A Dynamic Cache Sub-block Design to Reduce False Sharing

Murali Kadiyala and Laxmi N. Bhuyan

Department of Computer Science
Texas A&M University
College Station, Texas 77843-3112
Tel. (409)845-5007
E-mail: {kadiyala, bhuyan}@cs.tamu.edu

ABSTRACT – Parallel applications suffer from significant bus traffic due to the transfer of shared data. Large block sizes exploit locality and decrease the effective memory access time. It also has a tendency to group data together even though only a part of it is needed by any one processor. This is known as the false sharing problem. This research presents a dynamic sub-block coherence protocol which minimizes false sharing by trying to dynamically locate the point of false reference. Sharing traffic is minimized by maintaining coherence on smaller blocks (sub-blocks) which are truly shared, whereas larger blocks are used as the basic units of transfer. Larger blocks exploit locality while coherence is maintained on sub-blocks which minimize bus traffic due to shared misses. The simulation results indicate that the dynamic sub-block protocol reduces the false sharing misses by 20 to 40 percent over the fixed sub-block scheme.

Keywords: cache memory, cache coherence protocols, shared memory mutiprocessor, false sharing, sub-block design

* This research has been partly supported by NSF grant MIP-9301959

A Dynamic Cache Sub-block Design to Reduce False Sharing

Abstract

Parallel applications suffer from significant bus traffic due to the transfer of shared data. Large block sizes exploit locality and decrease the effective memory access time. It also has a tendency to group data together even though only a part of it is needed by any one processor. This is known as the false sharing problem. This research presents a dynamic sub-block coherence protocol which minimizes false sharing by trying to dynamically locate the point of false reference. Sharing traffic is minimized by maintaining coherence on smaller blocks (sub-blocks) which are truly shared, whereas larger blocks are used as the basic units of transfer. Larger blocks exploit locality while coherence is maintained on sub-blocks which minimize bus traffic due to shared misses. The simulation results indicate that the dynamic sub-block protocol reduces the false sharing misses by 20 to 30 percent over the fixed sub-block scheme.

1 Introduction

Block size has a significant effect on the performance of a shared memory multiprocessor system. Parallel programs exhibit varied sharing patterns and incur substantially higher miss ratios and bus utilization than comparable uniprocessor programs. A cache block can be shared by several processors either on a per-processor basis or be finely shared by several processors. Programs are said to exhibit per-processor locality when a processor makes multiple writes to a block without being interrupted by other processors. When processors contend for one or more words within the block, fine grain sharing is said to exist.

The performance of coherence protocols is largely dependent on the sharing behavior of parallel programs. Coherence schemes on larger blocks perform better when memory referencing patterns exhibit *per-processor* and *spatial locality* while smaller blocks are preferred for applications that exhibit false sharing. If the size of the cache block is smaller than the data object, a single reference to the data object will cause multiple cache and invalidation misses. For such situations increasing the block size will, in general, reduce the number of cache and invalidation misses. If the size of the cache block is larger than the size of the data object then two or more data objects which are not shared may occupy the same cache line. This is known as the *false sharing problem*. If two processors modify the two data objects independently at the same time, additional invalidation misses are created. Since the size of the data objects and sharing patterns can vary widely for different applications, a single fixed block size does not perform satisfactorily for all applications.

Eggers and Katz [1] have examined the effect of cache block size on cache and bus performance using traces of four application programs. They showed that increasing the block size decreased the miss ratio for applications that show larger per-processor locality while increasing it for others that exhibit a large degree of false sharing. Using the *Berkeley* protocol for maintaining coherence, they show that increasing the block size increases the bus traffic by as much as 56% to 94% for applications that have false sharing and decreases by 40% to 60% for applications with large spatial and processor locality. A similar variation in invalidation patterns with block size is also observed in directory based coherence schemes [2]. In some cases, larger blocks increased the number of invalidations, while in others they reduced it, as was observed in the case of the shared bus. These variations are also attributed to the different sharing patterns. Simulation of three parallel applications for the VMP multiprocessor [3] also show that no single block size is suitable for all applications. These studies indicate that sharing patterns largely influence the best block size for a given application.

Ideally we want to benefit from the pre-fetching effects of larger blocks and at the same time minimize the invalidation traffic due to false sharing. Previous approaches for solving the false sharing problem have focused on matching the data objects with the cache block size. Compiler technique like padding [4] is used to include redundant bytes to align object sizes with block boundaries. Several program restructuring techniques, which utilize the knowledge of the cache organization, achieve significant improvement in the cache miss ratio for large block sizes [5].

For situations that exhibit fine-grain sharing, grouping objects according to sharing patterns [4] and properly allocating objects to separate cache lines [6] show substantial improvement in cache miss ratios. Hardware schemes that minimize false sharing miss rates do so by adjusting the effective block size or by using large address blocks and smaller transfer blocks. Dubnicki and LeBlanc [7] propose a scheme in which the block size is dynamically changed based on the sharing behavior. A block is split in half when only one half is used by one processor and merged if two adjacent cache blocks are accessed. A counter associated with each line determines if a block is to be merged or split.

Chen and Dubois have investigated the effect of cache protocols with partial block invalidations [8]. In their protocol (C & D), the address block is divided into smaller sub-blocks which are the units of invalidation. A state bit is used with each cache block and a valid bit *V bit* is associated with each sub-block. The *V* bit along with the state bit keeps track of the valid sub-blocks in a cache block. In their scheme, invalidations affect only parts of the cache block allowing reads on valid sub-blocks to be completed without delay. For applications that exhibit fine grain sharing, they show that partial block invalidation scheme provides significant improvement in memory traffic and miss rates over complete block schemes.

The sub-block coherence scheme, in general, is an excellent idea that reduces the false sharing to a large extent [8]. The main limitations of the fixed sub-block scheme is that the extent of false sharing is still limited by the size of the invalidation blocks. False sharing can always exist in any fixed size sub-block greater than one byte. While such techniques have the effect of minimizing invalidations traffic when accessing valid parts of the block, it also has the tendency to break up truly shared cache blocks. This has a performance degradation since reads or writes to shared blocks that are split across several sub-blocks will cause additional cache misses and hence more cache transfers. The choice for the sub-block size becomes an important parameter while designing the cache. Different applications will perform better for different sub-block sizes and no one sub-block size will perform well for all applications.

We propose an architecture that takes advantage of maintaining coherence on sub-blocks which are truly shared. The sub-block sizes are dynamically determined with the objective of maximizing the size of a truly shared sub-block. By dividing the cache block into as many true sub-blocks as needed and maintaining coherence on true sub-blocks, we minimize the invalidation misses due to false partitioning of true shared blocks. We also reduce false sharing misses which can occur within each sub-block in any fixed sub-block scheme. The reduction in false sharing miss rate occurs because any fixed sized sub-block cannot prevent false references within each sub-block unless the size of the sub-block is one.

In our protocol, since sub-block sizes are determined dynamically, it will settle to a value which is most suitable for the application being run.

This paper is organized as follows. In section 2, we present the overall architecture and the new cache organization. In section 3, we describe the detailed protocol for the dynamic sub-block scheme. Section 4 discusses the simulation methodology and the benchmark applications used to evaluate our proposed scheme in comparison to the fixed sub-block and the complete block scheme. Results and comparisons are presented in section 5 and and the conclusions are given in section 6.

