Clemson University
TigerPrints

Publications | School of Computing

8-1993

Measuring the Effects of Thread Placement on the Kendall Square KSR1

Amy Apon
Clemson University, aapon@clemson.edu

T D. Wagner
Vanderbilt University

E Smirni
Vanderbilt University

M Madhukar
Vanderbilt University

L W. Dowdy
Vanderbilt University

Follow this and additional works at: https://tigerprints.clemson.edu/computing_pubs

Part of the Computer Sciences Commons

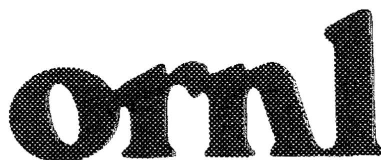
Recommended Citation

Please use publisher's recommended citation.

This Article is brought to you for free and open access by the School of Computing at TigerPrints. It has been accepted for inclusion in Publications by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

MARTIN MARIETTA ENERGY SYSTEMS LIBRARIES

3 4456 0377137 2



ORNL/TM-12462

OAK RIDGE NATIONAL LABORATORY

MARTIN MARIETTA

Measuring the Effects of Thread Placement on the Kendall Square KSR1

T. D. Wagner
E. Smirni
A. W. Apon
M. Madhukar
L. W. Dowdy

MANAGED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

OAK RIDGE NATIONAL LABORATORY
CENTRAL RESEARCH LIBRARY
CIRCULATION SECTION
4500N ROOM 175
LIBRARY LOAN COPY
DO NOT TRANSFER TO ANOTHER PERSON
If you wish someone else to see this report, send in name with report and the library will arrange a loan.

This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831; prices available from (615) 576-8401, FTS 626-8401.

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-12462

Engineering Physics and Mathematics Division

Mathematical Sciences Section

405
32

MEASURING THE EFFECTS OF THREAD PLACEMENT ON THE KENDALL SQUARE KSR1

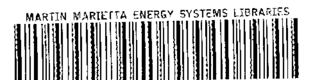
T. D. Wagner
E. Smirni
A. W. Apon
M. Madhukar
L. W. Dowdy

Computer Science Department
Vanderbilt University
Box 1679, Station B
Nashville, TN 37235

Date Published: August 1993

Research was supported by the Scientific Computing Program of the Office of Energy Research, U.S. Department of Energy via subcontract 19X-SL131V from the Oak Ridge National Laboratory.

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400


MARTIN MARIETTA ENERGY SYSTEMS LIBRARIES

3 4456 0377137 2

Contents

1 Introduction 1
2 Description of the ALLCACHE Memory Structure 3
  2.1 General Architecture 3
  2.2 Memory Architecture 4
  2.3 Automatic Prefetching 7
3 Experiments 8
  3.1 Workload Description 8
  3.2 Experiment 0 10
  3.3 Suite I 11
  3.4 Suite II 15
  3.5 Suite III 15
  3.6 Suite IV 18
4 Interpretation of Results 20
5 Conclusions and Future Work 22
6 References 23

MEASURING THE EFFECTS OF THREAD PLACEMENT ON THE KENDALL SQUARE KSR1

T. D. Wagner

E. Smirni

A. W. Apon

M. Madhukar

L. W. Dowdy

Abstract

This paper describes a measurement study of the effects of thread placement on memory access times on the Kendall Square multiprocessor, the KSR1. The KSR1 uses a conventional shared memory programming model in a distributed memory architecture. The architecture is based on a ring of rings of 64-bit superscalar microprocessors. The KSR1 has a Cache-Only Memory Architecture (COMA). Memory consists of the local cache memories attached to each processor. Whenever an address is accessed, the data item is automatically copied to the local cache memory module, so that access times for subsequent references will be minimal.

If a local cache has space allocated for a particular data item, but does not have a current valid copy of that data item, then it is possible for the cache to acquire a valid read-only copy before it is requested by the local processor due to a request by a different processor that happens to pass by on the ring. This automatic prefetching can greatly reduce the average time for a thread to acquire data items. Because of the automatic prefetching, the time required to obtain a valid copy of a data item does not depend simply on the distance from the owner of the data item, but also depends on the placement and number of other processing threads which share the same data item. Also, the strategic placement of processing threads helps programs take advantage of the unique features of the memory architecture which help eliminate memory access bottlenecks for shared data sets. Experiments run on the KSR1 across a wide variety of thread configurations show that shared memory access is accelerated through strategic placement of threads which share data. The results indicate strategies for improving the performance of applications programs, and illustrate that KSR1 memory access times can remain nearly constant even when the number of participating threads increases.

1. Introduction

Typically, as the number of processors increases in a shared memory multiprocessor, memory access times also increase due to contention on a common communication path or at the shared memory. This increased access time limits the additional processing (either the size of the problem or the speed of execution) which can occur as additional processors are allocated to a program. Kendall Square Research introduced the KSR1 system in 1991 [Ken91] with an architecture and distributed memory scheme that makes it possible for average memory access times to remain fairly constant as the number of processors grows. The architecture is based on a ring of rings of processing cells. Each processing cell has a 64-bit microprocessor and its own local memory that is managed as a cache. Up to thirty-two processing cells are connected on a level 0 ring. Up to thirty-four rings may be connected in a level 1 ring, so that the KSR1 may contain as many as 32*34, or 1088, processing cells.

The KSR1 has a shared memory programming environment, but all memory is contained in the caches of the processors. A valid copy of a data item must exist in the local cache of the processor in order to be accessed. Attached to each processor is the ALLCACHE Engine (ACE), which is the distributed mechanism responsible for finding a valid copy of a data item, copying it to a processor's local cache when it is referenced by that processor, and maintaining sequential consistency between caches.

One processor is the designated owner of each data item, but this ownership is not bound to any particular processor. When a processor writes a data item, it first obtains ownership of the data item in its local cache. At the time of writing to an item, all other copies of the item in other processor caches are marked as invalid, but the memory space for that data item may remain allocated. If a processor reads a data item and a valid copy is not available in the local cache of that processor, then a read-only copy of the data item is obtained via the ALLCACHE Engine. Many processors may have a valid read-only copy of a data item in their local cache. Each subsequent read of the same data item will be to the local copy and will require a minimal amount of time.

Repeated memory accesses to the same, shared, read-only data item require a fixed, minimal amount of time to access. Therefore, an important factor in keeping average memory access times small is to discover techniques for increasing the probability that a processor will find a copy of the shared item in its local cache. Three features of the memory architecture which move a read-only copy of a data item to a local cache prior to a request are the two programmer options *poststore* and *prefetch*, and one architectural feature *automatic prefetching*.

When a variable is updated by a write, a poststore by the writing thread will cause a valid read-only copy of the variable to be sent to all other processors which have a memory location allocated for that variable. The tradeoffs in using poststore have been studied [RSW+93]. In a

lightly loaded system, poststore can be effective in reducing memory access times. In a heavily loaded system, the cost to perform the poststore can be larger than the cost for copying the variable into the local cache at the time of access. Also, when the read-only copy of the variable sent by the poststore command is received by a processing cell, the variable may not be updated if the cell is busy making other memory accesses [Ken91].

The prefetch command allows a thread to request a valid copy of a data item before it is actually required by the thread. The prefetch command has been used to study the data rate for multi-ring memory performance [DHT93].

Automatic prefetching occurs when a processor has space allocated for a particular data item, but does not have a current valid copy of it. If a different processor makes a request for this data item, it is possible for this processor to acquire (i.e., "snoop") a valid read-only copy of it as the response to the request passes by on the rings. Automatic prefetching can greatly reduce the average time for a thread to acquire data items, since its occurs before the processor requests the data items. It also allows several processors to acquire read-only copies of a data item in parallel. Because of the automatic prefetching, the time required to obtain a read-only copy of a data item does not depend simply on the distance from the owner of the data item, but also depends on the placement and number of other processing threads which share the same data item.

Since the KSR is a ring of rings, the method of communication between rings is an important factor to consider when a large number of threads share data. Because of the unique architecture, the time to communicate between rings does not depend only on the distance between rings, but also on the type and patterns of data access.

The focus of this paper is to examine the combined effects of automatic prefetching and thread placement in reducing average memory access times for shared data in a multi-threaded application. The results indicate that strategic thread placement across multiple rings of the KSR1 can substantially reduce memory access time for shared data items. Other observed, but not reported, behavior, is that multiprogramming of the processing cells can reduce the measured access times for individual threads due to the overlap of fetch times with the time that a thread is context switched.

The goals of this paper are:

- to describe key features of the KSR memory architecture (Section 2),
- to design and execute a series of experiments which examine the effects of placement of threads on memory access time (Section 3),
- to identify programming techniques and thread placement options which can help to minimize memory access time (Section 4), and

- to summarize the findings and to outline how these results can assist applications programmers in optimizing their codes for the KSR multiprocessor (Section 5).

