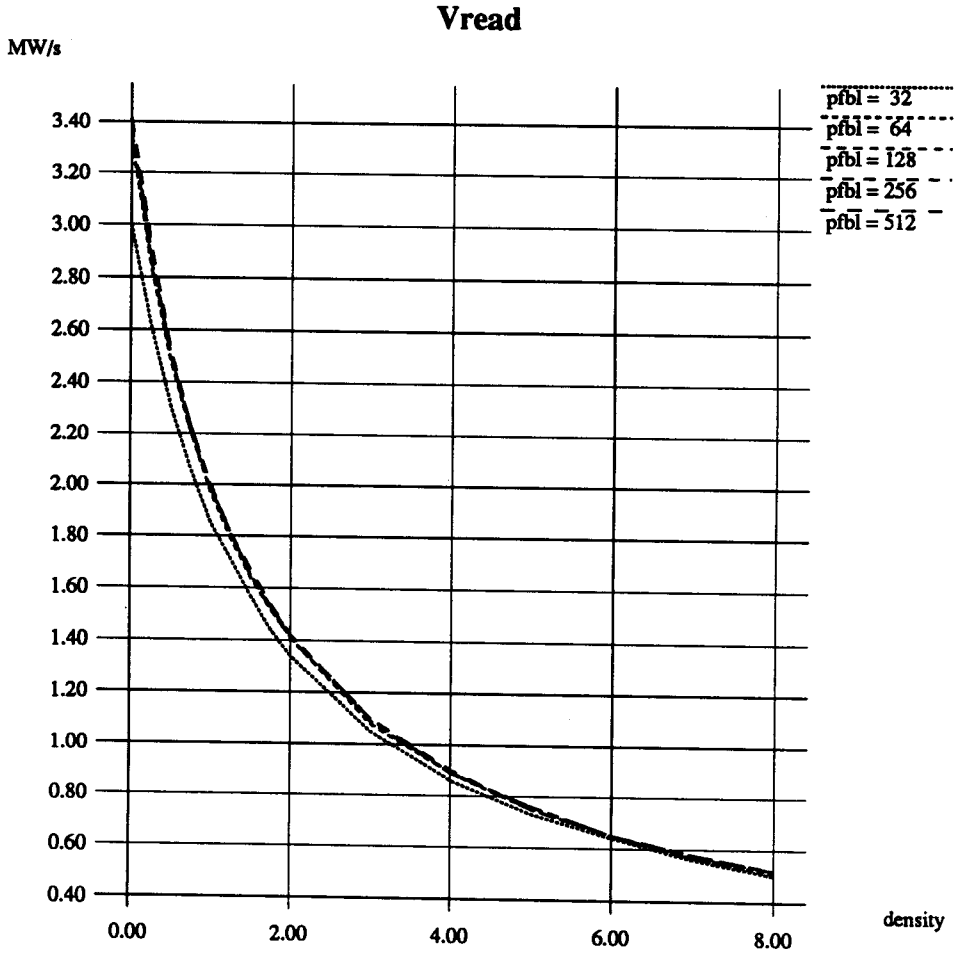

Figure 4: Implicit prefetch vector read bandwidth $p = 1$.

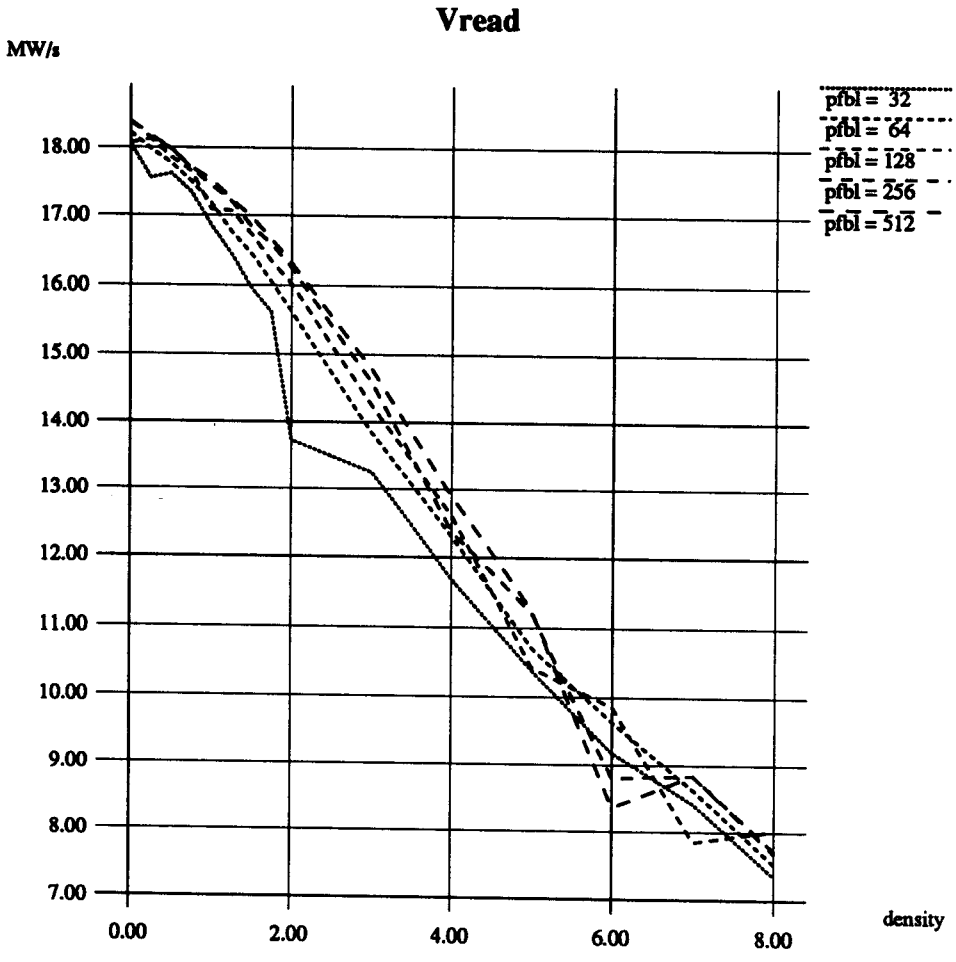

Figure 5: Implicit prefetch vector read bandwidth $p = 16$.

Total CEs	Clusters	Bandwidth
1	1	4.08 (1.0)
2	2	7.35 (1.8)
3	3	9.51 (2.3)
4	1	14.4 (3.5)
4	4	11.0 (2.7)
8	2	19.8 (4.9)
12	3	21.3 (5.2)
16	4	20.9 (5.1)

Table 8: Basic global memory vector stores bandwidths in MW/s and speedups.

longer the concave region persists along the sparsity axis.

The slight variation in performance due to the change of prefetch block size is due to the fact that the overhead and latency (setting up the registers in the prefetch unit and waiting for the return of the first word back) are less significant. The longer prefetch block sizes also imply more intense use of the global network. Therefore, when contention is significant the larger prefetch block sizes will be more sensitive to ρ. However, since the small number of interstage links are a very strong bottleneck the effect is much less pronounced than expected.

For a fixed prefetch block size, the data cube formed by triples, *(bandwidth, density, processors)*, illustrates the trends in performance of the global memory as the amount of contention varies. Figures 6 and 7 show the bandwidth vs. sparsity face of the cube for prefetch block sizes of 32 and 512 using implicit prefetching.

For up to 4 CE's, contention is relatively insignificant. It is interesting to note, however, that 4 CE's active across 4 clusters (i.e. 1 CE per cluster) is much worse than 4 CE's within a cluster. Such a difference in behavior is explained by the initial shuffle connection between the CE's and the first stage crossbars. In the first case, the 4 active CE's are connected to the same crossbar switch while in the second case, the 4 active CE's are connected to two different crossbar switches. Therefore, the first connection pattern will create a much worse local contention problem in the associated crossbar switch. It can be seen that if the sparsity increases this effect diminishes as expected. Therefore, the bandwidths of the two 4 CE configurations converge.

