IEICE TRANS. INF. & SYST., VOL.E83–D, NO.9 SEPTEMBER 2000

PAPER

A Scheduling Policy for Blocked Programs in Multiprogrammed Shared-Memory Multiprocessors

Inbum JUNG[†], ***Student Member***, **Jongwoong HYUN**[†], ***and*** **Joonwon LEE**[†], ***Nonmembers***

SUMMARY Shared memory multiprocessors are frequently used as compute servers with multiple parallel programs executing at the same time. In such environments, an operating system switches the contexts of multiple processes. When the operating system switches contexts, in addition to the cost of saving the context of the process being swapped out and that of bringing in the context of the new process to be run, the cache performance of processors also can be affected. The blocked algorithm improves cache performance by increasing the locality of memory references. In a blocked program using this algorithm, program performance can be significantly affected by the reuse of a block loaded into a cache memory. If frequent context switching replaces the block before it is completely reused, the cache locality in a blocked program cannot be successfully exploited. To address this problem, we propose a preemption-safe policy to utilize the cache locality of blocked programs in a multiprogrammed system. The proposed policy delays context switching until a block is fully reused within a program, but also compensates for the monopolized processor time on processor scheduling mechanisms. Our simulation results show that in a situation where blocked programs are run on multiprogrammed shared-memory multiprocessors, the proposed policy improves the performance of these programs due to a decrease in cache misses. In such situations, it also has a beneficial impact on the overall system performance due to the enhanced processor utilization*.

key words: *time sharing system, cache pollution, process scheduling, cache locality, preemption-safe*

1. Introduction

Cache performance depends on the locality of references. If the sequence of addresses referenced by programs cannot all be stored in the cache, cache misses occur. It is not possible to build a cache that is large enough to hold the working sets of all programs, nor is it possible to code all programs to avoid all cache misses. However, several optimization techniques based on small source-code changes are used for improving cache performance [5], [6], [10]. A blocked algorithm is a well-known optimization technique to reduce cache misses via improved temporal locality in programs. This algorithm operates on submatrices or blocks matched with the processor's cache size instead of operating on entire rows or columns of an array. Thus, this algorithm maximizes reuse of the data loaded into the cache before the data are replaced.

Much research has investigated a number of effective methods for exploiting the resources of multiprocessor machines [2], [4], [7], [14]. While this is helpful when one has unique access to a dedicated parallel supercomputer, there is no way for a new program to enter the machine until the previous program finishes. Thus, many such machines are in fact shared among many users. In particular, since multiprocessors with low-cost high performance have become increasingly available in recent years, it is attractive to use them as compute servers that allows the users to have simultaneous access. The most common and easily-used type of multiprocessors are the shared-memory bus-based machines, which contain multiple processors communicating through a shared bus. These machines are generally of small to moderate scale (usually less than 16–32 processors) due to problems with bus saturation with higher numbers of processors. For many uses requesting simultaneous interactive access to the machine, time sharing for the machine is a widely used for multiprogramming. When programs run on this multiprogrammed system, multiple programs must share each processor. However, in such environments, an operating system performs the context switching, since many programs may be executing simultaneously. When using multiprogramming, in addition to the overhead of context switching between the multiple processes, the frequent context switching itself can affect process cache behavior. After a context switch, a process may be rescheduled on another processor, without the cache data it had loaded into the cache of the previous processor. Even if the process is rescheduled onto the same processor, intervening processes may have overwritten some or all of the cache data.

In particular, when programs exploit the blocked algorithm to improve cache locality in a multiprogrammed system, the blocks loaded into the cache memory can be polluted between context switches. Thus, the advantage of the blocked algorithm suffers from the damage in cache locality. To address this problem, we propose a preemption-safe policy to exploit the cache locality of blocked programs in a multiprogrammed system. The proposed policy delays con-

Manuscript received December 1, 1999.
Manuscript revised February 16, 2000.

[†]The authors are with the Department of Computer Science, Korea Advanced Institute of Science and Technology, Taejon, Korea.