Related work includes studies which focus on the implementation of specific applications on the KSR1 [Char90,Char92,Ford92,Sumn92,WBHA93]. Each of these studies found that in order to obtain optimal performance on the KSR1, it is important to understand and take advantage of the peculiar characteristics of the architecture. For example, code which is optimized for a vector machine scales poorly on the KSR1, since many memory references are made only once and each such reference causes a local cache miss. When the code is rewritten to take advantage of the unique memory architecture, significant speedup improvements are possible [Sumn92]. Other related work includes initial benchmarking studies [Duni92], experimentation and modeling of the effective use of poststore [RSW+93], and studies of multi-ring memory performance [DHT93]. The results shown here assist in understanding how to optimize application codes.

2. Description of the ALLCACHE Memory Structure

2.1. General Architecture

The general KSR architecture is a multiprocessor system composed of a hierarchy of rings and processors. At the lowest level is the processor cell, which contains a 64-bit superscalar processor and 32 MB of local cache memory. Each processor also has .5 MB of traditional cache which is referred to as subcache memory. Each processor cell is connected to two neighbors to form the lowest level ring. The lowest level ring, termed ALLCACHE Engine:0 (ACE:0), is a unidirectional slotted ring with a peak data transfer rate of 1 GB per second. Each ACE:0 ring includes at least one cell, termed the ALLCACHE Routing and Directory (ARD) cell, which is responsible for routing to the next higher level ring. Up to 32 processing cells may be connected on a single ACE:0 ring. The next higher level ring, termed ACE:1, is composed of up to 34 ACE:0 rings. It is possible for an ACE:0 ring to contain multiple ARD cells, each of which can connect the ACE:0 ring to a different ACE:1 ring. The general KSR architecture provides for a third level which connects 32 ACE:1 rings into an ACE:2 ring [Ken91].

The KSR1 system used in this study consists of two identical ACE:0 rings, each containing 32 processing cells, linked on an ACE:1 ring. The system used in this study is shown in Figure 1. For ease of notation, the two ACE:0 rings are labeled Ring A and Ring B, and the processing cells are numbered 0 through 63 on the two rings.

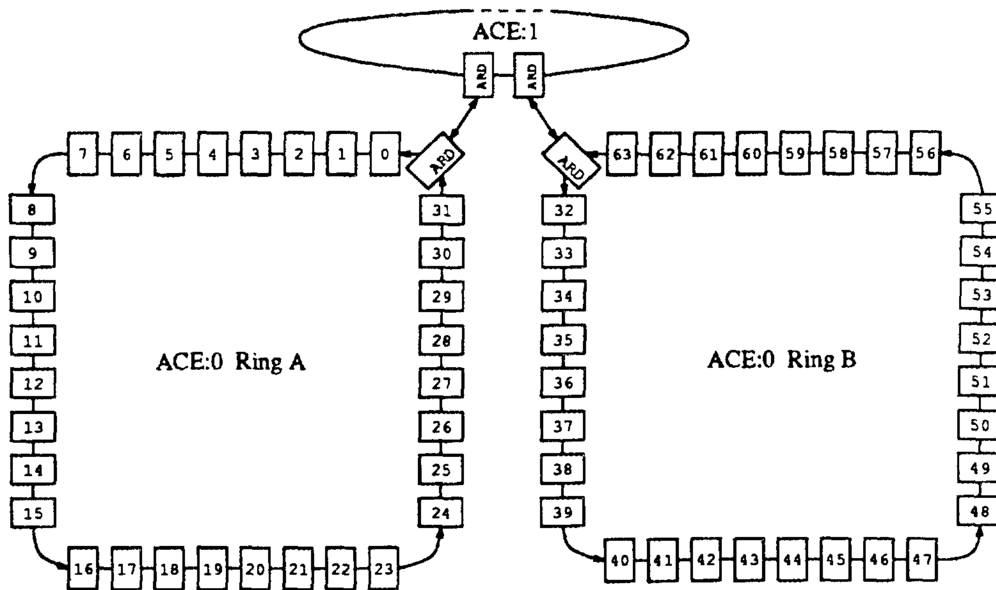


Figure 1: KSR1 with ACE:0 Ring A and ACE:0 Ring B

2.2. Memory Architecture

The KSR memory architecture is composed of two levels of related address space. The programmer's interface to memory is through the Context Address (CA) space. The System Virtual Address (SVA) space stores the data from all Context Address spaces. As in virtual memory systems on uniprocessors, a data item referenced in an application program is translated from the CA space of the application to a SVA through system software.

The 32 MB in each local cache is divided into 2048 pages of SVA space, each containing 2^{14} (16,386) bytes. The local cache is organized as a 16-way set associative cache. Each page is subdivided into 128 subpages, each containing 2^7 (128) bytes. Memory in the local cache is allocated on a page basis, but is copied between local caches in units of subpages.

The .5 MB in each local subcache is divided into a 256K instruction cache, and a 256K data cache. Memory in the subcache is allocated on a subpage basis, but is copied from the local cache to the subcache in units of subblocks (64 bytes). There are two subblocks per subpage. A processor may not directly access memory in another processor's subcache. For most instructions, a data item must be present in the subcache of the processor before being accessed. A few instructions operate directly on memory in the local cache of the processor.

Each local cache has a cache directory which contains a descriptor for each page of memory in that cache. At any particular time, a page descriptor may be invalid or valid. If a descriptor is invalid, then the corresponding cache memory page has not been allocated for a page of SVA memory. If a descriptor is valid, then space in the cache has been allocated for all subpages

in that page of SVA memory but a valid copy of each subpage may not be present. A 4-bit entry for the state of each subpage in the page is included in the descriptor for each page in the local cache. One bit of the entry indicates if the subpage is also held in the subcache of the processor. The other three bits give the state of the subpage.

There are three classes of subpage states:

- Invalid States

  When a subpage is not present in the local cache then it is in an invalid state with respect to that cache. The subpage is *invalid* if the page containing the subpage has a descriptor in the local cache, but the subpage is not present. The subpage is in *invalid-descriptor* state if the particular cache has no descriptor for the page containing the subpage. When a reference is made to a subpage which is in invalid-descriptor state, then the local cache must allocate a page of memory before a copy of the subpage can be obtained.

- Read-Only State

  There is only one read-only state. If any processor holds a subpage in *read-only* state, then any number of other caches can also hold the subpage in read-only state. The owner holds the subpage in *non-exclusive* state. If the owner of the subpage holds it in any other state, then no read-only copies exist.

- Owner States

  The owner of a subpage may hold it in a non-exclusive owner state, or in an exclusive owner state. When the subpage is in *non-exclusive* state, then the owner may read the subpage, but may not modify it. There are three exclusive-owner states: *exclusive*, *atomic*, and *transient-atomic*. When a processor holds a subpage in exclusive state, then it is the only valid copy of the subpage in the entire system and the processor may read or modify it. The atomic and transient-atomic states provide locks.

A memory request that is made for a subpage must include the state that the subpage will hold after it is acquired. The actions taken by any processors which hold a descriptor for that subpage depend on the requested state of the subpage. If a request is made to the owner processor for a read-only copy of a subpage which is currently held in the exclusive state, then the owner state changes to the non-exclusive state before the request is satisfied. The state of the subpage in other local caches which hold a descriptor for the subpage is not directly affected. It is possible for the state of the subpage in other local caches which hold a descriptor for the subpage to also change to read-only due to a copy being acquired through automatic prefetching.

Before a processor writes to a subpage, it must first obtain a copy of that subpage in exclusive owner state. At the same time, all other copies of the subpage in all other local caches change

Table 1: Latencies and capacities

Location of subpage	Total capacity (MB)	Latency in cycles (50 ns)
Local subcache	0.5	2
Local cache	32	18
ACE:0	1,024	175
ACE:1	34,816	600
Disk		400,000

to the invalid state. If a request is made to the owner processor for an exclusive copy of a subpage (for purposes of modification), then the state of that subpage at the relinquishing owner changes to invalid, and the ownership is transferred.

The ARD cells on each ACE:0 ring are responsible for directing requests between the ACE:0 ring and the higher level ACE:1 ring. The ARD is an important component of the memory architecture that helps eliminate memory access bottlenecks. Each ARD maintains a directory that includes the state of every subpage present on the ACE:0 ring to which it is attached. This state information specifies:

1. whether or not the owner of the subpage is on this ring,
2. whether or not there are valid read-only copies on this ring, and
3. whether or not there are valid read-only copies on other rings.

The state information is sufficient for an ARD to know whether or not a request can be satisfied on its local ACE:0 ring. If a request arrives at the ARD from its local ACE:0 ring, then it will be passed to the higher level ACE:1 ring if and only if it cannot be satisfied at its lower level ACE:0 ring. If a request arrives at the ARD from the higher level ACE:1 ring, then the ARD will extract the request from the higher level ring and pass it to the lower level ACE:0 ring if and only if it can be satisfied on its ACE:0 ring.

