, , 1–19 ()

Latency Hiding on COMA Multiprocessors

TAREK S. ABDELRAHMAN tsa@eecg.toronto.edu
Department of Electrical and Computer Engineering
The University of Toronto
Toronto, Ontario, Canada M5S 3G4

Received May 1, 1991

Editor:

Abstract. Cache-only memory access (COMA) multiprocessors support scalable coherent shared memory with a uniform memory access programming model. The local portion of shared memory associated with a processor is organized as a cache. This cache-based organization of memory results in long remote memory access latencies. Latency-hiding mechanisms can reduce effective remote memory access latency by making data present in a processor's local memory by the time the data are 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 the mechanisms for latency hiding are applied. Results from a 52-processor system indicate that these mechanisms hide a significant portion of the latency of remote memory accesses. The results also quantify benefits in overall application performance.

Keywords: Cache-only memory access (COMA) multiprocessors, latency hiding, prefetching, broadcasting, memory system performance, performance evaluation.

1. Introduction

Cache-only memory access (COMA) multiprocessors scale to large numbers of processors yet support a single coherent view of memory [3, 4]. Scalability is achieved by employing a physically distributed memory structure and a scalable low-latency interconnection network to support 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 the shared memory 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; that is, there is no designated "home" location for data. 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 [4] and the Data Diffusion Machine (DDM) [3].

The replication and migration of data can reduce the extent to which programmers must manage data in order to exploit locality of reference in applications. The caching of data on demand into the local cache of a processor brings the data closer

to where they are used and reduces the 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 [7]. Since there is no association between the physical address of data and the location of data in shared memory, data not in the local cache of a processor (i.e., remote data) must be located by systematically searching the cache directories of other 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, typically using a write-invalidate protocol [7]. This also requires a systematic search for all valid 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 can be used by application programmers and/or compilers to have data that are not in the local cache of a processor be local by the time the data are needed. This effectively overlaps remote memory access latency with computations, hence reducing the impact of remote accesses. 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 benefits. Unlike traditional latency-hiding mechanisms [2] that make data present in the processor's private cache, COMA latency-hiding mechanisms only move data into a processor's local cache, which is accessed at memory (i.e., 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. In the focus of our investigation are three applications: a program for solving dense linear optimization problems using the Simplex method; an LU factorization program; and a two-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

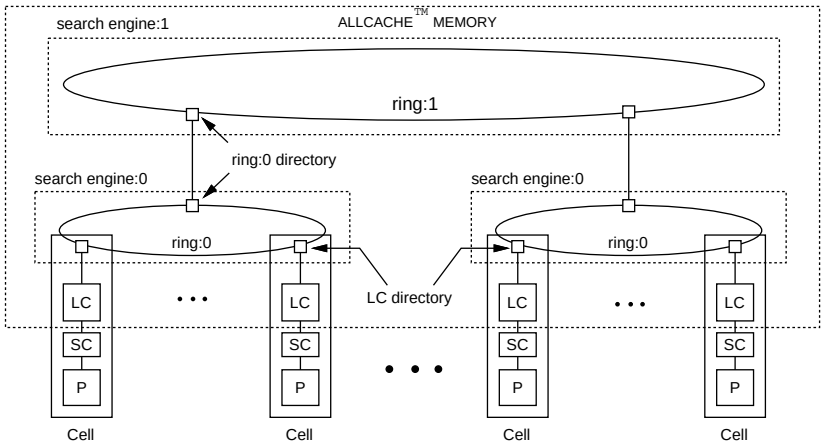

Figure 1. The KSR2 multiprocessor.

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 Gbytes/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 are allocated in the local cache in units of pages of 16 Kbytes. However, data move 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. Each entry specifies the state of each of the 128 subpages in the corresponding page. There is also one ring:0 cache directory for each ring:0 that contains an entry for each of the 65536 pages in the local caches of the 32 cells attached to 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 a remote memory access, or a *subpage miss*. The cell forms a request message and places it on the ring:0 to which it 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.

