To Appear in 2000 SCS Symposium on Performance Evaluation of Computer and Telecommunication Systems, July 16-20, 2000, Vancouver, BC, Canada

The Effects of Granularity and Scheduling Policies on the Performance of Data Parallel Programs

Inbum Jung, Kyoungchul Kim, and Joonwon Lee

Department of Computer Science,
Korea Advanced Institute of Science and Technology
373-1, Kusong-Dong, Yusung-Gu, Taejon 305-701, Korea
e-mail : {jib,ddaeng7,joon}@calab.kaist.ac.kr

Partial List of Frequently Used Keywords: Grain Size, Grain Scheduling, Cache Miss, Synchronization.

Abstract

The performance of data parallel programs suffers from the latencies caused by the memory access and synchronization. These latencies are affected by the grain size and the scheduling policy for the grains since they influence the memory reference patterns and the time required for synchronization among processors. In this paper, to explore the causes of these latencies, data parallel programs are experimenting with varying their grain sizes under two scheduling policies through execution-driven simulations. From this simulation, we found the best grain size and scheduling policy for each simulated parallel programs and analyze the causes inducing those results[1].

1 Introduction

Shared memory multiprocessors are often used for the execution of parallel programs. Each processor usually accesses the main memory via its cache memories to fetch the data, and the workload of parallel programs is partitioned into many grains based on the grain size chosen by programmers. However, the grain sizes that ignore caching effects may degrade the performance since the address interferences among the partitioned grains increase cache misses. The scheduling of grains onto processors also affects cache performance since they exhibit a wide variety of memory reference patterns.

Many parallel programs are usually written using synchronization primitives. In particular, the barrier and spin-lock primitives are commonly used for synchronization among processes. The amount of time spent at barriers depends on the grain size and caching behavior of the partitioned grains. The waiting time of spin-lock can be decomposed into the time for lock contentions and the time to maintain the cache coherence for the synchronization variable. Scheduling policies of grains also affects the spin-lock waiting time since the spin-locks are needed for handling the grain queues.

In this paper, the interaction between grain sizes and the scheduling policies of grains is examined in data parallel programs. The simulation results show that the grain sizes and the scheduling policies selected by programmers have impacts on the cache behavior and the synchronization latency in tested parallel programs. On the basis of these simulation studies, we suggest the best grain size and scheduling policy for each parallel program, and analyze the causes inducing those results.

The paper is structured as follows. Section 2 describes benchmark data parallel programs used in this study and their grain sizes and scheduling policies of grains. Section 3 presents the simulation environment used in our study. In Section 4 we measure the performance of our tested parallel programs under various grain sizes and scheduling policies, and analyze their results. In Section 5 related work is presented. Finally, the conclusion is presented in Section 6.

2 Grain Size and Scheduling Policies on Data Parallel Programs

The granularity or grain size determines the basic program segment chosen for parallel processing. Grain sizes are commonly described as fine, medium, or coarse, depending on the computation amount involved. In particular, data parallel programs operate on large data items and perform identical processing on data. Since data elements are subject to identical processing, such programs are parallelized by assigning data elements to processors. Thus, a grain size means the portion of data items and occupies the contiguous blocks of memory as much as its size. Furthermore, since parallel loops carry the reuse of the grains allocated to each processor, cache locality is important to achieve good performance on these programs.

In this paper, we choose several grain sizes to maintain the load balance in benchmark programs. Thus, the coarsest

[1]This work was supported in part by National Research Laboratory Program funded by Ministry of Science and Technology and university S/W research center program by Ministry of Information and Communication, Republic of Korea

grain size is found by dividing the workload of a program by the number of available processors. Other grain sizes including the finest grain size are achieved by dividing the workload by a multiple of the number of processors.

To explore the variation of the memory access patterns introduced by scheduling policies, the partitioned grains are executed by static and dynamic scheduling policies between iterations of a parallel loop. The static scheduling policy designates the grains to each process before a program is executed. Since the grains allocated to each process are not changed until the program finishes, cache locality can be fully exploited. On the other hand, the dynamic scheduling policy performs scheduling activities to the grains at runtime. As soon as a process completes the computation of a grain, the process begins executing the next grain in the grain queue. The dynamic scheduling improves the processor utilization because a program uses the available processors released by other processes during its execution; however, it results in scheduling overheads to handle the grain queue and the loss of cache locality. The grain sizes chosen and scheduling policies respectively induce the spatial and time variations to the memory reference pattern of data parallel programs. These combined variations have impacts on the cache behavior and synchronization operations of parallel programs.