The total memory capacities and hardware latencies for data transfer specified by the manufacturer for a KSR1 containing 34 ACE:0 rings, each containing 32 processing cells (1088 processors total), are given in Table 1. Results in this study indicate that the measured latencies for requests that are satisfied on a remote ACE:0 ring are smaller when there are less than 32 ACE:0 rings. When a memory request is made, the latency required does not depend simply on the location of the data item, but also on the number of other similar requests. An important task of the ARD is to serve as a filter for multiple requests to the ACE:1 ring for the same subpage by allowing only one of several identical requests to pass to the ACE:1 ring.

When a request is placed on a ring, the responding cell removes the packet from its local ring and places it into a FIFO extract buffer in the cell. If the FIFO extract buffers of the cell

are full when a packet comes along destined to the cell, then that packet is marked "responder busy". The originating cell (or sometimes the ARD) clears this bit and the packet continues to circulate on the same ACE:0 ring in another attempt. The amount of delay incurred in the case where the owner and the requesting cell are on separate ACE:0 rings is about 15 microseconds. The number of message packets which the FIFO extract buffers on a cell can hold is variable, depending on the message type. A processing cell can hold about 15 packets, while an ARD can hold about 256.

2.3. Automatic Prefetching

When there are many shared subpages in the system, the ALLCACHE architecture allows subpages to be copied prior to a request through *automatic prefetching*. In automatic prefetching, when a copy of a subpage is sent through the search engine to satisfy a request, any processor whose cache has a descriptor for that subpage which is invalid may acquire (i.e., "snoop") a read-only copy as the subpage passes by on the ACE:0 ring. Automatic prefetching takes place as long as the processor is not "too busy" performing other memory accesses [Ken91]. Automatic prefetching is a powerful mechanism which reduces memory access time in applications with a high degree of read-only sharing.

As an example of how automatic prefetching can reduce access time, consider the KSR1 system as illustrated in Figure 1. Suppose that the owner of a subpage is located at cell 0, and that cell 1 and cell 2 require read-only access to the same data. Suppose that cells 1 and 2 both have a page allocated in their local cache for the data, and that the state of each requested subpage is invalid. Thus, a descriptor exists for every subpage to be requested, but a valid copy of the subpage does not exist on cells 1 and 2. Suppose that processing is such that it can be guaranteed that the thread in cell 2 will access the data before the thread in cell 1. As cell 2 makes each request on the ACE:0 ring, the owner (cell 0) will respond to it. When the response passes by cell 1, it will see it. Since cell 1 has a descriptor of the subpage allocated, it will make a copy of the subpage to its local cache. The response message will not be delayed and will be passed to cell 2. Cell 2 will acquire the subpage and remove the message from the ring. When cell 1 finally accesses the data, it will find a valid copy of the subpage in its local cache, and will not place a request message on the ACE:0 ring. The memory access time for cell 1 will be minimal.

If the placement of the threads is changed in this example, then the benefits of automatic prefetching will not be seen. For example, if the thread in cell 1 accesses the data before cell 2, then cell 2 will see the request message on the ACE:0 ring. The request will pass to cell 0, which will respond to it. However, cell 1 will see the response message and remove it from the ACE:0 ring. Cell 2 will not see the response. When cell 2 finally accesses the data, it will not

have a valid copy and will have to request a copy of each subpage through the ACE:0 ring. The average memory access time for the two threads is much higher because automatic prefetching is not performed.

In general, the order of data access is not known *a priori* in a multiprocessor environment. However, because of increased latency across different ACE:0 rings and delays introduced by the ordering of the cells on the ring, it is possible to increase the likelihood that automatic prefetching will occur through strategic placement of owner and reader threads across different ACE:0 rings. The combined effect of automatic prefetching and the placement of processing threads is the focus of the experiments in the following section.

3. Experiments

3.1. Workload Description

Four suites of experiments examine the effects of the number and placement of processing threads. A synthetic workload is constructed which is executed in each suite of experiments. Two types of processing threads are used in the synthetic workload. An *owner thread* has the task of writing each subpage in its portion of the data set, so that it has the only valid copy in memory of the data set (i.e., is the owner of each subpage) at the start of each experiment. A *reader thread* has a descriptor for every subpage of the data set, but these descriptors will be made invalid when the owner thread writes to the subpages. A reader thread requests a read-only copy of each subpage in its portion of the data set after the owner thread has written the entire data set.

A preliminary experiment, Experiment 0, illustrates the performance in the case that no reader threads share data. In Suite I through Suite IV, all reader threads share a single large data set. The synthetic workload is designed to systematically measure the average access time per subpage for the reader threads under a variety of owner and reader thread placements.

The workload performs the following steps:

- Initialization Phase (executed at the beginning of each suite of experiments)

  1. A number of reader and owner threads are spawned, each of which binds to a unique processor for the duration of the experiments.

  2. Each reader thread and owner thread reads a predetermined portion of the data set.

- Measurement Phase (executed for each experiment in the suite of experiments)

  1. Each owner writes its portion of the data set.

  2. A barrier synchronization is performed for all threads.

3. Timing begins for each reader thread.
4. Each reader thread sequentially reads its portion of the data set.
5. Timing ends for each reader thread.

The Initialization Phase represents the overhead required for spawning threads, binding them to a processor, and allocating pages of local cache memory for the data set. Initialization Step 2 ensures that each local cache has a valid descriptor of every subpage in the data set so that all subsequent accesses to a subpage require only data movement, and do not require local cache memory allocation. The size of the data set for Experiment 0 depends on the number of reader threads, but is at most 16 MB. The size of the data set for Suite I through Suite IV is 50K subpages (6.4 MB). The entire data set fits into the local cache of the owner, so that no disk accesses are required during the measurement phase. Similar experiments on data sets of other sizes show similar results as long as the data set fits in the local cache and is significantly larger than the data subcache.

The Measurement Phase is repeated for each experiment in the suite of experiments. Measurement Step 1 sets the state of each subpage in each owner thread to exclusive owner and sets the state of each subpage in all other threads to invalid. The first access by a reader thread to a subpage will cause the state of the subpage to become non-exclusive owner in the owner thread, but the owner of the subpage does not change during the measurement phase. During the measurement phase, one word in each subpage is read, so that one entire subpage is copied for each read operation. This is the maximum rate of data copying possible, and emphasizes the effects of thread placement. Timing is done using the *pmon* library call from within the thread code.

A system library call is used for the barrier release mechanism in Measurement Step 2 which synchronizes the reader threads. In the barrier release mechanism, a master thread holds a lock for one or more slave threads. The master thread waits until all slave processes reach the barrier. Then, the master releases all slave threads at the same time. The barrier release performs as follows:

1. While waiting on the barrier, all slaves spin on a shared location in memory, the "go signal".
2. At the time of release, the master updates the go signal. This sends an invalidate to each slave thread.
3. The go signal is invalidated at each slave thread. The next attempt by the slave to read the go signal causes a request to be issued on the ring.
4. When each slave thread acquires the new value of the go signal it begins to execute the work on the other side of the barrier.

Because of the ring architecture, not all cells see the new value of the go signal at exactly the same instant. Thus, the relative time for passing the barrier can vary, depending on the location of the slave threads with respect to the master thread. The location of the master thread is at cell 0 in all experiments.

The experiments and the synthetic workload are specifically designed to analyze how memory access times can be improved when only thread placement is varied. The memory access pattern is simple so that the behavior under different thread placements is emphasized. Even though this is a simple access pattern, many application codes contain similar patterns. For example, a database update followed by the execution of a number of client programs which read and use the latest value exhibits such an access pattern. Even when application codes contain more complicated memory access patterns, the results shown here illustrate that thread placement is a factor to consider for improving the performance of application codes.

The suites of experiments progressively illustrate the effects of placement of reader and owner threads on the KSR. The performance metric of interest in all experiments is the average read time per subpage. All experiments were run on the KSR1 system as illustrated in Figure 1. All results presented are averaged over at least 5 runs of the same synthetic workload.

3.2. Experiment 0

The goal of Experiment 0 is to identify the performance when no automatic prefetching or filtering by the ARD occurs, and all reader threads access the data from the same common owner. The methodology of Experiment 0 is to eliminate the advantages of automatic prefetching by partitioning the data set among the reader threads, and measuring average access time per subpage as the number of reader threads varies from 1 up to 63. The owner thread is placed on Ring A in cell 31. The first 31 reader threads are placed on Ring A in processor order. The next 32 reader threads are placed on Ring B in processor order.

