Latency Hiding on COMA Multiprocessors

Tarek S. Abdelrahman
Department of Electrical and Computer Engineering
The University of Toronto
Toronto, Ontario, Canada M5S 1A4

Abstract

Cache Only Memory Access (COMA) multiprocessors support scalable coherent shared memory with a uniform memory access programming model. The cache-based organization of memory results in long memory access latencies. Latency hiding mechanisms can reduce effective memory latency by making data present in a processor's local memory by the time the data is needed. In this paper, we study the effectiveness of latency hiding mechanisms on the KSR2 multiprocessor in improving the performance of three programs. The communication patterns of each program are analyzed and mechanisms for latency hiding are applied. Results from a 52-processor system indicate that the use of these mechanisms hides a significant portion of remote memory accesses and that application performance benefits. The overhead associated with the use of these mechanisms can limit the extent of this benefit.

1 Introduction

Cache Only Memory Access (COMA) multiprocessors scale to large numbers of processors yet support a single coherent view of memory [2, 3]. Scalability is achieved by employing a physically distributed memory structure and a scalable low-latency interconnection network that implements distributed-shared memory. The local portion of memory associated with a processor is organized as a large cache, referred to as the *local cache*. Data can reside in any local cache in the system, and there can be multiple valid copies of the data in shared memory. The hardware allows for the replication and migration of data in units of cache lines; data requested by a processor is automatically cached by the hardware into the local cache of the processor requesting the data. There is no association between the physical address of data and the location of data in shared memory. Instead, the physical address corresponds to any shared memory location (or locations) whose cache tag matches that of the physical address. Examples of COMA multiprocessors include the KSR1/2 multiprocessors [3] and the Data Diffusion Machine (DDM) [2].

The replication and migration of data allows locality of reference to be exploited with little or no effort on the part of application programmers. The caching of data on demand into the local cache of a processor brings the data closer to where it is used, and reduces access time of future references to the data. Additionally, the existence of multiple valid copies of data in shared memory reduces hot-spots that arise when multiple processors simultaneously access a single memory [1]; multiple requests to the same data can be satisfied from multiple local caches. The use of hardware to automatically replicate data is particularly advantageous in applications that have dynamic and irregular access patterns.

However, the cache-based organization of memory gives rise to higher remote memory access latency [6]. Since there is no association between the physical address of data and the location of data in shared memory, data must be located by systematically searching the cache directories of the local caches in the system for a valid copy of the data. In addition, the consistency of the multiple copies of data that can exist in the system must be maintained. This also requires a systematic search for all copies of the data in the cache directories. The increased remote memory access latency can severely limit application scalability.

In order to reduce the impact of the higher remote memory access latency, COMA architectures employ mechanisms to overlap memory access with computations. These mechanisms allow application programmers and/or compilers to optimize programs so that data not in the local cache of a processor is made local by the time the data is needed, hence, effectively hiding the impact of remote memory access latency. However, these mechanisms have overhead associated with their use, and may degrade performance due to increased network and memory contention. In addition, these mechanisms have limited performance benefit since they do not move data into a processor's private cache; rather only into the local cache, which is accessed at memory (DRAM) speed. It is unclear the extent to which these mechanisms can be effective in improving program performance.

In this paper, we investigate possible performance gains that can be attained by hiding remote memory access latency on the KSR2 COMA multiprocessor. The focus of our investigation is three applications: a program for solving dense linear optimization problems using the simplex method; an LU factorization program; and a 2-dimensional FFT program. The programs are parallelized on a 52-processor KSR2 system, and latency hiding mechanisms are applied. Experimental results indicate that the performance of the program benefits from latency hiding, particularly when the number of processors is large. However, care must be used when applying these mechanisms— the overhead associated with their use may outweigh their benefit and degrade performance.