The three data parallel programs for this study are BMM (Blocked Matrix Multiplication), FFT(Fast Fourier Transform), LU(LU Decomposition) [1]. These programs show data parallelism, since they perform identical operations on all data elements and these elements are assigned to various processors to parallelize the computation. In particular, since these programs do not generate new data elements dynamically during the execution, they are easy to evaluate the performance variations according to the decided grain sizes. All these programs are written in C and use the synchronization and sharing primitives provided by the SGI's parallel macros package. All programs are run with eight processes on eight processors.

2.1 Blocked Matrix Multiplication(BMM)

The BMM is a matrix multiplication based on a blocked algorithm, which is a well-known optimal technique for improving the temporal locality.

```
1: Procedure BMM(X, Y, Z, N, B)  {
2:     for(kk = 0; kk < N ; kk += B){
3:         for(jj = 0; jj < N ; jj += B){
4:             for(i = 0; i < N ; i++){
5:                 for(k = kk; k < min(kk+B, N) ; k++){
6:                     r = X[i][k];        /* register allocated */
7:                     for(j = jj; j < min(jj+B, N) ; j++)
8:                         Z[i][j] += r * Y[k][j];
9:                 }
10:            }
11:        }
12:        barrier;      /* only in the dynamic scheduling policy */
13:    }
14: }
```

Example 1: Blocked Matrix Multiplication Algorithm

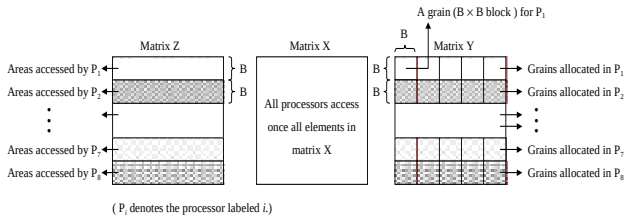


Figure 1: Data Partitioning and Distribution of the three arrays X, Y, and Z

Primary data structures are three two-dimensional arrays that execute the matrix multiplication. The Example 1 is the BMM program code. To multiply two matrices X and Y to produce Z, a $B \times B$ block of matrix Y is repeatedly used to generate the corresponding data of matrix Z. When all computation associated with the current block of matrix Y is finished, the next block is loaded and used similarly. To parallelized this BMM program, the block size B is to be used as a grain size for parallel processing. Thus, $B \times B$ blocks in matrix Y are assigned to each processor on the line 3.

Figure 1 shows that when the static scheduling policy is used, the memory reference patterns for the three matrices used and the grains partitioned into eight processors. All $B \times B$ blocks on the matrix Y are executed in parallel. This Figure shows that each processor accesses its fixed location on the two matrices Y and Z during the execution. In particular, the elements of two matrices Y and Z are reused between the iterations. On the other hand, the elements of the matrix X are accessed once by all processors during the execution. Thus, the reuse for the matrices Y and Z play a significant role in improving cache performance via cache locality.

On the other hand, when the dynamic scheduling policy is used, each processor does not access its fixed areas like the static scheduling policy. When each processor completes a block, it begins the next block in the grain queue. Due to this characteristic, the processor utilization increases since the processors released back to the system immediately participate in the remaining work. However, due to data dependence, the barrier primitive should be used whenever a parallel loop is iterated. Spin lock primitive is allowed to be used for mutual exclusion to the grain queue.

For our experiments, we use matrices of 256×256 elements(1.5 Mbytes in size). The coarsest grain size is a 32×32 block because the rows of the matrix Y are divided by eight processors. The finest grain size is a 2×2 block and other grain sizes considered are a 4×4 block, a 8×8 block and a 16×16 block. These grain sizes balance the loads among processors, since they provide the same number of blocks with eight processors.

