Copyright April 1993

Analysis of Memory Latency Factors and their Impact on KSR1 MPP Performance

Bassam Kahhaleh *

Advanced Computer Architecture Lab.†
Department of Electrical Engineering and Computer Science
1301 Beal Avenue, Room 2312 EECS
The University of Michigan
Ann Arbor, MI 48109-2122

Phone: (313) 763-6970
FAX: (313) 763-4617
email: kahhaleh@eecs.umich.edu

Apr. 15, 1993

Abstract

The Kendall Square Research KSR1 MPP system has a shared address space, which spreads over physically distributed memory modules. Thus, memory access time can vary over a wide range even when accessing the same variable, depending on how this variable is being referenced and updated by the various processors. Since the processor stalls during this access time, the KSR1 performance depends considerably on the program's locality of reference. The KSR1 provides two novel features to reduce such long memory latencies: prefetch and post-store instructions. This paper analyzes the various memory latency factors which stalls the processor during program execution. A suitable model for evaluating these factors is developed for the execution of FORTRAN DO-loops parallelized with the Tile construct using the Slice strategy. The DO-loops used in the benchmark program perform sparse matrix-vector multiply, vector-vector dot product, and vector-vector addition, which are typically executed in an iterative sparse solver. Memory references generated by such loops are analyzed and their memory latencies are experimentally evaluated. Thus, the performance of the KSR1 and its unique memory system is determined. Furthermore, the prefetch and post-store operations are evaluated and their effects on performance and memory latencies are determined. The limited size of the prefetch queue is shown to stall the processor for a long period of time, which reduces the benefit of prefetch considerably. The post-store operation is evaluated with two placements: immediate and delayed post-store. In both cases, the post-store operation has a high overhead. However, it is shown that delaying the post-store operation improved performance considerably.

*On sabbatical leave from University of Jordan, Amman, Jordan.

†This work uses facilities partially funded by NSF grant CDA-92-14296.

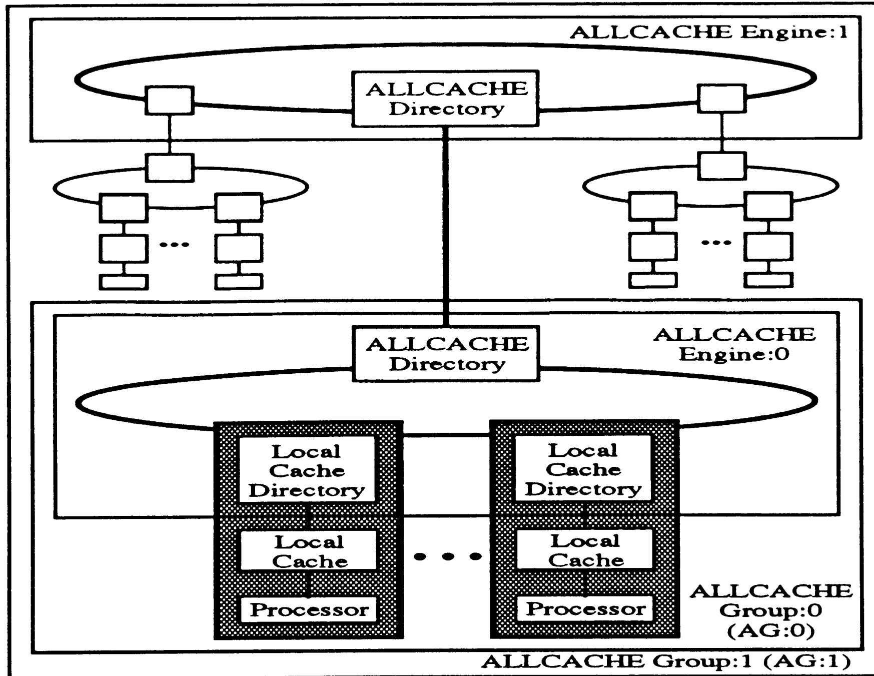

Figure 1: Scalability and hierarchy of KSR1.

1 Introduction

The Kendall Square Research machine 'KSR1' is a Massively Parallel Processors (MPP) system with shared address space and distributed physical memory modules. The shared address space provides the ease of use of the shared memory programming model, without worrying about allocating storage, managing sharable data or passing messages from one processor to another, while the distributed memory design provides the ease of scalability. These two features are provided by the unique design of KSR1's distributed memory scheme, the $ALLCACHE^{TM}$ memory system.

1.1 KSR1 Architecture

The KSR1's physical memory is distributed among all nodes equally. Each node contributes 32 MB of storage to the total cache capacity of the system. The total address space is completely sharable among all processors, with special hardware to perform the required accesses and dynamic storage allocation and management. Thus, each processor can access any data element independent of its physical location.

The system is highly scalable, up to 1088 processor nodes referred to as 'cells', with communication based on a hierarchy of 2 levels of rings, as seen in Figure 1. The first level, which is referred to as 'ALLCACHE ENGINE:0' or AE:0, can connect a maximum of 32 cells. The second level, 'ALLCACHE ENGINE:1' or AE:1, can connect a maximum of 34 AE:0 rings together. The ALLCACHE ENGINE is capable of a peak data transfer rate of 1 GByte/sec, with a packet length of 128 data bytes and a 16-byte header [6]. However, the maximum achievable data transfer bandwidth is reported as 731 MBytes/sec [2].

Each cell consists of a 64-bit superscalar processor, 0.25 MB instruction 'sub-cache', 0.25 MB data 'sub-cache', and 32 MB 'local-cache'. The processor is clocked at the rate of 20 MHz, and can perform a maximum of two floating point operations per clock. It is designed with a VLIW format that allows two instructions to be issued every clock: a data computation instruction and a memory access instruction. The peak performance of each cell is rated at 20 MIPS (VLIW instructions) and 40 MFLOPS, with 64 floating point registers, 32 integer registers, and 32 address registers [5].

The two sub-caches are organized as a 64-set, 2-way set associative caches with random replacement strategy. Storage is allocated with tag descriptors in 2 KBytes 'block' units, while data movement between sub-cache and local cache is based on 64-Byte 'subblock' units. The local cache is organized as a 128-set, 16-way set associative cache with LRU replacement strategy. Storage is allocated in 16 KBytes 'page' units, while data movement

between local caches is based on 128-Byte 'subpage' units. With the main memory of each processor made as a cache memory, the system is considered as a Cache-Only Memory Architecture (COMA) [4], with guaranteed cache coherency and automatic data movement between caches. Other cache protocols also employ automatic update feature, for example [3].

Once a program is loaded in the KSR1 memory and starts to run, a reference to a data element is usually satisfied by the processor's own data sub-cache. The sub-cache memory latency time is masked out by the fact that such a memory reference must be made ahead of time before the data element is actually required. Thus, the processor finds the required data already loaded in a register and does not stall, unless the reference misses the sub-cache. If a sub-cache miss occurs, the memory reference is usually satisfied by the local cache associated with the referencing processor and the referencing processor stalls until the reference is satisfied. If a local cache miss occurs, the memory reference is usually satisfied by a remote cache associated with another processor. This memory reference is usually associated with a rather long delay, which includes the time to search other caches and transfer the required data once found to the requesting processor. Assuming a page is already loaded in memory, this time delay is minimum if the data element is found in the local cache of a processor on the same ring as the requesting processor and the ring is lightly loaded, and maximum if the processor is on a remote ring and the rings are heavily loaded. Once the required data element is found, the *ALLCACHE* memory system automatically moves or copies the subpage containing this data to the local cache of the referencing processor. The local cache then loads the sub-cache with the subblock containing the required data. Thus, the KSR1 is considered as a Non-Uniform Memory Access (NUMA) system.

To reduce the delay time associated with multiple processors reading the same data, the *ALLCACHE* memory system provides multiple 'Read-only' copies of that data to all relevant processors. However, if a new value is to be written into a data element, all existing 'Read-only' copies of this data element in the system are invalidated automatically. This invalidation is essential to maintain cache coherence. Therefore, data which is referenced by one processor tend to remain only within the associated local cache and sub-cache, providing locality of reference. Shared data which is updated by multiple processors, on the other hand, is migrated automatically by the special hardware from one processor to another as each processor writes its new value. Therefore, no explicit move data instructions are required here to enhance the performance when referencing data in remote memories, as with other NUMA architecture systems.