The remainder of this paper is organized as follows. In section 2 the architecture of the KSR2 multiprocessor is briefly described. In section 3 the three applications and their parallelization on the KSR2 are described. In section 4 the use of the KSR2 latency hiding mechanisms in the parallel programs is discussed. In section 5 our experimental results are presented and analyzed. In section 6 previous work relevant to our study is reviewed. Finally, in section 7 concluding remarks are given.

2 The KSR2 Multiprocessor

The KSR2 consists of 32 to 1088 processing cells interconnected by a two-level ring hierarchy, as shown in Figure 1. Thirty two processing cells are connected to a first-level ring, which is referred to as ring:0. Up to 34 ring:0 rings can be connected to a second-level ring, which is referred to as ring:1. The rings are unidirectional and support data transfer rates of up to 1 Gbyte/s on ring:0 and 2 Gbyte/s on ring:1. A processing cell consists of a 64-bit superscalar processor, a 512-Kbyte first level cache referred to as the *subcache*, and a 32-Mbyte second level cache, which serves as the cell's main memory, and is referred to as the *local cache*. The subcache is divided into a 256-Kbyte subcache for instructions and a 256-Kbyte subcache for data. The local cache is 16-way set associative and its 128-byte cache line is referred to as a *subpage*.

Data is allocated in the local cache in units of pages of 16 Kbytes. However, data moves from one local cache to another in units of subpages. A local cache directory (LCD) is maintained by each cell and contains an entry for each of the 2048 pages in the local cache.

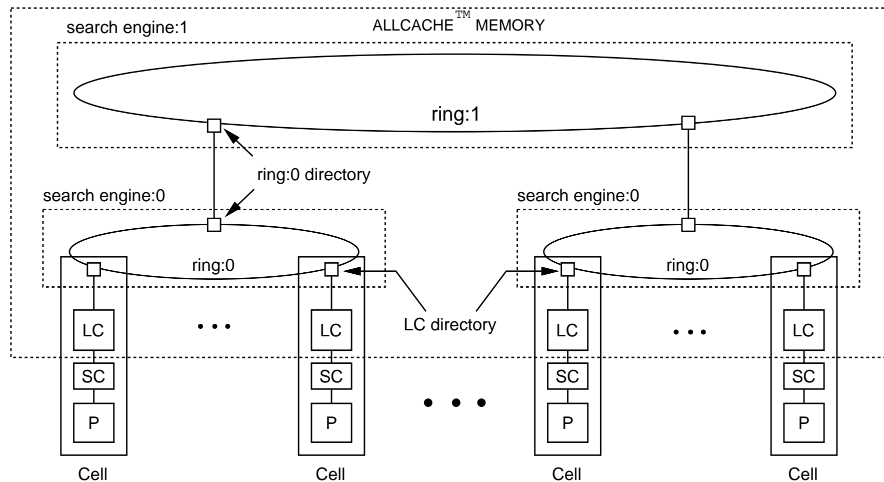

Figure 1: The KSR2 multiprocessor.

Each entry specifies the state of each of the 128 subpages in its corresponding page. There is also one ring:0 cache directory for each ring:0 which contains an entry for each of the 65536 pages in the local caches of the 32 cells attached to the ring:0. A duplicate ring:0 directory is attached to ring:1 for each of the ring:0 directories, as shown in Figure 1.

A reference that cannot be satisfied from the local cache of a cell causes the cell to form a request message and place it on the ring:0 to which the cell is connected. As the request message travels on the ring, each cell connected to the ring determines if the request can be satisfied from its local cache. If it can, the request message is removed from the ring, and a response message with the subpage containing the requested data is inserted in its place. The response message travels on the ring until it reaches the requesting cell, at which point it is removed from the ring, and its contents are placed in the local cache (and subcache) of the cell to complete the memory transaction.