In Figures 8 and 9 the speedup versus CE's is plotted for the prefetch block sizes of 32 and 512. Each curve corresponds to a different sparsity of requests. These two figures clearly show that the speedup becomes near linear if the sparsity is greater or equal to 4. Another effect which can be observed is that the the increase in speedup gain is more pronounced for prefetch block size of 512 rather than for 32. This is caused by the fact that the amount of loop overhead is much less for the

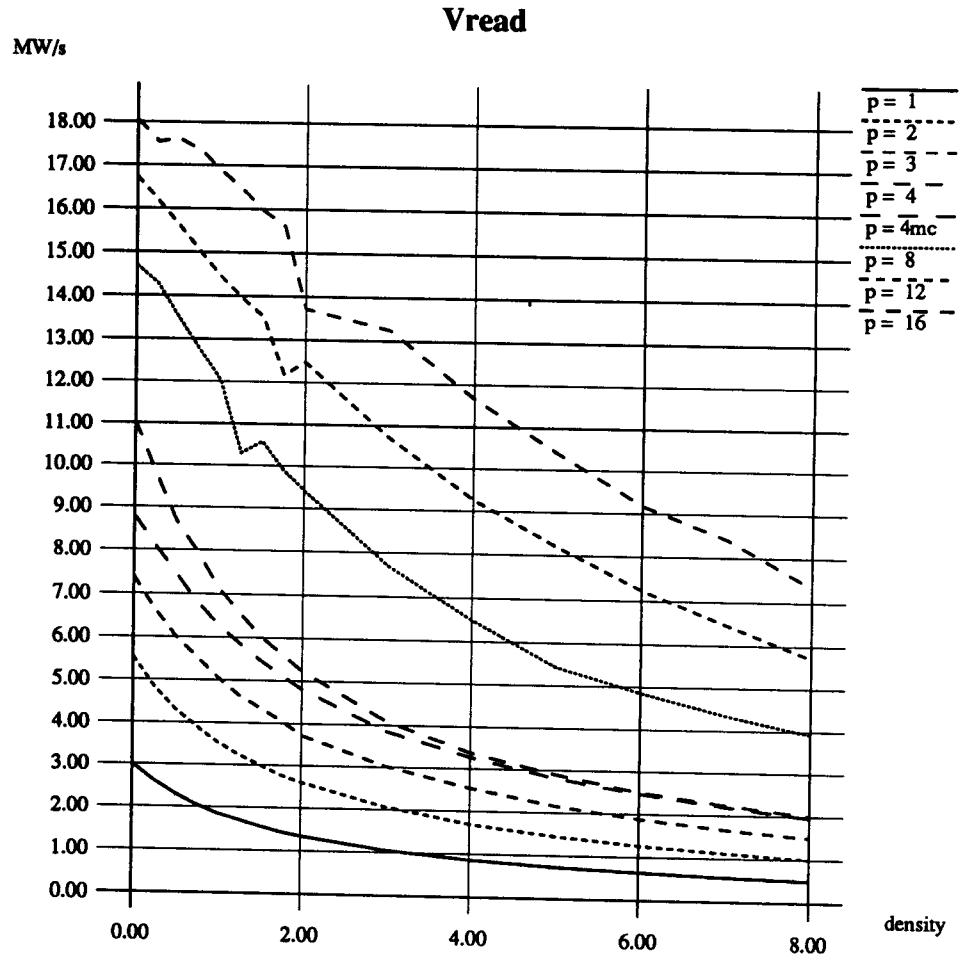

Figure 6: Implicit prefetch vector read bandwidth $pf = 32$.

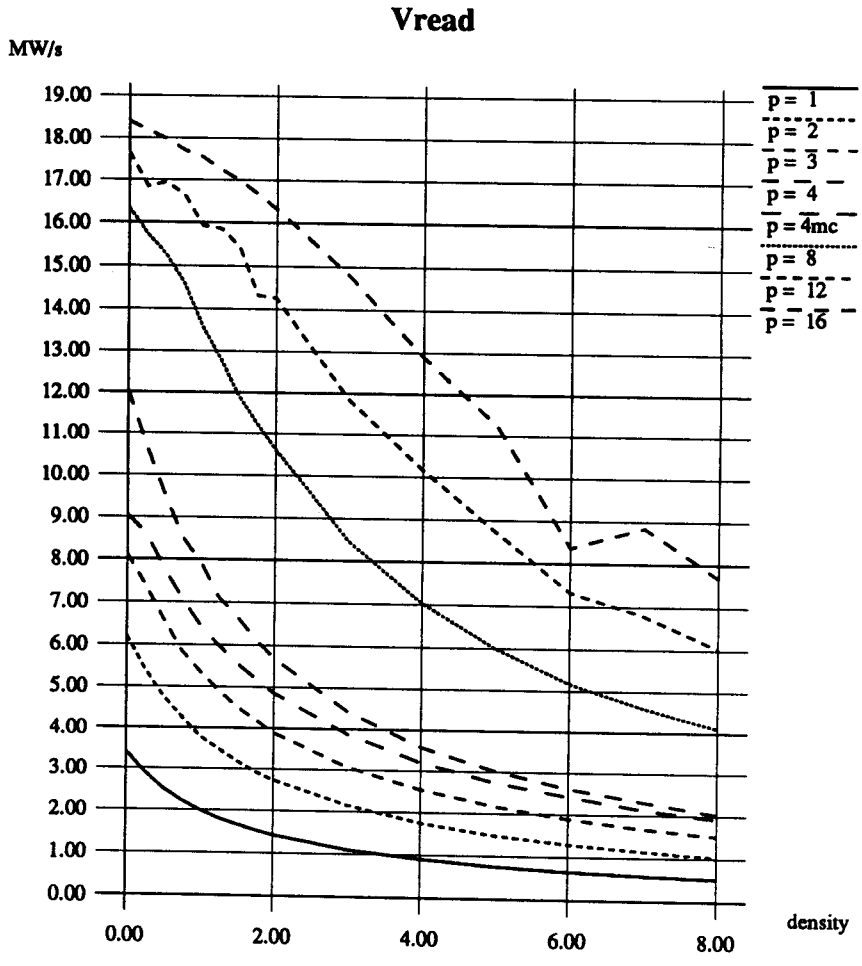

Figure 7: Implicit prefetch vector read bandwidth $pf = 512$.

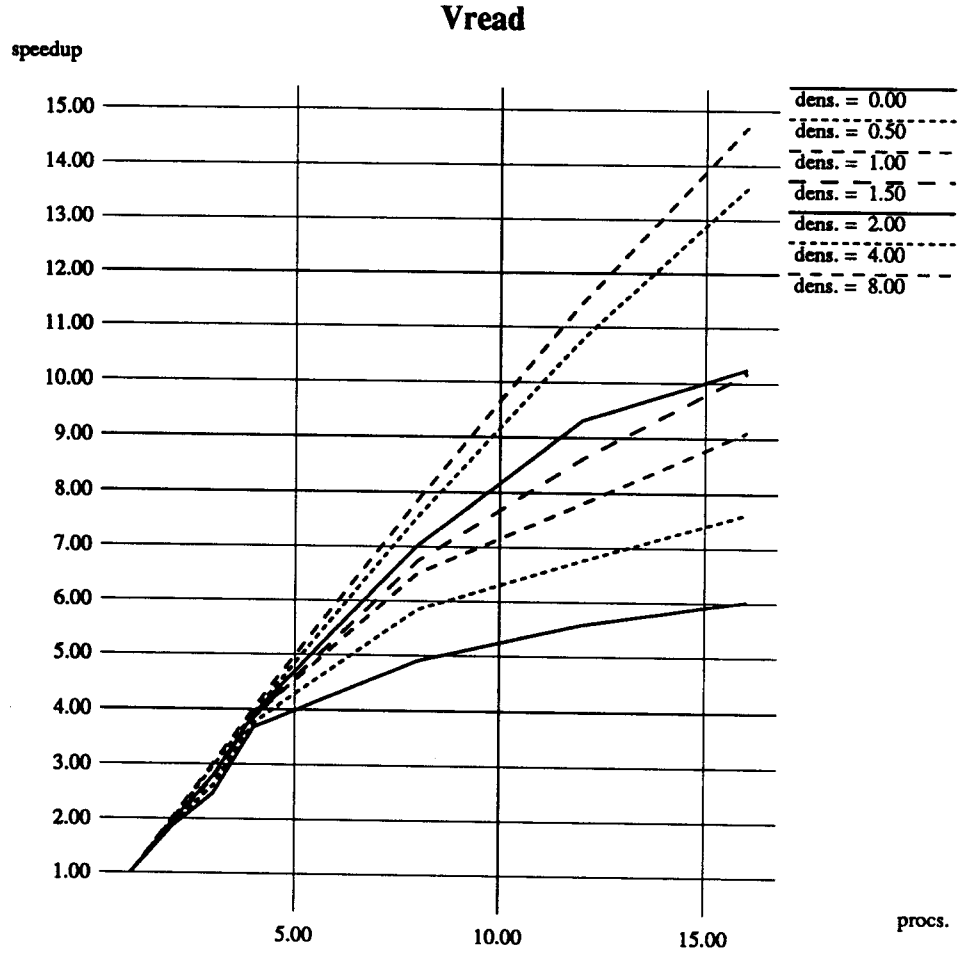

Figure 8: Implicit prefetch vector read speedup $pf = 32$.

larger prefetch block size, increasing the effect of inserting NOP's instructions in the kernel.

Another representation of the speedup curves is given in Figures 10 and 11, where the speedup is being presented as a function of the sparsity. Each curve corresponds to a different number of CE's. (Note that the data for 4 CE's split across 4 clusters is included in these curves.) In this case a perfect linear speedup would be reflected by horizontal curves around 2, 3, 4, 8, 12, and 16, and as such the deviation from optimal speedup is clearly indicated by these curves. For 2, 3, and 4 CE's the speedup is near optimal. Again it can be seen that the multi–cluster 4 CE's speedup is far from optimal when sparsity is small.