To reduce the time penalty associated with shared data references, KSR provided two instructions that can be used directly by the user (or compiler): the *prefetch* and *post-store* instructions. The prefetch instruction is usually inserted at an earlier stage to bring the required data elements (one subpage) to the local cache ahead of time. The processor continues to execute following instructions normally without waiting for the prefetch instruction to finish. Thus, the high delay penalty associated with memory accesses to remote caches is now overlapped with the execution time of many instructions. The post-store instruction, on the other hand, provides a simple means of broadcasting the value of one subpage of data elements to the rest of the processors. Post-store is usually performed after a processor updates a data element to send the new value to other processors which would require to read this data element in the future. This broadcast is especially useful if other cells have invalid copies of the broadcasted subpage and they are not too busy to ignore it [5]. Thus, subsequent references to this data element by those processors which utilized this broadcast would avoid the long stall time normally incurred in bringing the new value from a remote cache on demand.

The KSR1 provides another built-in feature to help reduce the long memory latency associated with shared data accesses: the *automatic update* feature. When a local cache misses on its processor's reference and a request is generated on the ring, the first cache on the ring that has the required subpage in non-invalid state responds with the data for this request. As the response goes from one cell to another on its way back to the requesting processor each local cache, with an invalid copy of the transmitted subpage, along the response path gets a chance to automatically update its copy, unless the local cache is "too busy."

1.2 Parallelizing FORTRAN DO-loops

The KSR1 provides three major parallel constructs to exploit parallelism in a high level program: parallel region, parallel section, and tile families. These constructs provide the necessary high level interface to pthreads [7]. The parallel section construct is used to execute multiple blocks of code in parallel. The parallel region construct is used to execute multiple instances of a block of code in parallel. Tiles, however, are used to parallelize DO-loops by decomposing the iteration space into tiles, and executing these tiles in parallel. Each tile is executed

by a pthread and each processor may execute one or more pthreads. Since all of the work in splitting the loop iteration space over a number of pthreads and managing their execution on multiple processors is done by the system software, tiling provides a simple and easy way to parallelize FORTRAN programs. Different tiling strategies are provided to handle various loop constructs, loop dependencies, and vector or matrix accessing patterns. The strategy may be static (determined at compile time) like the *slice, mod,* and *wave* strategies, or dynamic (determined at run time) like the *grab* strategy [6]. The *slice* strategy slices the iteration space of the DO-loop so that each pthread executes the DO-loop over a certain part of the iteration space. This is the simplest strategy and has the lowest overhead, but depending on how much work each thread performs the *slice* strategy might not give best performance.

1.3 Programming benchmark

When executing a parallel construct, the KSR1 performance depends on several factors. Since the KSR processor does not have hardware interlocks to stall it when an operand is not ready, explicit no-op instructions are used to provide the required processor delay until all operands become available. To increase the MFLOPS of the system, each processor should execute as many floating point operations as possible with minimum no-ops inserted. This objective depends usually on the application program in terms of the required computation, register allocation and usage, and data dependency. When each processor executes the generated code, the throughput becomes a function of memory performance. To maintain maximum throughput, the memory system should provide the required instructions and data at the required rate with minimum possible latency.

This paper investigates the performance of the KSR1 when executing some common operations such as matrix-vector multiply, vector-vector dot product, and vector-vector addition/subtraction, as found in an iterative sparse solver. The benchmark program used is a Finite Element Method (FEM) radiation backscatter modeling application, using the diagonal-preconditioned symmetric biconjugate gradient method to solve a system of complex linear equations [1]. All DO-loops in this program were tiled with the slice strategy. Figure 2 shows the main loop outlining the performed vector operations. The execution time of these operations is analyzed to determine the effect of memory references on performance. In an early study [8], the performance of KSR1 executing this benchmark was reported with emphasis on the effects of using latency hiding techniques. The observed performance improvement due to employing prefetch and post-store techniques was much reduced by the unexpected high overhead of using these two instructions. This paper analyzes the prefetch and post-store operations in detail and determines the factors of their high overhead and their effects on performance and memory latencies. The KSR1 system used in these experiments is configured with 32 processors on one ring.

2 Tile Execution Model

Each processor in the KSR1 system consists of several units (custom VLSI chips) that perform the required functions such as instruction execution, cache control, and ring interface. Basically, the Floating Point Unit (FPU), the Integer Processing Unit (IPU), and the Cell Execution Unit (CEU) can be accessed and programmed directly by the user. The FPU executes the required arithmetic operations on floating point operands stored in the floating point register file. The IPU executes the required integer and logical operations on integer operands stored in the integer register file. Both FPU and IPU units are capable of multiplication and addition, and both are pipelined to different levels. Only one arithmetic operation can be initiated each clock except for the multiply-and-add instructions which perform two floating point operations in a single instruction. The CEU performs the required instruction fetch, operand loading and storing, branching, and address calculation. It contains 32 address registers used to handle all operations dealing with memory. Each instruction consists of two parts: one part goes to either the FPU or IPU, and the other part goes to the CEU (or the XIU which handles system input/output). The two sub-caches and the local cache are also managed and controlled by other custom VLSI chips, which are transparent to the user.

Although the KSR1 processor is designed to execute one VLIW instruction every clock, data movements between different units including memory take several clocks. Thus, loading or storing an operand from or to memory or moving an operand from an integer to a floating point register takes additional two clocks to finish the instruction. Thus, when moving an operand to a register, this time delay is masked by executing two other instructions (or no-ops) before accessing the loaded register. When moving an operand to memory, the processor continues to execute subsequent instructions, but no instruction should access this memory location until the write period is over. When no-ops are used to mask the data movement delay, the performance of the KSR1 in terms

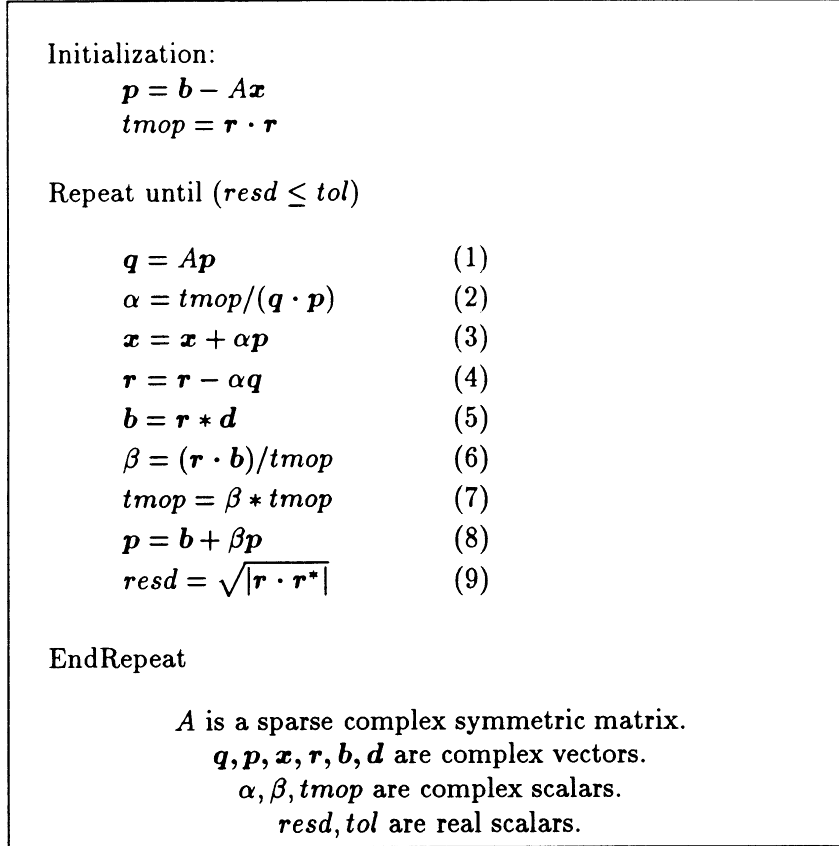

Figure 2: Main loop of a sparse solver using symmetric biconjugate gradient method.

of MFLOPS decreases accordingly, but the processor does not stall unless a sub-cache miss occurs. Therefore, a memory access resulting in a sub-cache miss introduces extra delay times that lower the performance.