The consistency of the data in the local caches is maintained using a write-invalidate protocol [3]. A subpage can be in one of four states: *exclusive* which indicates that the subpage is owned by the cell and that it is the only valid copy; *copy* which indicates that the subpage is a read-only copy; *invalid* which indicates that the subpage is an invalid copy; and *non-exclusive*, which indicates that the subpage is owned by the cell, but other valid copies exist in memory. A subpage must be in the exclusive state in a cell before the processor of that cell can write to it. The state of each subpage is changed in the LCD when a request for the subpage is received by the cell. The cell also monitors (or snoops on) the traffic on the ring and updates the state of relevant subpages it has. The memory access latencies for the various levels of the memory hierarchy of the KSR2 are shown in Table 1.

The KSR2 system has three mechanisms to reduce remote memory access latency. *Automatic update* is a feature of the snooping cache consistency protocol. When a cell issues a read request for data, and that request is placed on the ring interconnect, the first cell that has the data in a valid state responds to that request. As the response message travels on the ring back to the requester, all cells on the ring between the responding cell and the requesting cell that have the data in an invalid state and that are not ‘busy’ automatically update their copies. The *prefetch* instruction requests the cell to search for a valid copy of a subpage and move that subpage into the local cache of the requesting cell. Up to 4 prefetch instructions can be pending at any time. The hardware provides for *blocking* and *non-blocking* prefetch instructions. Blocking prefetch instructions stall the processor when more than 4 prefetchs are pending. The non-blocking prefetch does not stall the processor, but is ignored when more than 4 prefetches are pending. The prefetch instruction stalls the cell that issues the prefetch for two cycles and prevents accesses to the local cache of the

Table 1: Local and remote access latencies on the KSR2 (in processor cycles).

Access Type	Relevant Cace	Location of Data			
		Subcache	Local Cache	ring:0 Cache	ring:1 Cache
Read	Closest	2	20–24	175	600
Write	Furthest	2	20–24	175	600

cell until the prefetch request is placed on the ring [3]. The *poststore* instruction broadcasts a copy of a subpage on the ring interconnect. Cells that have a copy of the subpage in an invalid state, and that are not "busy" update their invalid copies. The poststore instruction also stalls the cell that issues the instruction and prevents accesses to the local cache of the cell. Automatic updates occurs for both prefetch and poststore instructions. This paper focuses mainly on the prefetch and poststore instructions.

3 The Applications

3.1 Simplex

This application uses the simplex method [5] to solve linear programming problems in the form:

$$\begin{array}{ll} \text{minimize} & z = cx \\ \text{Subject to} & Ax = b \\ \text{where} & b \geq 0 \quad \text{and} \quad x \geq 0 \end{array}$$

The problem is represented in a matrix form known as the simplex *tableau*, T, which contains the matrix A, the vectors b and c, and the value of the cost function z as shown below.

$$T = \begin{pmatrix} A & b \\ c & z \end{pmatrix}.$$

The application iterates over the tableau in three main steps, as shown in Figure 2. In step 1, a pivot column (p) is located. In the second step, a pivot row (r) is determined. The tableau element at the pivot row and the pivot column is referred to as the *pivot*. In the third step, Gauss-Jordan elimination is performed and the tableau is updated.

The main source of parallelism in Simplex is in the Gauss-Jordan elimination step. Given P processors, each processor becomes responsible for performing this step on a set of $\frac{m+1}{P}$ contiguous rows of the tableau. Processor P is designated as a *master*. It is also responsible for performing steps 1 and 2 of each iteration. In order to allow the processors to perform the Gauss-Jordan elimination step in parallel, the master performs the additional step of copying the pivot row into a pivot row buffer. The processors use this buffer to perform the elimination step instead of the pivot row itself, which allows the processor that owns the pivot row to update the pivot row without waiting for the rest of the processors to finish using it. Two points of barrier synchronization per iteration are needed. The first is needed after step 2 to ensure that the pivot column and row have been located and that the pivot row has been copied before step 3 is performed. The second barrier is needed after step 3 to ensure that all processors complete step 3 before a new iteration is started.

The time taken by the master to perform steps 1 and 2 and to copy the pivot row into the pivot row buffer is referred to as the *serial* time of the parallel application. It represents the Amdahl sequential fraction of the application and limits its scalability when the number of processors is large.