The bandwidth can be normalized such that the amount of time is shown per data element fetched. The result of this normalization is shown in Figures 12 and 13, where the number of CE's times the reciprocal of the bandwidth for implicit prefetching with prefetch block sizes 32 and 512 are plotted against density. From these

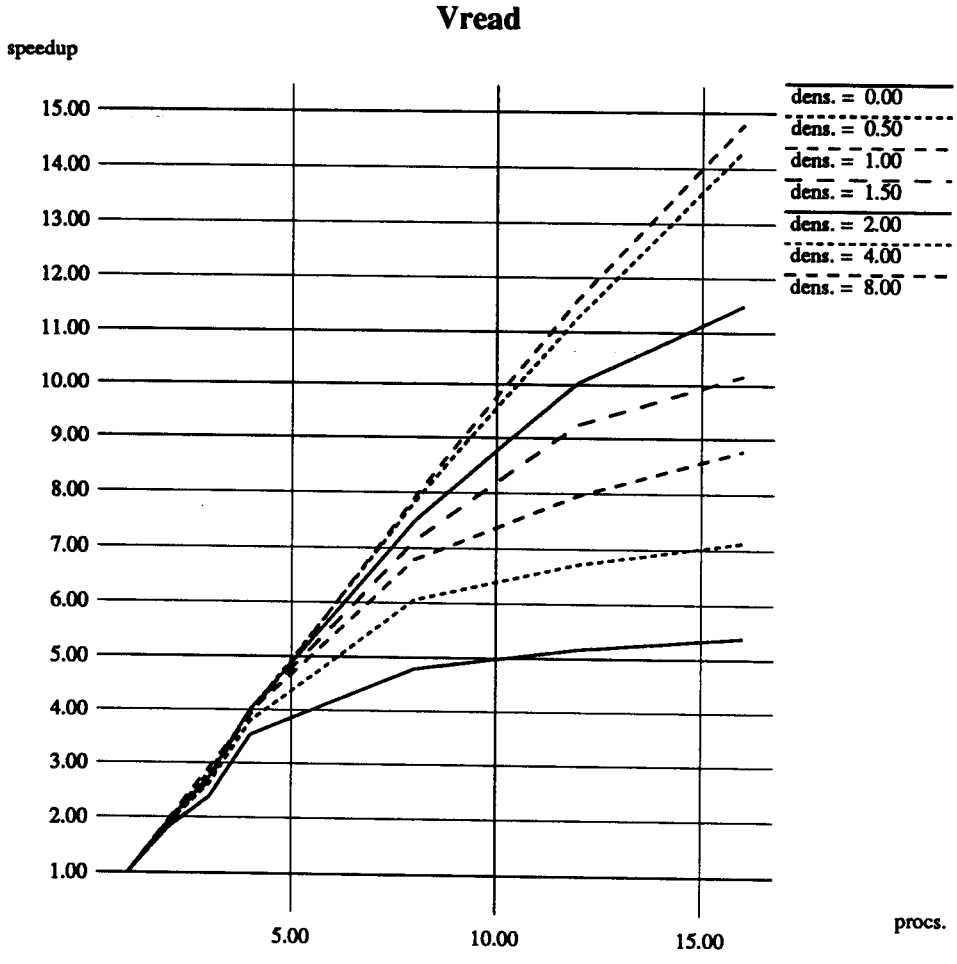

Figure 9: Implicit prefetch vector read speedup $pf = 512$.

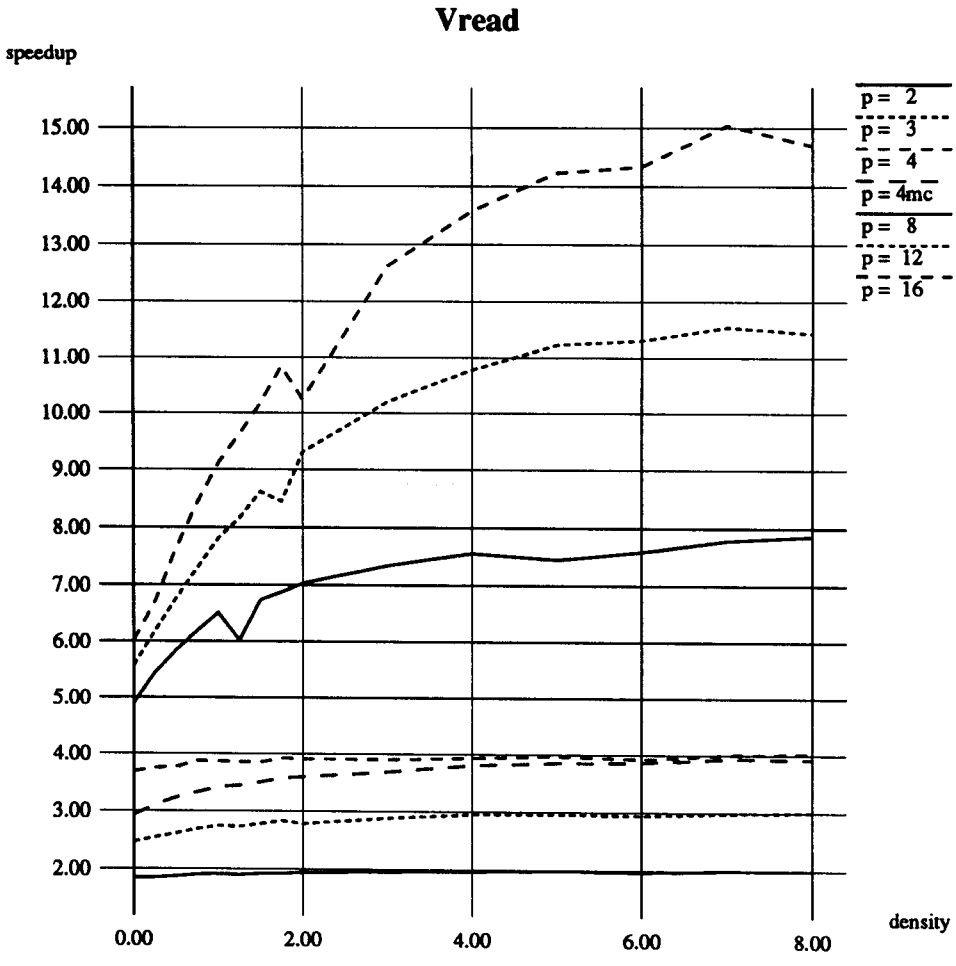

Figure 10: Implicit prefetch vector read speedup $pf = 32$.

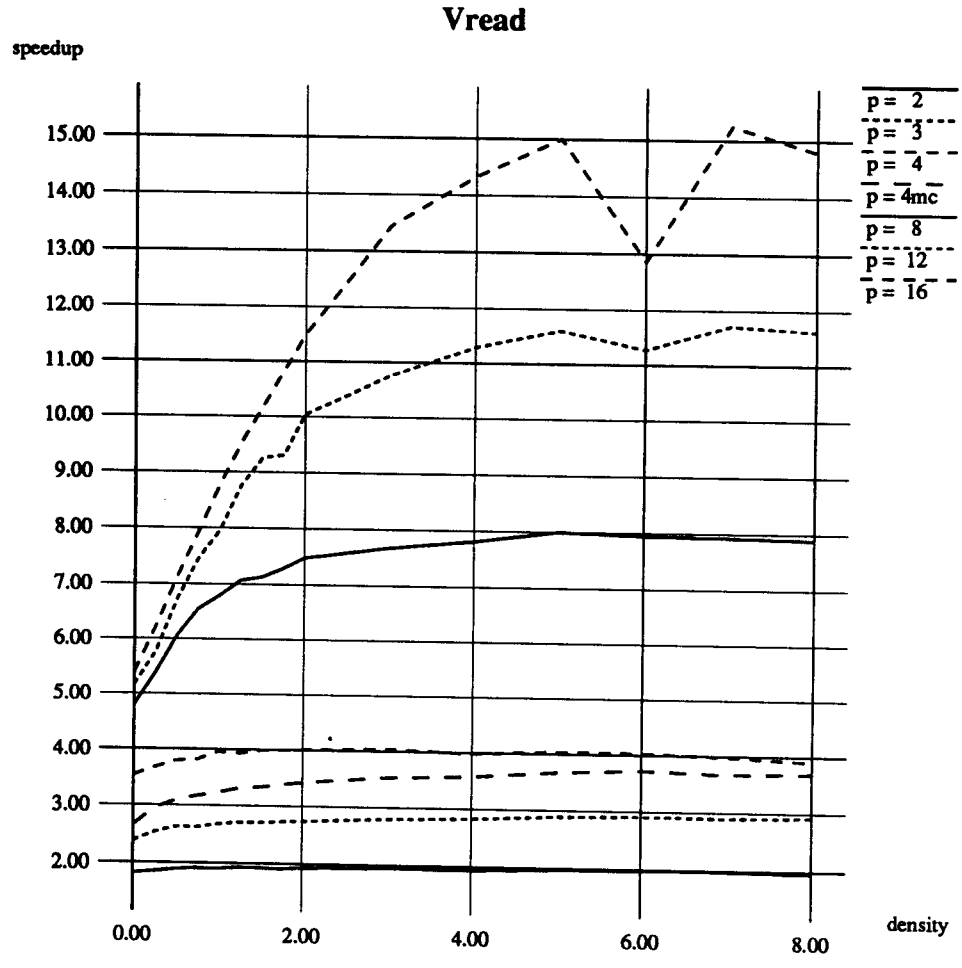

Figure 11: Implicit prefetch vector read speedup $pf = 512$.

curves can be seen that the normalized bandwidth of 1 CE and 4 CE's closely match each other and that the deviation from these curves occurs at 8 CE's or more. This is not surprising because the bandwidth of the four interstage network links are the bottleneck of the memory system. Thus, the request rate of up to 4 CE's can be easily satisfied by the network.