When a request cannot be satisfied from any of the local caches on the ring:0 to which the cell is connected, it is pulled up to ring:1. As the request message travels on ring:1, each duplicate directory on ring:1 determines if the request can be satisfied from any of the local caches on the ring:0 to which the directory is attached. If it can, the request message is pushed down to that ring:0 and allowed to travel until it is satisfied. The response message is then sent back to the requester.

The consistency of data in the local caches is maintained using a write-invalidate protocol [4]. 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. Each cell also monitors (or snoops on) the traffic on the ring and updates in its LCD the state of subpages it has copies of.

The hierarchal memory structure of the KSR2 leads to non-uniform memory access latencies. The time to access data in the local cache of a cell is shorter than the time to access data in a remote local cache. Additionally, the time to access data nonlocal to a cell depends on the distance between the cell and the (relevant) local cache that contains a valid copy of the data. The memory access latencies for the various levels of the memory hierarchy of the KSR2 are shown in Table 1. The relevant cache is the local cache that determines how far a request must travel before it can be satisfied. Hence, for a read operation the closest local cache responds with a copy of the data. For a write (or exclusive read) operation, all valid copies of the data must be invalidated, and the furthest local cache that contains a copy of the data determines how far the request must travel.

The KSR2 system has three mechanisms to reduce effective remote memory access latency: *automatic update*, *prefetch* and *poststore*. 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 the 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 subpages with copies of the data in an invalid state and that are not "busy" automatically update their subpages.

The *prefetch* instruction requests the cell to asynchronously search for a valid copy of a subpage and to copy that subpage into the local cache of the requesting cell. Once the request is completed, the processor continues with computation without having to wait for the response to the prefetch. Only up to four prefetch

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

			Location of Data		
Access Type	Relevant Cache	Subcache	Local Cache	Ring:0 Cache	Ring:1 Cache
Read	Closest	2	20–24	175–200	600
Write	Furthest	2	20–24	175–200	600

instructions can be pending at any time. The hardware provides for *blocking* and *nonblocking* prefetch instructions. Blocking prefetch instructions stall the processor when more than four prefetches are pending. The nonblocking prefetch does not stall the processor, but is ignored when more than four prefetches are pending. Data may be prefetched in either an exclusive or a non-exclusive state.

The *poststore* instruction asynchronously broadcasts a copy of a subpage on the ring interconnect. Once the broadcast is issued, the processor continues with computation. As the broadcast subpage travels on the ring interconnect, cells that have copies of the subpage in an invalid state and that are not "busy" update their invalid copies. Automatic updates occur for both prefetch and poststore instructions. This paper focuses mainly on the prefetch and poststore instructions.

The use of the prefetch and poststore instructions carries overhead. The prefetch instruction stalls the cell that issues the prefetch for 2 cycles and prevents accesses to the local cache of the cell until the prefetch request is placed on the ring [4]. Similarly, the poststore instruction stalls the cell that issues the instruction for up to 12 cycles and prevents access to the local cache of the cell for up to 28 cycles [4]. The use of the instructions adds to the traffic on the ring interconnect and hence may add to the access latency of other requests. Finally, the use of the prefetch and poststore may require additional instructions to compute the address of the prefetched or poststored subpage, which increases compute time.

3. The Applications

In this section the three applications used in the study and their parallelization methods on the KSR2 are described.

3.1. Simplex

This application uses the Simplex method [6] to solve linear programming problems in the form:

minimize	$z = cx$		
subject to	$Ax = b$		
where	$b \geq 0$	and	$x \geq 0$

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

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

3. **Gaussian elimination**:
   for $i = 1, m+1$
     for $j = 1, n+1$
       if $i \neq r$ then $t_{ij} = t_{ij} - \frac{t_{ip}}{t_{rp}} t_{rj}$
       else $t_{rj} = \frac{t_{rj}}{t_{rp}}$

Figure 2. Main steps of a Simplex iteration.