2 The New Cache Organization

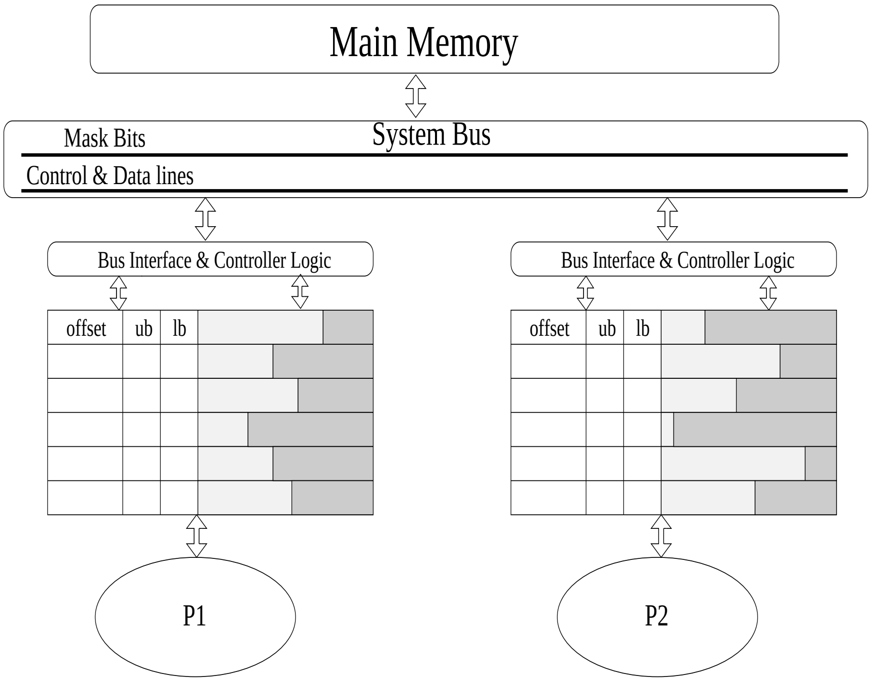

Figure 1: New Architecture for the Dynamic Sub-block Coherence Scheme.

Figure 1 shows the overall cache architecture and the modified bus structure. *P1* and *P2* are two processors with their own private caches. The caches are assumed to be a write back-write invalidate cache with a simple fetch on miss policy. Each row in the cache corresponds to a cache entry. Only those cache fields, which are not present in the conventional caches are shown explicitly in each cache entry. P1 and P2 access the shared memory through a shared bus. The bus consists of **mask** bits in addition to the usual data and control lines. These mask lines broadcast the offset information needed for maintaining coherence. Each cache block shown in Figure 1 is divided into two sub-blocks while in general it can be partitioned into as many sub-blocks as desired. Sub-block sizes can vary anywhere between zero and the block size. The **offset** field associated with each cache line maintains the byte location at which the cache block is partitioned. To allow the partition at any byte offset in the block we need $\log_2 B$ bits for the *offset* field, where B is the block size. In Figure 1, the lightly shaded portions of the cache block represent the physical extent of the upper sub-blocks and the darkly shaded portions represent the lower sub-blocks. Each cache line has two additional fields, **ub** and **lb** for storing the state of the upper and lower sub-blocks respectively. The size of the state field is dependent on the number of bits needed for the base protocol and is usually two bits. The exact use of these cache fields is presented later while describing the protocol actions in detail. Next we present a brief overview of our proposed protocol, its expected performance and its generality that makes it attractive to be used with existing protocols.

In our cache design we dynamically locate the point of false reference and partition the block into smaller sub-blocks based on local and remote cache references. To start with, we assume that each sub-block is half the size of the complete block and that each of them is truly shared. The success of our protocol lies in the ability to quickly partition the block into truly shared sub-blocks. In our scheme, a sub-block size is changed only if a processor modifies a sub-block and also at a later time wishes to modify a part of the other sub-block. In such cases the size of the sub-block is changed so as to include the modified part of the other sub-block. This will cluster together all the data that is truly shared. Since reads to shared blocks are to be maximized, and will not cause additional invalidation traffic, we do not change sub-block sizes on shared reads.

The proposed scheme is very general and can be adapted to any protocol. The adaptability of our scheme with the existing protocols stems from the fact that every protocol has an exclusive state which allows local processor writes to proceed without creating additional bus or invalidation traffic. The cache block in an *exclusive* state is the only copy in the system. Using this state to modify the offset field makes our scheme attractive without causing additional coherency overheads. The situation requiring the modification of the offset value is easily identified by a processor's write to a sub-block when the other sub-block is in an exclusive state. The offset value is changed to the most recent byte modified in the block. Since both the sub-blocks are in the exclusive state, they will be the only copies in the system and hence we are free to change sub-block boundaries with no additional traffic created.

On a miss to a cache block, the owner in addition to supplying the data will also supply the offset information so that the requesting cache can update its offset field. If memory is the owner, the offset is placed at the middle of the cache block. Writes to sub-blocks when both the sub-blocks are in the exclusively owned state will require careful thought in changing sub-blocks boundaries. In such situations we cannot update the offset field based on the last write alone. In our implementation the larger sub-block is allowed to grow while the smaller is only allowed to shrink. This restriction allows the protocol to quickly settle the partition to that value which will eliminate false referencing. For programs with true sharing block size comparable to the cache block size, the proposed scheme will perform as good as protocols on complete blocks. This is because if the block is a private block or a truly shared block, the offset will take a value of zero or the block size corresponding to the lower block or the upper block being equal to the block size respectively. In the next section, we describe in detail how the DSB scheme operates with the Berkeley protocol [9]. Wherever appropriate we have pointed out the differences of our dynamic sub-block protocol (**DSB**) with that of the complete block protocol (**CB**) and the fixed sub-block protocol (**FSB**). In the following subsection, we investigate the additional hardware complexity needed to implement the proposed dynamic sub-block cache organization.

2.1 Cost Analysis

A cache block can be divided into n sub-blocks by using $n-1$ offset fields in each cache line. The number of sub-blocks that a cache block can be partitioned into is defined as the *degree of partitioning* (D_p). We define the *degree of false sharing* (D_f) as the number of data objects which are placed in the same cache line but are used by different processors. It is apparent that D_f depends on the application, the block size and on the actual sharing behavior at run time. Ideally if each sub-block corresponds to a data object referenced by any one processor then references by any remote processor to other valid objects in the same cache block will not create unnecessary invalidation traffic due to false sharing. If the size of the data object is larger than the block size then D_f is equal to one. The maximum value for D_p and D_f is limited by the block size since we cannot have more partitions than the number of words in a block and the minimum size of any data object is at least one word. To present a quantitative evaluation of our scheme with respect to fixed sub-block schemes we will make an assumption that for all applications $D_f \leq 2$. This assumption allows us to choose a value of D_p equal to two. We do this with an understanding that a large D_f for an application can be counteracted by increasing D_p. In practice we can choose a value for D_p which is comparable to the maximum of D_f for all applications and dynamically set D_p to be equal to the run time D_f for that application.

In our new cache organization we need additional storage bits in the cache directory to maintain the offset field and the state bits for each sub-block. Let B be the block size and let $D_p = n$. Also let s be the number of state bits needed to encode the states of the base protocol. For a cache size of C we will need $(\frac{C}{B}) \times ((n-1) \times \log_2 B + n \times s)$ additional bits in the cache directory as compared to the cache organization on complete blocks. If the size of the cache directory is of concern then we can limit the minimum size of the sub-block (w) to be some power of two such that $w = 2^m$. The number of additional bits needed with the restriction on the minimum sub-block size is $(\frac{C}{B}) \times ((\frac{B}{w} - 1) \times (\log_2 B - m) + \frac{B}{w} \times s)$. Choosing large values for w reduces our storage overhead substantially but making $w = \log_2 B$ is identical to the situation in which the sub-block size is equal to the line size.

The hardware complexity of the cache and memory controllers is also increased since we have to partially update the cache or memory. This can be achieved by selectively assembling the cache block with data from the bus and data already present in the local cache. A multiplexor is used for each word to select data from either the bus or the processors local cache. Updating the offset field will not need an additional level of address decoding since it can be done in parallel while accessing the data in the cache. For an efficient implementation, the protocol relies on some means of being able to freely communicate the offset information (in addition to the data) so that the cache requesting the data can appropriately mask out the irrelevant portions of the transmitted data. This is achieved by means of additional offset mask bit lines as shown in Figure. 1. We will need $\log_2 B$ additional lines without the restriction on the minimal sub-block size and $\log_2 B - m$ with the restriction. Multiplexing the offset bits with the data lines allows us to completely eliminate these additional bus lines. The price we pay is an additional bus cycle whenever we need to transfer the offset information.