The above described results were all for the implicit prefetch experiments. From the memory standpoint of view, the behavior of the implicit and explicit prefetch should behave very similar because in essence the issue rate is the same. In figure 14 the ratio of explicit over implicit prefetch performance is shown. The main advantage of the explicit prefetch is to utilize the cycles lost, due to stalls in the CE, for other computational activities. As long as the amount of contention is small, the difference between the implicit and explicit prefetch performance is only due to the masking of the latency for the first word back. As can be seen from figure 14 the improvement of the explicit prefetch over the implicit prefetch is very minor in this case, i.e. the curve representing 4 CE's in one cluster). This points to the fact that latency for the first word is not that significant for the long prefetch block size of 512. As contention increases stalls due to out of order arrival of data becomes significant. This effect is can be masked using explicit prefetch. This is why the performance increase for the 4 CE's in different cluster and 8 CE's is more pronounced. It can be seen, that, when the sparsity is low (more contention), the performance of the explicit prefetch is about 20 %. However, if sparsity increases this effect is nullified, and even a performance drop of about 20 % occurs.

3.4 Load/Store combination templates

Next to the LOAD and STORE experiments as described before, experiments were also run with different combinations of LOAD/STORE instructions, i.e. L,LL,LS, and LLS. These experiments were done in steady state mode from global memory, with a vector of a given length sent to each cluster and accessed using all of the CE's available in each. The concurrent loop had each CE accessing a template of length 32 with a given number of NOPs per iteration. These NOPs per iteration relate to the sparsity of the previous sections by

$$\rho = \mathrm{NOPs}/(32 * k),$$

where k is the total number of vector register LOADs and STOREs, e.g. for LLS k would be equal to 3. The prefetching strategy used was that of the Cedar Fortran compiler, i.e., the prefetch registers are loaded for each vector access, prefetch is started immediately before the vmove instruction and turned off immediately after. So this form of prefetching is related to implicit prefetching except that the length of the prefetch block is always equal to the length of the vector register, i.e. 32. Figures 15 through 17 illustrate the behavior of the different LOAD/STORE combinations. As can be seen The LS–combination is outperforming the LL–combination

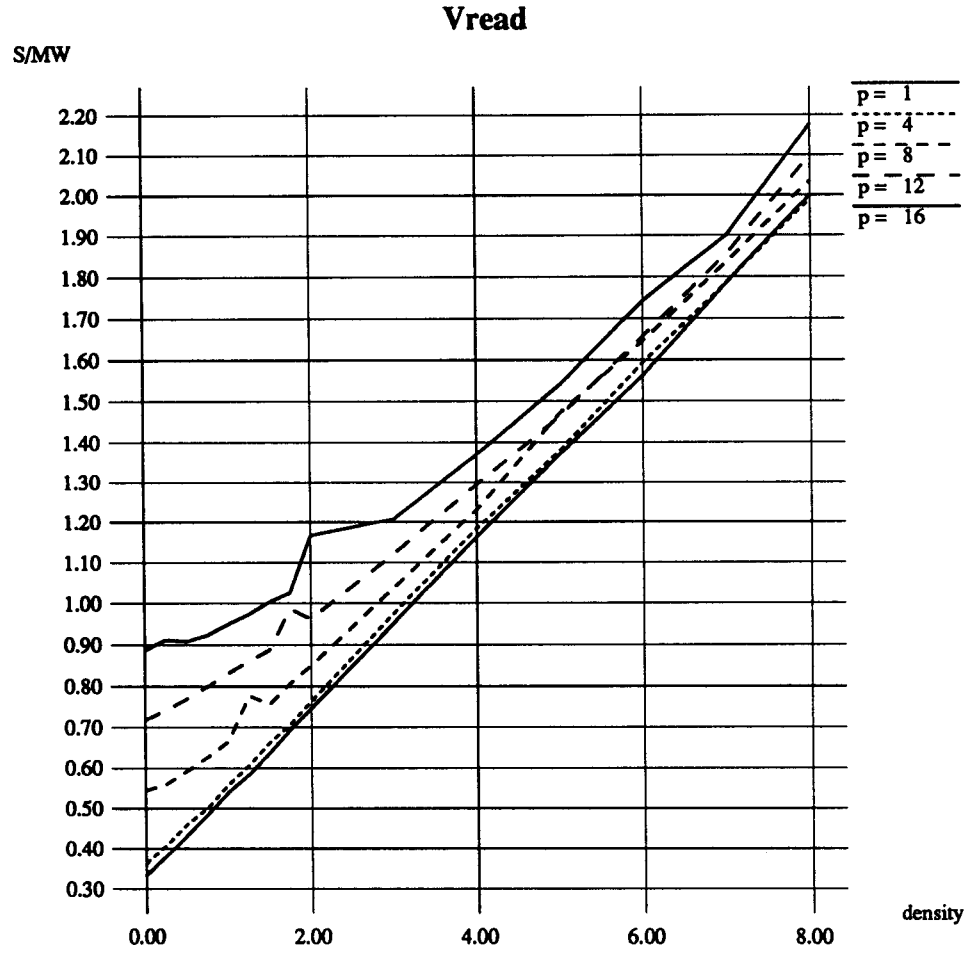

Figure 12: Implicit prefetch vector read p/BW $pf = 32$.

with a factor of 50 % if the sparsity is low. This is because the LS–combination the balances the data traffic volume evenly over the forward and backward network, whereas the LL–combination doubles the volume of data traffic requirements on the forward network. Because of the saturation of the network bandwidth, there is no observed difference in performance between the LS–combination and the LLS–combination.

3.5 Stride effects

The basic load template was run with 0 nops and the stride varying from 1 to 16 by increments of 1 to assess the effect of stride on contention. All of the CE's access their first element in bank 0, i.e. an offset of 0 for all CE's. Figures 18 to 21 illustrate the performance as stride varies for a given number of processors.

The effect of striding is directly related to the number of memory banks being

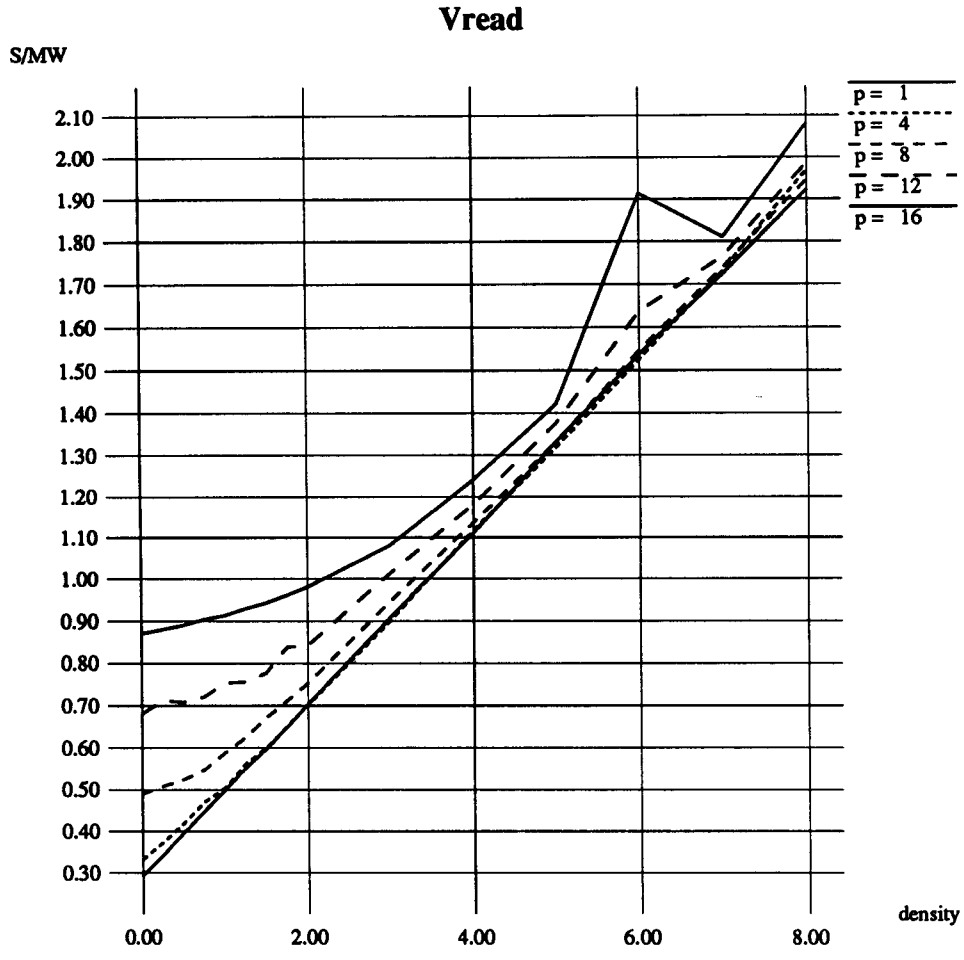

Figure 13: Implicit prefetch vector read p/BW $pf = 512$.

addressed by the CE's. This number of memory banks is equal to

$$16/\mathrm{GCD}(\mathrm{stride}, 16),$$

and the curves clearly demonstrate this, with the 16 CE case to be the most pronounced. For all the curves the performance of the stride 16 experiments turn out to be extremely close to the hardware limiting bandwidth for one global memory bank, i.e. 2.94 MW/s. The difference of the bandwidth obtained when addressing 8 memory banks (with stride, such that $16/\mathrm{GCD}(\mathrm{stride}, 16) = 2$), compared to addressing 16 memory banks is far less than a factor of 2. This again is caused by the limiting bandwidth of the 4 interstage network links.