3.2 LU Factorization

This application decomposes a matrix A into upper and lower triangular matrices, U and L, using Gauss-Jordan elimination without pivoting. The algorithm consists of three loops as

1. **Locate pivot column,** p:
   $p = \text{index}\left(c_{min} = \min_j \left\{t_{(m+1)j} \mid t_{(m+1)j} \geq 0\right\}\right)$.
   if $t_{(m+1)j} < 0 \;\; \forall\, j$ then optimal solution found.

2. **Locate pivot row,** r:
   $r = \text{index}\left(b_{min} = \min_i \left\{\frac{t_{i(n+1)}}{t_{ip}} \mid t_{ip} > 0\right\}\right)$.
   if $t_{ip} \leq 0 \;\; \forall\, i$ then no optimal solution exists.

3. **Gauss-Jordan elimination**:

```
for i=1,m+1
  for j=1,n+1
    if i ≠ r then t_ij = t_ij - (t_ip / t_rp) t_rj
    else t_rj = t_rj / t_rp
```

Figure 2: Main steps of a Simplex iteration.

```
for k=1,n-1
  for i=k+1,n
  m_i = -a_i,k / a_k,k
    for j=k+1,n
      a_i,j = a_i,j + m_i * a_k,j
```

Figure 3: Main steps of LU.

```
for i=1,n
  perform 1-dimensional FFT
  on row i
a_i,j = a_j,i
for i=1,n
  perform 1-dimensional FFT
  on row i
```

Figure 4: Main steps of FFT.

shown in Figure 3. It is parallelized by performing the Guass-Jordan elimination in parallel. The rows of A are partitioned among the processors cyclicly to balance the load, and each processor becomes responsible for performing the Gauss-Jordan elimination on its subset of rows. A barrier synchronization is needed after each iteration of the k loop to ensure the Gauss-Jordan elimination is complete before the new set of multipliers m_i are computed using row k, which is referred to as the *pivot row*.

3.3 FFT

This application computes the 2-dimensional Fast Fourier Transform (FFT) of an $n \times n$ matrix A. The main steps are shown in Figure 4. A 1-dimensional FFT is applied to each row of A, the matrix is transposed, and the 1-dimensional FFTs are applied again.

The main source of parallelism in FFT is the application of the 1-dimensional FFT. Each processor is responsible for performing the 1-dimensional FFT on a subset of $\frac{n}{P}$ rows. Each processor also participates in performing the transpose. A temporary array is used for this purpose. Two barrier synchronizations are needed. The first is after the first application of the 1-dimensional FFT, to ensure that all FFT computations are performed before the transpose takes place. The second is after the transpose, to ensure that it is complete before the second application of the 1-dimensional FFT.

4 Latency Hiding

4.1 Simplex

The number of remote memory accesses incurred by a processor in each step of Simplex is summarized in Table 2. S is the number of tableau elements per subpage[1].

There are no remote memory accesses in step 1 of the application. The master processor locates the pivot column by finding the minimum value in the last row of the simplex tableau. Since the master processor is responsible for updating this row of the tableau, the data is already in exclusive state, and no remote memory accesses are necessary.

There are three sources of remote memory accesses in step 2. Remote memory accesses arise when the master processor reads both the b vector and the pivot column to determine the pivot row index. Since both vectors have been updated by the processors during the Gauss-Jordan elimination step of the preceding iteration, a portion of the local copies of

[1] In our implementation, each tableau element is 64-bits long, and hence $S = 16$.

Table 2: Subpage misses in Simplex.