3 The Cache Coherence Protocol

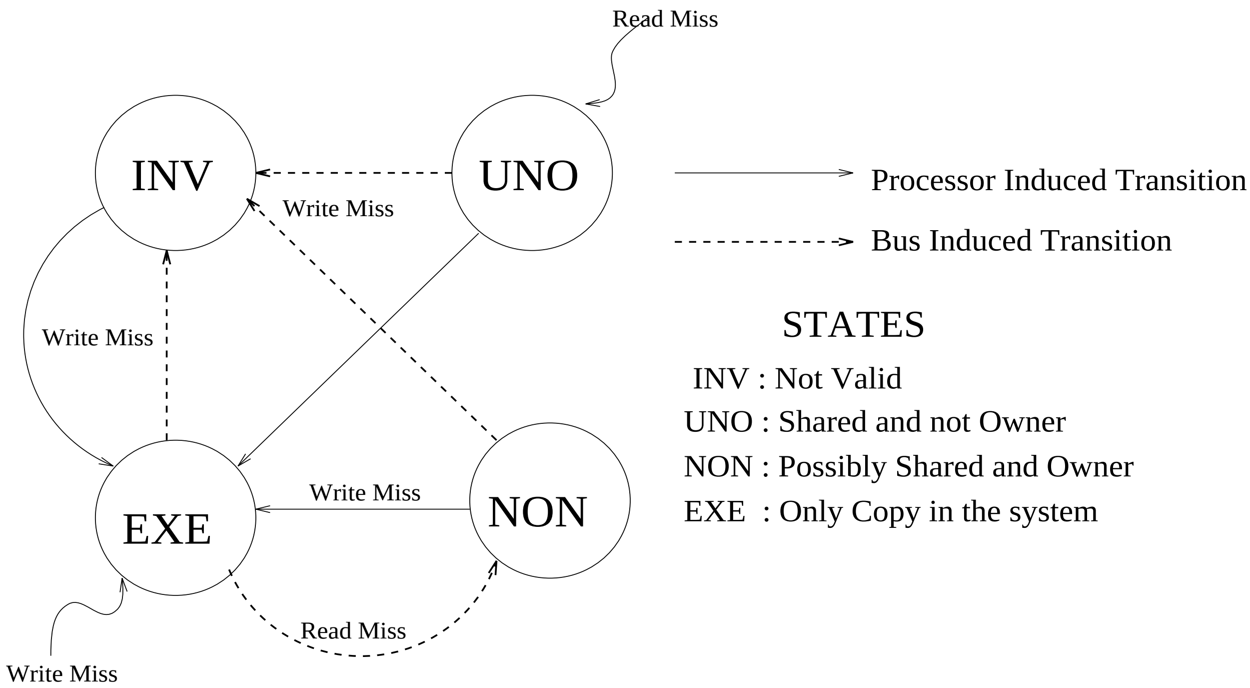

Figure 2: State Transition Diagram For the Berkeley Protocol.

Without loss of generality, we have chosen the Berkeley protocol [9] as the base protocol in this paper. The protocol is briefly reviewed and differences between the new architecture and the fixed block scheme is presented. The base protocol is represented using the conventional state transition diagram, as shown in Figure 2. The *Berkeley Protocol* is essentially an ownership protocol in which the owner of a block is responsible for supplying the block to the requesting cache or for updating the memory with the latest copy. Dirty blocks are not written back to memory when they become shared. In addition the protocol supports direct cache-cache transfers. The four states of the Berkeley protocol are the *Owned Exclusively* (EXE), *Owned Non Exclusively* (NON), *Invalid* (INV) and the *Valid* (UNO) state [10, 11]. A cache block in the state EXE means that it is the only modified copy in the system. A block in the state NON is owned by the cache and is possibly modified and shared by several processors. An INV state means that the cache block contains invalid data. A cache block in the UNO state is not owned and is possibly shared. We present

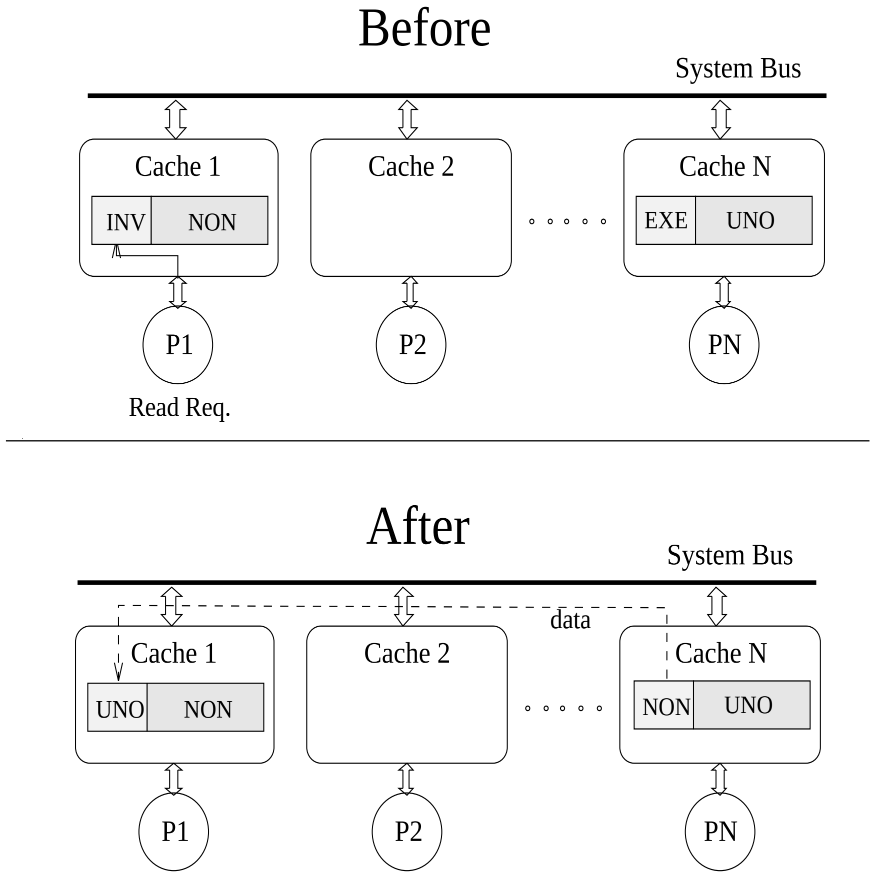

Figure 3: Read Miss : Processor Read Request.

below the read/write coherence actions with the new dynamic sub-block cache organization and compare with those in fixed sub-block organization employing the same Berkeley protocol. We show coherency actions in a system having N processors, which will be referred to as $P_1, P_2, \ldots, P_N$ respectively. The snapshot of the cache's before and after a coherence action is shown in Figures 3 through 6. Only the cache block of interest is shown in a processor's local cache. If the cache shows no blocks, then either the block under consideration is not present or is invalidated. The upper and lower sub-blocks can be differentiated by the shades assigned to them. The value of the offset field is not explicitly shown but is implied by the physical size of the upper and lower sub-blocks. The processor's request which triggered the coherence action is indicated below the corresponding processor. An arrow within each processor's cache indicates if the cache access is to the lower or upper sub-block.

