Evaluating the Effect of Auto-update on the Kendall Square KSR1 *

Karim Harzallah, Hui Li, and Kenneth C. Sevcik

e-mail: {karim,hui,kcs}@csri.toronto.edu

Technical Report CSRI-291

October 1993

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A4

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

*This research was supported by the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre of Ontario.

Abstract

The performance of a computation on a machine with a hierarchical memory organization depends to a great extent on the success of tolerating high latencies. The Kendall Square Research KSR1 provides four architectural features that hide long remote memory accesses: auto-update, combining, poststore, and prefetch. In this paper we investigate the benefits and limitations of the auto-update. We also derive accurate analytical models that capture the interaction of the different components of the memory system. Finally, we apply the model to predict the memory access times for several architectural alternatives.

1 Introduction and Goals

The effectiveness of large scale computing on scalable multiprocessors depends to a great degree on how well the memory hierarchy is managed, and to what extent locality of communication can be achieved automatically. The KSR1 multiprocessor includes four architectural features that address the memory latency problem: (i) automatic update, (ii) prefetch, (iii) poststore, and (iv) combining. Ultimately, we would like to develop a methodology for modeling the time-varying characteristics of a real application reference pattern and to reflect the effects of the latency hiding techniques both individually and in combinations. Here, our goal is, within the context of a hot-spot workload, to understand and to model the auto-update feature, and to investigate the sensitivity of memory performance to system parameters.

There are many advantages to having a simple tractable model for the behavior of a complex system. For instance, an understanding of the relationship between the memory access pattern and the latency hiding techniques can reveal those aspects of the program behavior where further optimization could improve overall performance. Our approach is to transform and to extend a model previously derived and validated for the Hector multiprocessor [HS93]. After parameterizing and validating our model using KSR1 measurements, we apply it to address "what if" questions that arise in design-tradeoffs.

The rest of the paper is organized as follows. In the next section, we give an overview of the KSR1 architecture and its novel features. In Section 3, we discuss the related work. Section 4 describes the synthetic workload used in our experiments. The model structure and parameterization are in Section 5, and its validation is described in Section 6. Section 7 presents a model for auto-update. In Section 8 we apply the model to explore the design space. Finally, we summarize our results and conclusions in the last Section.

2 Overview of the KSR1 Architecture

KSR1 architecture consists of a hierarchy of unidirectional rings. The ring at the bottom of the hierarchy is referred to as Ring:0. It consists of 32 processing elements and two cells dedicated to off-ring routing. Each processing cell consists of an integer and floating point unit, 0.25 MB instruction and data subcache, 32 MB local cache, a Co-Execution unit, some cell interfaces, and an external I/O unit. Two or more Ring:0's can be interconnected by another ring (known as Ring:1) at the next level of the hierarchy. Figure 1 shows a two ring configuration of the KSR1 multiprocessor. The Allcache Engine moves the data to the requesting processor on demand. This hardware management of memory allocation and movement provides a shared programming model.

The papers by Dunigan [Dun92] and by Wagner *et. al.* [WSA+93] provide more details on the KSR1 architecture.

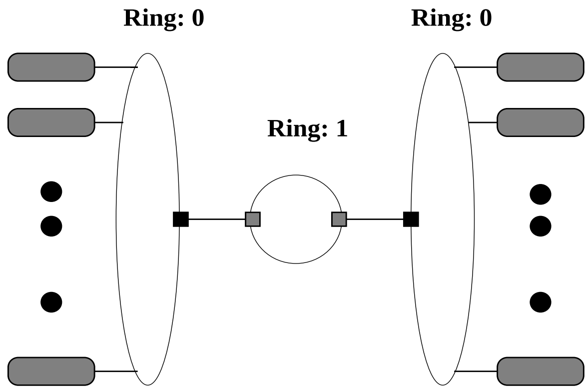


Figure 1: KSR1 system configuration.

A key to scalability is to keep data close to the processor that is using it. In the following section we describe KSR1 's four latencies hiding techniques that are intended to maximize locality of reference. The first two are hardware features, and the other two are software techniques.

Auto-Update: A processor with an invalid copy of some data can potentially update its own copy by observing a request from a different processor that happens to pass by on the ring. Then, when the processor next accesses the data, a valid copy will already be in its local memory. For the auto-update to work, the processor has to be in the return path of the requester, and the local cache has to be ready to exploit the availability of the new copy. So the effect of auto-update depends both on the status of the processors, and on their relative placements with respect to the owner of the valid data. Assume that all the valid copies of data are owned by a single processor (denoted by R_0). Then ideally, when reader R_i issues a read request, readers R_j (where $j < i$) can update their own copy as the valid copy returns to R_i. When more than one valid copy of an item is available, the first one encountered by a read request will respond. Figure 2 illustrates the impact of the auto-update after every processor R_i reads data i.

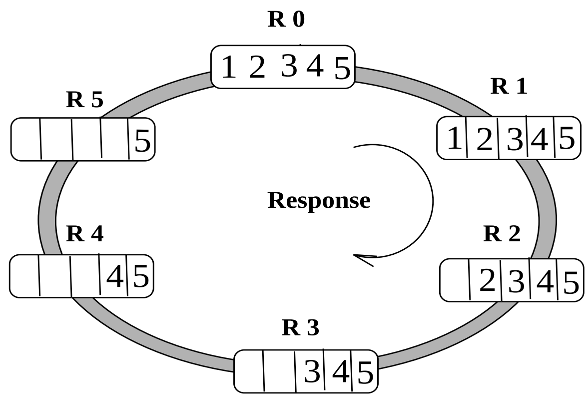


Figure 2: Benefit of auto-update when each processor R_i reads data i. (All data is initially held only by processor R_0.)

Combining: When multiple off-ring requests for the same data reach the Allcache Router Directory (ARD) device, which is responsible for directing requests between rings, they are combined such that only a single request makes it through to the destination ring. Irrespective of the number of off-ring readers, the load seen by the target memory is equivalent to the one placed by a single off-ring request. This technique reduces the load on the memory and the traffic on the target ring, and consequently reduces the average memory access time for off-ring requests destined to the same data.

Prefetch: The prefetch feature allows the overlap of memory access with computation by initiating data references before the data is needed rather than stalling the processor to wait for the data to be fetched. The local cache queues the prefetch requests generated by the processor in a prefetch queue and waits for their response to return. Because the processor issuing the prefetch instruction does not have to wait for completion of the request, access latency can be reduced.

Poststore: The poststore instruction enables the broadcast of an updated value to other caches that may need it. The advantage stems from the fact that the processors with an invalid copy may no longer need to carry out a remote access once the new value has been poststored.

The KSR1 and Hector architectures have many important features in common. They are both ring-based, hierarchical shared-memory multiprocessors, with physically distributed memory and a scalable interconnection network. In addition, both systems support hardware techniques for hiding and tolerating memory latency. However, the architectures have important differences as well. In KSR1 the allocation, movement and coherence of data are managed automatically by the hardware, while Hector's cache coherence is provided only by software [HS93].

3 Related Work

Rosti *et. al.* [RSW+93] analyze the poststore, and show that the advisability of using poststore depends both on the application and the current system load. Their experimental results point out that for the poststore to be useful there must be enough readers to off-set the poststore overhead. The authors also indicate that, similar to sender-initiated transfers [ELZ86], the poststore is most effective under light loads. Although their analytical and experimental data are in good agreement, their justification for their method (which is based on using an M/M/6 queueing model) is based on empirical observations rather than formal definition of the interaction of the individual components as they contribute to the overall performance of the system and the mechanics of the architecture being modelled. Furthermore, it is not clear how their Petri net model can be extended to account for a deeper hierarchy.

Windheiser *et. al.* [WBH+93] experimentally evaluate KSR1 latency hiding techniques, the prefetch, the poststore, and the auto-update. Their quantitative analysis of an iterative sparse linear equations solver reveals that the benefits of the auto-update are significant. However, the effects of the prefetch and the poststore on the overall performance are negligible for this algorithm. The authors suggest that in order for an application to take advantage of the prefetch and the poststore, the programmer may need to restructure the application significantly. Because their experiments were limited to a single ring based KSR1 system, the issue of scalability was not addressed. It is difficult to project the benefits of these hiding techniques with larger configurations.

Related work by Kahhaleh [Kah93] evaluates the overhead associated with using the prefetch and the poststore. The author points out that the benefit of the prefetch is limited by the size of the prefetch queue, and that the use of the poststore is only justified for a high number of processors. While it has been shown that delaying the poststore operation was found to increase the probability of benefiting from the broadcast, thereby improving the performance, the issue of how long this delay should be and whether this optimization can be automated is still an open question.

Boyd *et. al.* [BWAD93] propose a methodology to evaluate the communication performance of MPP's. They consider a single ring KSR1 system, and construct a set of synthetic workloads for which they vary the average amount of communication per processor, the degree of sharing among the processors, and the computation to communication ratio. Their results demonstrate the importance of the automatic update feature, and its limitations in the face of contention.