The data set is divided into disjoint subsets of size 2K subpages each, and each reader thread accesses a unique subset of the data set. Each reader thread reads the same number of subpages, irrespective of the total number of readers, in order to compare with the experiments in Suite I through Suite IV. The performance metric calculated is the average read time per subpage. The total size of the data set which is read is equal to the number of readers times 2K subpages. When 25 readers are executing, the size of the data set is 50K subpages, or 6.4MB. When 63 readers are executing, the size of the data set is 126K subpages, or 16MB. This size is small enough to ensure that the entire data set fits into the local cache of the owner thread (32 MB), thus eliminating the effects of paging to disk.

The results of Experiment 0 are shown in Figure 2. When the number of readers is larger than 32, then at least one reader is placed on Ring B. The graph in Figure 2 shows the average

access time per subpage for readers on Ring A, the average access time per subpage for readers on Ring B, and the overall average access per subpage for all readers. The graph shows that the owner thread can satisfy requests in nearly constant time until the owner thread saturates at roughly 8 reader threads. As additional reader threads are added the average access time per subpage increases almost linearly.

Each reader thread accesses each subpage in its unique subset of the data set exactly one time, so that each reference to a subpage results in one request being sent to the owner. As the number of reader threads increases, the number of requests increases proportionally, the FIFO extract buffers used for extracting messages from the ring at the owner fill, and requests must be denied. The denied requests circulate around the ring for another try. Thus, queueing effects are seen, and the average access times per subpage increase proportionally to the number of reader threads. This is consistent with the behavior of an M/M/c model of the system as reported in [RSW+93].

The performance in this experiment is a worst case example. No data is shared among the reader threads, so no automatic prefetching occurs, and queueing effects are maximum. Also, since no two requests are for the same subpage, the ARD on Ring B cannot filter requests from Ring B destined for the owner on Ring A. When all readers share a global data set, the effects of automatic prefetching and the filtering by the ARD are introduced. Average read times per subpage reduce substantially, as shown in the experiments in Suite I through Suite IV.

3.3. Suite I

In all experiments in Suite I through Suite IV, all readers share a single large data set. The size of this data set is 50K subpages (6.4 MB). All readers read the entire data set.

The goal of the experiments in Suite I is to examine memory access times as the location of the owner of the data set is varied on the same ACE:0 ring. The methodology of this experiment is to measure the average read time per subpage as the number of reader threads varies from 1 to 63. The owner thread is placed on Ring A. The first 31 reader threads are placed on Ring A in processor order. The second 32 reader threads are placed on Ring B in processor order. The location of the owner thread is varied from cell 0 to cell 31 on Ring A.

Figures 3 and 4 illustrate the results of Suite I. Results were obtained for each possible location of the owner from cell 1 to cell 31, but are not presented for the sake of brevity. In Figure 3 the owner is on cell 1. This figure is characteristic of the performance obtained when the owner thread is placed on cells 1 through 11. In Figure 4, the owner is on cell 31. Figure 4 is characteristic of performance obtained when the owner thread is placed on cells 12 through 31.

When the number of readers is less than 32, all readers are on the same ACE:0 ring, Ring A.

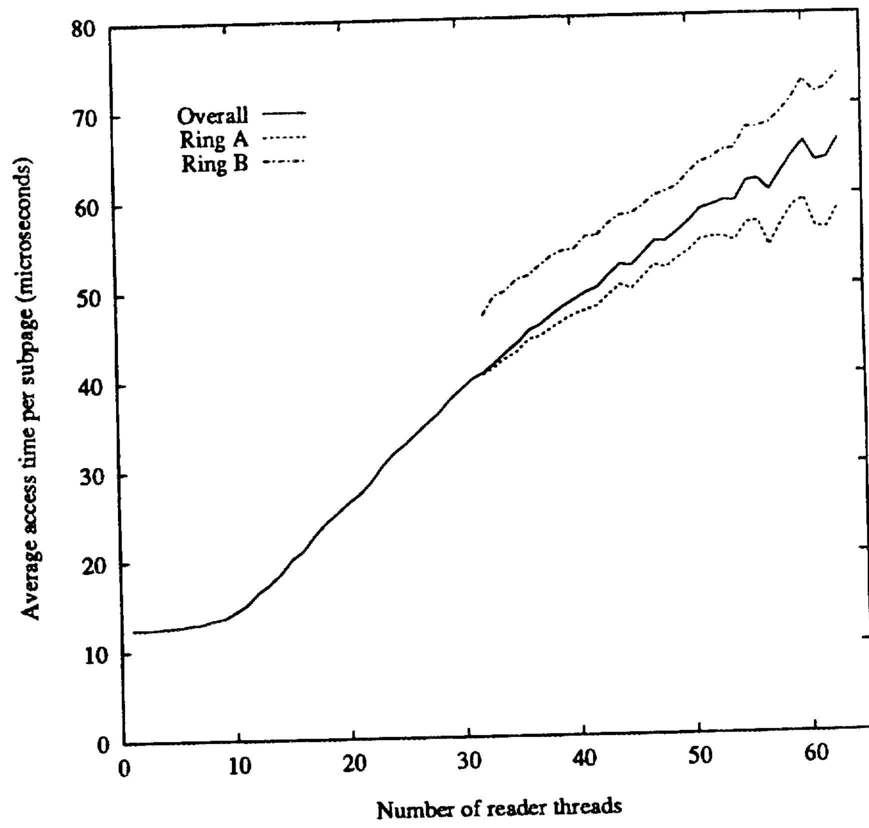


Figure 2: Suite 0: No automatic prefetching, one owner thread

When all readers are on Ring A, the results of this experiment show that the average read time per subpage increases as the number of additional threads increases. Further, when all reader threads are on Ring A, the average read time per subpage is insensitive to the placement of the owner thread, as illustrated in both Figures 3 and 4. Since all reader threads are accessing the same subpage at about the same time, little automatic prefetching takes place. This situation is similar to that shown in Figure 2. As the number of requests increases, more requests are denied at the owner, and queueing effects are seen.

When the number of reader threads is 32 or more, then at least one reader thread is placed on Ring B. When the owner thread is at cell 1 and at least one reader thread is placed on Ring B, then average access times are constant as the number of readers is increased beyond 32, as shown in Figure 3. Figure 3 shows the effect of the ARD filtering additional identical requests from Ring B. As the number of readers increases on Ring B, the number of requests for the same subpage also increases. However, the ARD passes to Ring A only a single request for all Ring B threads, so that the additional demand to the owner is only one request, irrespective of the number of reader threads on Ring B.

When the location of the owner thread is varied around the ring, the average read times *decrease* for all threads on *both* Ring A and Ring B as the number of reader threads is increased beyond 31. This effect is observed in the experiments when the owner cell is on cells 13 through 31. Figure 4 shows average access times when the owner thread is on cell 31. At first, this seems paradoxical. When threads are added on a ring which is remote to the owner (Ring B) which make requests for data on Ring A, the average access times for both Ring A and Ring B threads decrease. The location of the ARD, the direction of ring traffic (which is also the order of the barrier release mechanism), and relative placement of the owner thread to the placement of reader threads on Ring B combine to increase the amount of automatic prefetching for threads on Ring A, and decrease queueing effects at the owner thread. The automatic prefetching increases the number of valid subpages which are found in the local cache of each reader thread in Ring A. This causes a decrease in the average read time per subpage. Further, as automatic prefetching causes valid copies of the subpages to be available for reader threads, the number of requests is reduced. Since the number of requests is decreased, demand at the owner thread drops, and queueing effects are reduced. Average read times per subpage are reduced for all reader threads. The amount of automatic prefetching is probabilistic, and depends on the relative rate of execution of each of the reader threads.