3.6 Synchronization Experiments

Three types of multicluster synchronization routines were benchmarked: `event_post/event_wait`, `lockon/lockoff`, and `set_qlock/clear_qlock`. Two

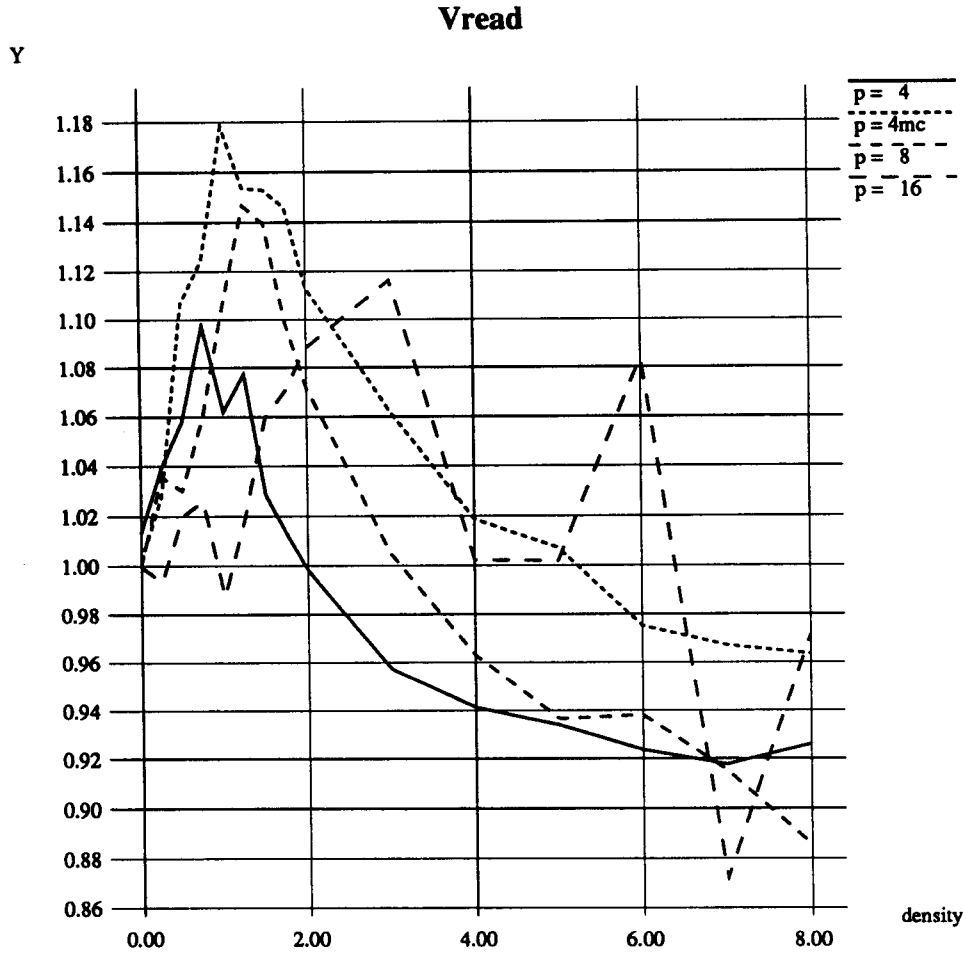

Figure 14: Explicit/Implicit $pf = 512$.

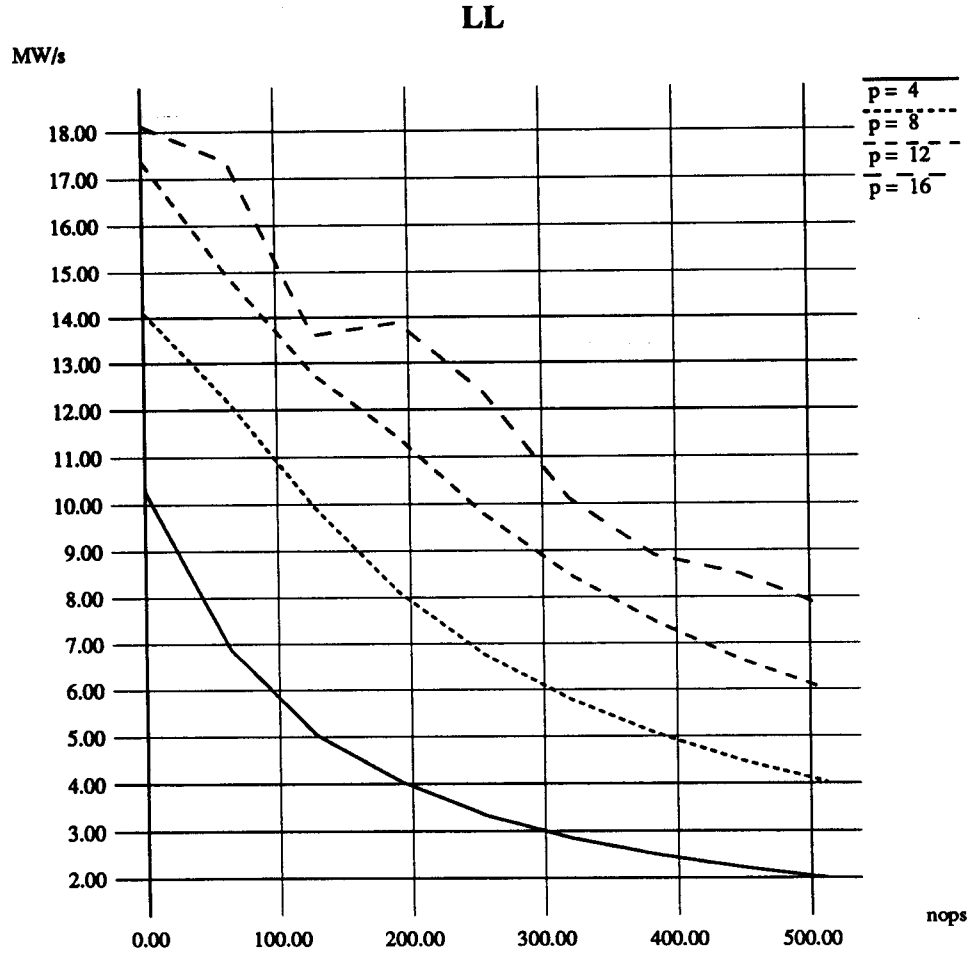

Figure 15: Load-Load bandwidth vs. nops.

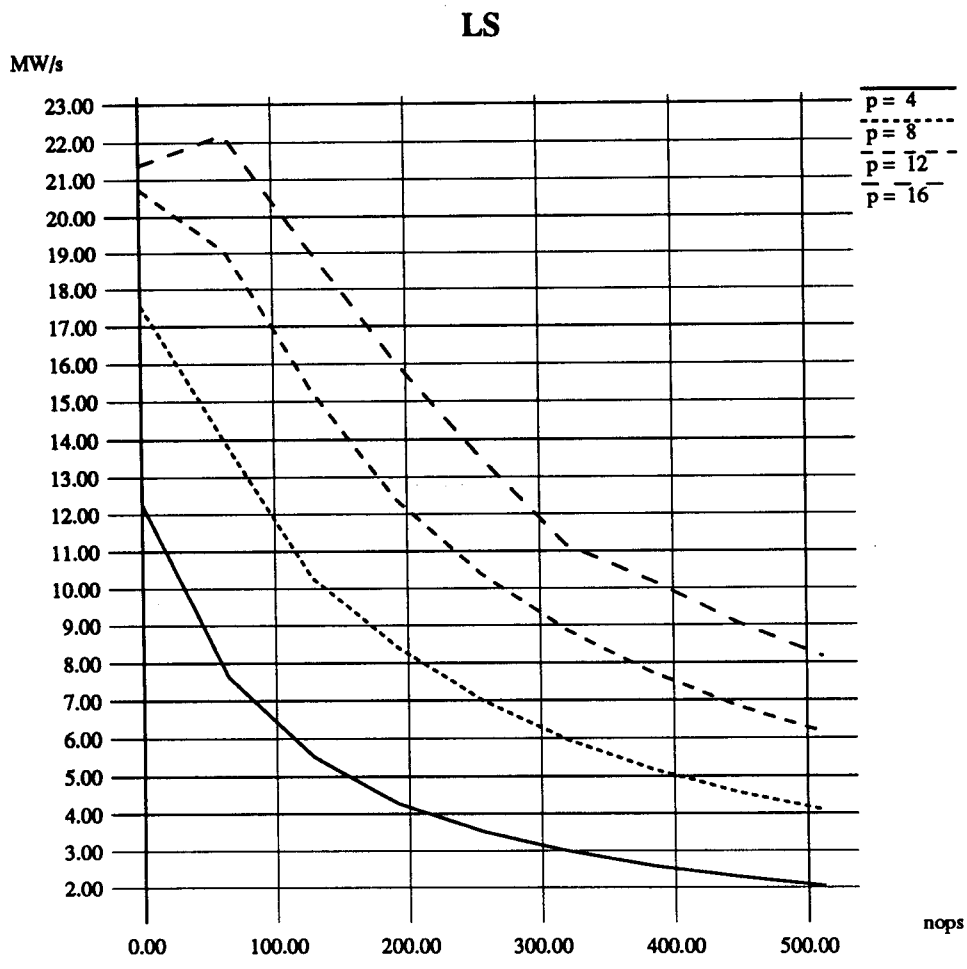

Figure 16: Load-Store bandwidth vs. nops.