Other work, by Harzallah and Sevcik [HS93], showed that non-uniform traffic distributions characterized by hot-spots can substantially increase the memory access times. They formulate a queueing model that includes the effects of increased contention for system resources, and the impact of the arbitration algorithm on the network traffic.

Our work complements the recent experimental study by Wagner *et. al.* [WSA+93]. The authors investigate the effects of thread placement on memory access times in a two ring KSR1 configuration. They show that additional benefits can result from the auto-update feature by staggering concurrent readers, and that the ARD's combining property is substantial enough that performance can be improved by placing the readers physically further from the owner.

Many researchers have conducted empirical comparisons of the performance of various applications on the KSR1 relative to other multiprocessors, including Dunigan [Dun92], Singh *et. al.* [SJHG93], and Stumm, *et. al.* [SVW+92].

4 Experimental Workload

Although the behavior of real applications is difficult to understand, it is often possible to partition the application into phases each involving a distinct data access pattern. Synthetic workloads can be created to mimic those access patterns. This provides insight into the overall behavior of complex applications. Our workload model expresses the access rate seen by an individual memory model in terms of the computational granularity of the application, the observed delay, and the architecture's ability to tolerate communication latencies. Controlling the number of participating processors of each class (processors are categorized according to their position in the system hierarchy with respect to a given memory module), and the average time between successive remote requests, allows us to examine an interesting set of access patterns.

In this paper, we focus our attention on memory access patterns that are most crucial to performance, in particular the hot-spot access pattern. Although this access pattern may not describe any complete real application, many program codes experience similar patterns of access during phases of their execution. For example, in the LU decomposition algorithm, processors simultaneously access the pivot row (right after it is normalized) creating a hot-spot.

Figure 3 shows the relative time consumed by the hot-spot phase when compared to an *ideal* (but unrealizable) LU decomposition for which the pivot row is local to every processor during the

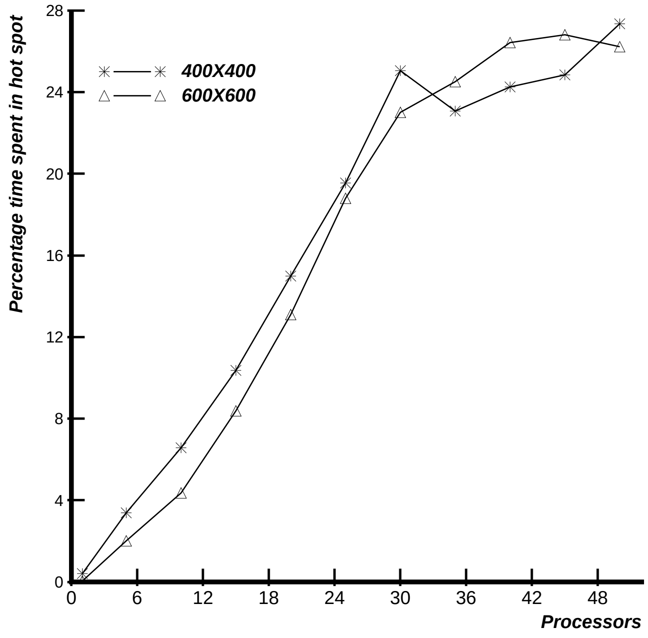


Figure 3: Impact of hot-spot phase on overall performance.

life time of the application. We note that the slope of the curves levels off when we move to the second ring. This is due to the optimizations performed at the Allcache Engine hierarchy level where off-ring requests accessing the same data are combined.

5 Analytic Model of Memory Access Latencies

5.1 Model Structure

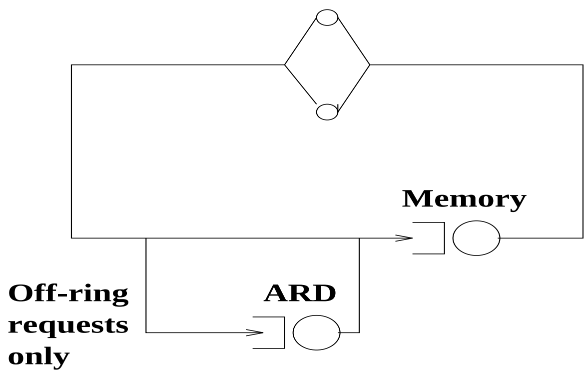


Figure 4: Model of KSR1 system.

Figure 4 illustrates the canonical queueing network model of a two-ring KSR1 configuration. Service centers are used to represent the memory module and the Allcache Router Directory (ARD) devices. The ARD cell is responsible for connections to the next higher level of the Allcache Engine hierarchy. The ARD has in fact two sections: the downlink side and the uplink side. The downlink side is part of Ring:0, and the uplink side is part of Ring:1. In order to reduce the size of the network to be solved, we took a hierarchical approach and replaced the entire aggregate by a single service center that represents the behavior of both ARD sections. The queueing network model considers two distinct classes of requests, each of which has its own workload intensity (specified by the level in the memory hierarchy, or the think time and the service demands). Although this model contains a memory and an ARD queue, it is still separable, and so can be evaluated directly by Mean Value Analysis (MVA) techniques [LZGS84]. Both the ARD and the memory have queues of capacity 32 [Fos93] which is enough to accommodate as many of the processors as might concurrently require it.

Here we present an analytical model of the memory structure of a KSR1 system. In this configuration, we distinguish two classes of remote readers: those from the same ring as the owner of the data (within-ring requests) and those from the other ring (off-ring requests). The memory access time includes the time it takes the request to traverse the network (in the absence of contention), the memory service time, and the contention observed both at the memory and at the network level. For many memory intensive applications it was observed that, relative to the memory contention, the contention for the ring appears to have little effect on the performance of the system. We denote by C_w and C_o the cycle time associated with within-ring and off-ring requests respectively. They are given by:

$$C_w = g + t_w + d_m \tag{1}$$
$$C_o = g + t_o + d_m + d_r \tag{2}$$

where t_w and t_o are the costs of accessing the memory in the absence of contention from within the same ring and from another ring, respectively. d_m is the average memory contention delay incured at the memory module, d_r the average delay observed at the ARD level, and g the average number of cycles between successive remote memory requests. Extending the model to accommodate different g values for different processors is straightforward. It follows that $1/C_w$ and $1/C_o$ are the injection rates of requests from processors of the two classes. In Table 1, the model notations are summarized.

We use an M/D/1 model to capture the queueing observed at the memory and at the ARD system. If we let U_m be the utilization of the memory, S_m the service time of the memory, n_w the number of readers from within the same ring, and n_o the number of readers from the other ring, then d_m and d_r are given by:

$$d_m = S_m \frac{n_w + n_o - 1}{n_w + n_o}[U_m + \frac{U_m^2}{2(1-U_m)}] \tag{3}$$
$$d_r = S_r \frac{n_o - 1}{n_o}[U_r + \frac{U_r^2}{2(1-U_r)}] \tag{4}$$
$$U_m = S_m[\frac{n_w}{C_w} + \frac{n_o}{C_o}] \tag{5}$$
$$U_r = \frac{S_r * n_o}{C_o} \tag{6}$$

Equations (3) and (4) estimate the average arrival instant queue length. The approximation is based on the assumption that the amount the queue length is diminished by the removal of a single request is equal to the amount the request contributes to the queue length. In fact the contribution

C_w	Within-ring cycle time.
C_o	Off-ring cycle time.
t_w	Within-ring cost of contention free memory access.
t_o	Off-ring cost of contention free memory access.
d_r	Average delay observed at the ARD level.
g	Average number of cycles between successive remote memory.
U_m	Memory utilization.
U_r	ARD utilization.
S_m	Memory service time.
S_r	ARD service time.
n_w	Number of readers from within the same ring.
n_o	Number of readers from the other ring.
d_m	Average delay observed at the memory module.
d_r	Average delay observed at the ARD level.

Table 1: Model Notation.

to the queue length depends on the rate of access and therefore on the class the request belongs to. We believe that extending the model to account for this distinction between classes at the queue level introduces more complexity to the solution without additional accuracy or insight. The average latency for within-ring and off-ring requests is simply:

$$L_w(g, n_w, n_o) = C_w - g \tag{7}$$

$$L_o(g, n_w, n_o) = C_o - g \tag{8}$$

These latencies can be calculated as a function of the workload intensity parameters, g, n_w, and n_o using equations (1) to (8). Since the utilization is known only after the model has been evaluated, we employ the iterative solution scheme shown below.

1. Set U_m to zero.
2. Evaluate the various delays C_w, C_o, d_m, d_r using eqs. (1), (2), (3) and (4) respectively.
3. Calculate the memory utilization based on the new values of the delays using eq. (5).
4. Calculate the router utilization based on the new values of the off-ring delays using eq. (6).

We iterate steps 2 through 4 until successive estimates of memory utilization, U_m, are sufficiently close, and then obtain performance measures from the final iteration.

5.2 Model Parameterization

In order to make the model realistic and to have the opportunity to validate it, we determined parameter values from measurements taken during use of KSR1 system with 56 processors. These measurements included per class average inter-request arrival time (i.e., $g + t_w$ for within-ring readers and $g + t_o$ for off-ring readers), average ARD processing time, and average memory requirements.