To evaluate the effect of such memory references on performance, the following model is used. Local variables are defined as those variables referenced by one processor throughout the given program. Shared variables are defined as those variables referenced cyclicly by more than one processor. As local variables are referenced, each processor generates a stream of references defined as *Local References.* Thus, a local reference might miss the sub-cache but will always hit the local cache. Referencing shared variables, on the other hand, may hit or miss the local cache depending on the referenced data and the performed operation. For the read operation, a processor referencing a shared data element, which was updated by another processor, would generate a *Remote Reference* that misses its local cache. This miss is then satisfied by acquiring a copy of the referenced data from a remote cache. However, any further read references to this data would hit the local cache, until another processor updates it. Therefore, a first time read reference to a shared variable, i.e. with a new value, is taken as a remote reference and any subsequent references to the same value are taken as local references. For the write operation, a processor referencing a shared data element which was copied to other processors would generate a remote reference that causes all other cells to invalidate their copies of this data. Any further write references to the same data would generate local references, until another processor obtains a copy of this data.

When a local reference misses the sub-cache, a subblock containing the required data element is copied from local cache to sub-cache. Therefore, local references to all data elements in the same subblock are combined together and taken in the model as one subblock reference. Similarly, a remote read reference is satisfied by copying a subpage containing the required data element from a remote cache, while a remote write reference is satisfied by invalidating all corresponding subpages in other remote caches. Therefore, remote references to all data elements in the same subpage are combined together and taken in the model as one subpage reference.

When a DO-loop with 'n' iterations is tiled with the slice strategy over 'p' threads, the iteration space is divided so that each thread executes the DO-loop over a given range of the loop index. For better performance, tiles

are aligned on subpage boundaries, i.e. the lower and upper bounds of the tile's loop index are computed so as to allow each thread to access whole subpages of shared variables. Therefore, the number of iterations executed by each thread is taken to be n/p with the assumption that the 'n' is divisible by 'p' and the resultant tiles are subpage aligned. If this is not the case, the next higher value of 'n' which satisfies this condition is used, and the last tile would stop when its index reaches the original value of 'n'.

Given a DO-loop with I instructions in its body and assuming no data references are generated, the time (in clocks) it takes to fetch and execute each tile of this loop is given by:

$$t_{thread} = \frac{n}{p} \; [\, I + \lceil \frac{I}{8} \rceil * T^{i} \,] \tag{1}$$

where T^{i} = average memory delay incurred in fetching an instruction subblock (8 VLIW instructions).

Now assume the DO-loop makes local references to 'L' distinct subblocks and remote references to 'R' distinct subpages of data, equation 1 becomes:

$$t_{thread} = \frac{n}{p} \; [I + \lceil \frac{I}{8} \rceil * T^{i} + L * T^{l} + R * T^{r} \,] \tag{2}$$

where
T^{i} = average memory delay in fetching an instruction subblock.
T^{l} = average memory delay in referencing a local data subblock.
T^{r} = average memory delay in referencing a remote data subpage.

The memory latency for a local subblock reference is 0 if the referenced subblock is in the data sub-cache, 23 clocks if the referenced block misses the sub-cache but it is allocated in the local cache, or 49 clocks otherwise. The memory latency for a remote subblock reference is $\geq$ 135 clocks depending on the ring load [8].

Since the same DO-loop in an iterative solver is executed repeatedly, some of the referenced instruction and local data subblocks may have remained in sub-cache from last iteration. In such a case, T^{i} and T^{l} decrease as more referenced subblocks are found in the instruction and data sub-caches. Furthermore, some of the referenced remote data subpages may have been acquired in the local cache as a result of the automatic update feature. In such a case, T^{r} decrease as more referenced subpages are found in the local cache.

To isolate each variable from the rest, the total number of local references 'L' is decomposed into 'L_1', 'L_2', ... etc, where:
L_1 = number of distinct subblocks referenced in variable 1.
L_2 = number of distinct subblocks referenced in variable 2.

.

.

Correspondingly, T^{l} is decomposed into:
T^{l}_{1} = average memory delay in referencing subblocks of variable 1.
T^{l}_{2} = average memory delay in referencing subblocks of variable 2.

.

.

Similarly, the total number of remote references 'R' is decomposed into 'R_1', 'R_2', ... etc, and correspondingly T^{r} is decomposed into T^{r}_{1}, T^{r}_{2}, ... etc. Assuming there are 'V_L' variables contributing to local references and 'V_R' variables contributing to remote references, equation 2 becomes:

$$t_{thread} = \frac{n}{p} \; [\, I + \lceil \frac{I}{8} \rceil * T^{i} + \sum_{j=1}^{V_L} L_j * T^{l}_{j} + \sum_{k=1}^{V_R} R_k * T^{r}_{k} \,] \tag{3}$$

Since the instruction sub-cache is separate from the data sub-cache and, practically, the instruction sub-cache is large enough to hold all or most of the instructions, T^{i} can be set to 0. The experimental value obtained for T^{i} was a very small fraction and can easily be ignored.

The parameters n, p, L_j and R_k are derived from the tile analytically. Unfortunately, the parameter T^{l}_{j} is much more difficult to compute since it depend on the behavior of the cache and its random replacement strategy. This

would also mean that each tile may have a different T_j^l. Therefore, T_j^l and T_k^r are measured experimentally and reported in this paper as an average value taken over all tiles except the last one. This gives a better accuracy since the last tile was executing less number of iterations than all other tiles due to the 'n/p' and subpage alignment constraints mentioned earlier. Furthermore, the timing measurements in these tiles were found in to be around the average value ± 9 %.

3 Performance Evaluation

With the sliced tiling strategy, each thread computes in parallel with the other threads its own part of each vector produced in steps (1), (3), (4), (5), and (8) in Figure 2. In addition, each thread computes its own part of the dot product calculations in steps (2), (6), and (9), then adds its partial result to a global sum to produce the final answer. All remaining scalar calculations are performed by one thread only.

Clearly, '$\boldsymbol{q}$', '$\boldsymbol{x}$', '$\boldsymbol{r}$', and '$\boldsymbol{b}$' vectors are local variables, since each thread would be accessing those elements which correspond to its part of iteration space. The 'A' matrix and the '$\boldsymbol{d}$' vector are read-only variables which are constant. Therefore, each thread would end up having its own copy of the required elements of 'A' and '$\boldsymbol{d}$' in its local cache. Only 'α', 'tmop' and '$\boldsymbol{p}$' are shared variables. The two scalar variables 'α' and 'tmop' are computed by one thread but used by all threads. The '$\boldsymbol{p}$' vector is produced partially by each thread in step (8) and used partly by all threads in step (1): the sparse matrix multiplication. Therefore, steps (1) and (8) are analyzed and their performance is measured experimentally to evaluate the various parameters that affect performance. The memory latency parameters T^l was evaluated by running the program several times, each time adding just one activity to memory. In order not to change the body of the loop, all memory references were replaced by no-op instructions. Then each time a memory reference instruction was restored back in the loop, the average execution time of all relevant tiles is computed and the increase in this time is then attributed to the added memory reference stream. The memory latency parameters T^r was evaluated by running the program with and without prefetching the remote references. The difference in the average execution time is then attributed to the remote references.