- **Read Miss:** On a read miss, in the Berkeley protocol the cache with the block in state *EXE or NON* must supply the block contents directly to the requesting cache and set its local state to *NON*. If no other cache has that block then the memory provides the required block. The state of the block in the requesting cache is always set to *UNO*. The coherence actions in the new architecture are performed exactly like described above, but are executed on sub-blocks. Figure 3 shows the actions performed upon a read miss by P_1 on the upper sub-block. The lower sub-blocks in P_1 cache is in the state *NON* and P_N has the only valid copy (*EXE*) for the upper sub-block. On seeing the read address on the bus, P_N determines that the memory location is to its upper sub-block which it owns. It will take control of the bus and supply the data. The state of the upper sub-block is changed to *NON* in P_N cache and to state *UNO* in P_1's cache. Using the information in its offset field, P_1 will mask out the irrelevant data and update only the upper sub-block. If the block was completely invalid then it also uses the offset bit lines (Figure 1) to update the offset field in its local cache. If memory supplies the cache block the offset is set to half the block size provided that no other cache block has a valid sub-block of the same cache block. The bus-arbitration logic enables only one of the several caches with valid copy of the sub-block to force the offset information which is used to update the offset field in the requesting cache. In the fixed sub-block scheme, we do not have the offset lines and the value of the offset line is implied to be at the center. Every other action in the FSB scheme is similar to that in the DSB scheme.

- **Write Miss:** Like a read miss, the block comes directly from the owner. All other caches with copies change the state to *INV* and the block in the requesting cache is loaded in state *EXE*. Figure 4 shows a cache block in P_1's cache with the state of the upper and lower sub-blocks being *UNO* and *INV* respectively. P2 is the owner of the lower sub-block with the state being *NON*. A write miss to the lower sub-block by P_1 will require P_2 to supply the cache sub-block and P_3 to invalidate its lower sub-block. The state of the lower sub-block in P_2 is set to *INV* and in P_1 to *EXE*

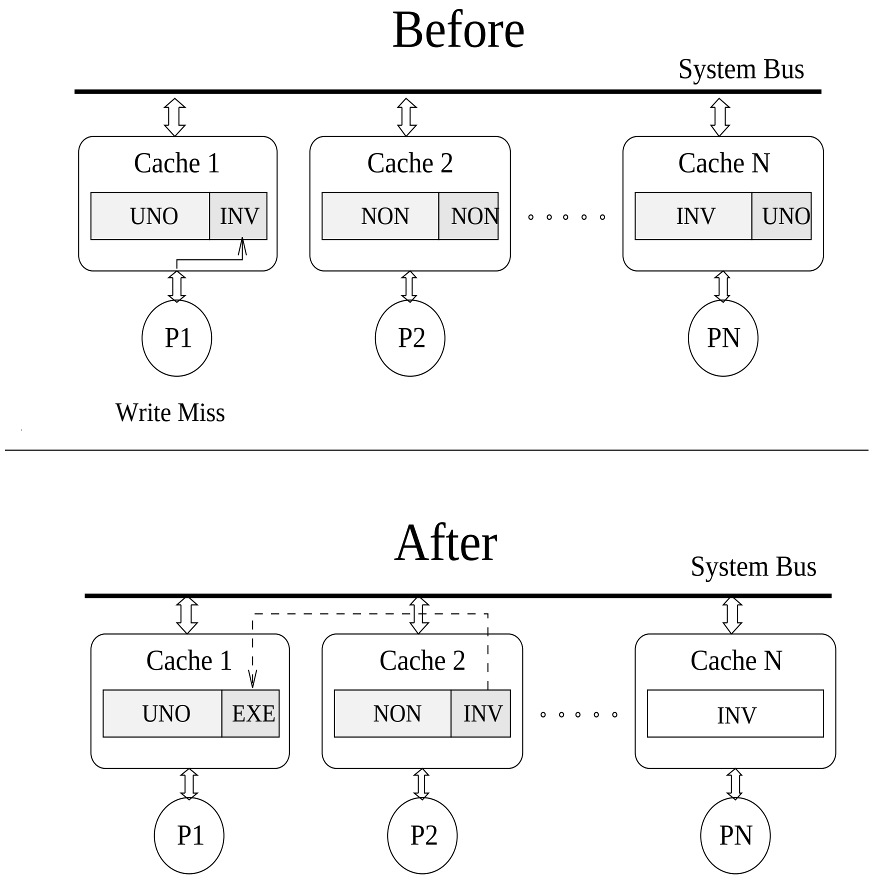

Figure 4: Write Miss : Processor Write Request.

respectively. Since P_2 has a valid upper sub-block, the offset field is still valid. The write miss invalidates the lower sub-block resulting in both sub-blocks being invalid. This situation corresponds to the complete block being invalid. The offset mask bits are not used since both P_1 and P_2 have the offset information. The offset information would have been used if both the sub-blocks of P_1 were in state INV. The offset field is not needed since the protocol guarantees that the offset field across all caches will be the same for a given cache block. This is true if at least one sub-block is not in state INV.

- **Write Hit:** In the Berkeley scheme, writes to blocks in state *EXE* can proceed without delay. If the block is *UNO or NON*, an invalidation signal must be sent on the bus before the write is allowed to proceed. All other caches invalidate their

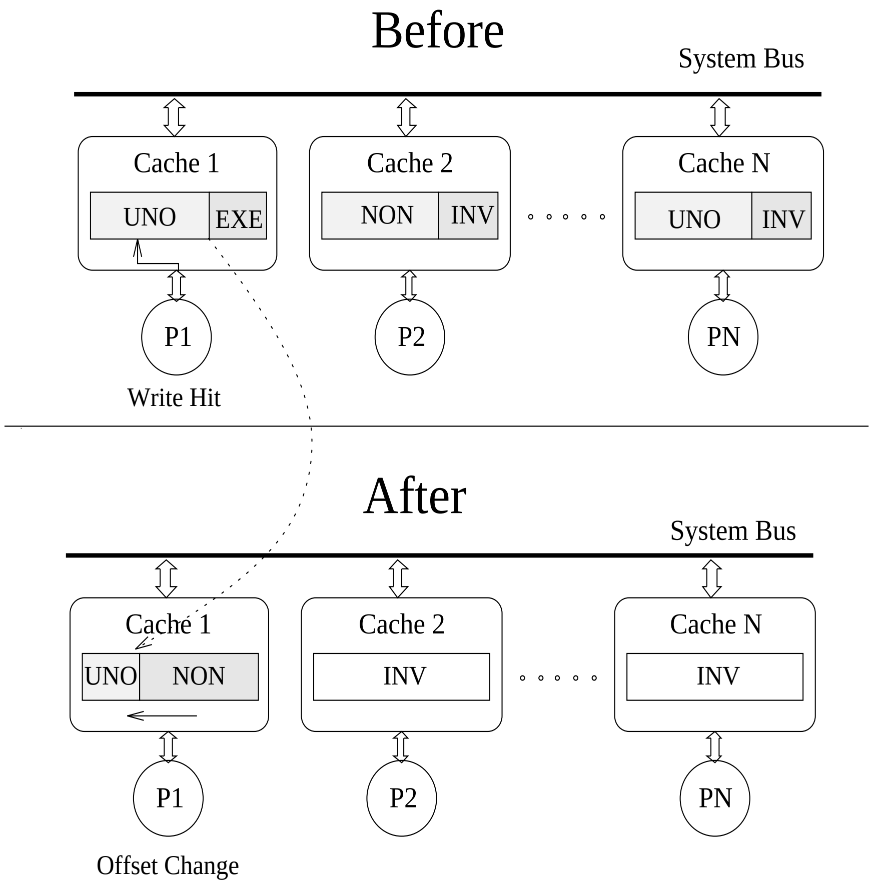

Figure 5: Write Hit : Changing the Offset Field.