Because of KSR1 architectural peculiarities, we had to carefully craft our experiments in order to obtain the desired parameter values. We had to isolate the effects of the auto-update, by forcing readers to access different data from the same memory module (in the case of the memory experiment). We also had to seclude some of the combining effects that tend to reduce the average off-ring memory access latency. For this, we mapped the off-ring readers to different memory modules, so that all the off-ring requests go through the ARD without being either combined or auto-updated as they will be accessing distinct data structures residing in distinct memory modules (in the case of the ARD experiment).

Figure 5(a) and Figure 5(b) show memory access times as a function of the number of processors. For various numbers of processors reading from a single memory module, the memory access times was observed and plotted in Figure 5(a). Figure 5(b) graphs the memory access times as the number of remote requests traversing the ARD varies from 15 to 30. As the workload intensity rises, congestion increases, and residence time along with it. Initially, this increase is gradual. As the number of readers grows, however, residence time increases at a faster rate, until the center approaches saturation. At saturation (i.e., when the server's pipeline is filled with a backlog of unprocessed requests), the throughput is equal to the inverse of the service time of the bottleneck center. Table 2 includes a summary of the model parameters. The values for t_w and t_o were obtained from simple experiments using two processors. We declared a shared array small enough to fit in the sub-cache, we let one processor write to the array, thus making it locally cached, next we let the other processor read the data remotely. The memory access time was measured both with the initializer placed on the same ring and on the other ring. Notice that with two processors there is no reason to worry about auto-update and combining effects.

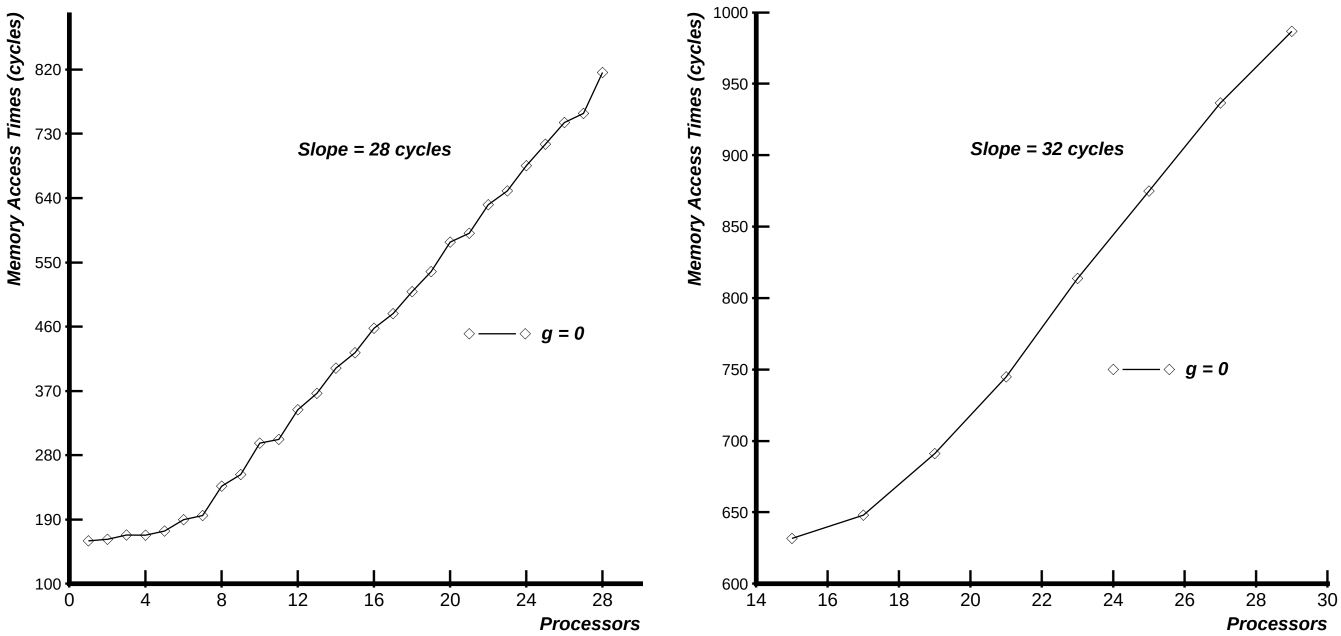
$S_m = 28$	$S_r = 32$	$t_w = 160$	$t_o = 540$

Table 2: The default values for model parameters (in cycles).

6 Validation

To validate our model, we conducted four experiments. The aim of the first experiment was to show that the model captures the additional contention due to increasing the number of participating readers. We further investigate the accuracy of our analytical predictions by examining the interference between within-ring and off-ring readers while keeping the total number of readers constant. The objective of the next two experiments is to test the scalability of the model by increasing the number of readers while observing the interference of the two classes.

In all the experiments, the shared data is a two dimensional array. We let one processor initialize this array, subsequently owning it, and we let the other processors read one word per subpage (to single out remote requests, since reading consecutive words of the same subpage will result in cache hits). Because we want to exclude the effects of auto-update, we assign different processors to different rows of the matrix.



(a) Determining memory service time.

(b) Determining ARD service time.

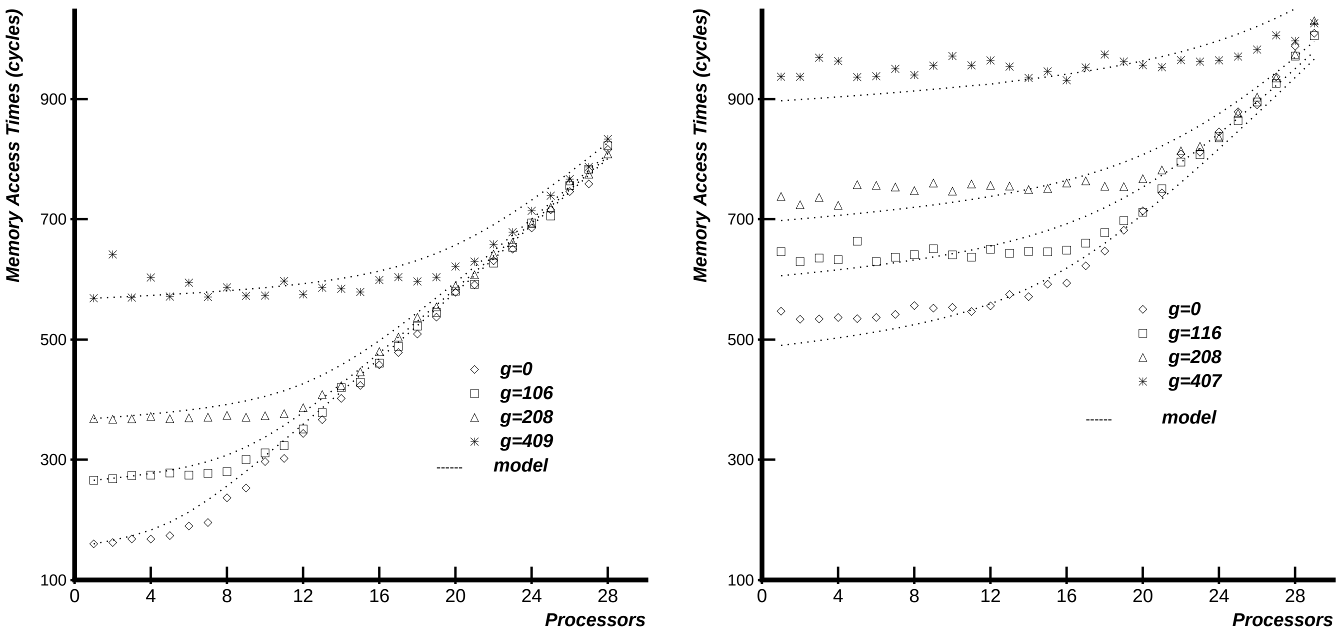
Figure 5: Determining service times of memory and ARD.

6.1 Effects of g on a single class request

Figures 6(a) and 6(b) show memory access times as a function of the number of reader processors for within-ring and off-ring accesses with various g values. There is a good agreement between measurements and the results of the analytical model. As the memory/ARD approaches saturation, the increase of response time is proportional to the number of readers. The model tends to be less accurate for higher values of g (possibly due to the exponential interarrival time assumption). By increasing g, we vary both the mean and the variance (proportionally) of the interarrival distribution. In practice, we observed that the variance does not change at the same rate as the mean. We are interested in memory intensive applications, for which values of g less than 50 are quite common, because there is room for pipelining many instructions, and g represents the non-overlapped portion of the time. We conclude that the model gives a good approximation for the cases we are interested in.