Step (1) consumes a major part of the convergence iteration (REPEAT-UNTIL) execution time. The sparse matrix 'A' is stored linearly as a long vector of non-zero values, with two other single dimensional matrices to store the required position information of each element. The 'A' matrix represents a matrix of 20033 rows with an average number of non-zero elements per row of 15.326. Analyzing step (1) reveals four local and one shared variables. The local variables consist of the '$\boldsymbol{q}$' vector and the three matrices which make up 'A'. The shared variable is the '$\boldsymbol{p}$' vector. Since the shared variable is generated iteratively in step (8) by all threads, each thread would reference certain subpages of the '$\boldsymbol{p}$' vector (which were produced by same thread) as local references and all other subpages (which were produced by other threads) as remote references. Furthermore, once a reference to a remote subpage is satisfied another later reference to the same subpage is treated as a local reference since a valid copy of this subpage is still available in the local cache. Hence, five variables contribute to local references and one variable contribute to remote references.

Each local variable in step (1) is referenced at the rate of one element per iteration. Therefore, an average of 0.125 subblocks are referenced per iteration for 8-byte (integer and real) operands and 0.25 subblocks for 16-byte (complex) operands. Let C be the average number of non-zero elements per row in 'A'. By analyzing the assembly code of step (1), the following parameters are computed:

$n = 20033$
$I = 26 + 15 * \mathrm{C}$
$L_1 = 0.25$
$L_2 = 0.125$
$L_3 = 0.125 * \mathrm{C}$
$L_4 = 0.25 * \mathrm{C}$
$L_5 = 0.25 * \mathrm{C}$
$R_1 =$ function { matrix 'A' , the number of threads p }.

L_1 represents references to the '$\boldsymbol{q}$' vector and L_2 through L_4 represent references to the three matrices comprising 'A'. L_5 represents all references made to the '$\boldsymbol{p}$' vector while R_1 represents references to the remote part of '$\boldsymbol{p}$'. Since R_1 is a function of the matrix 'A' and number of threads 'p', the value of R_1 is obtained by monitoring

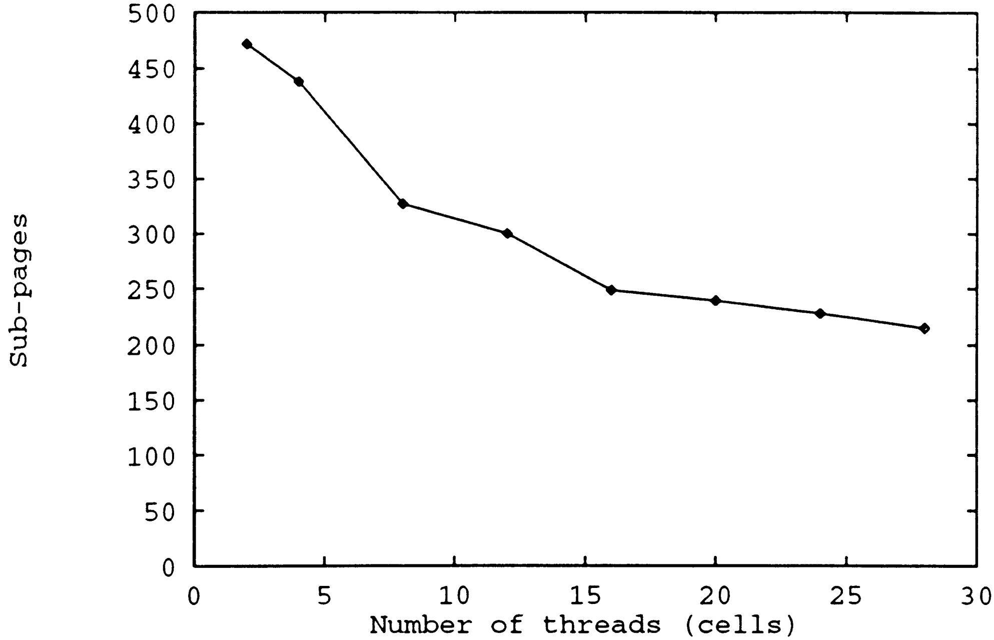

Figure 3: Average number of remote references per thread, R_1, in step (1).

all references made to 'p' in step (1) and isolating those with subscripts that are not within the thread iteration range. Figure 3 shows R_1 versus number of threads. As the number of threads increases and the tile size decreases, R_1 approaches L_5.

Figure 4 shows the average memory latency for L_1 to L_4 as a function of number of threads. The average memory latency for referencing these four variables with small number of threads is around 23 clocks per subblock, i.e. almost every reference to a distinct subblock resulted in a sub-cache miss. As the number of threads increases the tile size decreases and each thread works with a smaller set of data. With higher number of threads, referenced subblocks are more likely to remain in the sub-cache from one convergence iteration to another. Therefore, the performance improves (the average memory latency decreases) with more number of threads. The total data size for local references $= \frac{n}{p} * (L_1 + L_2 + L_3 + L_4 + L_5)$. The total data size for remote references = the number of subpages in figure 3. For 2 threads, the total data size is 6300 KBytes, and for 28 threads it is 475 KBytes. In both cases the total data size is greater than the 256 KBytes data sub-cache size, and therefore most of the distinct references miss the sub-cache.

Figure 5 shows the average memory latency for L_5 and R_1, which represent the shared variable 'p', as a function of number of threads. R_1 = the first time references to the remote part of 'p'. These remote references have an average memory latency of 170 clocks. L_5 consists of all distinct references to the local part of 'p' and any second time references to the remote part of 'p'. The high memory latency for these local references with smaller number of threads is due to multiple sub-cache misses when referencing the same subblock during tile execution. With a small number of threads, the tile size is bigger and the sub-cache can not keep all referenced subblocks simultaneously.

Step (8) is different than all other steps in that each thread updates its part of the shared variable 'p'. Before step (8) is executed, each processor would have read-only copies of certain elements of the 'p' vector, used in the multiplication process in step (1). Therefore, when executing step (8) each processor would be referencing subpages in its local cache with two types of states: non-exclusive and exclusive-ownership. The former state is associated with all subpages that were copied to other processors in step (1). Before a processor can write to a subpage with non-exclusive ownership state the *ALLCACHE* memory system has to invalidates all existing copies of this subpage first then give the writing processor the required exclusive-ownership state to perform the update operation. This process takes extra time which stalls the writing processor for a time similar to a local cache

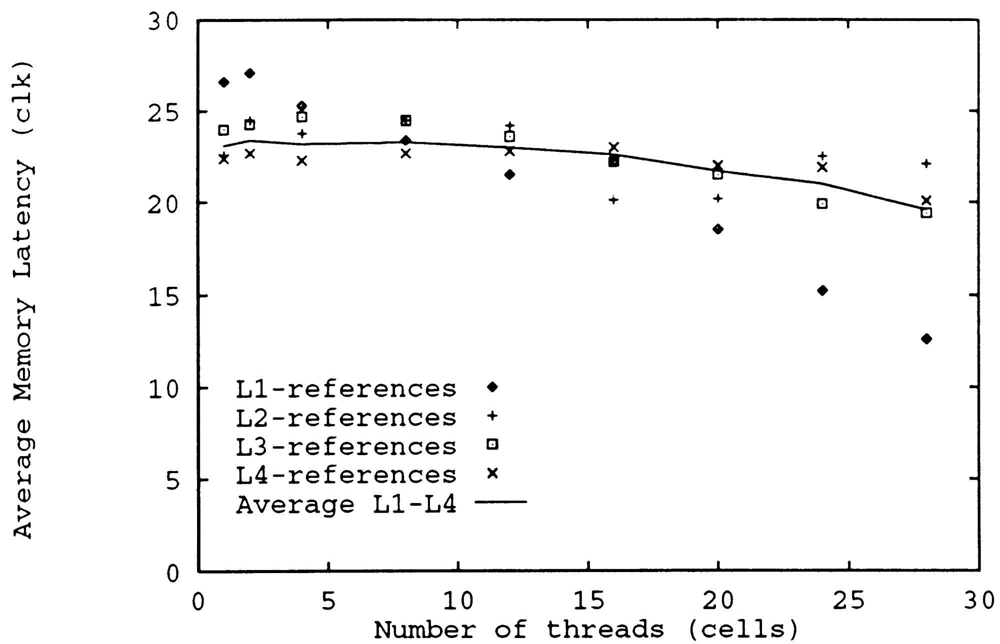

Figure 4: Average memory latency for referencing local variables in step (1).