copies upon matching the block address and the local state is changed to *EXE* in the requesting cache. In the dynamic sub-block scheme, writes to sub-blocks in the state *UNO or NON* will also need to send an invalidation for that sub-block. Other caches can easily determine which sub-block to invalidate by looking at the referenced address and comparing it with their offset value. Figure 5 shows P_1 generating a write reference to the upper sub-block in state *UNO*. This will require P_1 to send an invalidation signal to invalidate the corresponding sub-blocks in the cache's of P_2 and P_N. The state of the sub-block in P_1 is changed to *EXE*. Since both the sub-blocks in P_1's cache are in state *EXE*, it is clear that it is the only valid copy in the system. Also since P_1 had previously modified the lower sub-block and at a later time has modified a part of the upper sub-block, we change the size of the sub-blocks such that the modified part of the sub-block is also included with the lower sub-block. This is

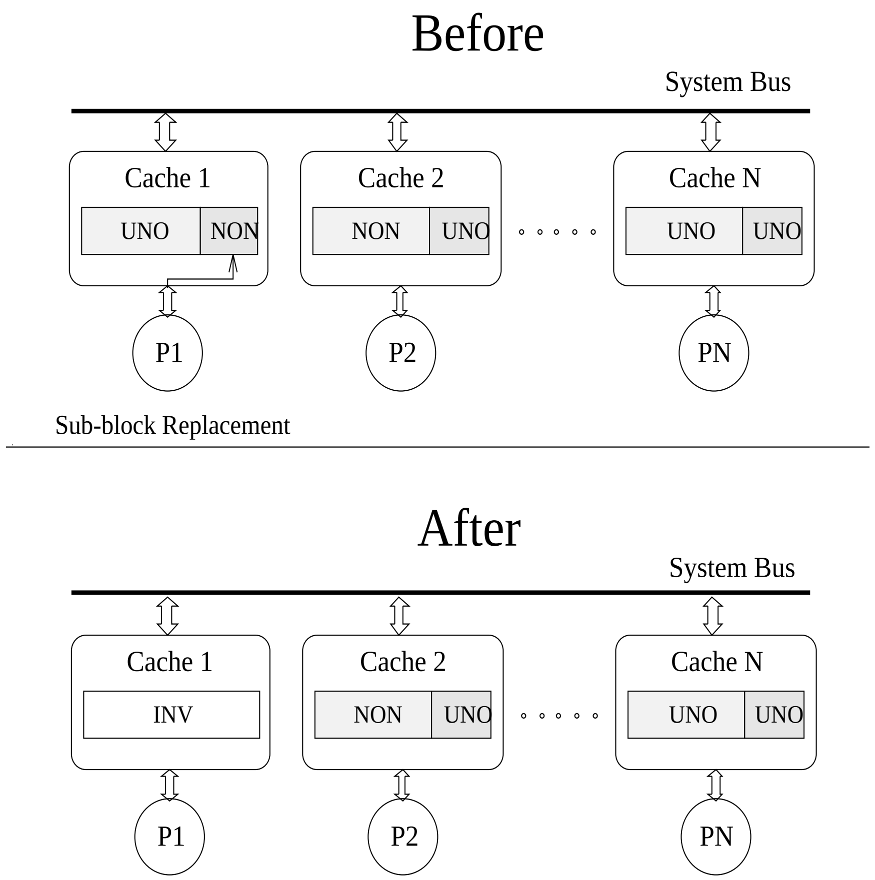

Figure 6: Sub-block Replacement.

the only time we change the offset field. We do not need to communicate this new offset information since this is the only valid copy in the system.

- **Sub-block Replacement:** Replacement in the Berkeley protocol is fairly straight forward since the owner is responsible for correctly writing back to the memory. If the state of the block is either *UNO* or *INV* no replacement is needed. Blocks in state *EXE* and *NON* can be replaced without creating any invalidation traffic. The replacement policy in the offset scheme is more complex since the block ownership is split across caches. In such situations a single block replacement is converted as a replacement to the individual sub-blocks. Treating a block replacement as two sub-block replacements provides a unified view of the DSB protocol with the base protocol. In terms of performance implications we do not lose any thing since under all conditions we will need to write back the block only once. The main disadvantage of such a replacement scheme is that the ownership can get split between the memory and the cache. Figure 6 shows P_1 wishing to replace the block in its local cache. Since the lower sub-block is owned by P_1, only the lower sub-block is updated in the main memory. If the states of both the sub-blocks are owned then we can write the entire block to the main memory and the state of the cache block in P_1's cache is set to invalid.

4 Simulation Methodology

We used a modified version of *Proteus* [12], which is an execution driven simulator for our analysis. The simulator was changed so that coherence actions are maintained using the Berkeley protocol as the base protocol. The protocol was modified according to the dynamic sub-block scheme as discussed in section 2.

The simulator can be easily configured to simulate a wide range of MIMD architectures on both bus and network based machines. In this paper, some parallel applications were simulated on a shared memory architecture with the bus as the interconnection network. Upto sixteen processors are connected to the bus with a memory access latency of four cycles and a cache to cache transfer taking two cycles. The cache is a fully associative cache of size 16K with varying block sizes. Cache reads, writes and invalidations take one cycle each.

Proteus provides accurate information about the timing and behavior of the parallel applications. Parallel execution of a program in several processors generates several events such as memory access events (both cache and main memory), network access events, execution events, interrupt request events, etc. The applications are written using a simple superset of *C* and a set of simulator calls. The proteus kernel combines the user application with user specified architecture to create the simulator which simulates on the generated events.

We classify misses as *false sharing* misses and other misses. Other misses include cold misses and true misses. The first time a processor references a block it causes a cold miss and every subsequent miss is either a true sharing miss or a false sharing miss. Such invalidation can be caused due to true or false sharing. We say a miss is a true sharing miss if the word referenced has been previously modified or will be modified by the current reference. All other misses are taken to be false sharing misses. In our simulation, we assume that the degree of false sharing is limited to two. This assumption is valid in general for relatively smaller block sizes but will cease to hold for larger blocks. In such situations we can allow cache block to be split into more number of sub-blocks. Under steady state conditions the offset will be located exactly at the point which will eliminate false sharing. Every reference to a sub-block is classified as true or false sharing miss depending on the information in the offset field and the state of the sub-blocks.

Let B be the block size with $B = 2^b$. The bytes in each block are numbered from 0 to $B-1$. Let C_{N_ij} be a reference to the jth byte of block N_i whose offset value is v_{N_i}. C_{N_ij} will correspond to a false sharing miss in the fixed sub-block scheme if the referenced bytes j is such that it satisfies $v_{N_i} \leq j < 2^{b-1}$ or $2^{b-1} < j \leq v_{N_i}$. False references on complete blocks are more readily obtained by identifying all cache references that access a sub-block in an owned exclusive state with the state of the other sub-block being invalid.

4.1 Benchmark Programs

In order to evaluate the effectiveness of the dynamic sub-block scheme to reduce the false sharing miss ratio, extensive simulations were done on benchmark applications using the complete block scheme, the fixed size sub-block scheme and the dynamic sub-block scheme. Two of these applications, MP3D and Water are from the SPLASH [13] benchmark suite. These applications were chosen to evaluate the performance of the dynamic sub-block schemes on applications that exhibit large amounts of true and false sharing. Floyd and Jacobi [14] were chosen to evaluate the the performance of the dynamic sub-block scheme when read sharing is large but the extent of write shared data is relatively small. A brief description of each of the programs is given below indicating the principal data structures and the general purpose of the program.

MP3D is a three dimensional particle simulator used in the study of rarefied fluid flow problems. Montecarlo simulations are used to simulate the trajectories of a collection of representative molecules. Primarily there are two large arrays of structures which account for more than 99% of the static data used by MP3D. The particle structure stores the state information for each molecule, and occupies 36 bytes per molecule. The space-cell structure stores the properties of each cell in the active space and requires 40 bytes per cell. Particle objects are finely divided among all the processors. During each iteration, all particles are moved requiring updating of the particle structure. True sharing miss is created when particles across cell boundaries collide and the cells are assigned to different processors. Collision requires updating five words (20 bytes) of the data structures of the two particles. False sharing occurs since consecutive particle objects belonging to different processors are grouped together in the same cache block. We have run MP3D with 3000 particles for 50 time steps on sixteen processor.