6.2 Class interference

For the rest of the experiments we will set g to 0. In this experiment, we fix the total number of readers to 30 and we vary the ratio of within-ring readers to off-ring readers. The goal is to examine the interaction between the two classes of accesses, and to study the impact this interference may have on the average memory access time of each class. Figure 7 shows that the evaluation of the model yields performance measures that are qualitatively consistent with the experimental data. As we increase the number of within-ring readers, the times for both within-ring accesses and off-ring accesses increase. Since the within-ring requests get to the memory faster, increasing



(a) Accesses to memory on same ring.

(b) Accesses to memory on the other ring.

Figure 6: Measured and modelled memory access times for various numbers of processors.

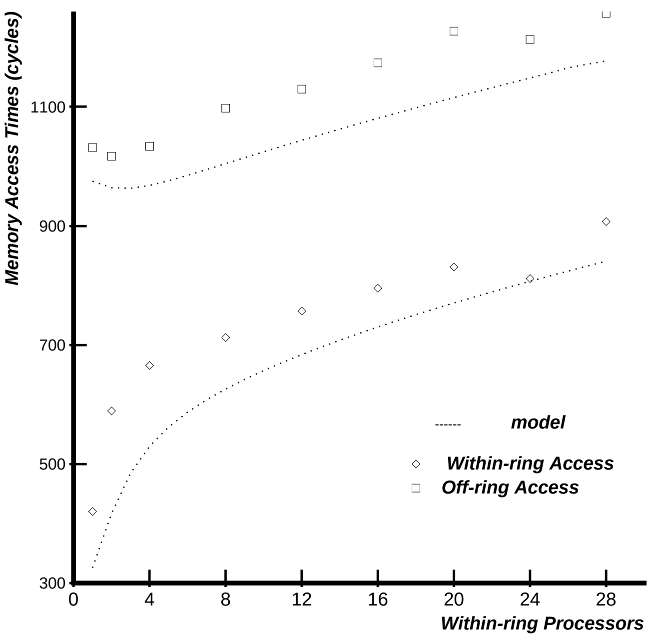


Figure 7: Measured versus modelled memory access times with the total number of readers set to 30.

the readers of this type keeps the memory busier (compared to off-ring readers) over a given period of time. The dip of the off-ring curve corresponds to shifting the bottleneck from the ARD to the memory by increasing the number of within-ring readers.

6.3 Steady state behavior

In this experiment, we measure the effect of having an increasing number of processors all continuously accessing a single memory module. We vary the number of processors from 1 to 50, starting with the within-ring processors. Once we use all 28 of them, we add readers from the other ring. It is conceivable that in order to balance the load some applications may take into account the difference in memory access time, and partition the workload accordingly, such that the processors take equivalent execution time. In this case, the memory performance is mostly defined by its steady state. Since we were interested in the steady state behavior of the memory, we made sure that the off-ring readers see the interference of the within-ring readers during the whole read phase. This was done by letting the within-ring readers access more data. (For the KSR1's parameters, letting the within-ring processors read twice as much data as is read by the off-ring processors was enough to ensure that the off-ring readers finish first.)

Figure 8(a) shows that when we start using off-ring readers the slope of the within-ring curve drops a little due to the competition for shared resources experienced from the off-ring requests. Within the context of our experiment, processors can only handle one outstanding request at a time. Thus, the readers can only proceed at rates appropriate to observed contention delays. The overall injection rate is therefore defined not only by the number of readers involved but also by their relative distance from the target memory. By introducing off-ring readers, the within-ring readers become subject to a lighter competition (when compared to the case where the new readers are of the within-ring type), which leads to a faster relative access time.

6.4 Reading equal amount of data

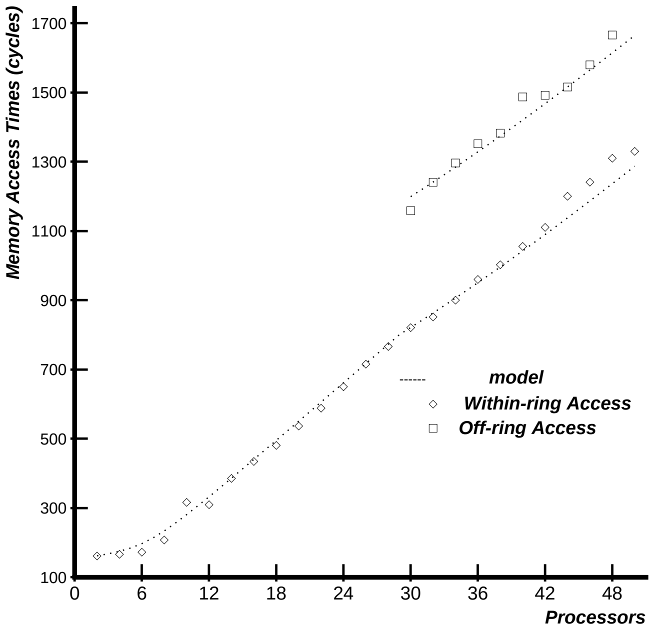
In practice, especially in SPMD type applications, it is very common to have all the readers read the same amount of information. Since within-ring processors progress faster than off-ring processors, the off-ring processors will face interference for a period of time (while competing for the same memory module with the within-ring processors) then they will be competing only among themselves as illustrated in Figure 9. We can distinguish here two phases, each of which can be modelled. Figure 8(b) shows the impact this may have on the average memory access times of both within-ring and off-ring requests.

If we denote by T_w and T_o the average memory access time for within-ring and off-ring requests for this case, then T_w is simply $L_w(0, n_w, n_o)$ and T_o is given by

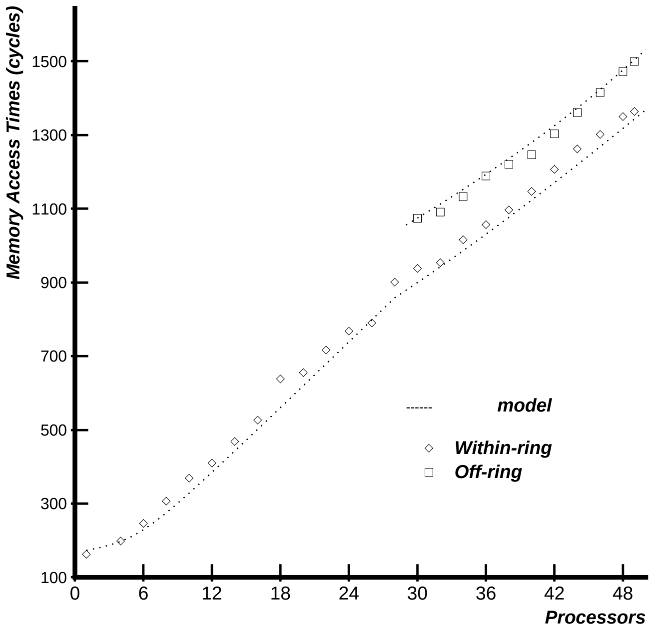
$$T_o = W * L_o(0, n_w, n_o) + (1 - W) * L_o(0, 0, n_o) \tag{9}$$

where W is the fraction of time spent by off-ring processors in the first phase. A good approximation of the weight is given by:

$$W = \frac{L_w(0, n_w, n_o)}{L_o(0, n_w, n_o)} \tag{10}$$



(a) Within-ring processors read more data.



(b) All processors read the same amount of data.

Figure 8: Measured and modelled average memory access times with various numbers of processors.

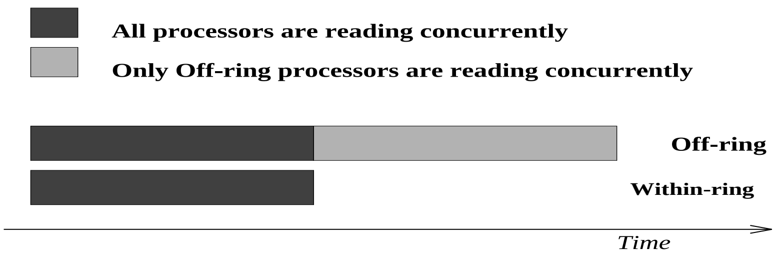


Figure 9: Using equal data size.

T_o thus reduces to

$$T_o = L_w(0, n_w, n_o) + (1 - \frac{L_w(0, n_w, n_o)}{L_o(0, n_w, n_o)}) * L_o(0, 0, n_o) \quad (11)$$

7 Modeling Auto-update

The memory reference behavior of an application depends largely on the intrinsic nature of the application. However, the programmer still has considerable flexibility in manipulating the algorithm, data structures, and program structure to change memory reference patterns in order to better exploit the memory hierarchy and the peculiarities of the architecture.

In this section, we explore the benefits offered by the auto-update feature of the KSR1, and we show how the hot-spot access pattern can be rechoreographed to achieve lower memory access times. Modeling the effects of auto-updates is difficult because it is hard to predict the exact number of readers that will benefit from the auto-update. In the presence of auto-update we can distinguish two phases. The first phase is a hot-spot access for a single memory module. The second phase is a mixture of local accesses and remote accesses (for those readers that did not fully benefit from the auto-update because of their relative position within the ring and within the hierarchy of rings, and/or because they were busy reading some other data).