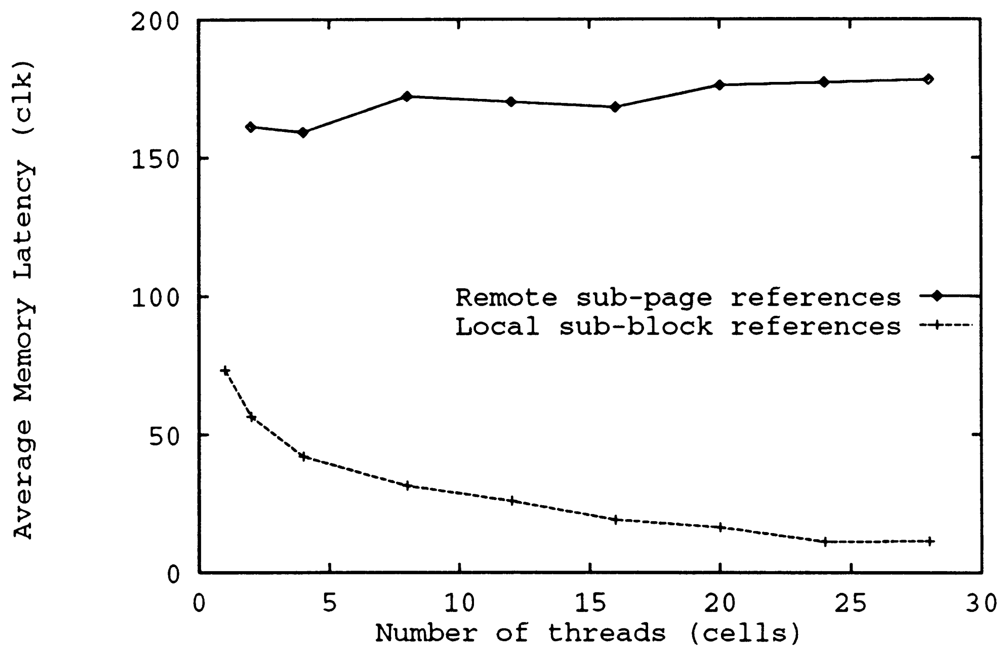

Figure 5: Average memory latency for referencing the shared variable in step (1).

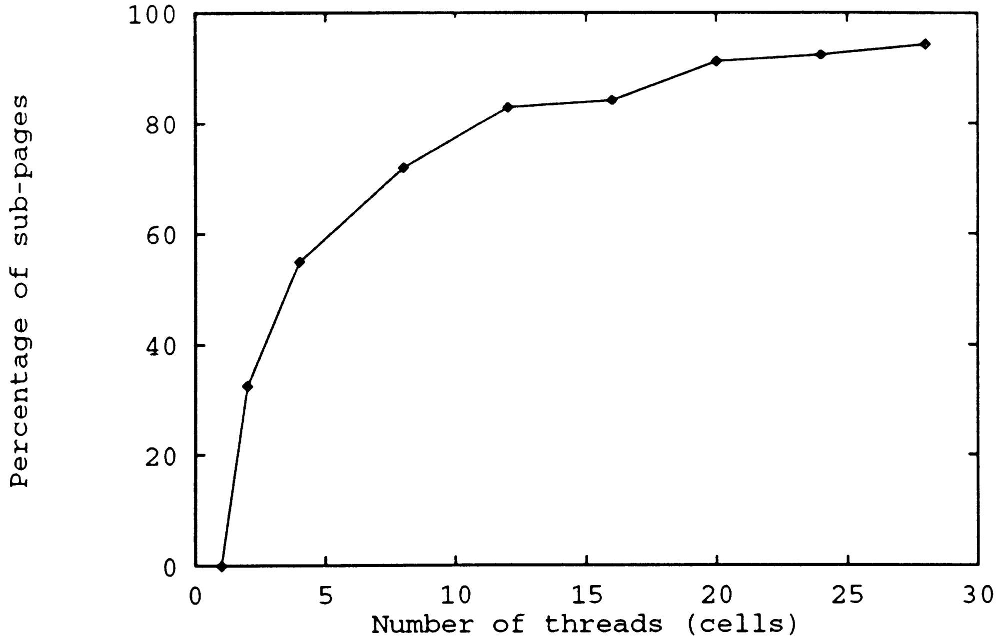

Figure 6: Percentage of subpages of '$\boldsymbol{p}$' which are shared among threads.

miss.

Analyzing step (8) reveals one local variable, '$\boldsymbol{b}$', and two shared variables, 'β' and '$\boldsymbol{p}$'. Since 'β' is a scalar, the time delay associated with its reference is negligible compared with other vector references, therefore it is ignored. All read references to '$\boldsymbol{p}$' are taken as local references as well as the write references to the nonshared part of '$\boldsymbol{p}$'. The write references to the shared part of '$\boldsymbol{p}$' are taken as remote references. Therefore, step (8) is now considered with two local and one remote references. By analyzing the code of step (8), the following parameters are computed:

$n = 20033$
$I = 12$
$L_1 = 0.25$
$L_2 = 0.25$
$R_1 =$ function { matrix 'A' , the number of threads p }.

L_1 represents references to the '$\boldsymbol{b}$' vector. L_2 and R_1 represent references made to the 'local' and 'remote' parts of '$\boldsymbol{p}$', respectively. Since R_1 is a function on the matrix 'A' and number of threads 'p', the value of R_1 is obtained by monitoring all references made to '$\boldsymbol{p}$' in step (8) and isolating those with subscripts that are used by other threads in step (1). Figure 6 shows the percentage of subpages of the '$\boldsymbol{p}$' vector which are shared among processors as a function of number of threads. R_1 can be easily computed from Figure 6.

Figure 7 shows the average memory latency for L_1 and L_2 as a function of number of threads. As the number of threads increases, the average memory latency for referencing '$\boldsymbol{b}$' drops from 15 to 7 clocks per subblock and is always less than the sub-cache miss delay. This indicates that part of references to '$\boldsymbol{b}$' hit the sub-cache every time step (8) is executed in the convergence iteration loop. As expected, this hit ratio increases as the number of threads increases and subsequently the data size per tile decreases.

The average memory latency for referencing the 'local' part of '$\boldsymbol{p}$' increases from 25 to 35.5 clocks per subblock at 12 to 16 threads, then decreases down to 26 clocks at 28 threads. This behavior is due to two opposite factors: referencing the two types of subpages of '$\boldsymbol{p}$'. The first factor is due to the improvement of cache performance with smaller data size. References to subpages with exclusive-ownership state is similar to references to any

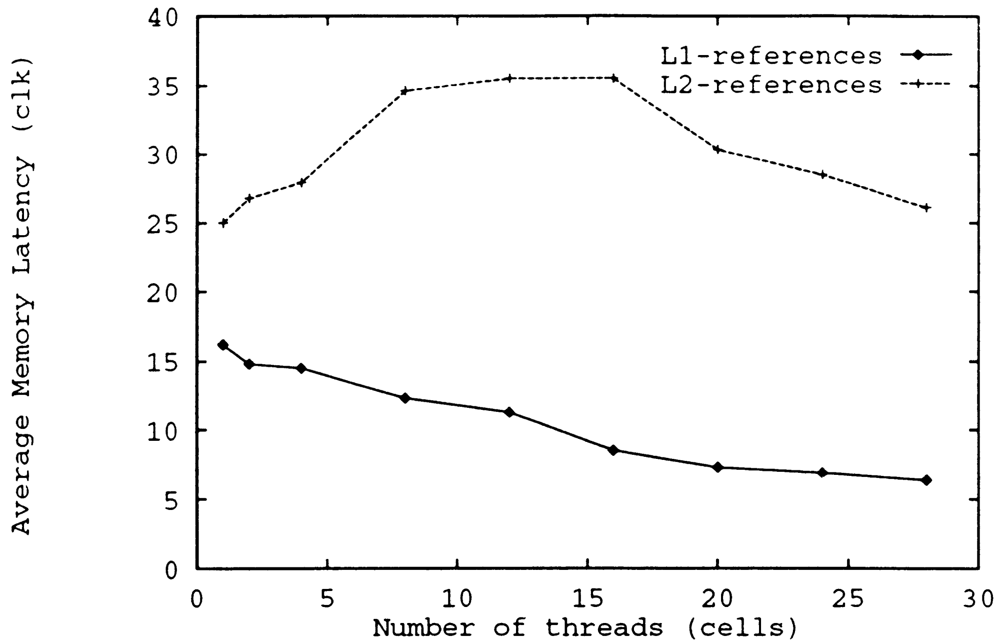

Figure 7: Average memory latency for referencing local variables in step (8).