Floyd is an "all pairs shortest path algorithm" used for computing the shortest distance between every pair of nodes. The algorithm proceeds in steps with each step computing the distance between every pair of nodes with a new intermediate node added. The graph is a random graph of 32 nodes. The basic data structures used in the algorithm are two 2-dimensional arrays for representing the predecessor matrix and the distance matrix. An additional two dimensional array is used for restoring the computed path. The program takes inputs form a weight matrix. All data items have a size of 4 bytes. Each processor is responsible for updating a few rows of the distance matrix. The entire matrix is write sharable with each element modified only once during each iteration. During each iteration a particular row is extensively read by all the processors causing large true sharing. False sharing occurs if the writable data assigned to different processors share the same cache block.

Water is an N-body application which computes the forces and potential for a system of water molecules. Computation proceeds in steps, with each step computing the interactions between the molecules. The data structure for one molecule occupies 600 bytes and a single molecular array structure maintains information about all the molecules used in the simulation. Processors are assigned molecules which are adjacent to each other in the molecular array structure, even though the molecules may not be spatially located near each other. This is done to exploit locality, facilitate data partitioning and minimize processor sharing. True sharing misses are created when molecules which are assigned to different processors, interact with each other, requiring the corresponding processor to examine each others data structures. Even though a molecule occupies 600 bytes of data only the force and potential fields which account for 72 bytes of data are truly shared by the processors. False sharing misses are created as the size of the cache block approaches the molecule's data structure. In such situations, several fields describing the two molecules could potentially overlap within the same cache block. Water was run with 4 molecules on 4 processors. Results are presented for 100 time steps taking into consideration only the parallel portion of the execution.

Jacobi is an iterative algorithm for solving a system of partial differential equations. The program uses two 16x16 arrays of floating point numbers (4 bytes). One of the arrays contain the values computed in the previous iteration. Each element in the current array is computed as the average of the four neighboring cells of the corresponding element in the previous array. In each iteration, a processor is responsible for updating a part (4x4 array) of the current array. Processors read the relevant portions of the previous array and modify the corresponding part of the current array. After each iteration, processors synchronize through barrier synchronization and the two arrays are compared for convergence. The roles of the current and previous array are interchanged. True sharing occurs on data objects (floats) of size 4 bytes. False sharing appears mainly when a processor updates the data elements at the boundary of the data partition. For large block sizes, data that has been partitioned between two processors can potentially reside in the same block.

4.2 Performance Metrics

The metrics are primarily chosen for comparing the relative performance of the proposed protocol with respect to the fixed sub-block scheme. Three performance measures are used for the comparison. Primarily we are interested in the additional reduction in false sharing misses obtained using the DSB scheme as compared to the FSB scheme. We express this additional reduction in misses as a percentage of the miss reduction obtained using the FSB scheme. Secondly, we compare the total execution time for four applications using the CB scheme, the FSB scheme and the DSB scheme. It has been shown that the FSB scheme shows substantial improvement in miss rate and network traffic over complete block schemes [8]. As a final measure, we compare the relative improvement in execution time obtained using the DSB scheme with respect to the FSB scheme. We normalize this gain over the improvement in execution time achieved by the FSB scheme as compared the CB scheme.

The *execution time* is measured in terms of the program execution cycles as simulated by proteus. False sharing misses are computed as discussed in section 4. For ease of discussion the following notations are introduced.

N_{cb} = number of execution cycles for the complete block (CB) scheme.
N_{fsb} = number of execution cycles for the fixed sub-block (FSB) scheme.
N_{dsb} = number of execution cycles for the dynamic sub-block (DSB) scheme.
M_{cb} = number of false sharing misses in the complete block (CB) scheme.
M_{fsb} = number of false sharing misses in the fixed sub-block (FSB) scheme.
M_{dsb} = number of false sharing misses in the dynamic sub-block (DSB) scheme.

The percentage reduction in false sharing miss rate of the DSB scheme compared to the FSB scheme is given by

$$\%\ reduction\ in\ FSM = \frac{M_{fsb} - M_{dsb}}{M_{cb} - M_{fsb}} \times 100 \tag{1}$$

Equation 2 gives the performance of fixed sub-block scheme (FSB) with respect to the complete block schemes (CB). Similarly, the performance gain of the DSB scheme relative to the FSB scheme with respect to the CB scheme is given by eq. 3. For both the FSB and the DSB schemes, the improvement in program execution time is normalized with respect to the total time taken on the complete block scheme.

$$G_{fsb} = \frac{N_{cb} - N_{fsb}}{N_{cb}} \tag{2}$$

$$G_{dsb} = \frac{N_{fsb} - N_{dsb}}{N_{cb}} \tag{3}$$

The *relative improvement* in execution time of the DSB scheme over the fixed sub-scheme is obtained from eq. 2 and eq. 3 as

$$R = \frac{G_{dsb}}{G_{fsb}} * 100 \tag{4}$$

5 Results and Comparisons

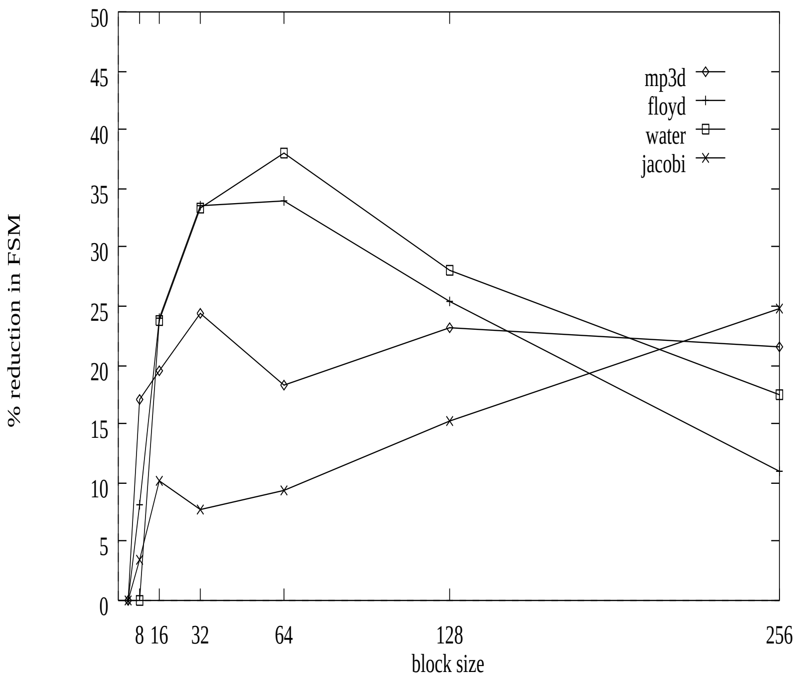

Figure 7: Percentage reduction in false sharing miss rate vs block size

In this section, we compare various experimental results for the complete block (CB), the fixed sub-block (FSB) and the new dynamic sub-block (DSB) scheme. It has been shown that sub-block coherence scheme with fixed sub-block sizes show significant improvement over coherence schemes on complete blocks [8]. This performance gain is due to reduction in the false sharing miss rate. The DSB scheme achieves better reduction in false sharing miss rate and show a better overall performance.