The n variables $x = (x_1, x_2, \cdots, x_n)$ are referred to as the *optimization variables*. The *cost function* $z(x)$ is a linear function of the optimization variables, and it represents the cost to be minimized. The $m \times n$ linear system of equations $Ax = b$ represents a set of m constraints on the optimization variables; each equation in the system defines a constraint on the n optimization variables.

The Simplex 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.

$$T = [t_{ij}] = \begin{pmatrix} A & b \\ c & z \end{pmatrix}.$$

The application iterates over the tableau in three main steps, as shown in Figure 2. In step 1 a pivot column (p) is located by finding the smallest non-negative coefficient of the vector c (the last row of the tableau). In the second step a pivot row (r) is determined by locating the smallest ratio of the elements of the vector b to the corresponding elements of the pivot column. The tableau element at the pivot row and the pivot column is referred to as the *pivot*. In the third step a Gaussian elimination is performed using the pivot row, and the tableau is updated.

The main source of parallelism in Simplex is in the Gaussian 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, as shown in Figure 3 for $P = 4$. 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 Gaussian 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. This allows the processor that owns

Figure 3. Parallelization of Simplex.

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 into the pivot row buffer, before step 3 is performed. The second barrier is needed after step 3 to ensure that all processors complete the Gaussian elimination 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 sequential fraction of the application. This sequential fraction limits scalability when the number of processors is large.

3.2. LU Factorization

This application decomposes an $n \times n$ matrix A into upper and lower triangular matrices, U and L, using Gaussian elimination without pivoting. The algorithm consists of three loops, as shown in Figure 4. The outermost loop iterates over the matrix A, performing the Gaussian elimination on the submatrix $A_{k,k}$. The element $a_{k,k}$ is referred to as the *pivot* element and the kth row of A is referred to as the *pivot row*. In each iteration a multiplier m_i is computed for each row i using the pivot element.

The application is parallelized by performing the Gaussian elimination in each iteration of the k loop in parallel. The computation is divided among the processors such that each processor is responsible for performing the Gaussian elimination on a subset of the rows of A. Since the Gaussian elimination is applied to a submatrix whose size decreases as k increases, the amount of parallel computation decreases as k increases. Consequently, the rows are statically divided among the processors in a cyclic fashion, as shown in Figure 5 (for $P = 2$), to balance the load. The static assignment of rows to processors is necessary to make each processor responsible for performing the Gaussian elimination on the same subset of rows in successive iterations of the k loop. This enhances local cache affinity and avoids unnecessary remote memory accesses. A barrier synchronization is needed after each iteration

Figure 5. Parallelization of LU.

of the k loop to ensure the Gaussian elimination is complete before the new set of multipliers is computed.

3.3. FFT

This application computes the two-dimensional fast Fourier transform (FFT) of an $n \times n$ matrix A. The main steps are shown in Figure 6. A one-dimensional FFT is applied to each row of A, the matrix is transposed, and the one-dimensional FFTs are applied to the rows again.

The main source of parallelism in an FFT is the independent application of one-dimensional FFTs to the rows of A. The computation is divided such that each processor is responsible for performing the one-dimensional FFTs on a subset of $\frac{n}{P}$ contiguous rows. A temporary array is used to implement the transpose, and hence each processor participates in performing the transpose. Two barrier synchronizations are needed. The first is after the first application of the one-dimensional FFTs, 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 one-dimensional FFTs.

4. Latency Hiding

In this section we describe the sources of remote memory accesses in each application and describe how latency-hiding mechanisms of the KSR2 can be used to hide the latency of these accesses.

```
for i = 1, n
  perform one-dimensional FFT on row i
a_{i,j} = a_{j,i}
for i = 1, n
  perform one-dimensional FFT on row i
```

Figure 6. Main steps of an FFT.