Shared data is initially stored in one processor (named processor 0) as a two dimensional array, where the number of rows is equal to the number of readers. To take full advantage of the auto-update feature, we try to avoid simultaneous access to the same subpage by letting processor i read the data starting at row i. This interleaving technique increases the amount of automatic update. The data movement among processors hence creates two distinct phases:

1. In the first phase, each processor i accesses row i from processor 0 creating a hot-spot. After this phase, ideally, row i is auto updated on processor j if $i < j$ (as suggested in Figure 2).
2. In the second phase, the processors access the auto updated data locally, and remotely access the rest (not necessarily from processor 0 as there are valid copies lying around, and the first encountered processor with a valid copy will respond). This tends to alleviate the contention at the memory module of processor 0.

Our analysis includes the following assumptions and simplifications.

- each processor makes similar progress during the 1st phase.
- each processor auto-updates its own copy whenever possible ("perfect update").
- new copies are well spread so that contention is negligible in the second phase.
- the benefit of the auto-update in the second phase is negligible.

T_w^1	Within-ring average access time in phase1
T_w^2	Within-ring average access time in phase2
T_o^1	Off-ring average access time in phase1
T_o^2	Off-ring average access time in phase2
t_l	Average access time of local data

Table 3: Notation.

7.1 Single Class Model

In Non-Uniform Memory Access (NUMA) architectures, requests that are issued from different memory hierarchies exhibit markedly differing resource usage. In the KSR1 for instance, the off-ring class is ARD bound while the within-ring class is memory bound. Such systems can be modelled adequately only by multiple class models, since the single class assumption that requests are indistinguishable is inappropriate. In the following subsection, we analyze the case where the customer population consists of either within-ring or off-ring requests. Later we build up on the single class analysis and consider a workload with a mixture of requests.

7.1.1 Within-Ring

$$
\begin{aligned}
T_w &= \frac{1}{n_w} * T_w^1 + (1 - \frac{1}{n_w}) * T_w^2 && (12)\\
T_w^1 &= L_w(0, n_w, 0) && (13)\\
T_w^2 &= \frac{1}{n_w * n_w} \sum_{i=1}^{n_w} [(i-1) * L_w(0,1,0) + (n_w - i) * t_l] && (14)
\end{aligned}
$$

The average access time is the weighted sum of the average access times during both phases. The duration of the first phase is proportional to the expected number of reads by individual processors, since the data structure is equally partitioned among all readers the weight of the first phase is thus $1/n_w$. The average access time of the second phase is the arithmetic mean of all access times during this phase. Each is a combination of local and remote components whose contributions depend on the relative distance between the reader and the owner.

7.1.2 Off-Ring

The equations for the off-ring case are established by arguments analogous to those of the within-ring case:

$$
\begin{aligned}
T_o &= \frac{1}{n_o} * T_o^1 + (1 - \frac{1}{n_o}) * T_o^2 && (15)\\
T_o^1 &= L_o(0, 0, n_o) && (16)\\
T_o^2 &= \frac{1}{n_o * n_o} \sum_{i=1}^{n_o} [(i-1) * L_w(0,1,0) + (n_o - i) * t_l] && (17)
\end{aligned}
$$

Figure 11(a) and Figure 11(b) compare predicted and actual memory access times for increasing number of readers. We can clearly observe that most of the benefit arises when we move from a single reader to as few as four readers. As we involve more and more processors, we extend the advantage of the auto-update to more readers, meanwhile we accentuate the contention during the first phase. This trade-off is well illustrated in both Figure 11(a) and Figure 11(b).

7.2 Two Class Model

In the presence of two rings, the interference of two classes of requests comes into play. Automatic update does not happen across the ARD. Within-ring readers can benefit from off-ring requests, yet off-ring requests cannot take advantage of the subpages requested by within-ring readers. We can distinguish two or three phases depending on the number of participating readers from each class (See Figure 10). The shared data structure is read such that each within-ring processor reads $1/n_w$ of the information, and each off-ring processor reads an $1/n_o$ th. The purpose of such partitioning is to reduce the number of remote accesses by benefiting from the auto-update property.

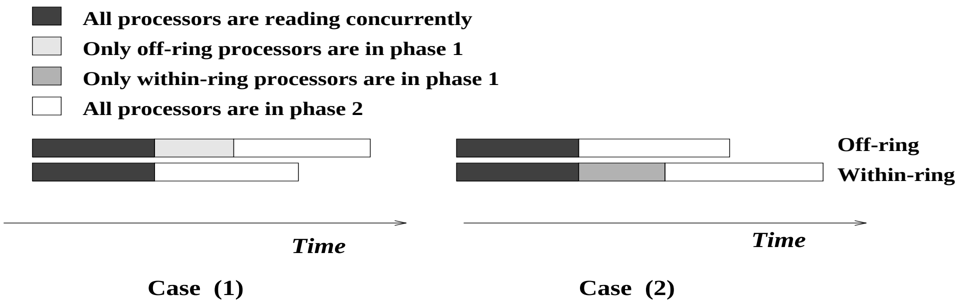


Figure 10: Interaction between within-ring and off-ring processors.

We will make additional assumptions to keep the model as simple as possible.

- no interference in the 2nd phase.
- maximum contention in the 1st phase.
- no auto-updates of within-ring copies occur while the data is moved around due to an off-ring request.

The last two assumptions make the model a rather pessimistic one.

Case 1: $\frac{L_w(0,n_w,n_o)}{n_w} < \frac{L_o(0,n_w,n_o)}{n_o}$

$$
\begin{aligned}
T_w &= \frac{1}{n_w} * T_w^1 + (1 - \frac{1}{n_w}) * T_w^2 && (18)\\
T_w^1 &= L_w(0, n_w, n_o) && (19)\\
T_w^2 &= \frac{1}{n_w * n_w} \sum_{i=1}^{n_w} [(i-1) * L_w(0,1,0) + (n_w - i) * t_l] && (20)\\
T_o &= W_1 * T_o^1 + W_2 * T_o^2 + W_3 * T_o^3 && (21)\\
&= \frac{1}{n_w} * L_w(0, n_w, n_o) + W_2 * T_o^2 + (1 - \frac{1}{n_o}) * T_o^3 && (22)\\
W_2 &= \frac{1}{n_o} - \frac{1}{n_w} \frac{L_w(0, n_w, n_o)}{L_o(0, n_w, n_o)} && (23)
\end{aligned}
$$

$$T_o^2 = L_o(0,0,n_o) \tag{24}$$

$$T_o^3 = \frac{1}{n_o * n_o} \sum_{i=1}^{n_o} [(i-1) * L_w(0,1,0) + (n_o - i) * t_l] \tag{25}$$

Case 2: $\frac{L_w(0,n_w,n_o)}{n_w} > \frac{L_o(0,n_w,n_o)}{n_o}$

$$T_w = W_1 * T_w^1 + W_2 * T_w^2 + W_3 * T_w^3 \tag{26}$$

$$T_w = \frac{1}{n_o} * L_o(0,n_w,n_o) + W_2 * T_w^2 + (1 - \frac{1}{n_w}) * T_w^3 \tag{27}$$

$$W_2 = \frac{1}{n_w} - \frac{1}{n_o} \frac{L_o(0,n_w,n_o)}{L_w(0,n_w,n_o)} \tag{28}$$

$$T_w^2 = L_o(0,n_w,0) \tag{29}$$

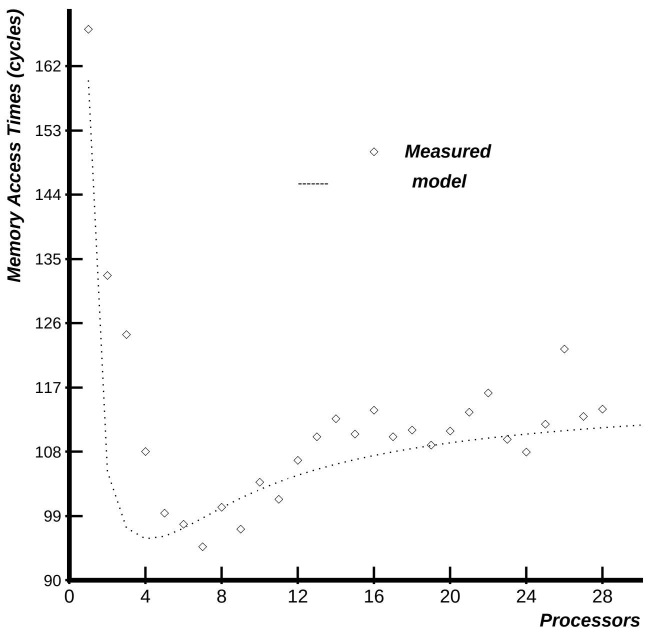
$$T_w^3 = \frac{1}{n_w * n_w} \sum_{i=1}^{n_w} [(i-1) * L_w(0,1,0) + (n_w - i) * t_l] \tag{30}$$