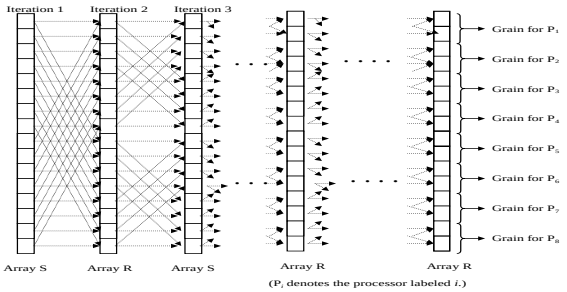


Figure 2: A snapshot of the FFT program on parallel processing

2.2 Fast Fourier Transform(FFT)

The FFT program we have used here is a classic iterative Cooley-Tukey algorithm for an n point; one-dimensional, unordered and radix-2 FFT. This program performs log$n$ iterations of the most outer loop. Each iteration does n complex multiplications and additions. Primary data structures are two one-dimensional arrays composed of both a source point array and a result point array. The next Example 2 is the FFT program code. The outer loop starting at line 3 is executed log$n$ times for an n point FFT, and the inner loop starting at line 7 is executed n times during each iteration of the outer loop. In every iteration of the outer loop, the array R is updated using the elements that were stored in the array S. Line 10 performs a crucial step in the FFT program. This step updates $R[i]$ by using $S[j]$ and $S[k]$, and also computes the powers of w known as *twiddle factors*. Line 7 is the right position for parallel processing. In this line, the array R is allocated to each processor according to the index i's stride as much as a grain size.

```
1: Procedure FFT(R, S, n)  {
2:      r = logn;
3:      for(m = 0; m < r − 1; m++){      /* outer loop */
4:           for(i = 0; i < n − 1; i++){
5:                S[i] = R[i];
6:           }
7:           for(i = 0; i < n − 1; i+grain_size)
8:                j = (b0 . . . bm−1 0bm+1 ...br−1);
9:                k = (b0 . . . bm−1 1bm+1 ...br−1);
10:               R[i] = S[j] + S[k] × w^(bm bm−1 ...b0 00...0);
11:          }
12:          barrier;
13:     }
14: }
```

Example 2: Fast Fourier Transform using the Cooley-Turkey Algorithm

When the static scheduling policy is used, Figure 2 shows the grains partitioned into eight processors and the iteration steps of a parallel loop during the execution. As shown in this Figure, arrays S and R in turn are used as a source point array or a result point array. Since this program results in the memory access pattern based on the divide-and-conquer characteristic, half the data accessed in an iteration does not reuse in the next iteration. On the other hand, using the dynamic scheduling policy, it does not fix the areas of each processor during the execution of the program. In both scheduling policies, a barrier primitive is used between each iteration of the outermost loop. The spin-lock primitive to handle the grain queue is needed in the dynamic scheduling policy.

For our experiments, we execute FFT on 65536 input points(1 Mbytes in size). The coarsest grain size is 8092 points, which is achieved by dividing a source point array by eight processors. The finest grain size is 2 points. Other grain sizes considered are 4096, 2048, 1024, 512, 128, 64, 32, 16, 8, 4, and 2 points. All grain sizes balance the loads among all processors, based on the number of partitioned grains.

2.3 LU Decomposition

This program decomposes matrix A as the product of a lower-triangular matrix L and an upper-triangular matrix U so that $A = L \times U$. After a pivot row is computed, the remaining rows underneath are modified by this pivot row. During each iteration, the pivot row gradually moves to the bottom and the number of remaining rows underneath decreases. As the computation proceeds in the LU program, the amount of computation decreases gradually.

To maintain the load balance in this intrinsic property of the LU program, whenever each iteration of the most outer loop begins, we recalculate the grain size by dividing the number of the remaining rows underneath the pivot-row by the number of processors and a *grain divisor* defined as a 32-bit integer variable. Thus, since the number of processors is fixed in our study, the grain divisor value represents the degree of granularity used in this program. The grain divisors considered are 1, 2, 3, 4, 5, 6, 7, 8, and 9. As larger grain divisors, the finer grain sizes are applied at the remaining rows underneath the pivot-row. Thus, the finest grain size is grain divisor 9, and the coarsest grain size is grain divisor 1. For example, the LU program code is shown in the Example 3.

```
1: Procedure LU(A, n){
2:      for(k = 0 ; k < n − 1 ; k++){      /* outer loop */
3:           for(j = k+1 ; j < n − 1 ; j++)
4:                A[k, j] = A[k, j] / A[k, k];
5:           for(i = k + 1 ; i < n − 1; i + grain_size){
6:                for(j = k + 1 ; j < n − 1 ; j++)
7:                     A[i, j] = A[i, j] - A[i, k] × A[k, j];
8:           }
9:           barrier;
10:     }
11: }
```