4.1. Simplex

The data accessed remotely in each step of an iteration of Simplex are shown in Figure 7. The number of remote memory accesses in each step is summarized in Table 2. S is the number of tableau elements per subpage.[1]

There are no remote memory accesses in step 1. The master processor locates the pivot column (p) by finding the minimum value in the last row of the Simplex tableau. Since the master processor is also responsible for updating this row of the tableau during the Gaussian elimination step of the preceding iteration, the subpages of the row are already in exclusive state in the local cache of the master, 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 elements of the b vector and the pivot column to determine the pivot row (r). Since both vectors have been updated by the processors during the Gaussian elimination step of the preceding iteration, some of the subpages of the local copies of these vectors in the local cache of the master are invalid. Indeed, the master must read tableau elements from $P-1$ remote local caches, as shown in Figure 7. 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 Gaussian elimination step of the preceding iteration, their 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 computation with the latency incurred by the master to access the next elements of b and the pivot column. 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. This 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 the $(j+1)$st element of the pivot row buffer can be prefetched

(c) Step 3.

Figure 7. Data accesses in the steps of Simplex.

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	$\lceil \frac{n+1}{S} \rceil$	$\lfloor \frac{m+1}{P} \rfloor + 1$

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.[2]

There are two sources of remote memory accesses in step 3. Each processor needs a copy of the pivot row buffer to perform the Gaussian 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 to update these elements. These values have been read by the master in step 2 and exist in a non-exclusive state in the local cache of every processor except the master. Hence, each processor must first 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 *in step 2* of the iteration. 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 to a valid one. The processors find and use this valid copy when they exit the first barrier to perform the Gaussian 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 used in the Gaussian elimination.

4.2. LU Factorization

Remote memory accesses occur in LU during the first access to the pivot row in every iteration of the k loop. Since the matrix has been updated in the previous 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 local cache, causing contention and exacerbating the latency of remote accesses. The average number of subpage misses per processor per iteration is approximately given by

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

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

The natural way to hide remote memory access latency in LU is to use the poststore instruction. This is because there is a single source of data (i.e., the local cache of the processor that last updated the pivot row) and multiple readers of these data (i.e., the remaining processors). Under these conditions the use of the poststore is likely to improve performance [5]. The poststore instruction can be used as follows. In a given iteration k_1 of the outermost k loop, the processor that computes the (k_1+1)st row poststores the subpages of the updated row. Since this row will become the pivot row in the next iteration of the k loop (i.e., (k_1+1)), each processor finds a valid copy of the pivot row in its local cache by the beginning of the next iteration.

However, for the above scheme to work, invalid copies of the subpages of the (k_1+1)st row must exist in the local caches of other processors when the processor that computes this row poststores; the poststore updates a subpage in a local cache only if a copy of the subpage *already* exists in this cache. In fact, at the k_1 iteration of the k loop, the (k_1+1)st row has been accessed only by one processor—the one that is assigned to compute it—and by none of the other processors. Nevertheless, invalid subpages of the row may exist in the local caches of other processors because adjacent pivot rows reside on the same (16-Kbyte) operating system page. Consider the first iteration of the k loop, in which row 1 is used as the pivot row by every processor, and row 2 is poststored by processor 2. Processors $2 \ldots P$ first miss on the subpages of row 1 and access them remotely from processor 1. This causes a page to be allocated in the local cache of each of processors $2 \ldots P$. This page can hold a copy of not only row 1, but also of adjacent rows. The constituent subpages of the page are initially all invalid. The responses to the remote accesses to row 1 update the subpages of this row on the page; subpages of adjacent rows remain invalid. Hence when processor 2 poststores the subpages of row 2, the poststores find invalid subpages of the row in the local caches of other processors and can update them. That is, missing on the subpages of one pivot row creates invalid subpages of adjacent rows in the local caches of the processors, making poststores of future pivot rows effective. The poststores are not effective when the current pivot row is on one page and the next pivot row is on the next page. Clearly, the effectiveness of the poststore instruction in reducing the total number of subpage misses depends on the number of adjacent rows in a page.