$$T_o = \frac{1}{n_o} * T_o^1 + (1 - \frac{1}{n_o}) * T_o^2 \tag{31}$$

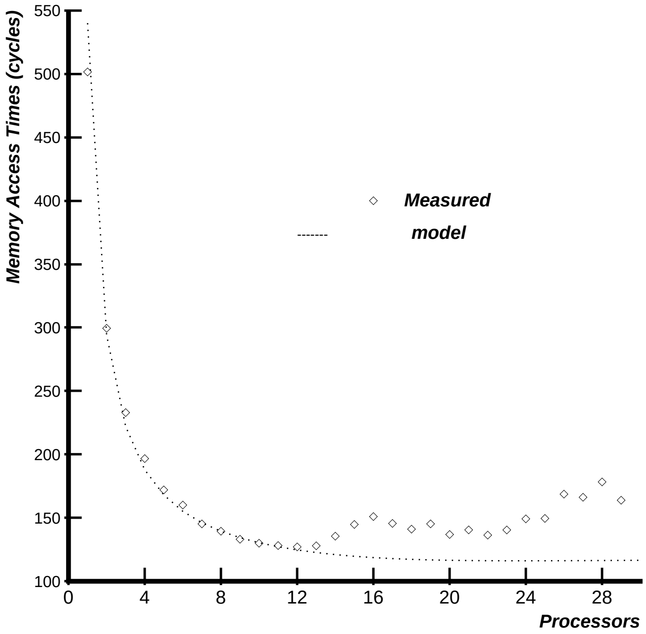
$$T_o^1 = L_o(0,n_w,n_o) \tag{32}$$

$$T_o^2 = \frac{1}{n_o * n_o} \sum_{i=1}^{n_o} [(i-1) * L_w(0,1,0) + (n_o - i) * t_l] \tag{33}$$

Figure 12(a) and Figure 12(b) show that the evaluation of the model yields performance measures that are qualitatively consistent with measurements. Our assumptions result in a pessimistic model (tends to be pronounced in Figure 12(a)) that still captures the dynamics of the auto-update feature.

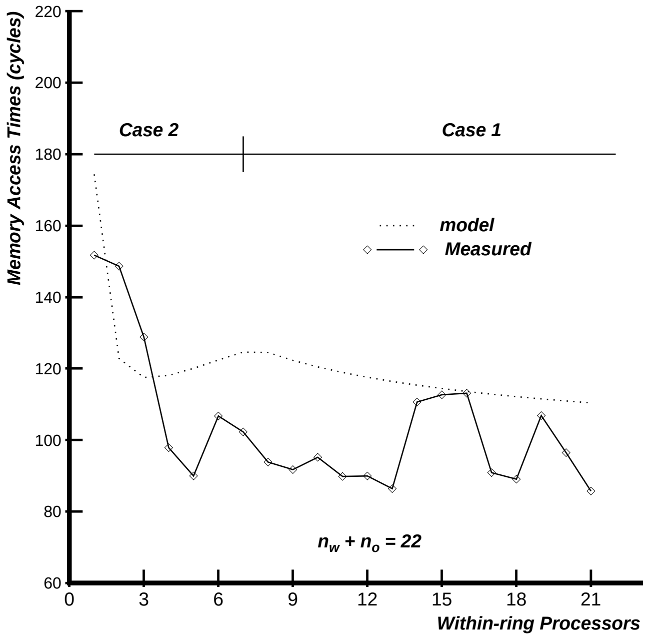


(a) Accessing data in the same ring.

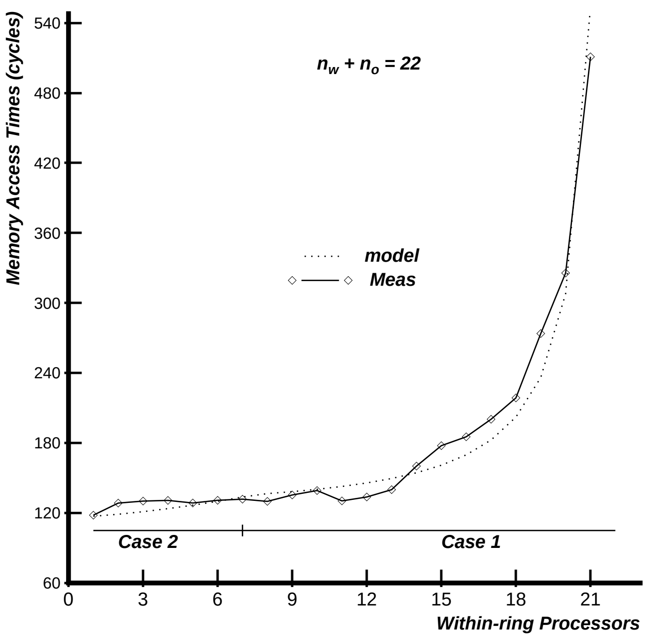


(b) Accessing data in a different ring.

Figure 11: Auto-update within a single ring.



(a) Average access time observed by within-ring readers.



(b) Average access time observed by off-ring readers.

Figure 12: Two Class Model for Auto-update.

8 Applications of the Model

We now use the model to estimate the effects of varying the parameters on memory access times. The first issue of interest is the multiprogramming environment, where parallel applications compete for the same resources, and the availability of processors changes dynamically with time. Future scalable multiprocessors will most likely be of the NUMA type, meaning that when scheduling such multiprocessors, we should consider not only the number of available processors but also their relative position within the memory distance hierarchy. Given that an application requires N processors, we address here the question of the sensitivity of performance to which N processors are allocated.

8.1 Impact of Workload on Memory Access Times

Figure 13(a) and Figure 13(b) depict the impact of varying n_w and n_o on memory access times. For both cases (i.e., $n_w + n_o = 40$ and $n_w + n_o = 50$), the analytical model assumes that there is enough buffering to accommodate outstanding requests and that an unlimited number of processors can be put on a single ring. In order to compare the curves within the same graph, the memory access latencies were normalized with respect to the memory access time when the number of within-ring processors is equal to $n_w + n_o - 1$ or the number of off-ring processors is equal to 1. It is important to consider Figure 13(a) and Figure 13(b) together. For instance, a horizontal line may intersect the off-ring curve in two points suggesting two equivalent solutions, yet, a further examination of the within-ring curve will identify the best one. Thus, determining a good processor allocation requires the definition of an objective function that allows for different memory access times. We shall explore two objective functions and their implications on placement decisions.

If we denote by Q_m and Q_r the average queue lengths of the memory and the ARD respectively, then under heavy loads (i.e., small g, t_w, and t_o, as these parameters tend to control the injection rate of requests), the Q_m and Q_r can be approximated by $n_w + n_o$ and n_o respectively. When we fix the total number of readers to $n_w + n_o$, the number of within-ring readers that minimizes C_w, n_w^*, satisfies:

$$Q_m * S_m = Q_r * S_r \tag{34}$$

$$\Longrightarrow \quad n_w^* = \frac{(n_w + n_o) * (S_r - S_m)}{S_r} \tag{35}$$

For light loads (i.e., large g, t_w, or t_o) $Q_m < n_w + n_o$ and $Q_r < n_o$, so

$$n_w^* < \frac{(n_w + n_o) * (S_r - S_m)}{S_r} \tag{36}$$

8.2 Impact of Processor Speed / Memory Speed Gap

It is believed that the gap between memory speed and cpu speed will continue to grow. The effects of this technological trend on future systems can be projected with our model. One way to capture this is to keep all the parameters fixed and to vary the value of t_o. Figure 14(a) and Figure 14(b) clearly illustrate the consequence of NUMAness on memory access times. Notice that the off-ring curve loses its local minimum with higher t_o values. As expected off-ring access times are more sensitivity to variation in t_o values than the within-ring access times.

8.3 Impact of ARD service time

Since the ARD was found to be a bottleneck under certain conditions, it is interesting to investigate the sensitivity of the performance due to variations of S_r. As we use more and more within-ring processors, the memory access times become less dependent on the ARD's performance which is illustrated by a convergence of the curves towards the left hand side of the axis, See Figure 15(a) and Figure 15(b). Figure 15(a) shows that higher values of S_r imply lower memory access times for within-ring readers, which is expected as S_r controls the flow of off-ring requests and consequently the extent of their interference with the rest of the readers. Figure 15(b) depicts the dynamics of n^* with various values of S_r.