In Figure 7, we show the percentage reduction in false sharing miss rate in the DSB scheme compared to the FSB scheme for MP3D, Floyd, Water and Jacobi. The percentage reduction in false sharing miss rate is calculated as in eq. 1. For a block size of 4 bytes, applications like MP3D, Floyd and Jacobi show the same reduction in miss rate as compared to the FSB scheme. Since the smallest data objects modified for these applications are about two bytes, allowing one byte sub-blocks as in the DSB scheme does not reduce the false sharing miss rate further. In water the smallest shared writable data is a double (8 bytes) because of which, for a block size of 8 bytes, the DSB scheme performs as good as but not better than the FSB scheme. For the same block size we see a substantial reduction in the false sharing miss rate by 17.1% for MP3D, 8.1% for Floyd and 3.4% for Jacobi. This significant improvement is due to frequent accesses to data locks (2 bytes) in MP3D and the misalignment of 4 byte data objects with the fixed sub-block (4 bytes) in Floyd and Jacobi. The reduction in miss rate is more pronounced for increasing block sizes upto 32 bytes. It reaches a maximum of 25% for MP3D and 34% for Floyd when the block size is 32 bytes and 39% for Water when the block size is 64 bytes. MP3D shows an almost constant percentage reduction in the false sharing miss rate (25%) for block sizes greater than 32 bytes. This indicates that the DSB scheme eliminates most of the false references in the application. Water and Floyd show smaller improvements for larger block sizes. As a general trend, we observed that for very large block sizes the percentage reduction in false sharing misses decreases since false sharing occurs in either of the two sub-blocks and that a degree of partitioning of two is insufficient to eliminate all the false references. In Figure 7 we see that Jacobi shows a greater reduction in false sharing misses for block sizes upto 256 bytes since the degree of sharing is limited to two.

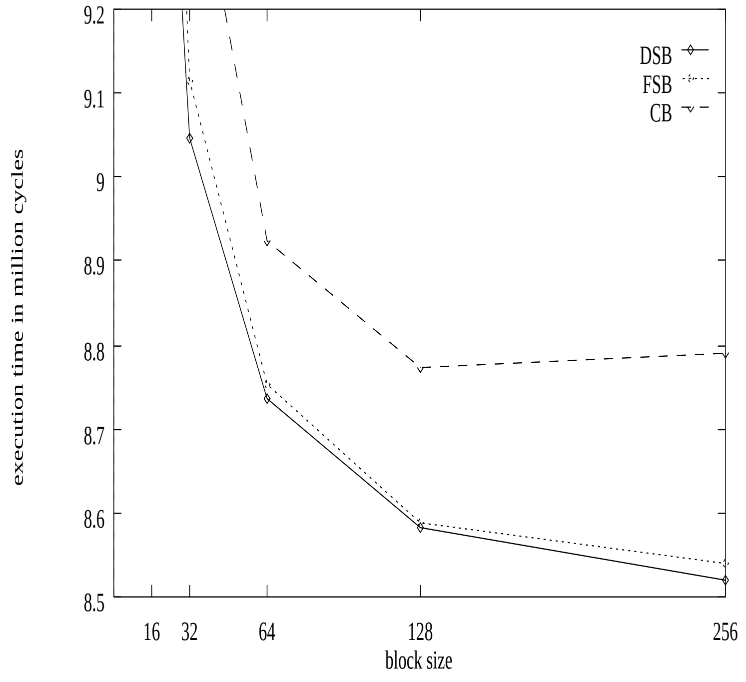

Figure 8: Proteus execution cycles vs block size for MP3D

Figures 8 through 11 shows the variation in execution time with block size for MP3D, Floyd, Jacobi and Water with the DSB, the FSB and the CB scheme. For applications like MP3D (Fig. 8), Water (Fig. 10) and Jacobi (Fig. 11) both the DSB and the FSB perform