Example 3: LU Factorization Algorithm

The main data structure is a two-dimensional matrix A being decomposed. For k varying from 0 to $n-1$, the LU program systematically eliminates the values of the row k from those of the rows $k+1$ to $n-1$ so that the matrix of coefficients becomes upper-triangular. As shown in this program, in the k^{th} iteration of the outer loop starting on line 2, the k^{th} row of matrix A is subtracted from each of

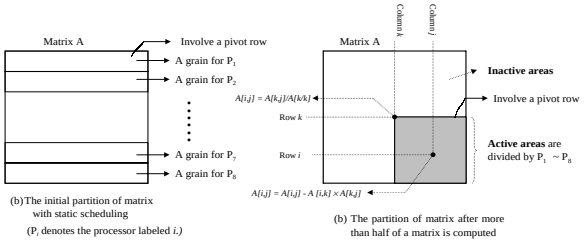


Figure 3: A snapshot of the LU program on parallel processing

the rows $k+1$ to $n-1$(loop starting on line 5). A typical computation of the LU program in the k^{th} iteration of the outer loop is shown in Figure 3(b), The k^{th} iteration of the outer loop does not involve any computation on rows 1 to $k-1$ or columns 1 to $k-1$. Thus, at this stage, only the lower-right $k \times k$ submatrix of A (the shaded portion in Figure 3(b)) is computationally active. Line 5 is the location for parallel processing. Since the remaining rows underneath the pivot row k are active areas, they are allocated to each processor according to the grain size.

The pivot row's computation executing on line 2 is a serial component in this program but the amounts of data accessed and work done per the pivot row decrease gradually. Thus, spin-lock primitive is used during the computation of a pivot row. Due to the data dependency between the outer loops, the barrier primitive is used in both static and dynamic scheduling policies. It locates on the line 9 of the Example 3.

Figure 3(a) shows the initial partition of a matrix when the static scheduling is used. Figure 3(b) shows that after more than half of a matrix is computed, only the remaining rows underneath the pivot row are active and they are divided into eight processors. As shown in these Figures, since the remaining rows under the pivot-row are decreased as the computation proceeds, the number of rows allocated into each processor is also diminished. For our experiments we run the LU program with a 256×256 matrix (512 Kbytes in size).

3 Simulation Environment

3.1 Simulated Multiprocessor

The simulation environment consists of a functional simulator that executes parallel programs and an architectural simulator that models the shared memory multiprocessor. An efficient program-driven simulator, MINT(Mips INTerpreter)[2] is used as a functional simulator. We construct an architectural simulator based on a multiprocessor with eight processors and a shared bus-based structure. Each processor is assumed to be a RISC processor with the same cache size and each instruction is executed in a single cycle except memory references.

Table 1: Timing Parameters

Events (operations)	Penalties (cycles)
A write on a shared line (The shared lines on other caches are invalidated)	3
A cache miss (A missed cache line is supplied by an another cache)	7
A cache miss (A missed cache line is supplied by the main memory)	22

3.2 Cache Parameters and Timing Considerations

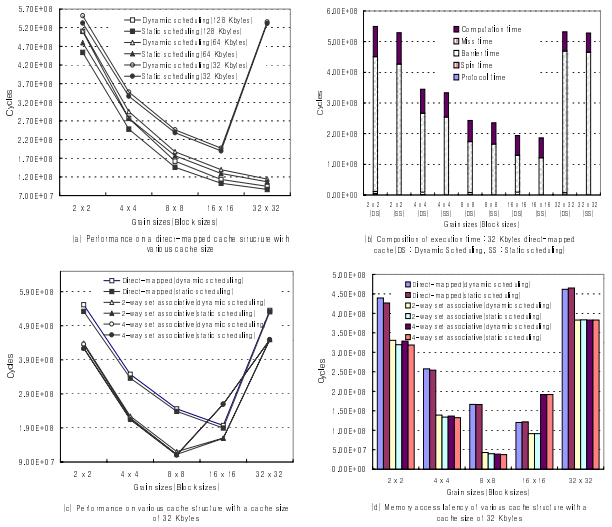
Parallel applications are executed on the various cache structures including direct-mapped, set associative caches(i.e., 2-way, and 4-way) with the LRU(Least-Recently-Used) replacement policy. The LRU policy is slightly more effective, but it is typically more expensive to implement. Set associative caches decrease cache conflict misses, but they may suffer from the cost of increased hit times. Hill [3] found about a 10% difference in hit times for direct-mapped caches versus 2-way set associative cache and a 12% difference for 4-way set associative cache. In our experiments, the cost of hit time for set associative caches is not reflected, but the variations of memory access pattern according to grain sizes can be explained by the amount of cache misses. The cache sizes are varied from 32K to 128 Kbytes with the cache line size of 16 bytes. The simulated cache coherency protocol is write invalidation scheme [4]. On current microprocessors, the main memory access-time is about 80 ns, the clock rate is 250 Mhz(e.g., MIPS R10000, UltraSparc-II) and the system bus width is 128 bits. Table 1 shows timing values used in the cache coherency protocol, based on these parameters including 1 address cycle and 1 bus operation cycle.