In Suite I, the owner is on Ring A. Reader threads on Ring A are on a *local* ring with respect to the owner of the data. Reader threads on Ring B are on a *remote* ring with respect to the owner of the data. It was found that placing some number of reader threads on a ring which is remote to the owner improves the performance of reader threads on both Ring A and Ring

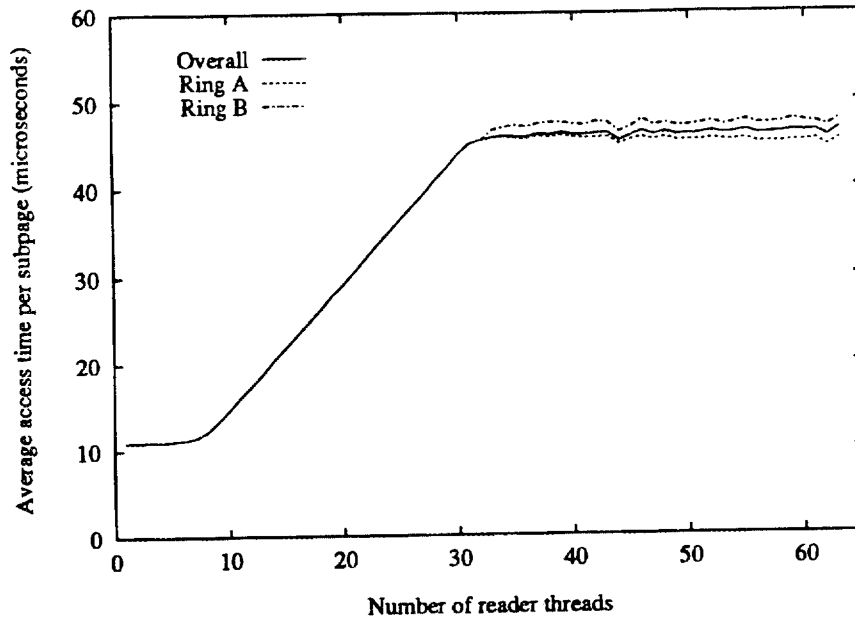


Figure 3: Suite I: Owner thread on cell 1

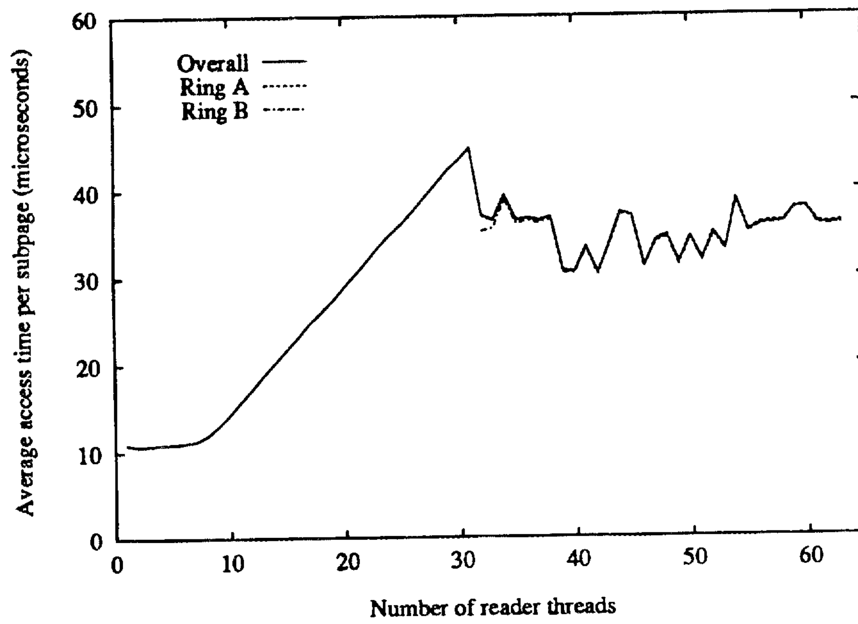


Figure 4: Suite I: Owner thread on cell 31

B. Suite II investigates tradeoffs in the number of remote threads versus the number of local threads.

3.4. Suite II

The goal of the experiments in Suite II is to measure average memory access times for remote threads as the number of local threads is varied. The owner of the data set is placed on Ring A. The methodology of this experiment is to vary the number of remote reader threads (reader threads on Ring B) from 1 up to 32. Four experiments in Suite II set the number of local reader threads (reader threads on Ring A) to be one of 0, 10, 20, or 30. The location of the owner thread is selected to be typical of the two types of performance which was observed from Suite I. The location of the owner thread is either cell 1 or cell 31.

Figure 5 illustrates average read times per subpage for Ring B threads as the number of reader threads on Ring B is increased from 1 up to 32. The owner of the data is on Ring A, and 0 reader threads are on Ring A. When all readers are remote to the owner of the data, the placement of the owner on Ring A has a small effect on the average read times per subpage.

Figure 6 illustrates average read times per subpage for Ring B threads as the number of reader threads on Ring B increases from 1 up to 32. The owner of the data is on Ring A, and 30 reader threads are on Ring A. This graph is similar to the right half of the graphs in Figures 3 and 4. When there are 30 local reader threads on Ring A, then the placement of the owner on Ring A has a significant effect on the average read times per subpage for the reader threads on Ring B. Figure 6 illustrates that average read times per subpage improve as much as 30%, depending on the placement of the owner thread. Experiments with 10 and 20 local reader threads give intermediate results and are not shown for the sake of brevity.

An interesting observation can be made for the graphs in Figure 5 and the left half of the graph in Figure 4. In both experiments the owner is on Ring A, and the total number of reader threads increases from 1 up to 31. In Figure 4 all readers are on Ring A, which is local to the owner. In Figure 5 all readers are on Ring B, which is remote to the owner. Figure 7 shows these two curves on the same graph. There is a crossover point in the graph. When more than 20 reader threads are executing, better performance is observed when all reader threads are on the *remote* ring as compared to when all readers are on the same ring as the owner. This effect is due to the behavior of the ARD, which filters requests from Ring B and the effects of automatic prefetching by the reader threads on Ring B.

3.5. Suite III

The goal of the experiments in Suite III is to fix the location of the owner thread, and examine the effects of various placements of reader threads. Cell 31 on Ring A is chosen as the location

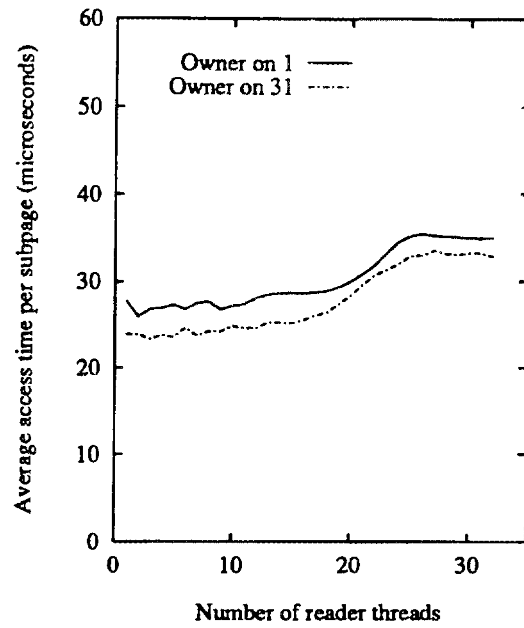


Figure 5: Suite II: 0 local reader threads

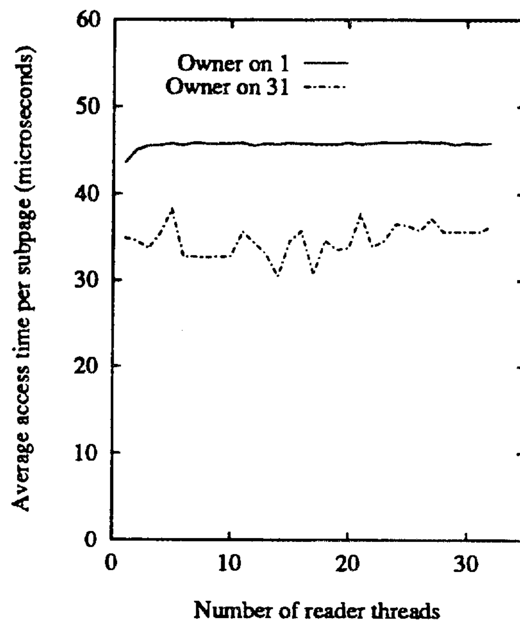


Figure 6: Suite II: 30 local reader threads

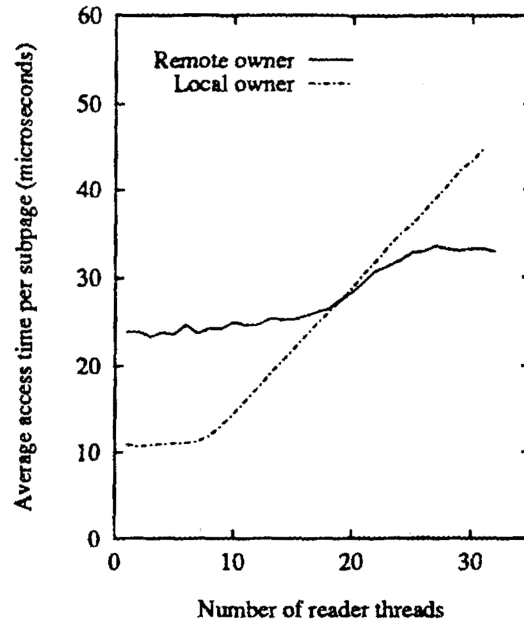


Figure 7: All reader threads on Ring A versus all reader threads on Ring B

for the owner (based on the results of Suite I). The methodology of this suite is to place from 1 up to 63 readers, one at a time, where the choice of ring is determined by the *batch size* of the experiment. The batch size is one of 1, 2, 4, 8, 16, or 32. If the batch size is N, then the first N readers are placed on Ring A, the second N readers are placed on Ring B, the third N readers are placed on Ring A, and so on, until all 63 reader threads are placed. For example, if the batch size is $N = 8$, then the first 8 reader threads are placed on cells 0 through 7 on Ring A, the next 8 reader threads are placed on cells 32 through 39 on Ring B, the next 8 reader threads are placed on cells 8 through 15 on Ring A, and so on. With a batch size of 32, the first 32 readers are placed on Ring A, and the second 32 reader threads are placed on Ring B. Thus, $N = 32$ corresponds to the Suite I experiments. The results of the run with batch size equal to 8 is shown in Figure 8. The overall average subpage access time and the access time for the processors on each ring is graphed in Figure 8. This case exhibits the most improvement over the original batch size of 32 from Suite I.