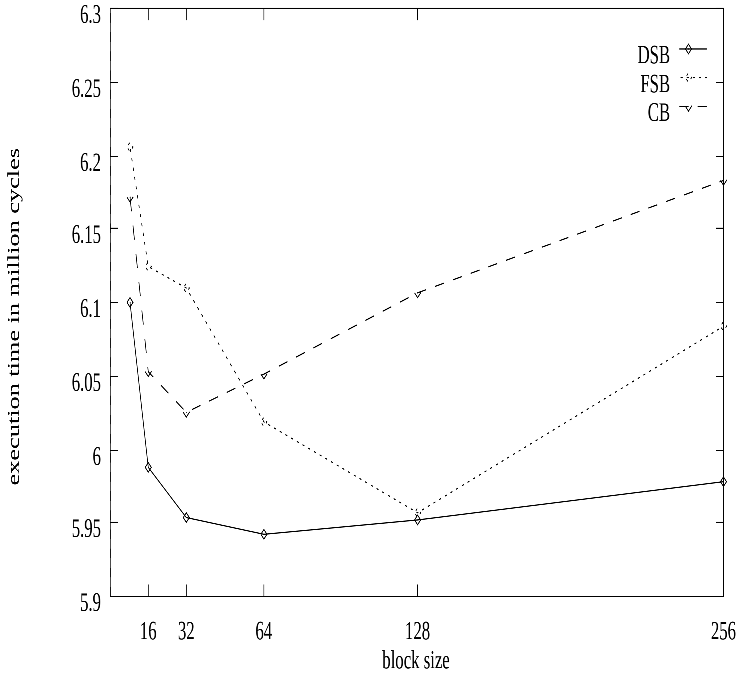

Figure 9: Proteus execution cycles vs block size for Floyd

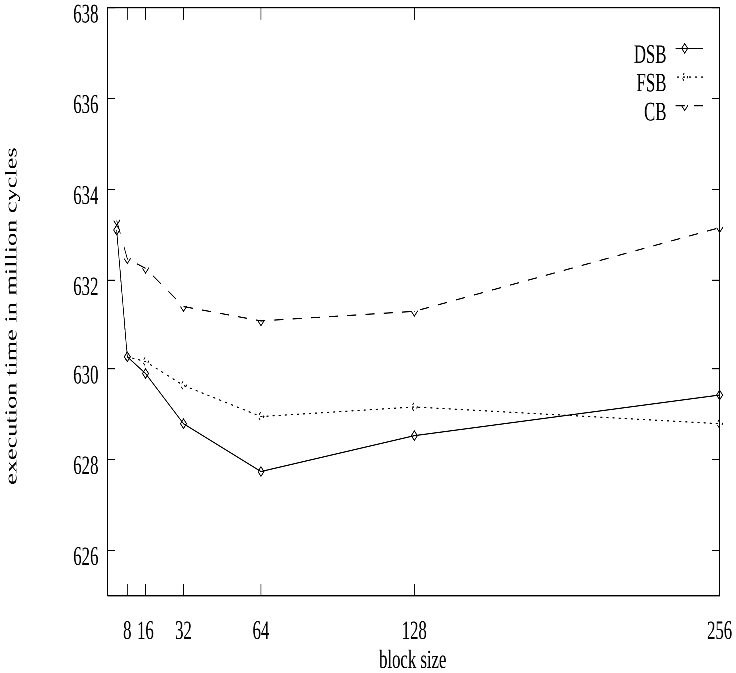

Figure 10: Proteus execution cycles vs block size for Water

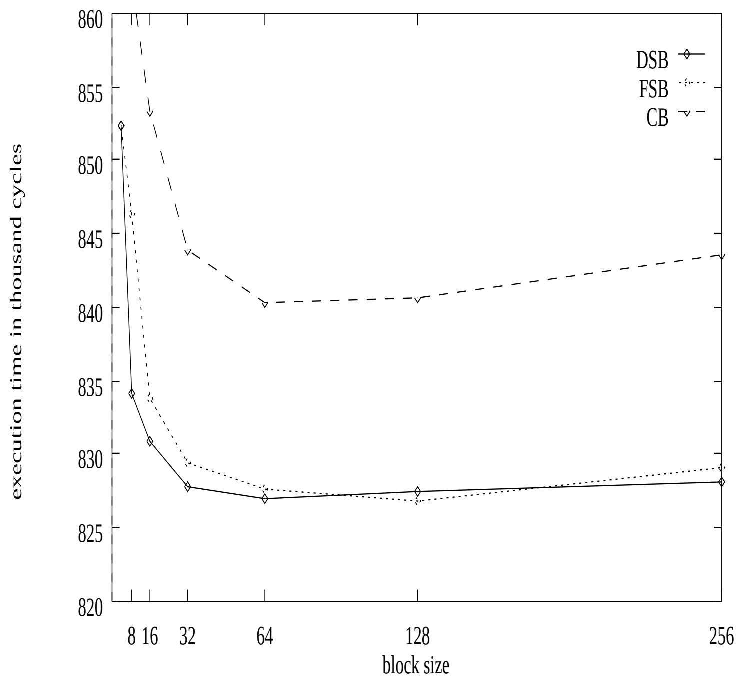

Figure 11: Proteus execution cycles vs block size for Jacobi

better than the complete block scheme for all block sizes. This is due to a large reduction in false sharing miss rate. However for Floyd (Fig. 9), the CB scheme does better than FSB scheme for block size less than 32 bytes but the DSB scheme does better than both for all block sizes. Even though the FSB scheme reduces the false sharing component for block sizes less than 32 bytes, the true sharing misses increase in the FSB scheme causing the poorer performance with respect to the CB scheme. The better performance of the DSB scheme is due to the additional 34% reduction in false sharing miss rate and the reduced true sharing misses obtained by making the sub-block size equal to the block size as in the complete block scheme. For block sizes greater than 32 bytes the decreased false sharing miss rate causes the FSB scheme to perform better than the CB scheme. For Water (Figure 10), we see that the FSB scheme does better than the DSB scheme for a block size of 256 bytes although the DSB scheme reduces the false sharing miss rate by 20%. This is due to the additional overhead in the Berkeley protocol whenever the ownership of a cache block is divided between a cache and the memory. In water, true sharing occurs on 72 bytes of data, which is accommodated in one sub-block (128 bytes). While the DSB scheme will isolate this as a true sub-block and any reference to the other sub-block will cause a cache miss

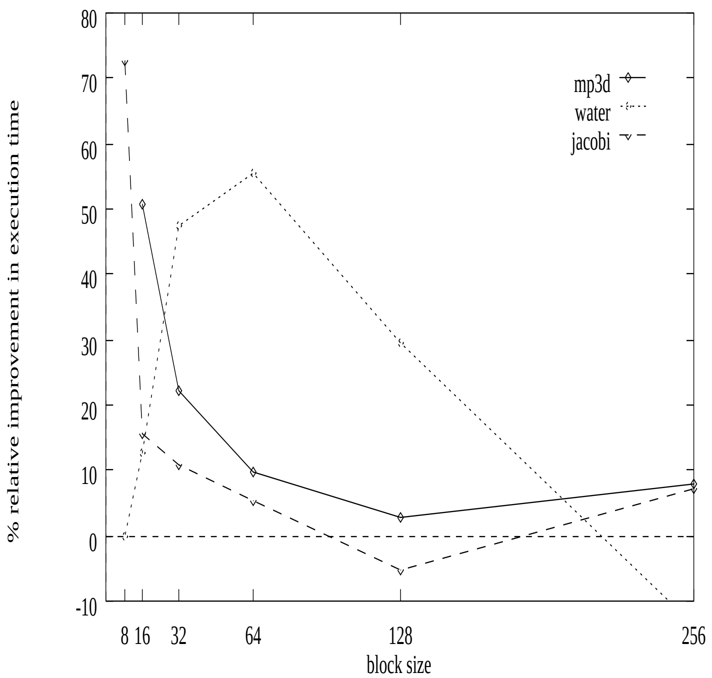

Figure 12: Relative improvement in execution time for MP3D, Water and Jacobi.

which has to be supplied from the memory causing the additional latency. For a similar reason, in Jacobi (Fig. 11) also the FSB scheme does better than the DSB for a block size of 128 bytes.

Figure 12 shows the relative improvement in execution time (eq. 4) of the DSB scheme relative to the FSB scheme for MP3D, Water and Jacobi. We get a large variation in the relative improvement (0%-300%) for Floyd and as a result we could not plot the curve for Floyd in Figure 12. With the degree of partitioning equal to two and for small block sizes, MP3D, water and Jacobi show a 60% relative improvement in execution time. As the block size increases we see a reduced gain of 10% due to the larger degree of false sharing. The FSB scheme performed marginally better in only two cases involving Jacobi and Water for a block size of 128 bytes and 256 bytes respectively. This behavior is due to the additional coherence overhead incurred by the Berkeley protocol.

6 Conclusions and Future Work

The paper presented a design for a dynamic sub-block implementation to reduce the effect of false sharing. A preliminary cost analysis shows marginal increase in the hardware complexity. While a block can be dynamically partitioned to the degree of false sharing, a more practical approach would be to divide a block into two sub-blocks and maintain the states of the sub-block individually for coherence actions. The approach was tested with different applications (MP3D, Floyd, Water and Jacobi) and the performance gain varied widely depending on the block size. A comparison was made with the complete block scheme and the fixed sub-block scheme to show that the proposed dynamic scheme performs the best for almost all the applications. A detailed hardware design of the FSB and DSB cache controllers and extensive simulations is necessary to properly evaluate the new sub-block scheme.

References

[1] Susan J. Eggers and Randy H. Katz, *"Effect of Sharing on the Cache and Bus Performance of Parallel Programs"*, ASPLOS , 1989, pp. 257-270.

[2] A. Gupta and W.D. Weber. *"Analysis of cache invalidation patterns in shared memory multiprocessors."*, In Cache and Interconnect Architecture in Multiprocessors, Kulwer Academic Publishers, 1990, pp. 83-108.

[3] D.R. Cheriton, et, al. *"The VMP Multiprocessor: Initial Experience, refinements, and performance evaluation."*, In 15th Intl. Symposium On Computer Architecture, June 1988, pp. 410-421.

[4] S.J.Eggers and T.E.Jeremaissen *"Eliminating False Sharing."*, Proc. of the 1991 Int. Conf. on Parallel Processing, pp. I-377-I381, Aug. 1991.

[5] D.R.Cheriton, H.A. Goosen, and P. Machanick *"Restructuring a parallel simulation to improve cache behavior in a shared-memory multiprocessor: A first experience."*, In Intl. Symp. On Shared Memory Multiprocessing, April 1991.

[6] J. Torrellas, M.S. Lam and J.L. Hennessy, *"Shared Data Placement Optimizations to Reduce Multiprocessor Cache Misses. "*, Proc. of the 1990 Int. Conf. on Parallel Processing, pp. 266-270, August 1990.

[7] Czarek Dubnicki and Thomas LeBlanc. *"Adjustable block size coherent caches."*, In Proc. Of 19th Intl. Symposium On Computer Architecture, 1992, pp. 170-180.

[8] Yung-Syau Chen and Michel Dubois, *"Cache Protocols with Partial Block Invalidations "*, Proc. Of 7th IPPS, April 1993.

[9] R. Katz, et. al., *"Implementing a cache consistency protocol"*, Proc. Of 12th International Symposium On Computer Architecture, 1985, pp. 276 -283.

[10] James Archibald and Jean-Loup Baer, *" Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model "*, ACM Transactions on Computer Systems, vol 4, No. 4, November 1986, pp 273-298.

[11] Qing Yang , Laxmi N. Bhuyan and Bao-Chyn Liu, *"Analysis and Comparison of Cache Coherence Protocols for a Packet-Switched Multiprocessor"*, IEEE Transactions On Computers, vol. 38, no. 8, August 1989, pp. 1143-1153.

[12] E.A Brewer,et. al., PROTEUS: *"A High-Performance Parallel-Architecture Simulator"*. MIT Technical Report MIT/ICS/TR-516. September 1991.

[13] J.P.Singh, W-D. Weber and A. Gupta *"SPLASH: Stanford Parallel Application for Shared Memory. "*, Technical Report, Dept. Of Computer Science, Stanford University, April 1991.

[14] D. Bertsekas and J. Tsitsiklis, *"Parallel and Distributed Computations: Numerical Methods"*, Prentice Hall, 1989.