local variable such as '*b*'; the more threads used the smaller the data size each thread works with and therefore the higher the probability of keeping this data in the sub-cache from the previous convergence iteration. The other factor is due to the apparent effect of missing the sub-cache with references to subpages with non-exclusive ownership state. For a small number of threads, the second factor dominates. As more threads are used, the degree of sharing of '*p*' vector increases, i.e. the percentage of subpages of '*p*' with the non-exclusive ownership state increases in each processor as seen in Figure 6. Hence, the average memory latency for referencing '*p*' increases. For a large number of threads, the first factor dominates and as more threads are used the percentage of shared subpages of '*p*' remains almost constant. Hence, the cache improvement with small data size shows more explicitly.

Figure 8 shows the average memory latency for R_1 as a function of number of threads. The average memory latency for updating the shared part of '*p*' is about 160 clocks per subpage for a small number of threads. As more threads are used, this latency increases due to the fact that the degree of sharing of '*p*' increases at the same time the tile size decreases. Therefore, as the number of threads increases R_1 increases, generating higher traffic on the ring. At the same time more threads are accessing the ring during the shorter tile execution time, producing higher traffic density on the ring.

4 Prefetch Operation Evaluation

The KSR1 provides special instructions to reduce the memory latency associated with accessing shared data, such as referencing the remote part of '*p*' in step (8). The prefetch instruction is typically used to bring the required subpages into the local cache and in the required state, before the data is actually referenced. To utilize this feature, step (6) is modified to include the code to generate the required prefetches to '*p*'. The inserted prefetches require all of the '*p*' subpages to be resident in the local cache with exclusive-ownership state. The local cache simply ignores the prefetch instructions for those subpages which are already in the required state. Figure 9 shows the new average memory latency for L_1 and L_2 in step (8). The average memory latency for R_1 in step (8) is now reduced to 0. As expected, the average memory latency for referencing '*p*' is much lower since most of '*p*' is now in the sub-cache. Furthermore, the average memory latency for referencing '*b*' is higher now probably due to removing some of its subblocks from the sub-cache in favor of '*p*' subblocks.

The cost of the prefetch operation comes from increasing the number of instructions to perform the required

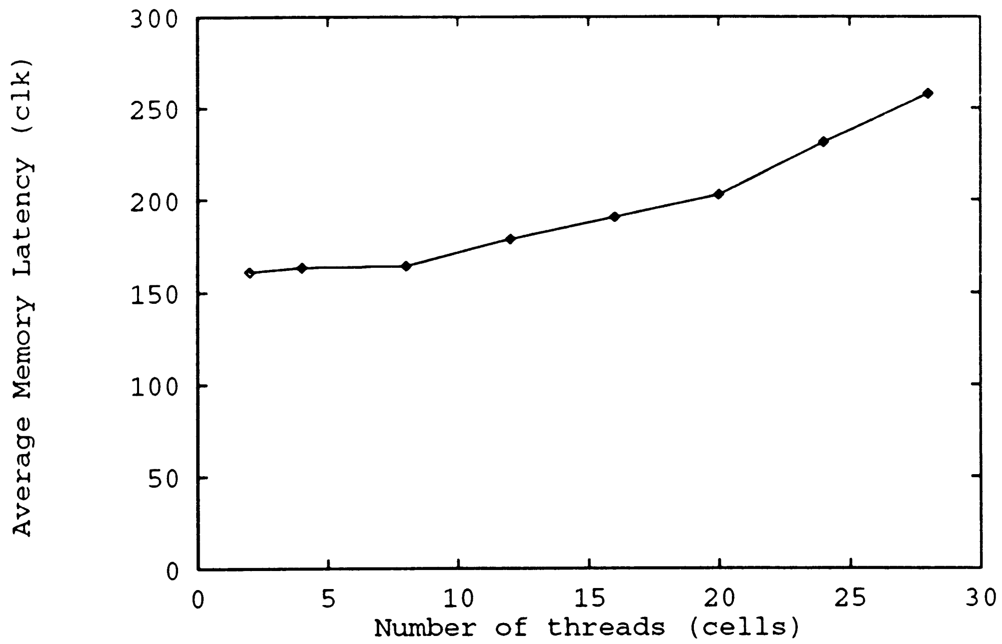

Figure 8: Average memory latency for referencing shared variable in step (8).

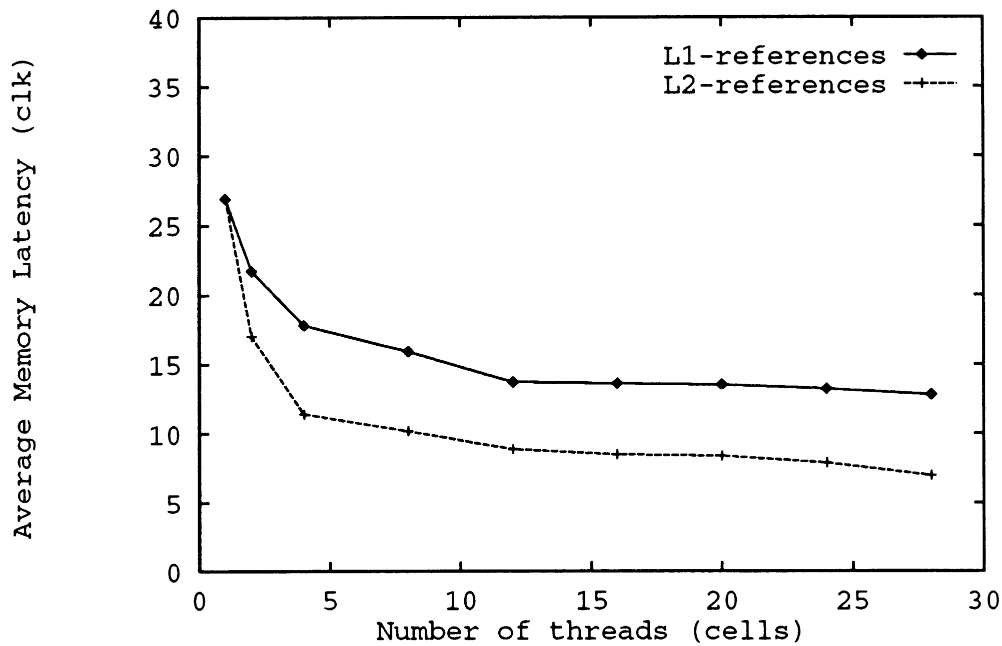

Figure 9: Average memory latency for referencing local variables in step (8) with *Prefetch*.