One result of the filtering performed by the ARD is that many reader threads on Ring B appear to the owner as only one additional reader thread. While the average access time for the first eight readers bound to Ring B is somewhat higher than the average access time for the first eight readers on Ring A, the addition of those eight readers on Ring B have the same effect on the Ring A reader threads as adding only one reader thread. When the second eight threads are added to Ring A, the Ring A curve takes on the characteristic shape seen in Suite

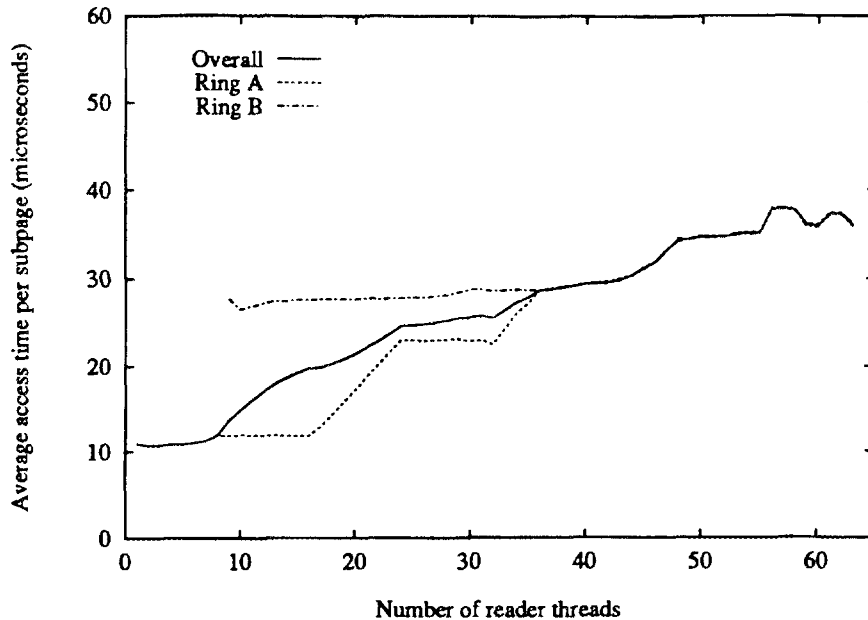


Figure 8: Suite III: Batches of 8 reader threads on each ring

I, but the times for Ring B do not change, as shown in Figure 8. Once Ring A is full, the curve is similar to the curve from Suite I, Figure 4, with the owner on cell 31. A comparison of the results using a batch size of 8 and the results from Suite I with the owner bound to cell 31 is shown in Figure 9. It is clear that the placement of the reader threads across the two rings can affect the overall average subpage access time, especially when the number of reader threads is less than the full complement of processing cells.

3.6. Suite IV

The goal of this suite of experiments is to demonstrate that further performance improvements can be achieved by increasing the amount of automatic prefetching using a simple programming technique. The performance improvement due to automatic prefetching is more pronounced if this technique is combined with a good placement strategy for the owner threads.

The methodology used in this experiment is that the reader threads do not access the same subpage simultaneously. Each reader thread starts reading at a different point in the data set. With this *staggered readers* technique each reader thread has a different access pattern from every other reader thread and the effect of automatic prefetching is more noticeable.

Figure 10 illustrates the average subpage access time of staggered readers as the number of reader threads varies from 1 to 63. The owner thread is placed on cell 31 on Ring A. The performance improvement is due to automatic prefetching of subpages as they pass by on the

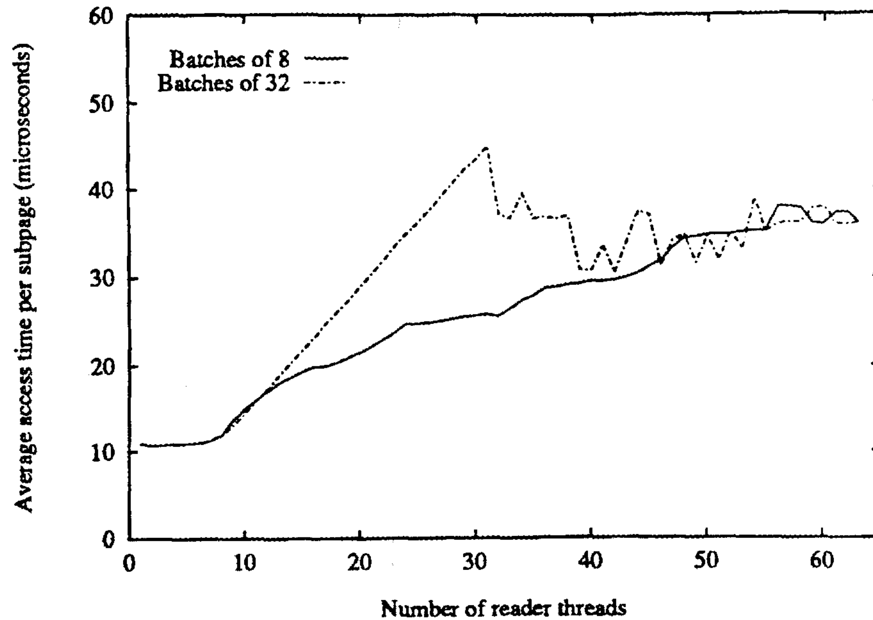


Figure 9: Comparison of Batches of 8 against Batches of 32

ring. This effect is noticeable from 1 to 2 reader threads. As the number of reader threads increases, the access time becomes flat, because a larger number of subpages that have not been requested yet are automatically prefetched by each reader thread. Thus, the owner thread can support more reader threads before it begins to saturate. When the number of readers is larger than 31 (i.e., at least one reader thread is not on the same ring as the owner), the overall average access time increases because of the requests from the second ring. However, the access time of the reader threads on Ring B reduces as the number of readers on Ring B increases because more subpages are brought by the ARD to Ring B and automating prefetching becomes effective on Ring B as well. Automatic prefetching does not take place between ACE:0 rings. Therefore, the readers on Ring B cannot take advantage of the subpages requested by Ring A processors. The access time of the reader threads on Ring A remains flat because the reader threads on Ring A are able to continue automatic prefetching of the responses to requests from Ring B as they circulate on Ring A. After a threshold is crossed (about 57 readers in both rings), the access time on Ring A begins to increase due to queueing effects.

Figure 11 illustrates that further performance improvements can be achieved if the staggered readers technique is combined with multiple owner threads. In this experiment, the data set is divided into four parts each owned by a different owner thread. This reduces the demand at each individual owner. The performance results of two cases are reported. In the first case, the total number of reader threads is equal to 28. The 28 reader threads and the 4 owner threads

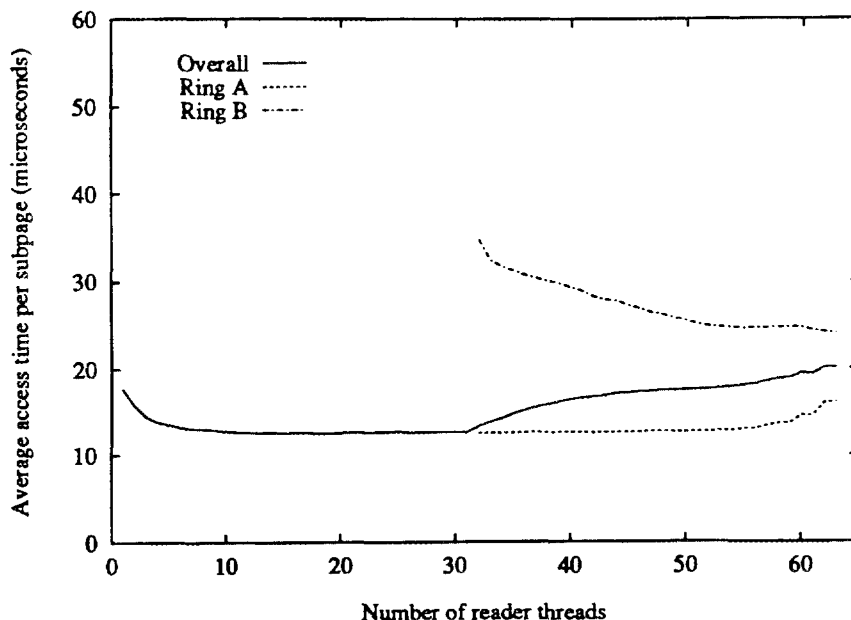


Figure 10: Suite IV: Staggered readers, owner thread on cell 31

reside on the same ring (Ring A). In the second case, the total number of readers is 60, with 28 reader threads and 4 owner threads on Ring A and the remaining 32 reader threads on Ring B. These cases were selected as extreme cases that have among the highest average access time per subpage.