Step	Invalid-subpage misses	Non-exclusive subpage misses
1	0	0
2	$2 \times \left(\lfloor \frac{m+1}{P} \rfloor + 1 \right) + \lceil \frac{n+1}{P} \rceil$	$\lceil \frac{n+1}{S} \rceil$
3	$\lceil \frac{n+1}{S} \rceil$	$\lfloor \frac{m+1}{P} \rfloor + 1$

these vectors is invalid. The master must read tableau elements from $P-1$ non-local caches. Remote memory accesses also arise when the master copies the pivot row from the tableau into the pivot row buffer. If the pivot row is not in the local cache of the master, it must be remotely read. Finally, remote memory accesses arise because exclusive ownership of the subpages of the pivot row buffer is required to write the elements of the buffer. Since these elements have been read by all other processors during the Gauss-Jordan elimination step of the preceding iteration, these subpages exist in the non-exclusive state, and the master must stall until exclusive ownership is acquired.

The prefetch instruction can be used to hide the remote access latency incurred by the master processor in three ways. First, the subpages containing $t_{(i+1)p}$ and $t_{(i+1)(n+1)}$ can be prefetched into the local cache of the master processor as the master computes the ratio $t_{i(n+1)}/t_{ip}$. This overlaps the latency incurred by the master to access the next values of b and the pivot column with the computation. Second, the subpage containing the $(j+1)^{st}$ element of the pivot row can be prefetched as the master copies the jth element from the pivot row into the pivot row buffer. The overlaps the latency incurred by the master to access the element of the pivot row needed next with the copying of an element. Finally, the subpage containing $(j+1)^{st}$ element of the pivot row buffer can prefetched in the exclusive state into the local cache of the master as the master writes the jth element of the pivot row buffer. This overlaps the latency incurred by the master to exclusively acquire the next element of the pivot row buffer with the copying of an element.

There are two sources of remote memory accesses in step 3. Each processor needs a copy of the pivot row buffer to perform the Guass-Jordan elimination step. This buffer has been updated in step 2 by the master processor, and, hence, is invalid in the local cache of every other processor. The processors must read the valid copy from the local cache of the master. Also, each processor needs exclusive ownership of its portion of the elements of the b vector. These values have been read by the master in step 2, and exist in a non-exclusive state. Hence, each processor must acquire the elements of its portion of b exclusively by invalidating copies of these elements in the local cache of the master.

The poststore instruction can be used to hide remote memory access latency incurred in step 3. The master processor poststores the subpages containing the elements of the pivot row buffer as the buffer is being updated. Since all the processors have an invalid copy of the buffer in their local caches from the preceding iteration, each processor upgrades its invalid copy into a valid one. The processors find and use this valid copy when they exit the first barrier to perform the Gauss-Jordan elimination. This eliminates remote access latency penalties incurred by the processors in accessing the elements of the pivot row buffer. This also has the advantage of alleviating the possible hot-spot that can result when all processors attempt to access the local cache of the master processor to obtain a copy of the buffer.

The prefetch instruction can also be used to hide remote memory access latency in step 3 by prefetching elements of the b vector in the exclusive state before they are needed for the Gauss-Jordan elimination.

4.2 LU Factorization

Remote memory accesses occur in LU during the first access to the pivot row in a k iteration. Since the matrix has been updated in the previous k iteration, the subpages of the pivot row exist in the exclusive state in only one processor. All other processors must remotely access

the subpages. These accesses are to all the same cache, and cause contention, exasperating the latency of remote accesses. The average number of subpage misses per processor is approximately given by

$$\frac{1}{n-1}\sum_{k=1}^{n-1}\left\lfloor\frac{n-k}{S}\right\rfloor,$$

where n is the size of the matrix A, and S is the number of matrix elements per subpage.

The poststore instruction can be used to hide remote memory access latency as follows. In a given k iteration of LU, the processor that computes the $(k+1)$st row poststores the subpages of the updated row. Since this row will become the pivot row in the next k iteration, each processor finds valid copies of the subpages of the pivot row in the $(k+1)$st iteration.