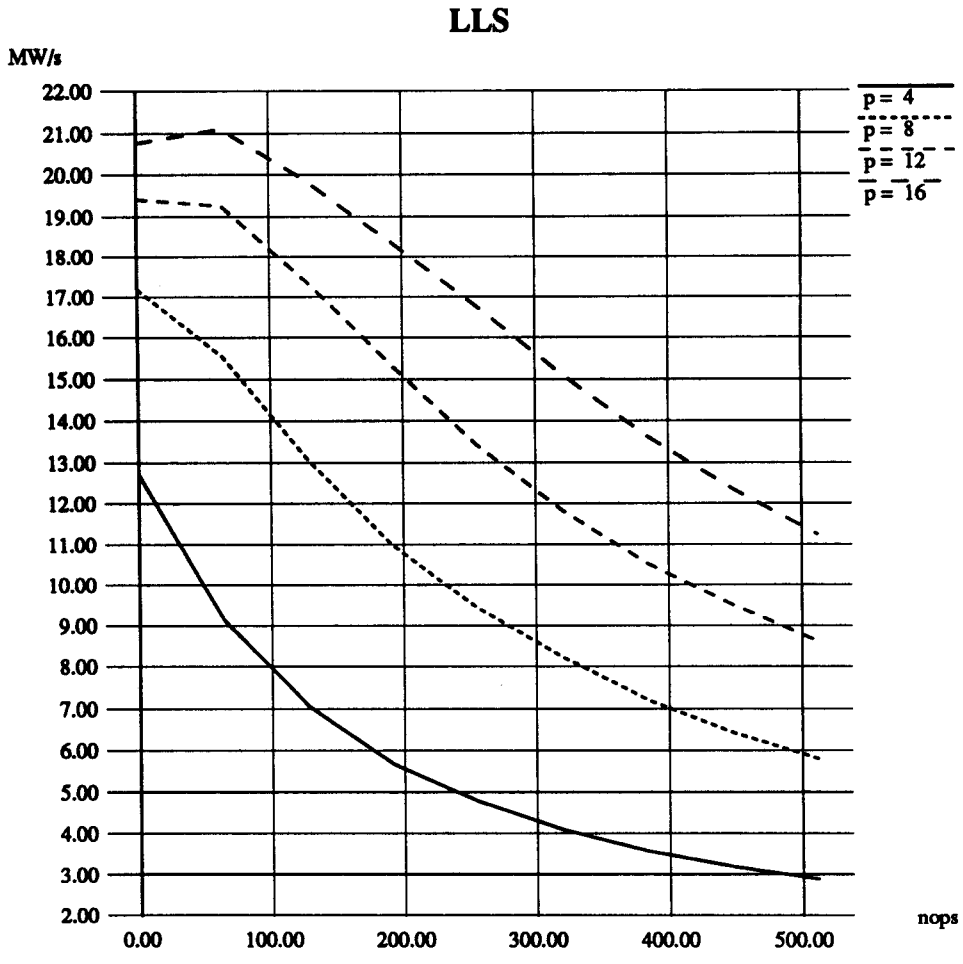

Figure 17: Load-Load-Store bandwidth vs. nops.

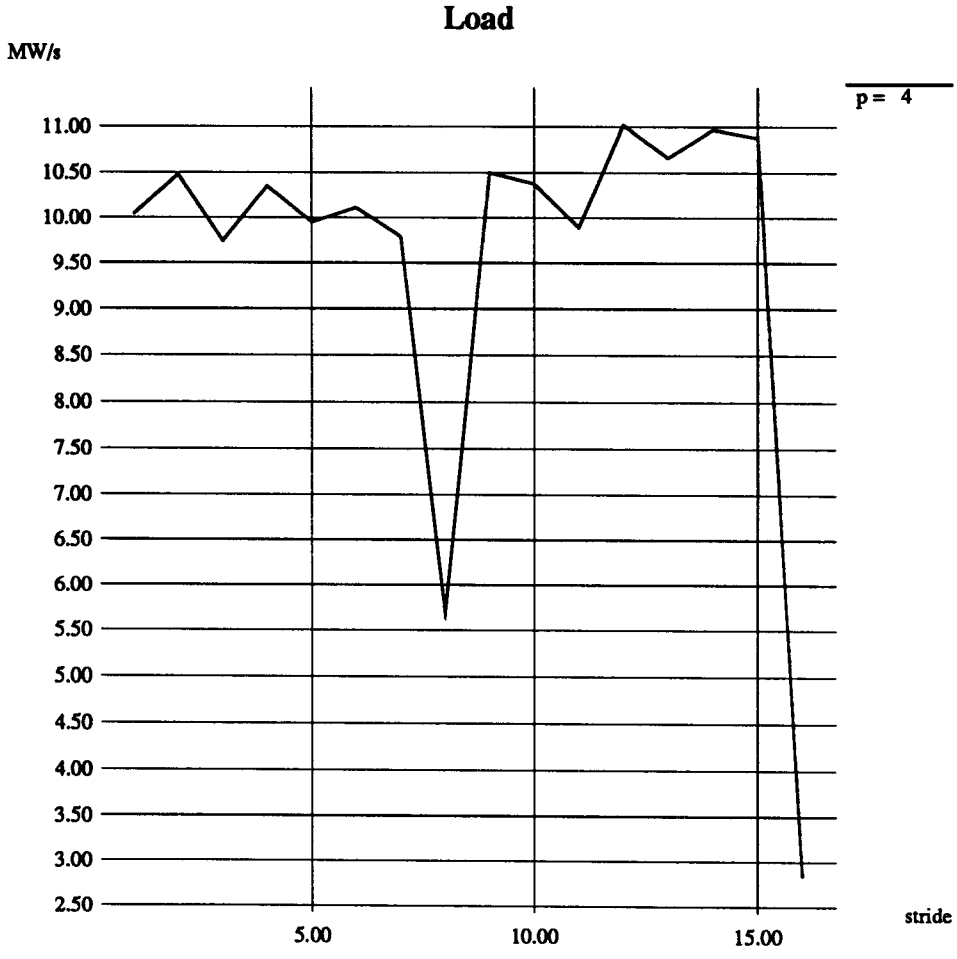

Figure 18: Load with 0 nops with nonunit strides, $p = 4$.

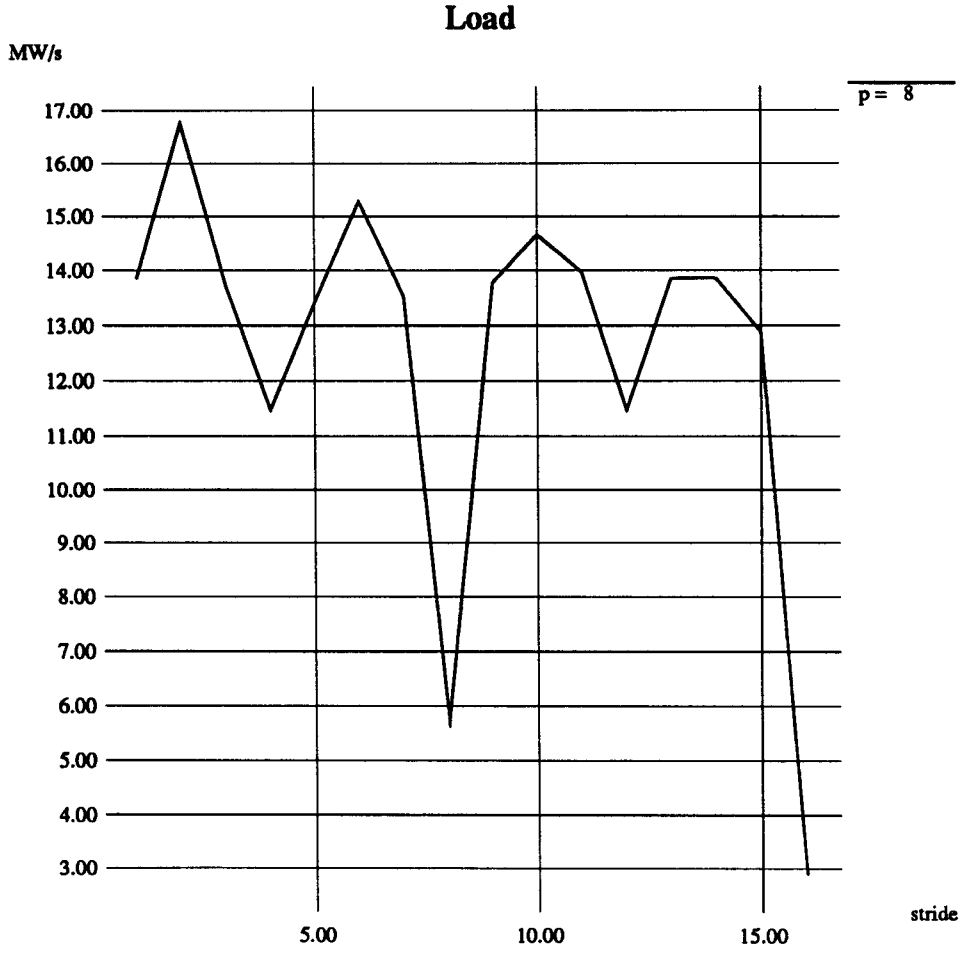

Figure 19: Load with 0 nops with nonunit strides, $p = 8$.