Two placement strategies of the owner threads were selected. The placements are characteristic of the two different performance trends observed in Suite I. As expected, if the owner threads are placed on processing cells 28 to 31 the performance is better than when the owners are placed on cells 0 to 3. If the owner placement is further combined with the staggered readers technique, a performance improvement of about 70% is shown in the 28 readers case. The performance improvement is smaller (20%) in the 60 readers case due to the overhead introduced by traversing the ACE:1 ring.

4. Interpretation of Results

The results of the experiments show that with strategic thread placement and coding that takes advantage of the architecture, memory access times on the KSR1 can remain fairly constant as the number of threads that share a data set increases. A number of implications for applications programmers can be identified:

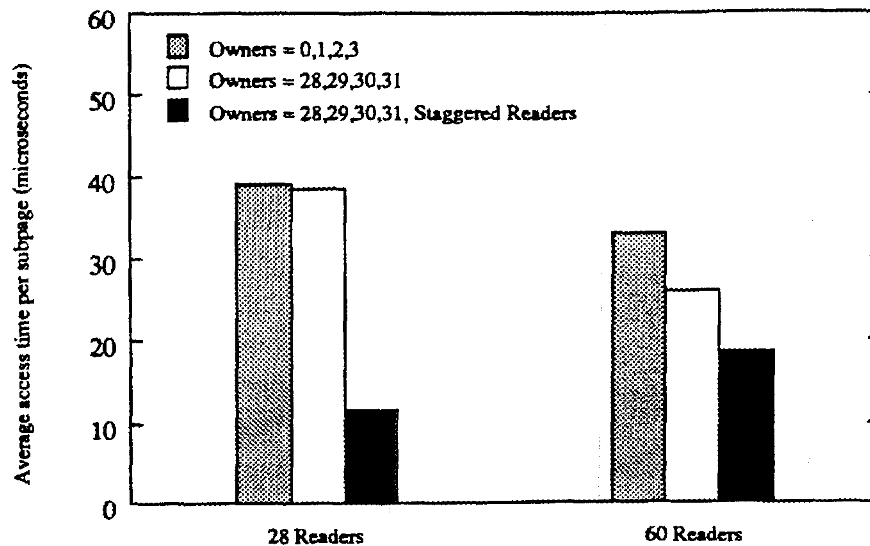


Figure 11: Suite IV: Performance improvements with multiple owner threads

- A good understanding of the architecture and its novel features are essential for improving the performance of application codes. For example, when the advantages of automatic prefetching and ARD filtering are eliminated from application code, then memory access times increase linearly as the number of participating threads increases, as shown in Experiment 0.
- The placement of the owner thread of a data set affects performance. For example, a 20% improvement is seen when the owner thread is moved from cell 1 to cell 31 in Suite I.
- Suite I also shows that additional threads that share a data set can actually improve performance, as long as the threads are placed strategically to take advantage of automatic prefetching and ARD filtering.
- The placement of the owner thread of a data set particularly affects the performance of reader threads that are placed on a remote ring, as shown in Suite II.
- The distribution of reader threads across multiple ACE:0 rings can improve performance substantially, as shown in Suite III.
- Code changes that allow a shared data set to be distributed among several owners, or stagger the access pattern among readers, can also substantially improve the performance, as shown in Suite IV. This effect is less noticeable as the number of reader threads increases.

5. Conclusions and Future Work

The key issue addressed in this paper is the impact of thread placement in a multiprocessor system. A measurement based approach is taken. The specific system considered is the Kendall Square Research KSR1. Even though all processors are physically identical, the specific thread placement affects performance due to unique architectural features. A series of controlled workloads is constructed and placed on the system. Various thread placement experiments are conducted and the results reported.

The primary contributions of this paper include: a description of the key features of the KSR architecture with emphasis on the ALLCACHE memory structure; the design and execution of a series of experiments which illustrate the unique memory access behaviors on the KSR1; and the identification of programming techniques and thread placement strategies which improve the performance of the system.

The experiments illustrate several interesting and unexpected results. These findings indicate that several extensions to this work are appropriate. Such extensions include:

- The testing of actual application codes. The controlled workloads considered in this paper are synthetically generated. Although designed to mimic certain application codes, it is necessary to identify and test actual codes to determine the effects of thread placement.

- The testing on KSR1 systems with multiple (i.e., more than two) ACE:0 rings. Although it is expected that the results reported here generalize to more rings, experimental verification is appropriate. Also, similar experimentation on a KSR2, with a extra level in the ring hierarchy, is needed.

- Analytic modeling and prediction of the KSR1/KSR2. This work represents a preliminary study of the effects of thread placement. Several graphs which represent various particular aspects of the system are given. These measurement figures should form the basis for the construction and validation of appropriate analytic models.

- The combined analysis of explicit prefetch (i.e., the programmer option), poststore, and automatic prefetch. This paper concentrates on automatic prefetch. Other papers have concentrated on explicit prefetch and still others have concentrated on poststore. Understanding when each feature is most beneficial would be worthwhile.

- The testing and analysis of the effects of multiprogramming. In this work, a single multi-threaded workload is considered. Understanding the effects of multiple multi-threaded workloads with respect to the overall thread placement strategy is desired.

Acknowledgments

The helpful information, criticisms, and suggestions provided by Tom Dunigan and Jim Rothnie have significantly improved this paper.

6. References

[Char90] Benjamin Charny. Peak performance in a cell of a KSR supercomputer for multiplication of real matrices. Technical report, Kendall Square Research, Waltham, Ma., October 1990.

[Char92] Benjamin Charny. A programming tool to improve performance and stability of testing results – example of parallel matrix multiply. Technical report, Kendall Square Research, Waltham, Ma., March 1992.

[DHT93] Thomas H. Dunigan, Adolfy Hoisie, and Anne E. Trefethen. Scalability evaluation of the KSR1. Technical report, Cornell University, 1993.

[Duni92] Thomas H. Dunigan. Kendall Square multiprocessor: Early experiences and performance. Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, April 1992.

[Ford92] Rupert Ford. Architecture and Simulation: Laminar to turbulent transition. Technical report, Centre for Novel Computing, University of Manchester, June 1992.

[Ken91] Kendall Square Research, Waltham, Ma. *KSR1 Principles of Operation*, Revision 5.5, October 1991.

[RSW+93] E. Rosti, E. Smirni, T. D. Wagner, A. W. Apon, and L. W. Dowdy. The KSR1: Experimentation and modeling of poststore. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1993.

[Sumn92] Robert Sumner. Architecture and Simulation: Transonic turbulent impinging jets codes. Technical report, Centre for Novel Computing, University of Manchester, June 1992.

[WBHA93] D. Windheiser, E. L. Boyd, E. Hao, and S. G. Abraham. KSR1 multiprocessor: Analysis of latency hiding techniques in a sparse solver. In *Proceedings of The International Parallel Processing Symposium*, Newport Beach, California, April 1993.

ORNL/TM-12462

INTERNAL DISTRIBUTION

1. B. R. Appleton
2. A. S. Bland
3–4. T. S. Darland
5. J. J. Dongarra
6. T. H. Dunigan
7. G. A. Geist
8. K. L. Kliewer
9. M. R. Leuze
10. R. A. Manning
11. C. E. Oliver
12–16. S. A. Raby
17. T. H. Rowan
18–22. R. F. Sincovec
23–27. R. C. Ward
28. P. H. Worley
29. Central Research Library
30. ORNL Patent Office
31. K-25 Applied Technology Library
32. Y-12 Technical Library
33. Laboratory Records - RC
34–35. Laboratory Records Department

EXTERNAL DISTRIBUTION

36. Amy W. Apon, Computer Science Department, Vanderbilt University, Nashville, TN 37235

37. Donald M. Austin, 6196 EECS Building, University of Minnesota, 200 Union Street, S.E., Minneapolis, MN 55455

38. Clive Baillie, Physics Department, Campus Box 390, University of Colorado, Boulder, CO 80309

39. Edward H. Barsis, Computer Science and Mathematics, P. O. Box 5800, Sandia National Laboratory, Albuquerque, NM 87185

40. Robert E. Benner, Parallel Processing Division 1413, Sandia National Laboratories, P. O. Box 5800, Albuquerque, NM 87185