4 Simulation Results

In this section, the performances of parallel programs are measured across a range of grain sizes on various hardware configurations(3 kinds of cache structures, 3 kinds of cache sizes) and the causes of performance variations are analyzed. The breakdown items of the processor execution time are composed of the protocol-time, spin-time, barrier-time, miss-time and computation-time. Protocol-time is the time wasted in maintaining the cache coherence protocol. Spin-time is the busy-waiting time due to spin-lock operations. Barrier-time is the time spent waiting at the barriers. Miss-time is the time spent waiting for data to be fetched into the cache. Computation-time is the time spent doing useful work.

4.1 BMM Behavior

Figure 4(a) shows the performances of the BMM program when it is run under the dynamic and static scheduling policies across a range of grain sizes. The performances are measured based on three different cache sizes with the direct-



(a) Performance on a direct-mapped cache structure with various cache size

(b) Comparision of execution time : 32 Kbytes direct-mapped cache (DS : Dynamic Scheduling, SS : Static scheduling)

(c) Performance on various cache structure with a cache size of 32 Kbytes

(d) Memory access latency of various cache structure with a cache size of 32 Kbytes

Figure 4: BMM : 8 processors, Matrices of 256×256

mapped structure. The smaller performance differences are observed on the fine grain sizes between the static and dynamic scheduling policy using the same cache size. These differences result from using synchronization primitives for the dynamic scheduling policy.

Figure 4(b) shows the breakdown of the processor execution time achieved by the static and dynamic scheduling policy on the 32 Kbytes direct-mapped cache. The portion spent waiting for data due to cache misses is the primary cause of performance variations on the overall grain sizes. The block size to be used as a grain size results in a tradeoff between the effects of the reused data and the cache misses caused by the block's own address interference [5]. From the Figure 4-(b), the grain sizes smaller than a 16×16 block increase the cache misses due to the small amount of reusable data. On the other hand, the grain sizes larger than a 16×16 block increase the cache misses caused by the block's own address interferences. In particular, when using the dynamic scheduling policy, the difference of cache misses among processors results in higher barrier waiting time, since processors suffering more cache misses arrive late at barriers.

Figures 4(c) and (d) illustrate varying performances and memory access latencies achieved when the BMM program executes on the various cache structures (direct-mapped, 2-way set associative, 4-way set associative) with a cache size of 32 Kbytes. These Figures show the performance variations attributed to the memory access latencies, and that set associative caches represent a better performance than the direct-mapped cache under all grain sizes except 16×16. In particular, the grain size showing the best performance on set associative caches is smaller than that of the direct-mapped cache. The reasons are that not only the set size of set-associative caches affects the tradeoff between the amount of reusable data and the cache misses due to a block's own address interference, but also the replacement policy for cache lines has an impact upon cache performance. As shown in Figure 4(c), when using set associative caches, the 8×8 block results in the best performance in two set associative caches due to the smallest cache misses as represented in Figure 4(d).

4.2 FFT Behavior

Figure 5(a) shows the performance levels obtained for three different cache sizes using the direct-mapped structure and two scheduling policies across a range of grain sizes. The performance differences between the dynamic and static scheduling policy are small under all grain sizes except some fine grain sizes.

Figure 5(b) and (c) show the breakdown of the processor execution time achieved by the static and dynamic scheduling policies on the 32 Kbytes direct-mapped cache. These figures show that the miss-time occupies the primary portion of the execution time. In particular, when using the static scheduling with fine grain sizes, the portions of computation time are higher than those of other grain sizes due to the overheads for treating fine grain sizes like frequent function calls. On the other hand, when using the dynamic scheduling, the spin times on fine grain sizes increase since the dynamic scheduling policy incurs the frequent grain reallocations at run-time.