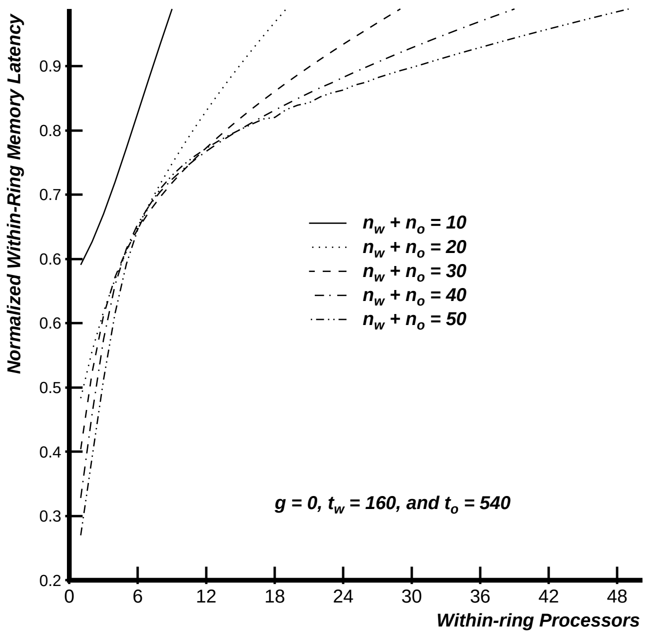
8.4 Impact of Processor Placement

The previous graphs depict a broad selection of processor placements. Choosing a favorable combination of processors depends on the objective function we seek to optimize. In this section, we consider two objective functions: The first one is simply $n_w * L_w(0, n_w, n_o) + n_o * L_o(0, n_w, n_o)$ which may apply when the work is evenly divided among the processors (with respect to time) thus reading shared data takes roughly the same time. The normalized function is a measure of the efficiency of the system, which we define as number of cycles used divided by number of cycles needed to read the same data. Here we assume that the application is coded taking into account the non-uniformity of memory access. Figure 16(a) shows that for KSR1's current parameters, the optimal combination that maximizes efficiency corresponds to maximizing the number of within-ring readers. Within the same context, minimizing response time however is equivalent to minimizing C_w.

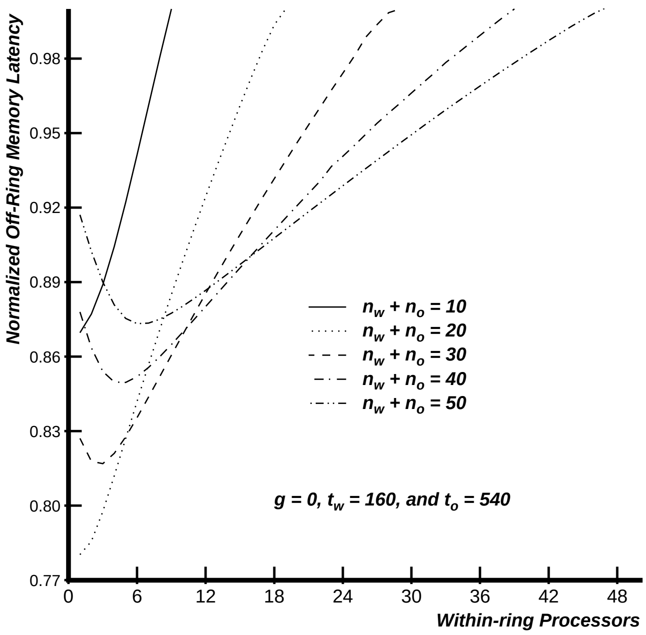
The second objective function assumes that the work is evenly divided among the processors with respect to the frequency of memory access. The natural way to scale applications without too much worrying about the underlying architecture leads to this even distribution. Figure 16(b) depicts cases of efficiency with local minima. Figure 17 shows the normalized response time. Notice that for light loads ($n_w + n_o < 30$), maximizing the response time corresponds to maximizing the processors of the off-ring type. Maximizing the number of readers of the within-ring type, however, results in maximum response time when ($n_w + n_o > 40$). The response times resulting from various processor placements differ by only 5% to 10%. This might be due to the fact that contention is a dominant component of memory access time, hence less responsive to NUMAness and thereby processor placements.

	Efficiency	Response Time
Function I	$\frac{n_w * L_w(0,n_w,n_o) + n_o * L_o(0,n_w,n_o)}{max \quad value}$	$L_w(0, n_w, n_o) = L_o(0, n_w, n_o)$
Function II	$\frac{n_w * T_w + n_o * T_o}{max \quad value}$	T_o

Table 4: Summary of Objective functions.

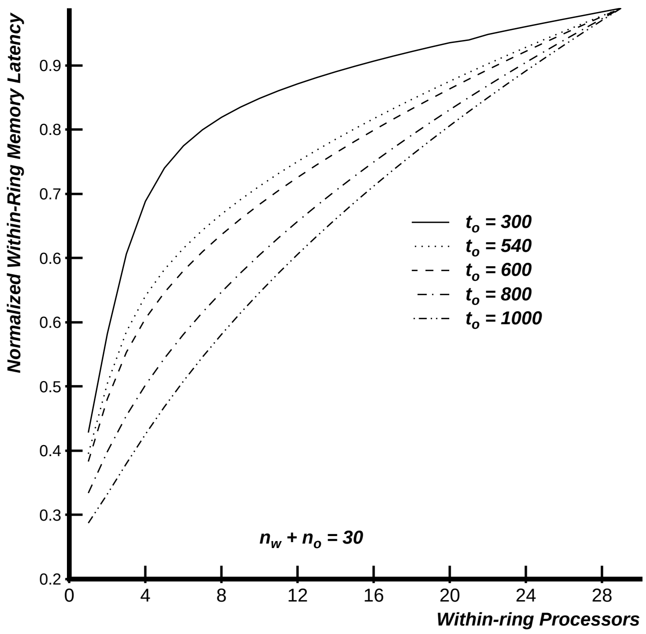


(a) Effects on within-ring readers.

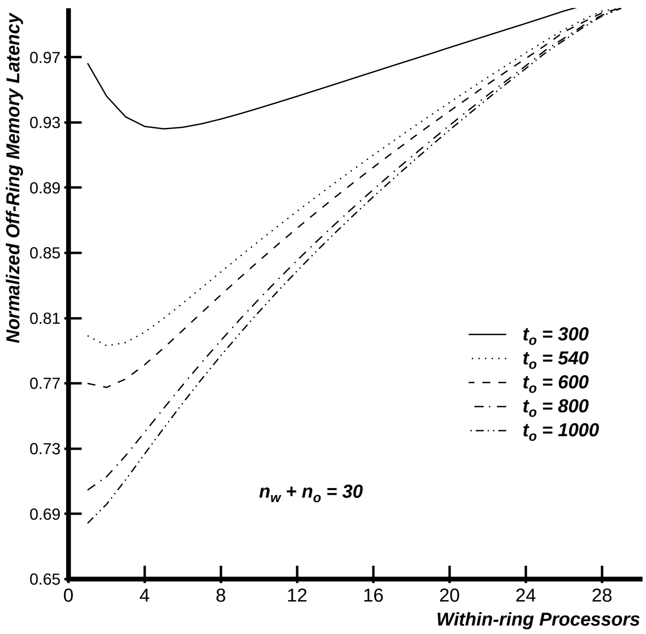


(b) Effects on off-ring readers.

Figure 13: Impact of workload on memory access times.

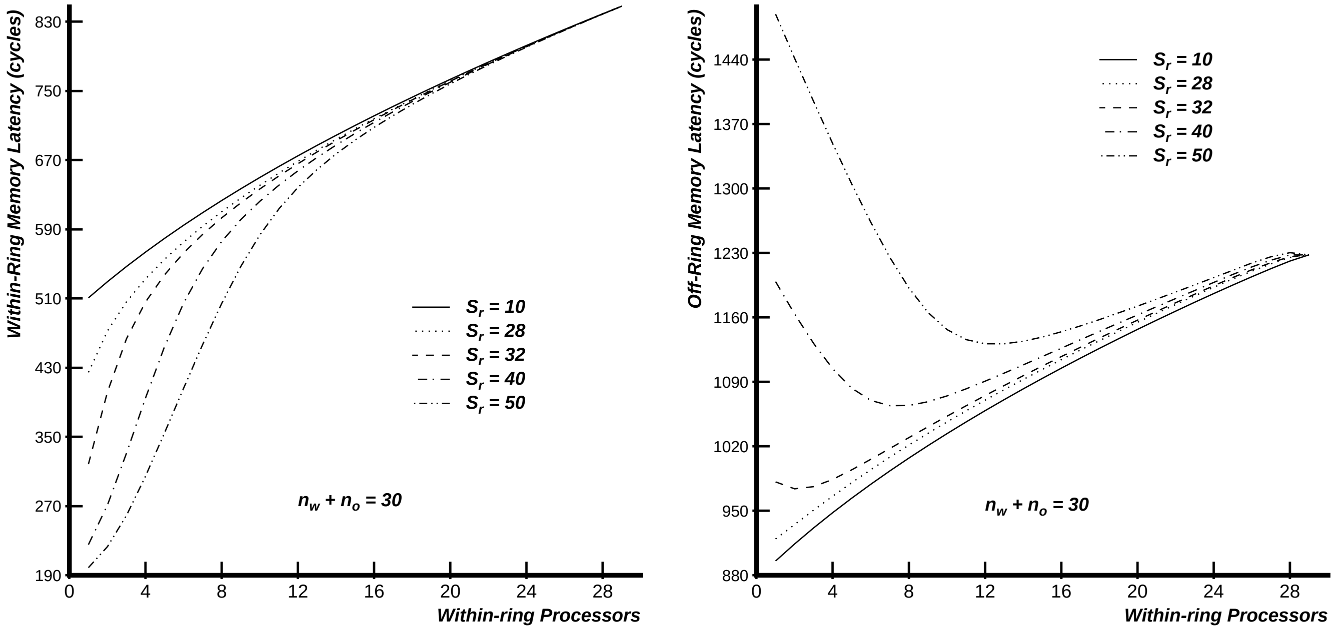


(a) Effects on within-ring readers.