4.3 FFT

Remote memory accesses occur in FFT during the matrix transpose step of the application. Each processor must read an $\frac{n}{P} \times \frac{n}{P}$ sub-block of the matrix A from each of the other $P-1$ processors. Since these sub-blocks are written to by their respective processors during the first 1-dimensional FFT, they are in the exclusive state in these processors and are invalid in the local cache of the processor. Each processor performing the transpose step must access the valid copies remotely. The total number of subpage misses per processor is approximately given by

$$(P-1)\times\frac{n}{P}\times\left\lfloor\frac{n}{S\times P}\right\rfloor,$$

where n is the size of the matrix A, and S is the number of matrix elements per subpage.

Both the poststore and prefetch instructions may be used to hide the latency of remote accesses. The poststore instruction can be used in the first step; each processor poststores the subpages of data to be used by other processors in the third step of the application. The prefetch instruction can be used in the transpose step itself. A processor prefetches the element of A it needs next as it copies the current element. The use of the prefetch instruction is more efficient, but adds overhead in the transpose step. The use of the poststore instruction is less efficient because it broadcasts data even though only one other processor needs it. It has the advantage of not adding overhead in the transpose step.

5 Experimental Results

The three applications were parallelized on a 52-processor KSR2 system using the C *pthreads* library; 26 threads on processors on one ring:0 ring and 26 threads on processors on a second ring:0 ring. Execution times and the number of subpage misses were collected using the *pmon* facility, which allows applications to read a set of hardware event counters. The counters accumulate information, such as the number of subcache and local cache misses for each thread. Simplex was used to solve a 714 $\times$ 1938 problem. LU was used to solve a 1000 $\times$ 1000 problem, and FFT was used to solve a 512 $\times$ 512 problem.

5.1 Simplex

The impact of the prefetch instruction is shown in Figures 5 and 6. Figure 5 shows the number of subpage misses occurring in step 2 with and without prefetch. It indicates a significant reduction in the number of subpages misses with the use of prefetch. Figure 6 shows a resulting 30% to 40% reduction in the serial time. There is a sharp increase in the serial time when the number of processors exceeds 24. In this case, the processors span the two rings of the system, requests traverse the ring:1 ring, and the penalty of remote memory access increases sharply. The prefetch instruction is successful in reducing the impact of this sharp increase.

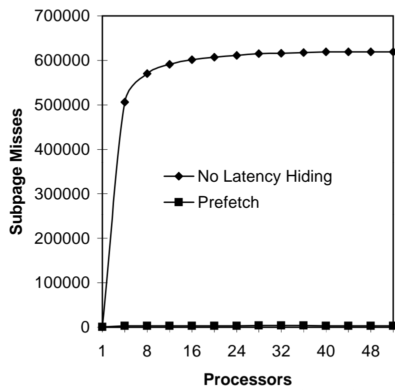

Figure 5: Subpage misses in Simplex.

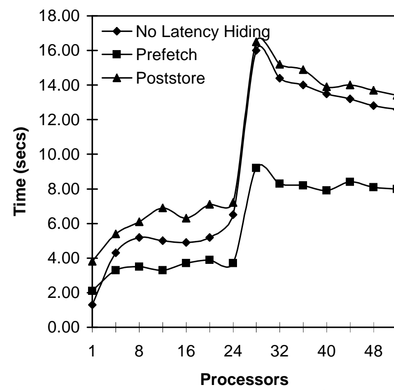

Figure 6: Serial time in Simplex.

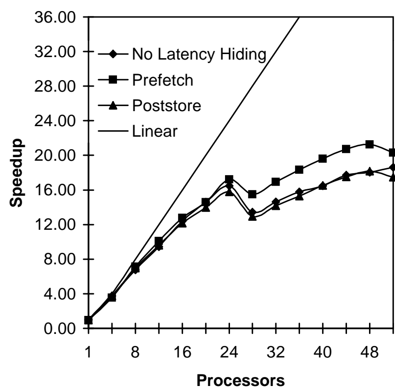

Figure 7: Speedup of Simplex.