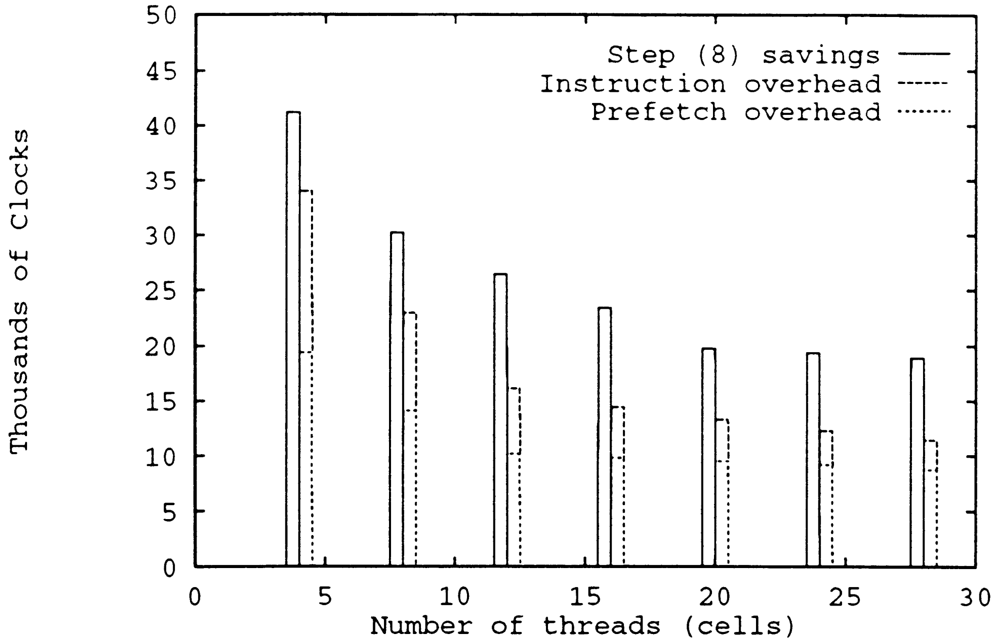

Figure 10: Savings in step (8) vs. cost of *Prefetch* in step (6).

prefetch plus the memory delay due to the execution of the prefetch instruction. The local cache queues the prefetch requests generated by the processor and waits for their response to arrive back later. The prefetch instruction added to step (6) is used with the blocking option to make sure that all prefetches are completed and nothing is dropped due to a full queue. Since the queue can hold only few outstanding prefetch requests, the processor would stall on the next local cache access when the queue is full. Figure 10 shows the savings in step (8) and the cost of using the prefetch instruction in step (6). As expected, both the savings and the overhead decrease with the number of threads as the tile size decreases. Although more instructions were added to use the prefetch operation, there is always an advantage (to a varying degree) of using the prefetch operation. This is basically due to overlapping some of the prefetch overhead (accessing remote caches) with the processor execution time.

5 Post-store Operation Evaluation

The KSR1 provides another special instruction (post-store) to reduce the memory latency associated with accessing shared (remote) data, such as the remote subpage reference latency in Figure 5. To reduce this memory latency, each remote subpage referenced in step (1) should be copied into the local cache before its is actually referenced. The post-store instruction is typically used to broadcast the required subpages around the ring, giving chance to all local caches to copy and store them locally for faster future references. Since the remote subpages of 'p' referenced in step (1) are computed in step (8), the new value of 'p' may be post-stored either in steps (8) or (9).

To evaluate the effect of post-store, Step (8) is programmed to perform one post-store operation on 'p' once a subpage is computed. The execution time of step (8) increases now due to the added instructions (to check and issue a post-store instruction every subpage) and due to the execution time of the post-store instruction itself. As a processor executes a post-store instruction, the referenced subpage is transmitted on the ring where all other cells can copy this subpage into their local cache. This copy operation involves the local cache only, i.e. it is done in parallel with processor operation, unless the local cache is too busy in which case the local cache ignores the broadcast.

To improve the chances of a local cache picking up the broadcasted subpages, two further modifications on step (8) are evaluated. First, step (8) is programmed to perform two post-store operations on 'p'. Once a subpage of

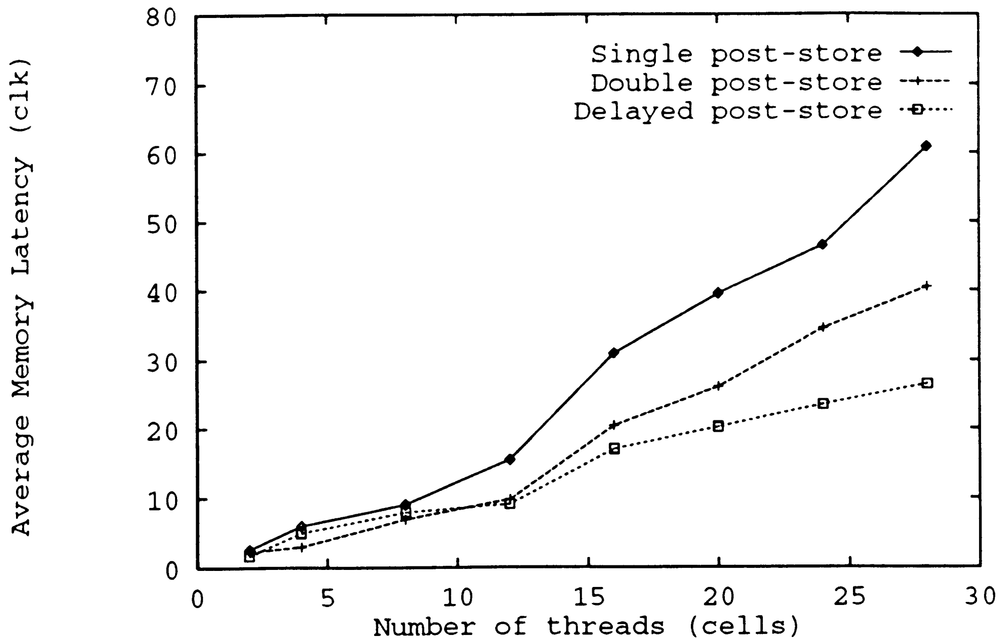

Figure 11: Average memory latency for referencing the shared variable in step (1) with *Post-Store.*

'p' is computed, the processor executes a post-store on this new subpage and another post-store on the previous subpage. Thus, each cell on the ring has two chances of picking up each distinct subpage transmitted by any other cell. Second, instead of performing the post-store immediately after computing the required subpage, the post-store is delayed to a later time.

Figure 11 shows the new average memory latency for R_1 in step (1), using three post-store schemes: single post-store in step (8), double post-store in step (8), and delayed single post-store in step (9). The average memory latency for remote references to the shared variable 'p' is now reduced considerably. However, as more threads are used, the traffic on the ring generated by post-store instructions is higher. With this higher traffic, the local caches are kept busier and subsequently unable to pick up part of the transmitted subpages and copy them for future references. Thus, when executing step (1), references to those subpages dropped in step (8) cause local cache misses, increasing the average memory latency for R_1 as seen in Figure 11.

The cost of the post-store operation comes from increasing the number of instructions to perform the required post-store plus an overhead delay due to the execution of the post-store instruction. When the processor issues a post-store request to its local cache, the processor stalls for a period of time ≥ 8 clocks, at the same time the local cache resource becomes unavailable for a period of time ≥ 24 [5]. Thus, a subsequent reference to the local cache might stall the processor until the local cache is free. Figure 12 shows the savings in step (1) and the cost of using the single post-store approach in step (8). For low number of threads, the total overhead of *Post-store* overcomes the savings, primarily due to the unability of the local cache to perform the post-store on a subpage immediately after it has been updated by the processor. Thus, the processor is stalled until the post-store request is accepted by the local cache.

Figure 13 shows the savings in step (1) and the cost of using the delayed post-store approach in step (9). Since the post-store operation is delayed well after all the required subpages have been computed and stored in memory, the memory latency overhead is now greatly reduced. Still, the processor is stalling for some time, probably due to keeping the local cache busy with the processing of post-store requests.