The prefetch instruction may be used to hide the latency of pivot row accesses by having $P-1$ processors prefetch pivot row subpages from the local cache of the

processor that just updated the row, when the row is first accessed in an iteration. However, this requires a prefetch message and a response message (for each subpage) for each of the $P-1$ processors, as opposed to a single poststore message (for each subpage) for all $P-1$ processors. More seriously, however, it results in the same contended data access pattern that exacerbates the latency of remote accesses. Another way to use the instruction is for each processor to prefetch the (k_1+1)st row while using the k_1th pivot row to update the rows assigned to it in an iteration k_1 of the k loop. However, this also generates more requests and response messages, and can result in contention, with no guarantee that the processor updating the (k_1+1)st row is finished updating the row before the remaining processors prefetch the subpages. Consequently, prefetched subpages may later be invalidated and the benefit of the prefetch may be lost. Hence the prefetch instruction is not considered for hiding the latency of pivot row accesses in LU.

4.3. FFT

Remote memory accesses occur in the 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 one-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 the 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, and can increase the time to finish the first step of the FFT. 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 the processors on one ring:0 ring and 26 threads on

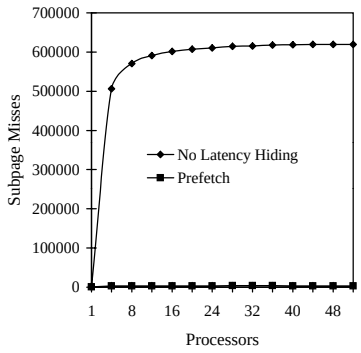

Figure 8. Subpage misses in Simplex.

the processors on a second ring:0 ring. Execution times and the number of remote memory accesses 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 714×1938 problem. LU was used to solve a 1000×1000 problem, and FFT was used to solve a 512×512 problem.

5.1. Simplex

The impact of the prefetch instruction is shown in Figures 8 and 9. Figure 8 shows the number of subpage misses occurring in step 2 with and without prefetch. It indicates a significant reduction in the number of subpage misses with the use of prefetch. Figure 9 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 grows from 24 to 28. This is because with 28 processors, the threads span the two rings of the system; 26 threads on ring:0 and 2 threads on ring:1. Remote memory 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.

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 9. 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 and the reuse of the pivot row buffer 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 10. The use of the prefetch instruction improves the speedup of the application by about 20% when the number of processors is large. This smaller benefit is mainly because

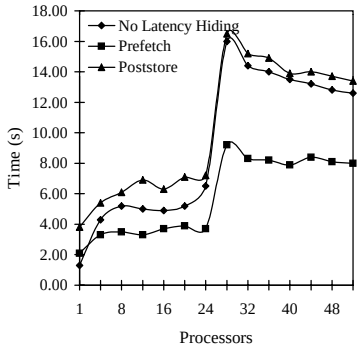

Figure 9. Serial time in Simplex.

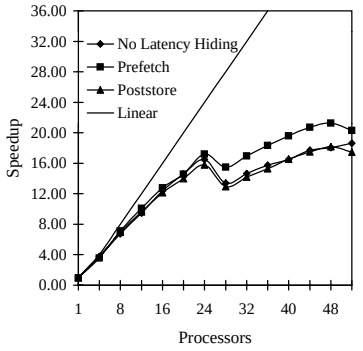

Figure 10. Speedup of Simplex.

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 per iteration (equation 1) is shown in Figure 11. It reflects a decrease in the number of subpage misses with the use of the poststore instruction. The figure also shows a decline in the average number of subpage misses as the number of processors increases, even though each processor references the same number of nonlocal subpages independent of the number of processors (see equation 1). We offer the following explanation based on the effect of automatic update [8].

The number of subpage misses incurred by each processor in an iteration[3] of LU using 20 processors is shown in Figure 12. Since all P processors read the pivot row immediately after leaving the barrier to begin a new iteration, and since valid copies of pivot row subpages exist only in the local cache of one processor (see Section 4.2), it is natural to expect that $P-1$ processors miss on each pivot row subpage. However, Figure 12 shows that while some processors incur the expected number of misses, some incur far fewer misses. If all processors request a pivot row subpage at the same time, then each would miss on that subpage. However, if the processors access the subpage at different times, then processors that request the subpage first miss on the subpage. The response to the miss request automatically updates invalid pivot row subpages in the local caches of processors that are on the path traveled by the response. Processors that are late accessing the subpage find it valid in their local caches and suffer no miss.