The barrier waiting times in the FFT program are affected by the differences of cache misses between the participating processors and those of the work in the grains, even if the same number of grains is allocated to each processor. When using the static scheduling policy, Figure 5(b) shows that the barrier times at fine grain sizes are higher than those at coarse grain sizes. The reason is that the use of fine grain sizes raises the possibility of cache conflict misses due to the address interference between the grains allocated to the same processor. Processors that incur few cache misses reach barriers earlier than other processors, therefore, the total barrier waiting time is increased.

However, when using the dynamic scheduling policy, the run-time support for the grain reallocation mitigates the difference of cache conflict misses among the processors. Thus, Figure 5(c) shows the small amounts of barrier waiting times under all grain sizes using the dynamic scheduling policy. On the other hand, the difference of the amount of work in the partitioned grains is due to the fact that the location of each grain within the data array affects the calculation of *twiddle factors* [1]. Thus, the processors suffering more computing time for the twiddle factors reach the barriers late. However, the barrier waiting times are not heavily influenced by computational imbalances due to the calculation of twiddle factors, since the caching effects described above place bigger impact on the total barrier waiting time.

Figure 5-(d) illustrates the varying performances when running on the various cache structures (direct-mapped, 2-way set associative, 4-way set associative) with a cache size

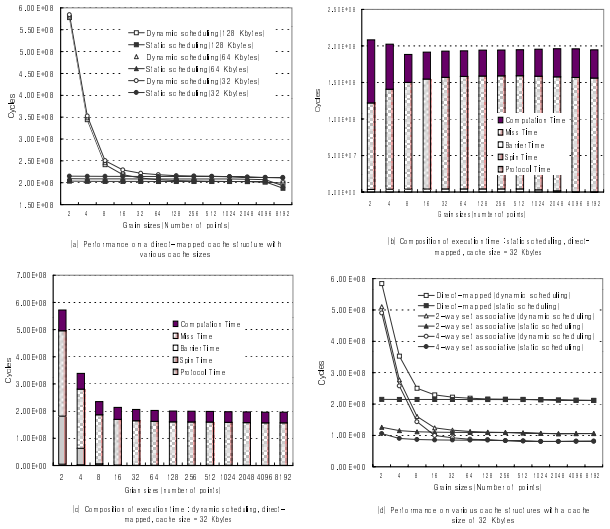


Figure 5: FFT : 8 processors, 65536 points

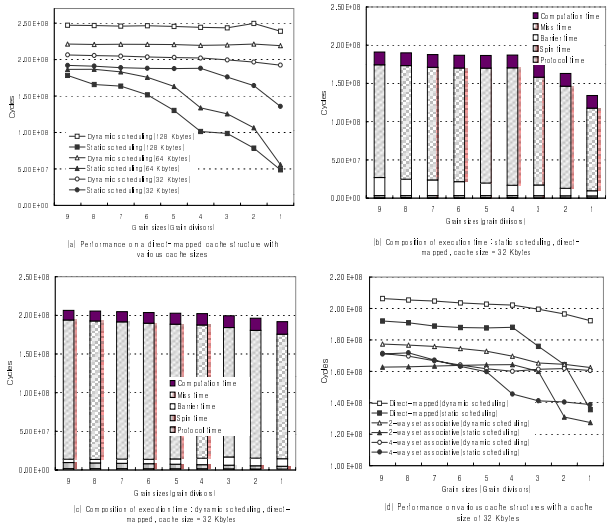


Figure 6: LU : 8 processors, a matrix of 256×256

of 32 Kbytes. The set associative caches perform better than the direct-mapped cache due to the reduction of cache conflict misses. However, since the form of the graphs induced by set associative caches is similar to that of the direct-mapped cache, the causes of the performance variations under the overall grain sizes can be analyzed by the above explanation for the direct-mapped cache.

4.3 LU Behavior

Figure 6(a) shows the performances of the LU program when it is run under the dynamic and static scheduling policies across a range of grain sizes. As described in Section 2, the larger grain divisors are used, the finer grain sizes are applied. Thus, the coarsest grain size is the grain divisor 1, and the finest grain size is the grain divisor 9. The performances are measured based on three different cache sizes with the direct-mapped structure. The performances of the static scheduling policy are better than those of the dynamic scheduling policy under all grain sizes.