41. Donna Bergmark, 745 E & TC Building, Hoy Road, Cornell University, Ithaca, NY 14853

42. Roger W. Brockett, Harvard University, Pierce Hall, 29 Oxford Street Cambridge, MA 02138

43. Bill L. Buzbee, Scientific Computing Division, National Center for Atmospheric Research, P. O. Box 3000, Boulder, CO 80307

44. Maria Calzarossa, Dipartimento di Informatica e Sistemistica, Università Degli Studi di Pavia, Via Abbiategrasso 209, I-27100 Pavia, Italy

45. Brian M. Carlson, Computer Systems Research Institute, University of Toronto, Toronto, Ontario M5S 1A1, Canada

46. Jagdish Chandra, Army Research Office, P. O. Box 12211, Research Triangle Park, NC 27709

47. Melvyn Ciment, National Science Foundation, 1800 G Street N.W., Washington, DC 20550

48–52. Lawrence Dowdy, Computer Science Department, Vanderbilt University, Nashville, TN 37235

53. Donald J. Dudziak, Department of Nuclear Engineering, 110B Burlington Engineering Labs, North Carolina State University, Raleigh, NC 27695-7909

54. Derek Eager, Department of Computer Science and Engineering, Sieg Hall, FR-35, University of Washington, Seattle, WA 98195

55. Edward Felten, Department of Computer Science, University of Washington, Seattle, WA 98195

56. Geoffrey C. Fox, NPAC, 111 College Place, Syracuse University, Syracuse, NY 13244-4100

57. Offir Frieder, George Mason University, Science and Technology Building, Computer Science Department, 4400 University Drive, Fairfax, Va 22030-4444

58. Dennis B. Gannon, Computer Science Department, Indiana University, Bloomington, IN 47401

59. C. William Gear, NEC Research Institute, 4 Independence Way, Princeton, NJ 08540

60. W. Morven Gentleman, Division of Electrical Engineering, National Research Council, Building M-50, Room 344, Montreal Road, Ottawa, Ontario, Canada K1A 0R8

61. Alan George, Vice President, Academic and Provost, Needles Hall, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

62. Gene Golub, Computer Science Department, Stanford University, Stanford, CA 94305

63. Andy Grant, Computer Graphics Unit, Manchester Computing Centre, University of Manchester, Oxford Rd, Manchester M13 9PL, United Kingdom

64. Eric Grosse, AT&T Bell Labs 2T-504, Murray Hill, NJ 07974

65. John L. Gustafson, Ames Laboratory, 236 Wilhelm Hall, Iowa State University, Ames, IA 50011-3020

66. Robert M. Haralick, Department of Electrical Engineering, Director, Intelligent Systems Lab, University of Washington, 402 Electrical Engineering Building, FT-10, Seattle, WA 98195

67. Michael T. Heath, National Center for Supercomputing Applications, 4157 Beckman Institute University of Illinois, 405 North Mathews Avenue, Urbana, IL 61801-2300

68. John L. Hennessy, CIS 208, Stanford University, Stanford, CA 94305

69. Charles J. Holland, Air Force Office of Scientific Research, Building 410, Bolling Air Force Base, Washington, DC 20332

70. Fred Howes, Office of Scientific Computing, ER-7, Applied Mathematical Sciences, Office of Energy Research, U.S. Department of Energy, Washington, DC 20585
71. Robert E. Huddleston, Computation Department, Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94550
72. Gary Johnson, Office of Scientific Computing, ER-7, Applied Mathematical Sciences, Office of Energy Research, U.S. Department of Energy, Washington, DC 20585
73. Lennart Johnsson, Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1214
74. Harry Jordan, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309
75. Malvyn Kalos, Cornell Theory Center, Engineering and Theory Center Building, Cornell University, Ithaca, NY 14853-3901
76. Kenneth Kennedy, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77001
77. Michael Langston, Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301
78. Richard Lau, Office of Naval Research, Code 111MA 800 Quincy Street, Boston Tower 1, Arlington, VA 22217-5000
79. Robert L. Launer, Army Research Office, P. O. Box 12211, Research Triangle Park, NC 27709
80. E. D. Lazowska, Department of Computer Science and Engineering, Sieg Hall, FR-35, University of Washington, Seattle, WA 98195
81. Tom Leighton, Lab for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139
82. James E. Leiss, Rt. 2, Box 142C, Broadway, VA 22815
83. Heather M. Liddell, Center for Parallel Computing, Department of Computer Science and Statistics, Queen Mary College, University of London, Mile End Road, London E1 4NS, England
84. Rik Littlefield, Pacific Northwest Laboratory, MS K1-87, P.O.Box 999, Richland, WA 99352
85. Ivo de Lotto, Dipartimento di Informatica e Sistemistica, Università Degli Studi di Pavia, Via Abbiategrasso 209, I-27100 Pavia, Italy
86. Manish Madhukar, Computer Science Department, Vanderbilt University, Nashville, TN 37235
87. Allen D. Malony, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403
88. Oliver McBryan, University of Colorado at Boulder, Department of Computer Science, Campus Box 425, Boulder, CO 80309-0425

89. James McGraw, Lawrence Livermore National Laboratory, L-306, P. O. Box 808, Livermore, CA 94550

90. Neville Moray, Department of Mechanical and Industrial Engineering, University of Illinois, 1206 West Green Street, Urbana, IL 61801

91. Richard Muntz, Computer Science Department, University of California at Los Angeles, Los Angeles, CA 90024

92. David Nelson, Director, Office of Scientific Computing, ER-7, Applied Mathematical Sciences, Office of Energy Research, U.S. Department of Energy, Washington, DC 20585

93. Randolph Nelson, IBM, P.O. Box 704, Room H2-D26, Yorktown Heights, NY 10598

94. James M. Ortega, Department of Applied Mathematics, Thornton Hall, University of Virginia, Charlottesville, VA 22901

95. Merrell Patrick, Department of Computer Science, Duke University, Durham, NC 27706

96. David A. Poplawski, Department of Computer Science, Michigan Technological University, Houghton, MI 49931

97. Daniel A. Reed, Computer Science Department, University of Illinois, Urbana, IL 61801

98. Emilia Rosti, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Via Comelico 39, 20135 Milano, Italy

99. Diane T. Rover, 155 Engineering Building, Department of Electrical Engineering, Michigan State University, East Lansing MI 48824

100. Ahmed H. Sameh, Department of Computer Science, University of Minnesota, 200 Union Street S.E., Minneapolis, MN 55455

101. Robert B. Schnabel, Department of Computer Science, University of Colorado at Boulder, ECOT 7-7 Engineering Center, Campus Box 430, Boulder, CO 80309-0430

102. Robert Schreiber, RIACS, MS 230-5, NASA Ames Research Center, Moffet Field, CA 94035

103. Martin H. Schultz, Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520

104. David S. Scott, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006

105. The Secretary, Department of Computer Science and Statistics, The University of Rhode Island, Kingston, RI 02881

106. Charles L. Seitz, Department of Computer Science, California Institute of Technology, Pasadena, CA 91125

107. Giuseppe Serazzi, Politecnico di Milano, Dipartimento di Elettronica e Informazione, Piazza Leonardo da Vinci 32, 20133 Milano, Italy

108. Kenneth C. Sevcik, Computer Systems Research Institute, 10 King's College Road, University of Toronto, Toronto, Ontario M5S 1A1, Canada

109. Horst D. Simon, NASA Ames Research Center, Mail Stop T045-1, Moffett Field, CA 94035

110. Evignia Smirni, Computer Science Department, Vanderbilt University, Nashville, TN 37235

111. Burton Smith, Tera Computer Company, 400 North 34th Street, Suite 300, Seattle, WA 98103

112. Marc Snir, IBM T.J. Watson Research Center, Department 420/36-241, P. O. Box 218, Yorktown Heights, NY 10598

113. Rick Stevens, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439

114. Paul N. Swarztrauber, National Center for Atmospheric Research, P. O. Box 3000, Boulder, CO 80307

115. Anne Trefethen, Engineering & Theory Center, Cornell University, Ithaca, NY 14853

116. Mary Vernon, Computer Sciences Department, University of Wisconsin, 1210 W. Dayton Street, Madison, WI 53706

117. Robert G. Voigt, National Science Foundation, Room 417, 1800 G Street N.W., Washington, DC 20550

118–122. Thomas Wagner, Computer Science Department, Vanderbilt University, Nashville, TN 37235

123. Mary F. Wheeler, Department of Mathematical Sciences, Rice University, P. O. Box 1892, Houston, TX 77251

124. Andrew B. White, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 87545

125. John Zahorjan, Department of Computer Science and Engineering, Sieg Hall, FR-35, University of Washington, Seattle, WA 98195

126. Office of Assistant Manager for Energy Research and Development, U.S. Department of Energy, Oak Ridge Operations Office, P. O. Box 2001, Oak Ridge, TN 37831-8600

127–136. Office of Scientific & Technical Information, P. O. Box 62, Oak Ridge, TN 37831