The number of processors that benefit from automatic update is nondeterministic and depends on the order in which processors request a subpage, the location on the ring of the local cache that contains a valid copy of the subpage and on the

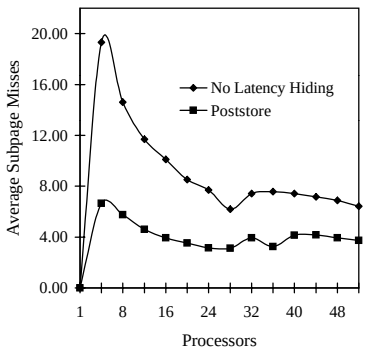

Figure 11. Average subpage misses in LU.

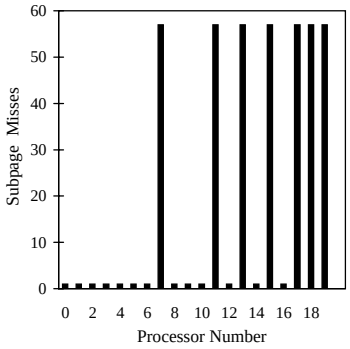

Figure 12. Processor misses in LU.

relative position of processors that first request the subpage with respect to this local cache. However, as the number of processors increases, the likelihood of more processors benefiting from the automatic update also increases, for two reasons. First, a single response to a miss request can update invalid pivot row subpages in more local caches since there can be more local caches on the path of the response when the number of processors is larger. Second, some processors encounter a longer delay in leaving the barrier when the number of processors is larger. Processors leaving the barrier later are more likely to see their subpages updated when they request the pivot row data. Consequently, as the number of processors increases and more processors benefit from the automatic update, the average number of subpage misses declines, as displayed in Figure 11.

The impact of the poststore instruction on the overall performance of LU is shown in Figure 13. 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.

The impact of the number of rows per operating system page on the effectiveness of the poststore instruction is shown in Figure 14. It shows the relative reduction in the number of subpage misses, defined by

$$\frac{\text{subpage misses without poststore} - \text{subpage misses with poststore}}{\text{subpage misses without poststore}} \times 100\%,$$

as a function of the size of the matrix n. It indicates that the relative reduction of subpage misses with poststore decreases as the size of the matrix increases, and consequently the number of rows per page decreases. This confirms our analysis in Section 4.2. The relatively large reduction in the number of subpage misses is reflected as a relatively smaller reduction in the overall speedup of the application. For example, Figure 14 indicates a 60% reduction in subpage misses for $n = 1000$,

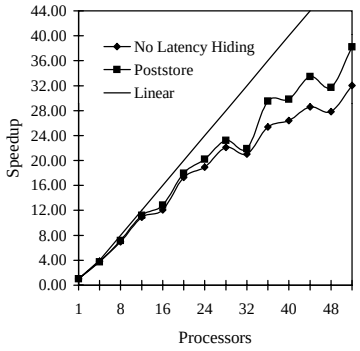

Figure 13. Speedup of LU.

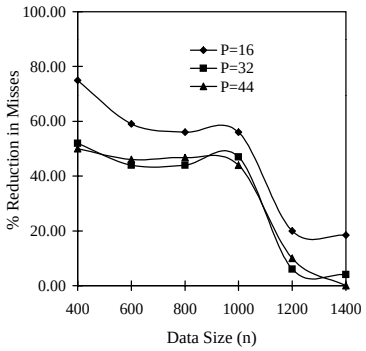

Figure 14. Impact of data size in LU.

whereas Figure 13 shows only a 15% to 20% improvement in overall speedup. This is because the latency of misses is only a fraction of total execution time; computations performed by the processor also contribute to execution time. Furthermore, misses to pivot row subpages occur only while updating the first row assigned to a processor. The now-local pivot row data are then reused to update the remaining rows.