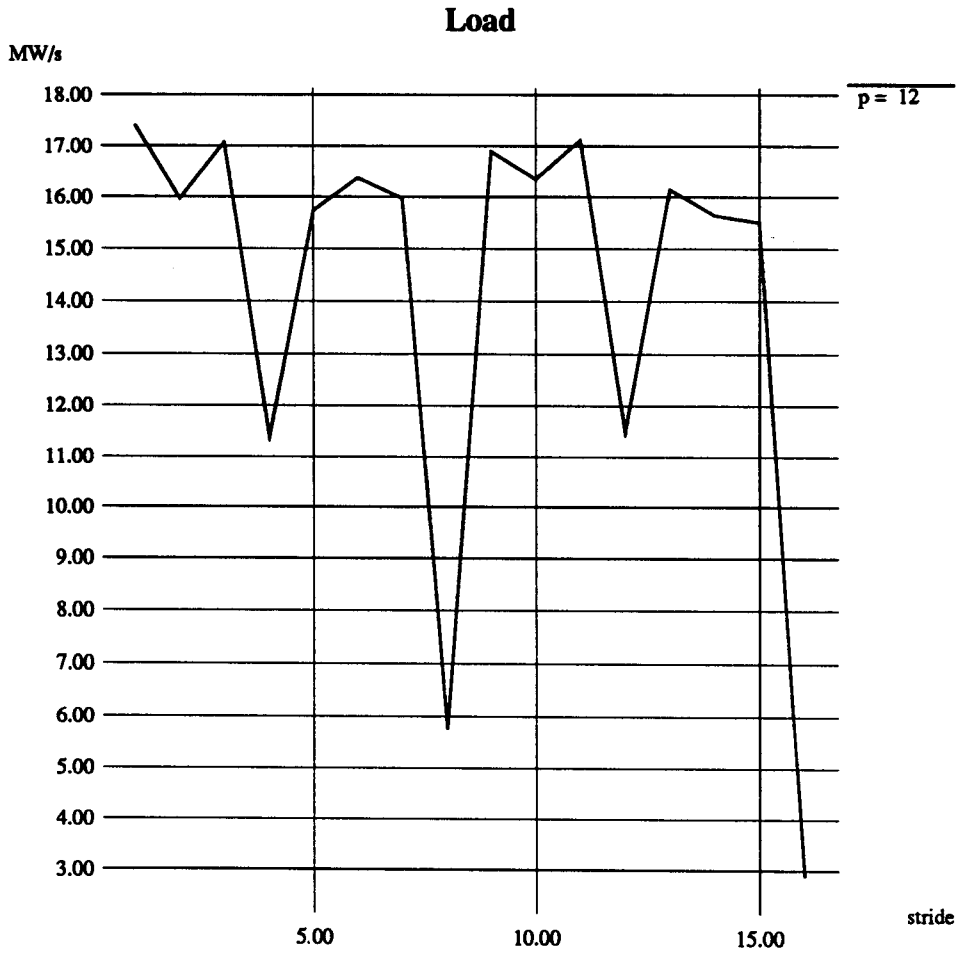

Figure 20: Load with 0 nops with nonunit strides, $p = 12$.

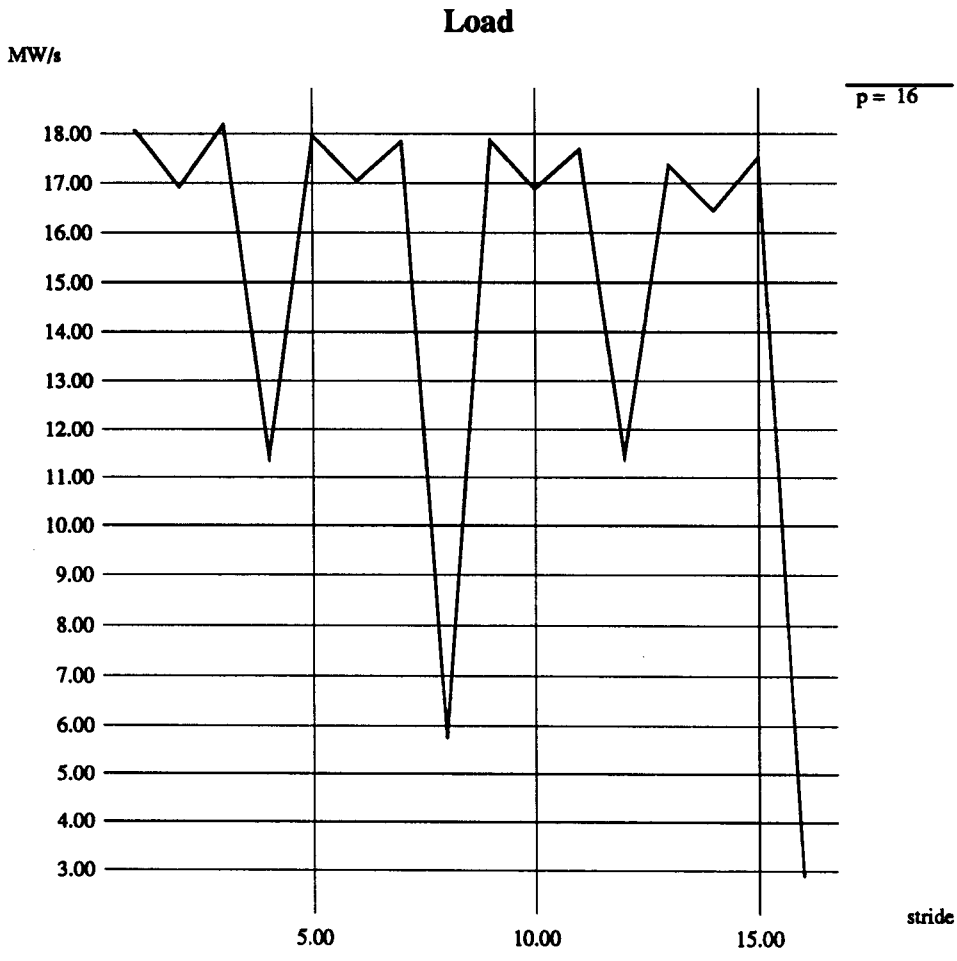

Figure 21: Load with 0 nops with nonunit strides, $p = 16$.

clusters were used to execute 10 and 50 iterations with each iteration having a given execution time τ. The average time, t, per iteration is plotted versus τ in the curves below. Table 9 contains the linear regression coefficients for each synchronization type, i.e., coefficients of the function $t \approx A * \tau + B$. The y-intercept B can be taken as an estimate of the synchronization overhead. Notice the substantial difference between those which involve the runtime status queues and the approach which uses `DAWDLE`. It should also be noted that the linear regression for the `DAWDLE`-based approach is biased towards iteration sizes which cause `DAWDLE` to be used. For more tightly couple situations where the initial buy-wait loop succeeds in seizing the lock the cost of synchronization can be reduced to a few microseconds.

The data upon which the regressions are based are given in Figures 22, 23, and 24. The expected trends are illustrated. The `event_post/event_wait` and `lockon/lockoff` curves show stability of results for large τ. The instability for small τ is due to the uncertainty of the polling mechanism discussed above. Note that the `set_qlock/clear_qlock` pair produces an extremely stable set of results.

Type	Samples	A	B
Events	32	1.008e-03	0.202
Lockon/off	32	1.015e-03	0.180
Qlocks	32	1.019e-03	0.002

Table 9: Linear regression coefficients for synchronization.

References

[1] K. GALLIVAN, D. GANNON, W. JALBY, A. MALONY, AND H. WIJSHOFF, *Behavioral characterization of multiprocessor memory systems*, in Proc. 1989 ACM SIGMETRICS Conf. on Measuring and Modeling Computer Systems, New York, 1989, ACM Press, pp. 79–89.