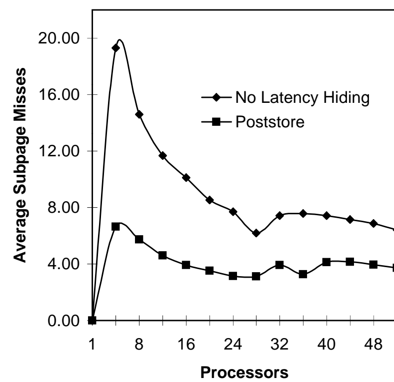

Figure 8: Subpage misses in LU.

The use of the poststore hides remote memory access latency in step 3, but the instruction must be used in step 2. The impact of the poststore instruction on the serial time is shown in Figure 6. There is an increase in the serial time, which is due to the overhead of the poststore instruction. The results reported by the pmon facility indicate a negligible reduction in the number of subpage misses and in the execution time of step 3 due to the poststore. This is due to the large amount of computation in step 3, which makes the impact of remote memory access latency small. Hence there is only a marginal improvement in the execution time of step 3 due to the poststore.

The impact of latency hiding on overall performance is depicted in Figure 7. The use of the prefetch instruction improves the speedup of the application by about 20% when the number of processors is large. This limited benefit is mainly because prefetch improves the performance of the second step of the application, which constitutes a relatively small fraction of overall execution time. The use of the poststore instruction slightly *degrades* overall performance because of the overhead added by the instruction with almost no gain from its use.

5.2 LU Factorization

The impact of the poststore instruction on the average number of subpage misses per processor is shown in Figure 8. It reflects a decrease in the number of subpage misses with the use of the poststore instruction.

It is interesting to note the decline of the average number of subpage misses as the number of processors increases, even though each processor references the same number of non-local subpages independent of the number of processors. This is attributed to the effect of automatic update. When the number of processors is small, processors leave the barrier at roughly the same time, and each misses on almost all the pivot row subpages. However, when the number of processors is large, processors leave the barrier at different times, causing some to access the pivot row subpages before others. Processors that are late

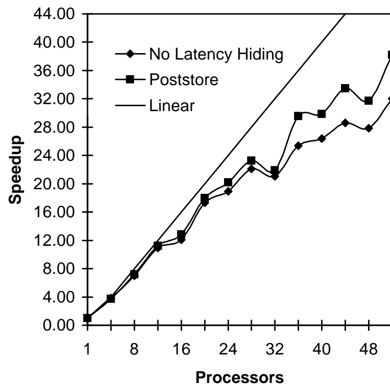

Figure 9: Speedup of LU.

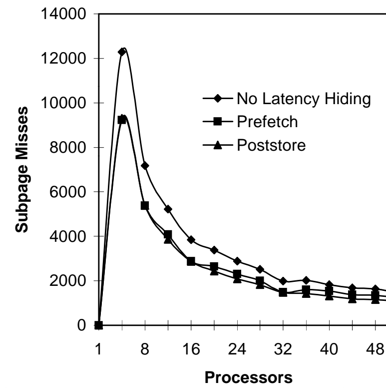

Figure 10: Subpage misses in FFT.

in leaving the barrier have their pivot row subpages automatically updated, and suffer no misses. This causes the average number of subpage misses to drop.

The impact of the poststore instruction on the overall performance of LU is shown in Figure 9. Improvements in the range of 15% to 20% can be achieved when the number of processors is large. There is little improvement when the number of processors is small due to the large amount of computation performed by each processor; the impact of remote memory access latency is small. When the number of processors becomes large, the amount of computation per processor becomes smaller, and the impact of the remote accesses to the pivot row become more pronounced.

5.3 FFT

The average number of subpage misses incurred in the transpose step of the application is shown in Figure 10, with and without the use of latency hiding. A similar reduction in the number of misses is achieved with either the prefetch or the poststore instructions. The impact of this reduction on overall performance is minimal, as indicated by the speedup of the application, shown in Figure 11. This is attributed to the only small reduction in the number of subpage misses, and due to the small impact of the transpose step on performance; less than 10% of the total time is spent in that step.