(b) Effects on off-ring readers.

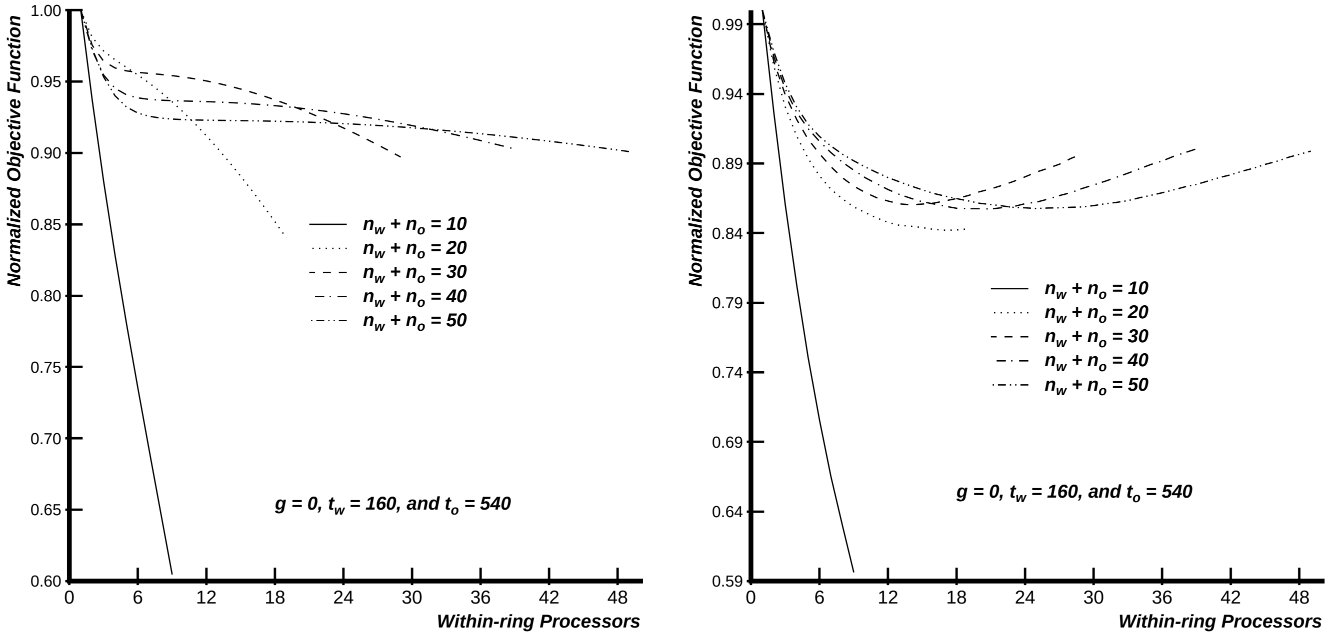
Figure 14: Impact of various values of t_o on memory access times.



(a) Effects on within-ring readers.

(b) Effects on off-ring readers.

Figure 15: Impact of various values of S_r on memory access times.



(a) Minimizing the efficiency of function I.

(b) Minimizing the efficiency of function II.

Figure 16: Importance of processor placement for various workloads.

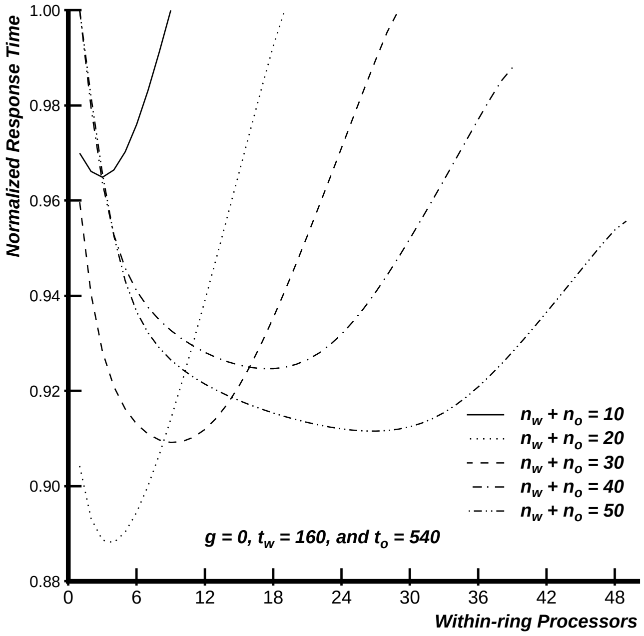


Figure 17: Implications of processor placement on response time (function II).

9 Summary and Conclusions

We have devised a modeling approach based upon the specification and the iterative solution of sets of equations which define the relationship and the mechanics of the architecture being modeled. The solution technique is found to be efficient (requiring only few iterations to reach convergence.) and accurate. We have demonstrated that simple and efficient models can be applied to complex real systems to study the performance of architectural alternatives. The utility of the model is demonstrated through a set of experiments on the KSR1 machine. Our work thus tends to foster the synergism between experimental and analytical methods for computer system evaluation.

Some of the equations we have derived may be of direct applicability in future modelling studies. The more important contribution of this work, however, is the analytical approach and methodology which involves the iterative solution of a set of non-linear equations that represent the system structure and operation. While little is known formally about the convergence properties of such models, empirically they have proven to be useful in a wide range of contexts.

We would like to extend and generalize our memory access model to describe a wider range of patterns. A future goal is to model the time-varying characteristics of an entire application and to reflect the effects of latency hiding techniques on the run time.

An interesting question to investigate is the issue of processor placement for an entire application rather than a single phase. Applications proceed from one phase to another, and for each phase there is an optimal distribution of within-ring and off-ring processors. In addition every phase has its own input parameters in a multiprogramming environment. In such environments, both the distribution of within-ring and off-ring processors and the total number of processors may change from phase to phase. An efficient scheduling strategy would be to find a good compromise between the choices at hand that minimizes the overall response time. This may involve assigning weights to different phases according to their contributions to the overall response time.

Acknowledgements: Our thanks to the Computer Science Department of the University of Washington for access to their KSR1 machine.

References

[BWAD93] E. L. Boyd, J. Wellman, S. G. Abraham, and E. S. Davidson. Evaluating the communication performance of MPPs using synthetic sparse matrix multiplication workloads. In *International Conference on Supercomputing*. ACM, July 1993.

[Dun92] T. Dunigan. Kendall Square multiprocessor: Early experiences and performance. Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, March 1992.

[ELZ86] D. Eager, E. Lazowska, and J. Zahorjan. A comparison of receiver-initiatied and sender-initiatied adaptive load sharing. *Performance Evaluation*, pages 53–68, February 1986.

[Fos93] R. Foster. Personal communication, 1993. Kendall Square Research.

[HS93] K. Harzallah and K. C. Sevcik. Evaluating memory system performance of a large scale NUMA multiprocessor. Technical Report 283, University of Toronto, CSRI, available for anonymous ftp from ftp.csri.toronto.edu, March 1993. *To appear in Proc MASCOTS'94.*

[Kah93] B. Kahhaleh. Analysis of memory latency factors and their impact on KSR1 MPP performance. Technical Report CSE–TR–157–93, University of Michigan, Department of Electrical Engineering and Computer Science, April 1993.

[LZGS84] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance.* Prentice-Hall, Englewood Cilffs, N.J, 1984.

[RSW+93] E. Rosti, E. Smiri, T. D. Wagner, A. W. Apon, and L. W. Dowdy. The KSR1: Experimentation and modeling of poststore. In *Proc. ACM Sigmetrics*, pages 74–85, May 1993.

[SJHG93] J. P. Singh, T. Joe, J. L. Hennessy, and A. Gupta. An empirical comparison of the kendall square research KSR-1 and stanford dash multiprocessors. Technical report, Stanford University, Computer Systems Laboratory, April 1993. *To appear in Supercomputing'93.*

[SVW+92] M. Stumm, Z. Vranesic, R. White, R. Unrau, and K. Farkas. Experiences with the hector multiprocessor. Technical Report 276, University of Toronto, CSRI, available for anonymous ftp from ftp.csri.toronto.edu, October 1992.

[WBH+93] D. Windheiser, E. L. Boyd, E. Hao, S. G. Abraham, and E. S. Davidson. KSR1 multiprocessor: Analysis of latency hiding techniques in a sparse solver. In *International Parallel Processing Symposium*, pages 454–461. ACM, April 1993.

[WSA+93] T. D. Wagner, E. Smirni, A. W. Apon, M. Madhukar, and L. W. Dowdy. Measuring the effects of thread placement on the Kendall Square KSR1. Technical Report ORNL/TM-12462, Oak Ridge National Laboratory, August 1993.