Figures 6(b) and (c) show the breakdown of the processor execution time achieved by static and dynamic scheduling policies on the 32 Kbytes direct-mapped cache. These Figures show that the miss-time is given much weight in the processor execution time. These results are explained as follows. After finishing an iteration of the outmost loop, the LU program performs a synchronization operation at a barrier and moves the current location of the pivot row to the bottom as much as a row. Since the remaining rows underneath the pivot row are reallocated into each processor, using the static scheduling policy, each processor has one different row within its grains when compared with the grains used in the previous iteration. The finer grain size becomes, the more grains are allocated to each processor, so the total number of different rows within in the grains increases. These different rows are the primary causes of the cache misses on the static scheduling policy. On the other hand, when the dynamic scheduling policy is used, the grains allocated to each processor for an iteration are not always identical to the grains used in the previous iteration. Thus, if the grains loaded into the cache in the previous iteration are not used, the first accesses for new grains result in cache misses(i.e., compulsory miss). This characteristic is the primary cause of the cache misses when using the dynamic scheduling policy.

The uniform spin waiting times of Figure 6(b) come from the portion of the sequential component executing pivot rows. On the other hand, the spin waiting time of Figure 6(c) involves both the execution time of these sequential components and the lock contention time caused by the spin-lock primitives that are used in the dynamic scheduling policy.

The barrier waiting time of the LU program depends on both the difference of cache misses among processors before reaching the barrier and the intrinsic load imbalance caused by gradually decreasing workload. Figure 6(c) shows that the dynamic scheduling policy results in the higher barrier waiting times as the grain size increases. As described above, using the dynamic scheduling policy, if some processors do not reuse the grain loaded into the cache in the previous iteration, they result in more cache misses than the other processors. Also, these cache misses increase as the large grain sizes are used, since they result in the more data loaded into caches without reuse. Since the processors suffering more cache misses reach the barriers late, the total barrier waiting times increase proportional to the grain size.

When using the dynamic scheduling policy, cache sizes also affect barrier waiting times. As shown in Figure 6(a), program performances are degraded as the cache size is increased under the dynamic scheduling policy. The reason is that the barrier waiting times are increased as larger caches

are used by the LU's own memory reference patterns under the dynamic scheduling policy. According to our experiment, large caches decreased total cache misses, but they also increased the difference in cache misses between participating processors. Thus, this difference resulted in higher barrier waiting times in large caches.

On the other hand, Figure 6(b) shows that the static scheduling policy produces the higher barrier waiting time in the fine grain sizes. The reason is that when the static scheduling policy is exploited in fine grain sizes, all processors are not provided with the exactly same number of grains. Even using the grain divisors for mitigating this phenomenon, it would be happened due to the characteristic of the static scheduling policy designating the grains to each processor at the compile-time. If some processors are assigned more number of grains as compared with other processors, they arrive late at barriers. Furthermore, since fine grain sizes allocate many small grains to each processor, the address interference among the grains allocated to the same processor causes the variability in cache misses among processors. These causes result in the higher barrier waiting time in the static scheduling policy using the fine grain sizes.

Figure 6(d) illustrates the performance variations across a range of grain sizes when the LU program is run on the various cache structures(direct-mapped, 2-way set associative, 4-way set associative) with a cache size of 32 Kbytes. The performances of the set associative caches are better than those of the direct-mapped cache, since the set associative caches reduce the cache conflict misses. However, when set associative caches are used, the performance variations issued by two scheduling policies are similar to those of the direct-mapped cache. Thus, the causes of these performance variations result from the combined effects of the scheduling and granularity policies, above which are equal to those under the direct-mapped cache.

In particular, the Figure 6(d) also shows that the 2-way set associative cache performs better than the 4-way set associative cache in several grain sizes using the static scheduling policy. Due to the LU's own memory reference nature under these grain sizes and the characteristic of the replacement policy on set associative caches to find victim cache lines, the 2-way set associative cache can result in smaller cache misses than the 4-way set associative cache in these grain sizes.

5 Related Works

Much research has investigated the cache effects and bus traffic pattern under various parallel programs[6, 7]. These studies have reported that the data sharing characteristics affected the performance of cache coherency protocol and that the performance of coherent caches relied on the quantity of sharing data and locality property in the program. Lam[5] presented techniques for blocked programs that were used for reducing the cache misses via improved temporal locality. Several optimizations to improve this performance were evaluated.

Several previous works have been proposed for operating system scheduling policies and synchronization primitives[8, 9]. To improve the cache utilization for a given scheduling policy, Torrellas[8] had evaluated several cache affinity scheduling policies on shared memory multiprocessors. Zahorjan[9] showed that performance was extremely degraded by the priority scheduling with busy waiting synchronization primitives. This study illustrated that the preemptive scheduling of processors with the use of busy waiting synchronization primitives degraded the performance since the running processes waited for the preempted processes with the lock. Tucker[10] suggested the process control approach focused on the processor utilization. This study addressed both synchronization and cache problems by partitioning processors into groups and dynamically ensuring that the number of runnable processes of an application matches the number of processors allocated to it.