[2] K. GALLIVAN, W. JALBY, A. MALONY, AND H. WIJSHOFF, *Performance prediction of loop constructs on multiprocessor hierarchical memory systems*, in Proc. 1989 Intl. Conf. Supercomputing, New York, 1989, ACM Press, pp. 433–442.

[3] C.-Q. ZHU AND P.-C. YEW, *A scheme to enforce data dependence on large multiprocessor systems*, IEEE Trans. Softw. Eng., SE-13 (June 1987), pp. 726–739.

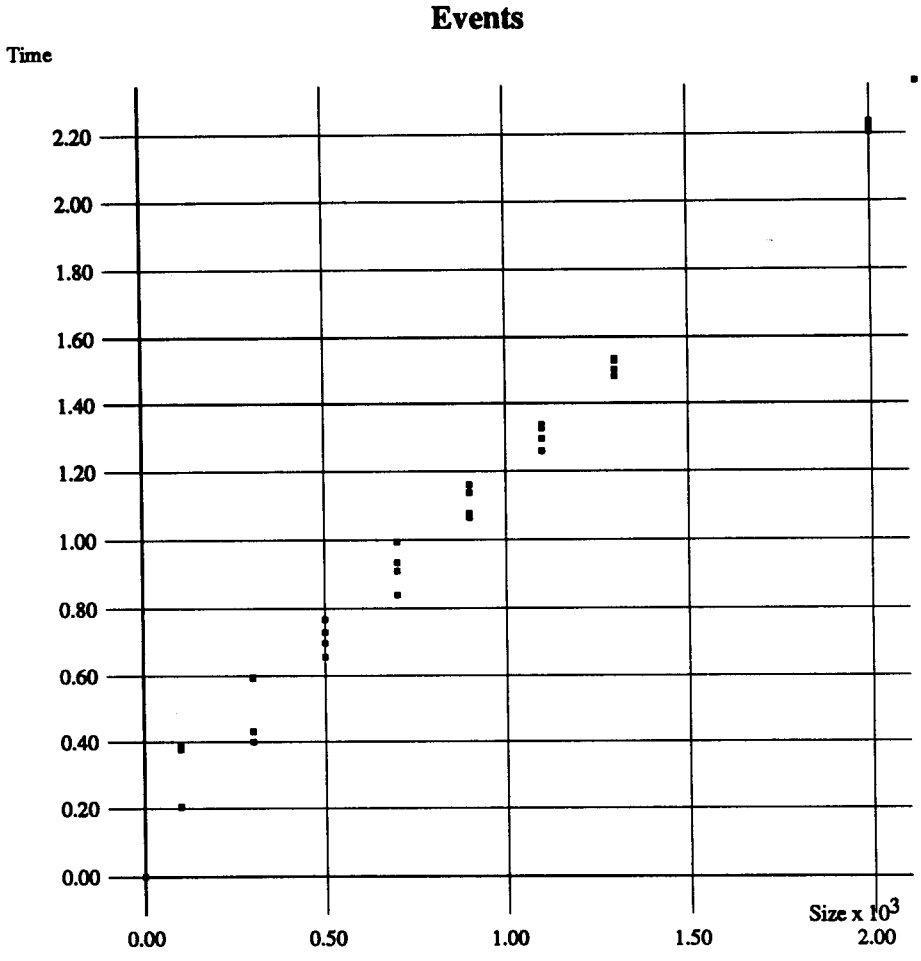

Figure 22: Event posting synchronization.

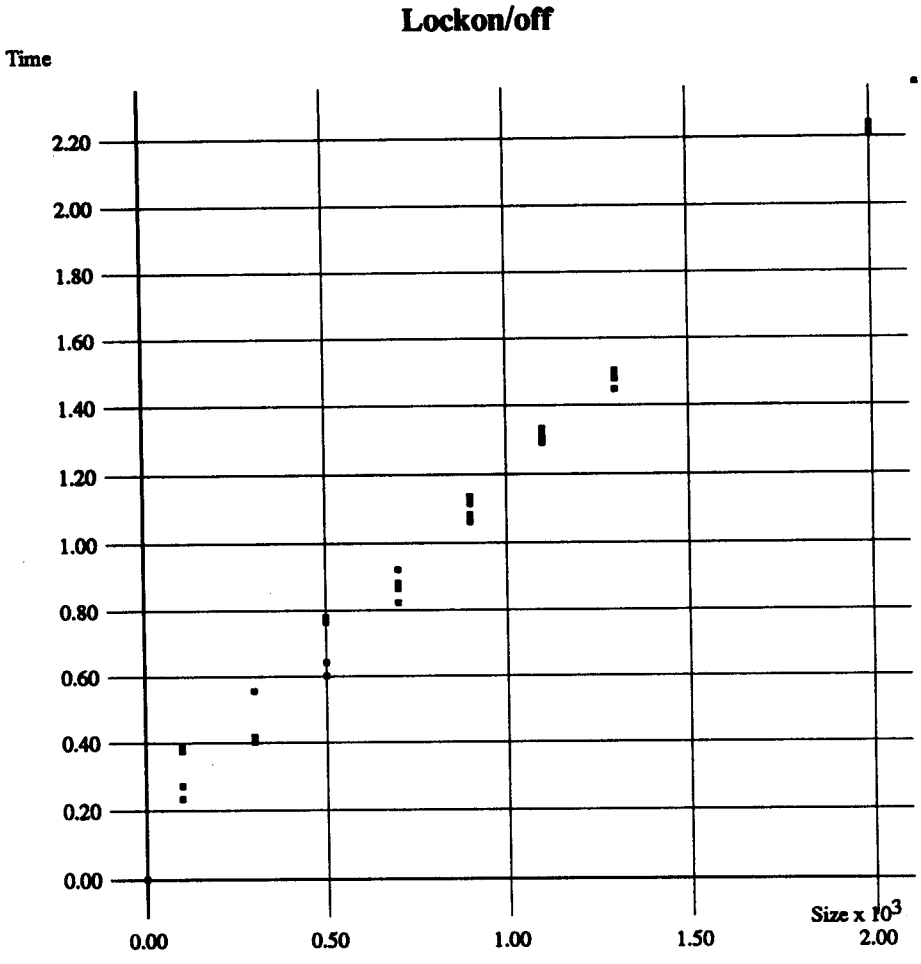

Figure 23: Lockon/off synchronization.

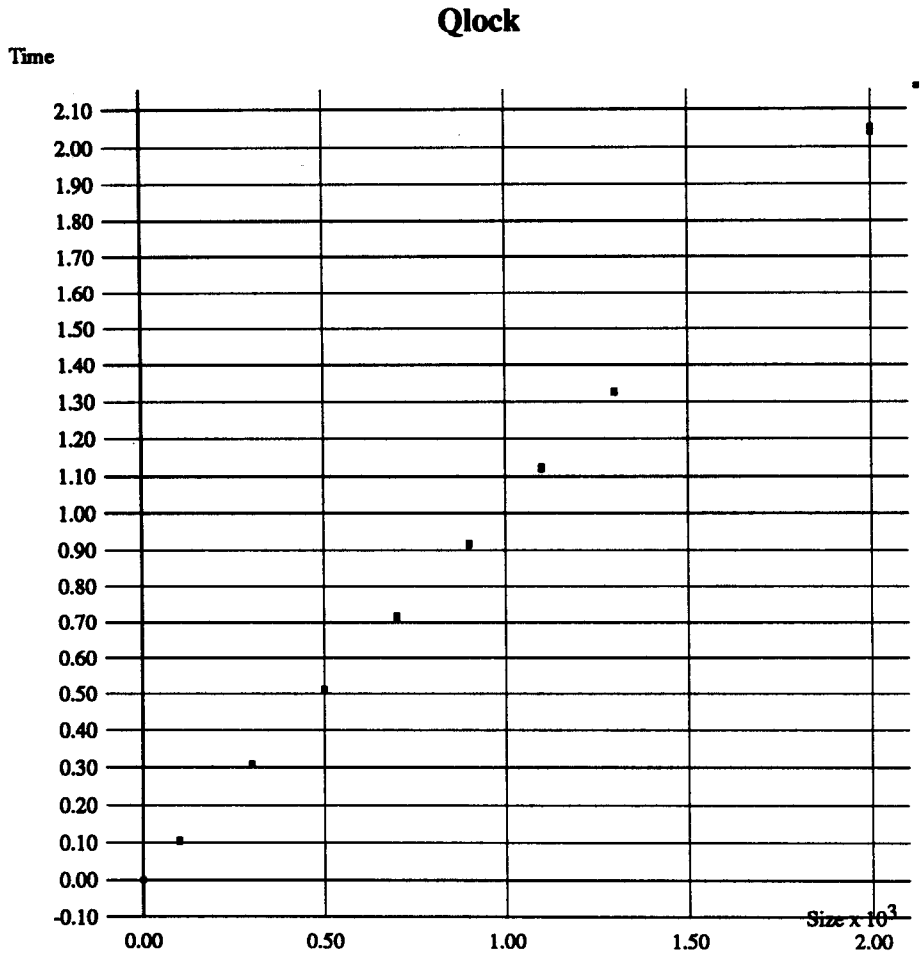

Figure 24: Qlock synchronization.