6 Conclusion

The KSR1 MPP is a shared address space, distributed physical memory system. Its memory consists of local caches and transparent sub-caches only, making it the first commercially available COMA system. Various

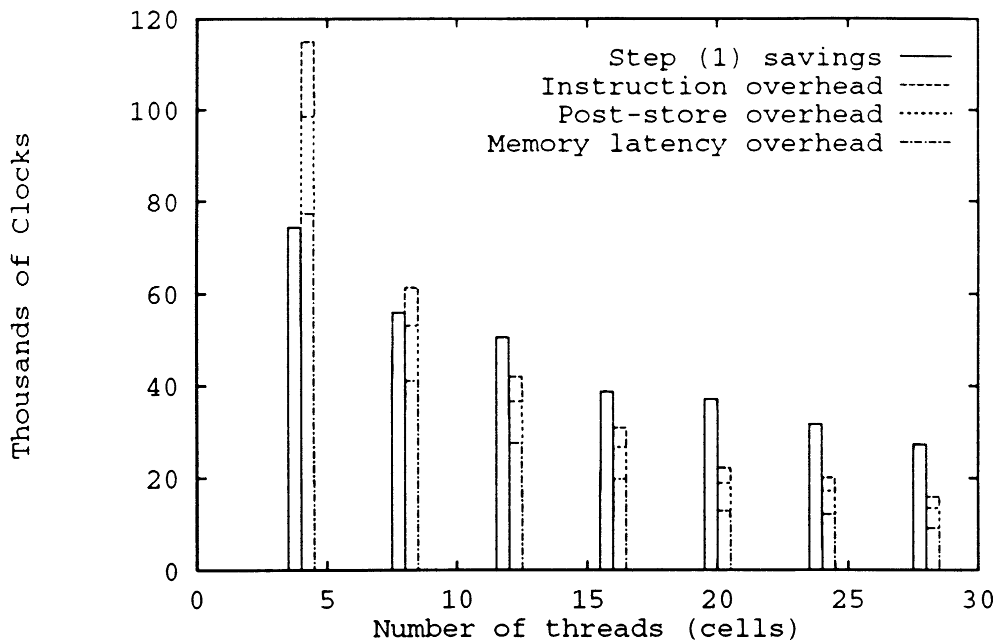

Figure 12: Savings in step (1) vs. cost of single *Post-Store* in step (8).

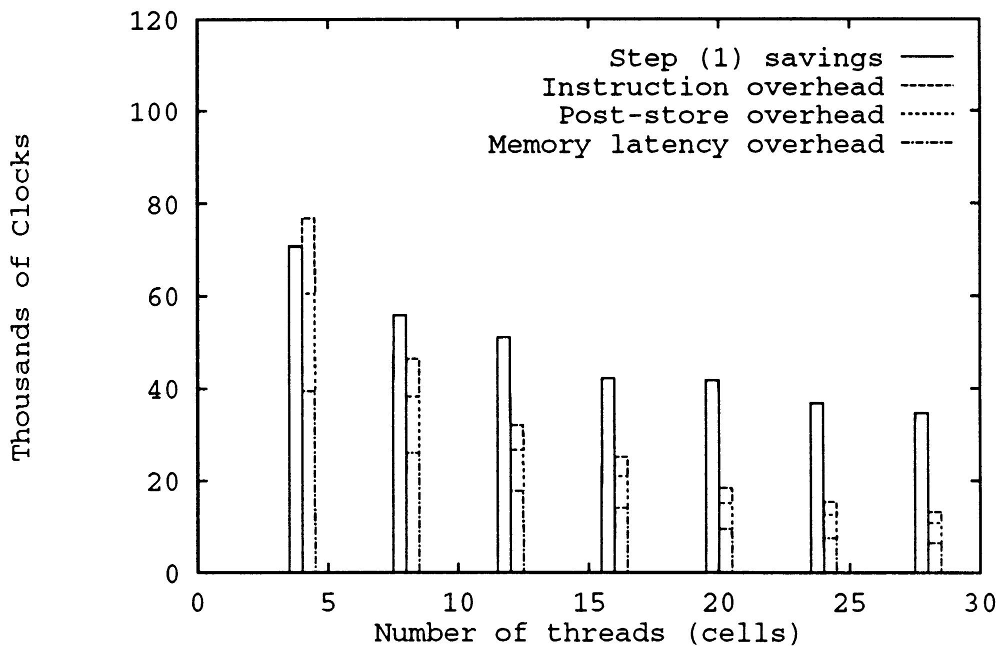

Figure 13: Savings in step (1) vs. cost of delayed *Post-Store* in step (9).

processors communicate over a shared ring, thus incurring long latencies with accesses to caches other than their local caches. This paper evaluated the performance of KSR1 using a Finite Element Method program as a benchmark and tiling DO-loops with the slice strategy. Specifically, memory latencies associated with memory references to both local and shared data were evaluated and plotted versus the number of processors used to execute the tile. KSR1 performance was evaluated for specific steps in the program, which represent the significant part of execution time and ring communication.

Memory references made while executing a tile were characterized as local and remote references. Local references are always satisfied by the local cache of each cell and therefore do not cause cache misses. Remote references are typically satisfied by accessing remote caches and therefore incurring higher memory latencies. Referenced variables in DO-loops were characterized as either local or shared. Local variables are those which are referenced by one processor only, such as referencing a vector with an index within the tile bounds. Hence, accessing local variables produces local memory references only. Shared variables are those which are referenced by more than one processor. Furthermore, shared variables can either exist as a single copy in one cache (exclusive-ownership state) or multiple copies in several caches (read-only and non-exclusive ownership states). Accessing a shared variable usually produces remote memory references, and some times additional local memory references, depending on the program. Since local memory references hit the local cache, the performance of the KSR1 in this case depends on the cache performance.

The average memory latency for accessing local variables decreased slightly as more processors were used due to reducing the tile size and subsequently the data size. The average memory latency for accessing remote variables, on the other hand, depends on the state of the accessed variable. Accessing a shared variable with exclusive-ownership state in a remote cache has a long average memory latency which increases slightly with the number of processors due to ring contention. Accessing a shared variable with non-exclusive ownership state in the local cache has an average memory latency of a sub-cache miss if the access is read. If the access is write, the average memory latency becomes that of a remote reference, and increases with the number of processors due to ring contention.

To reduce the long memory latencies associated with remote memory references, the *Prefetch* operation was used to access shared variables ahead of time. The benefit of *Prefetch* was much limited by the small number of pending requests each local cache may have at one time. Although the *Prefetch* operation was supposed to run in parallel with the processor to mask the long memory latency, much of this latency time showed up as processor-stall time waiting for the prefetch queue to accept a new request. To utilize the *Prefetch* operation, the prefetch requests should be spread over longer period of time so as to allow the prefetch queue to accept a new request once the processor makes one. Nevertheless, the *Prefetch* operation showed some overall improvements which was function of the number of processors and the degree of data sharing.

The *Post-store* operation was used and evaluated also. The *Post-store* instruction showed a high overhead if used to broadcast a subpage of data immediately after that subpage was computed. In such a case, the local cache is kept busy for longer periods of time, thus stalling the processor. For lower number of processors, the *Post-store* overhead is considerably more than its benefit, making *Post-store* performance worse. For higher number of processors, not all post-store broadcasts were successful, i.e. the average memory latency for referencing the broadcasted subpages was not completely eliminated. This is basically due to having higher ring traffic, which increases the probability of a local cache ignoring a broadcast due to being too busy. This *Post-store* overhead was considerably reduced when the *post-store* operation was delayed till a later stage. Thus, the performance of the *Post-store* operation was improved in both effects: the memory latency overhead was reduced when accessing the referenced variables during the post-store operation, and less number of broadcasts were ignored by the local cache.

References

[1] A. Chatterjee, J. Jin, and J. Volakis, "Application of edge-based finite elements and ABCs to 3-D scattering," *To appear in the IEEE Transactions on Antennas and Propagation*, 1993.

[2] T. H. Dunigan, "Kendall square multiprocessor: Early experiences and performance," Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, April 1992.

[3] J. R. Goodman and P. J. Woest, "The Wisconsin multicube: A new large-scale cache-coherent multiprocessor," in *Proceedings of the Fifteenth Annual International Symposium on Computer Architecture*, pp. 422–431, 1988.

[4] E. Hagersten, A. Landin, and S. Haridi, "DDM — a cache-only memory architecture," *Computer*, September 1992.

[5] *KSR1 Principles of Operation*, Kendall Square Research Corporation, 1991.

[6] *KSR1 Technical Summary*, Kendall Square Research Corporation, 1992.

[7] IEEE Technical Committee on Operating Systems, "Threads execution for portable operating systems," *draft P1003.4a/D4*, 1990.

[8] D. Windheiser, E. Boyd, E.Hao, S. Abraham, and E. Davidson, "KSR1 multiprocessor: Analysis of latency hiding techniques in a sparse solver," *The 7th International Parallel Processing Symposium*, April 1993.