6 Relevant Work

Windheiser et al. [8] conducted a similar study using an iterative sparse linear equation solver. They concluded that while performance improvements can be obtained through automatic update, the prefetch and poststore instructions have little impact. Their experiments were limited to one application and a single ring KSR1. In contrast, our experiments use a two ring KSR2 system, in which the prefetch and poststore instructions have lower overhead. Our experiments show that both prefetch and poststore have a significant impact on performance.

Wagner et al. [7] studied the impact of thread placement on performance using a set of synthetic benchmarks. They conducted experiments on a two-ring 64-processor system. Their results indicated performance benefits from automatic update, but suggested no systematic way of exploiting it in real applications.

Rosti et al. [4] analytically modeled the poststore instruction to determine conditions underwhich the instruction would be useful. They concluded that the poststore instruction is only effective when the system load is light and when there are a large number of readers. Their experiments, which used synthetic workloads, agreed with their analytical models.

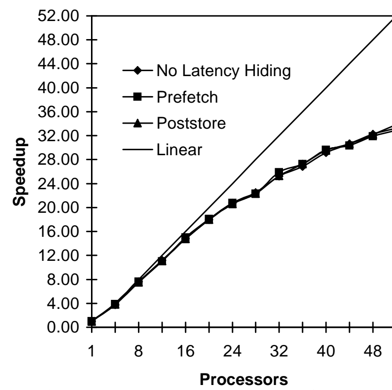

Figure 11: Speedup of FFT.

7 Summary and Conclusions

COMA multiprocessors provide coherent shared memory that can scale to large numbers of processors. However, the cache-based organization can lead to high remote memory access latencies in large systems, which can severely limit scalability. Latency hiding mechanisms can reduce effective memory access time, hence, improving scalability. In this paper we conducted an experimental study of mapping three applications on a KSR2 COMA multiprocessor. The objective of the study was to determine the effectiveness of the prefetch and poststore latency hiding instructions in improving program performance. The results of the study indicate that the use of these instructions hides a significant portion of remote memory accesses, and benefits application performance. The instructions must be used with care, weighing their expected benefit against their overhead. The use of the instruction in critical parts of the program, such as the sequential fraction may degrade overall performance instead of benefiting it.

References

[1] R. Bianchini, M. Crovella, L. Kontoothanassis and T. LeBlanc, Memory contention in scalable cache-coherent multiprocessors. Technical Report # 448, Department of Computer Science, The University of Rochester, 1993.

[2] E. Hagersten, A. Landin and S. Haridi, "DDM– A cache-only memory architecture," *IEEE Computer,* vol. 25, no. 9, pp. 44–54, 1992.

[3] Kendall Square Research, KSR1 principles of operation manual, Kendall Square Research Corporation, Boston, MA, 1992.

[4] E. Rosti, E. Smirni, T. Wagner, A. Apon and L. Dowdy, "The KSR1: experimentation and modeling of poststore," *Proceedings of ACM SIGMETRICS,* pp. 74–85, 1993.

[5] D. Solow, *Linear programming: An introduction to finite improvement algorithms,* North-Holland, New York, 1984.

[6] P. Stenström, T. Joe and A. Gupta, "Comparative performance evaluation of cache-coherent NUMA and COMA architectures," *Proceedings of the International Symposium on Computer Architecture,* pp. 80–91, 1992.

[7] T. Wagner, E. Smirni, A. Apon, M. Madhukar and L. Dowdy, Measuring the effects of thread placement on the Kendall Square KSR1, Technical Report # ORNL/TM-12462. Oak Ridge National Laboratory, Mathematical Sciences Section, 1993.

[8] D. Windheiser, E. Boyed, E. Hao, S. Abraham and E. Davidson, "KSR1 multiprocessor: analysis of latency hiding techniques in a sparse solver," *Proceedings of the Int'l Parallel Processing Symposium,* pp. 454–461, 1993.