5.3. FFT

The average number of subpage misses incurred in the transpose step of the application is shown in Figure 15, 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 16. This is attributed to only a small reduction in the number of subpage misses and 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. [9] conducted a similar study using an iterative sparse linear equation solver. They concluded that while performance improvements can be obtained through the 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. [8] studied the impact of thread placement on performance using a set of synthetic benchmarks. They conducted experiments on a two-ring 64-

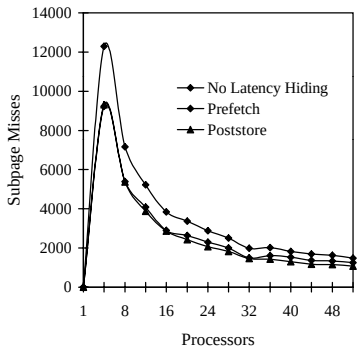

Figure 15. Subpage misses in FFT.

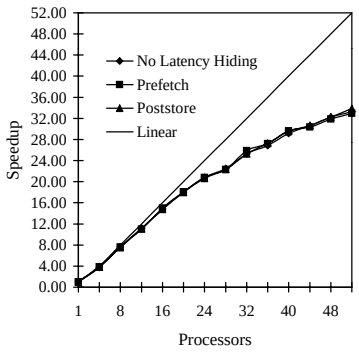

Figure 16. Speedup of FFT.

processor system. Their results indicated performance benefits from the automatic update, but suggested no systematic way of exploiting it in real applications.

Rosti et al. [5] analytically modeled the poststore instruction to determine conditions under which 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.

7. Summary and Conclusion

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 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, and one should weigh their expected benefit against their overhead. The use of the instructions in critical parts of the program, such as the sequential fraction, may degrade overall performance instead of improving it.

Acknowledgments

This research is supported by grants from NSERC (Canada) and ITRC (Ontario). The use of the KSR2 was provided by the University of Michigan Center for Parallel Computing.

Notes

1. In our implementation, each tableau element is 64 bits long and hence $S = 16$.
2. Since there are S elements on a subpage, the prefetch is done once per subpage, not once per element.
3. We arbitrarily chose iteration $K = 100$ in LU without poststore to collect these results. The size of the pivot row for this iteration is 900 elements, which corresponds to 57 subpages.

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, Rochester, NY., 1993.
2. T. Chen, and J. Baer. A performance study of software and hardware data prefetching schemes. *The 21st Annual International Symposium on Computer Architecture,* pp. 223–232. IEEE Computer Society Press, Los Alamitos, Calif., 1994.
3. E. Hagersten, A. Landin, and S. Haridi. DDM– A cache-only memory architecture. *IEEE Computer,* 25:44–54, 1992.
4. Kendall Square Research. *KSR1 principles of operation manual.* Kendall Square Research Corporation, Waltham, MA., 1992.
5. E. Rosti, E. Smirni, T. Wagner, A. Apon, and L. Dowdy. The KSR1: experimentation and modeling of poststore. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* pp. 74–85. The Association for Computing Machinery, 1993.
6. D. Solow. *Linear programming: An introduction to finite improvement algorithms.* North-Holland, New York, NY., 1984.
7. P. Stenström, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures, *The 19th Annual International Symposium on Computer Architecture,* pp. 80–91. IEEE Computer Society Press, Los Alamitos, Calif., 1992.
8. T. Wagner, E. Smirni, A. Apon, A. 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, Oak Ridge, TN., 1993.
9. D. Windheiser, E. Boyed, E. Hao, S. Abraham, and E. Davidson. KSR1 multiprocessor: analysis of latency hiding techniques in a sparse solver. *The International Parallel Processing Symposium,* pp. 454–461. IEEE Computer Society Press, Los Alamitos, Calif., 1993.

Received Date
Accepted Date
Final Manuscript Date