McCann [11] suggested using the dynamic processor allocation policy on the multiprogrammed shared memory multiprocessors. This study reported that the performance of a dynamic scheduling policy with the processor reallocation is superior to that of a static scheduling policy even if the system overhead due to the processor reallocation increases. Gupta [12] used the simulations to investigate the relationship between scheduling policies and synchronization primitives and their impact on the system throughput. This study also reported the impacts of the scheduling strategies on the caching behavior of the parallel applications. However, the effects of granularity on parallel applications were not considered.

In the works described above, the impact of grain size and scheduling policies on parallel applications were not examined, though these affect the cache behaviors and synchronization operations of parallel applications.

6 Conclusion

In this paper, we studied the effects of the granularity and scheduling policies in data parallel programs. We applied various grain sizes to our benchmark applications and executed the partitioned grains on the dynamic and static scheduling policy.

In the BMM program, the grain size greatly affected the cache miss rates. These cache misses resulted in the performance variations across a range of grain sizes. Even if the synchronization primitives were exploited by the dynamic scheduling policy, synchronization latencies were not given much weight in the total execution times. From the experiments, we found that the best grain size depended on the amount of cache misses varied by the block size chosen. In particular, set associative caches showed the best performance in a smaller grain size than the direct-mapped cache, since they have the small set size and a different replace-

ment policy for choosing a victim cache line, as compared with the direct-mapped cache.

In the FFT program, the performance differences between the dynamic and static scheduling policy were small across a range of grain sizes except some fine grain sizes. When using the fine grain sizes, the dynamic scheduling policy resulted in the high spin-lock time caused by the frequent grain reallocation at run-time. Since the memory access latencies due to the cache misses were given much weight in the total elapsed time, the best performance was achieved on the static scheduling policy using the coarsest grain size, which resulted in the smallest cache misses.

In the LU program, the wasted time due to the cache misses occupied most of the totals elapsed time along with the overall grain sizes. The best performance was achieved on the static scheduling using the coarsest grain size, which incurred fewer cache misses than other grain sizes. Using the static scheduling policy with the fine grain sizes, the higher barrier waiting time resulted from the load imbalances and caching effects among the grains. On the other hand, using the dynamic scheduling policy with coarse grain sizes, the higher barrier waiting time resulted from the caching effects due to the run-time grain reallocation.

Our simulation results showed that combined effects of the granularity and scheduling policies were taken into account for the parallel processing, since the interactions between the granularity and scheduling policies had great impacts on the memory access latency and synchronization latency.

References

[1] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing(Design and Analysis of Algorithms)*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[2] J.E. Veenstra and R.J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *In Proceeding of 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems(MASCOTS)*, pages 201–207, January 1994.

[3] M. D. Hill. A case for direct mapped caches. *Computer*, 21(12):25–40, December 1988.

[4] J. Archibald and J-L. Baer. Cache coherence protocols : Evaluation Using a Multiprocessors Simulation Model. *ACM Transaction on Computer Systems*, 4(4):273–298, November 1986.

[5] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[6] S.J. Eggers and R.H. Katz. A Characterization of Sharing in Parallel Programs and its Applicability to Coherency Protocol Evaluation. In *In proceedings of the 15th International Symposium on Computer Architecture*, pages 373–382, May 1988.

[7] A. Gupta and W.D. Weber. *Analysis of Cache Invalidation Patterns in Shared Memory Multiprocessors, In Cache and Interconnect Architectures in Multiprocessors*. Kulwer Academic Publishers, 1990.

[8] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, Feb. 1992.

[9] J. Zahorjan, E. D. Lazowska, and D. L. Eager. Spinning versus Blocking in Parallel Systems with Uncertainty. In *In Proceedings of the International Seminar on Performance of Distributed and Parallel Systems*, pages 455–472, December 1988.

[10] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Multiprocessors. In *In Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 159–166, December 1989.

[11] C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Strategy for Multiprogrammed, Shared Memory Multiprocessors. *ACM Transaction on Computer Systems*, 11(2):146–178, May 1993.

[12] A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. *In Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer System*, pages 120–132, May 1991.