[*]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.

text switching until only a block is fully reused in the program. However, since the delayed context switching can affect the running of other programs waiting on a run queue, their waiting time should be compensated. Thus, the proposed policy compensates the waiting time of other programs with subtracting any extra time blocked programs received from their next quantum. This procedure ensures that the characteristic of a blocked program itself can be utilized in a multiprogramming environment without greatly affecting the running of other programs. Our simulation results show that in a situation where blocked programs are run on multiprogrammed shared-memory multiprocessors, the proposed policy improves the performance of these programs due to a decrease in cache misses. In such situations, it also has a beneficial impact on the overall system performance by improving the processor utilization of other programs executing at the same time.

The remainder of the paper is organized as follows. Section 2 explains a blocked algorithm. Section 3 proposes a preemption-safe policy. Section 4 presents the simulation environment used in this study. In Sect. 5, we measure the performance of a preemption-safe policy on multiprogrammed shared-memory multiprocessors. Finally, related work is presented in Sect. 6 and we conclude in Sect. 7.

2. Blocked Algorithm

For some scientific programs, when accessing entire rows or columns of large arrays in every iteration of the loop, the cache misses due to limited cache capacity can severely hurt performance. The Example 1 is a naive matrix multiplication for matrices of size $N \times N$ in C programming language. To produce the matrix Z, the matrix X is multiplied by the matrix Y. The register variable r is also used to reduce memory accesses. Since this program operates on large data elements and performs identical processing on data, it is parallelized by assigning data elements to processors at the outmost loop(i.e., line 1) according to the grain size. The grain size means the fraction of matrix Z classified as coarse, medium, or fine, depending on the computation amount involved.

```
1:  for(i = 0 ; i < N ; i++){ /* parallelized location */
2:      for(j = 0 ; j < N ; j++){
3:          r = X[i][j];     /* register allocated */
4:          for(k = 0 ; k < N ; k++)
5:              Z[i][k] += r * Y[j][k];
6:      }
7:  }
```

Example 1 : A naive matrix multiplication in C language

Figure 1 shows that four processors are used for parallel processing with the coarsest grain size(i.e., $N/4$). In particular, the shaded elements of matrices illustrate the fractions being accessed by processor 1. From the Example 1, the shaded elements of matrix Z are reused N times each time in which the data are brought. On the other hand, all elements of matrix Y are reused $N/4$ times corresponding to the grain size chosen. From this figure, we know that if the cache size of a processor is not large enough to hold at least $N \times N$ matrix, the data of matrix Y would have been displaced before reuse. If the cache cannot hold even one row of the matrix Z, then the data of matrix Z in the cache cannot be reused. Thus, in the worst case, $2N^3 + N^2$ words of data should be read from memory in N^3 iterations. The high miss rate on the reuse can significantly slow down the system due to the increased memory fetches to numerical operations.

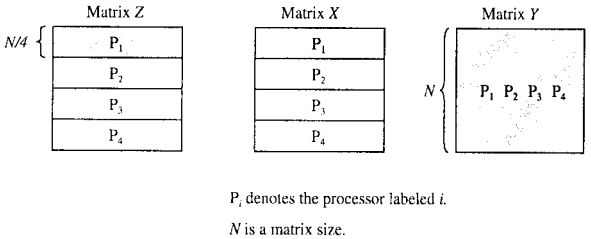

Fig. 1 Memory access patterns of a native matrix multiplication program.

To avoid this phenomenon, a blocked algorithm can be exploited for some scientific applications [5], [10]. The blocked algorithm ensures that the elements being reused can fit in the cache, since the original source code is changed to perform on only submatrices of size B, instead of operating on individual matrix entries. Example 2 is a blocked matrix multiplication code.

```
1:  for(kk = 0; kk < N ; kk += B){
2:      for(jj = 0; jj < N ; jj += B){
3:          for(i = 0; i < N ; i++){
4:              for(k = kk; k < min(kk+B, N) ; k++){
5:                  r = X[i][k];       /* register allocated */
6:                  for(j = jj; j < min(jj+B, N) ; j++)
7:                      Z[i][j] += r * Y[k][j];
8:              }
8:          }
8:      }
9:  }
```

Example 2 : A blocked matrix multiplication in C language

At the line 7 of this blocked program, we know that if a block size B is chosen so that a $B \times B$ submatrix of Y and a row of length B of Z can fit in the cache memory, both Y and Z are reused B times each time in which the data are brought. Thus, the total memory words accessed is $2N^3/B + N^2$ if there is no address

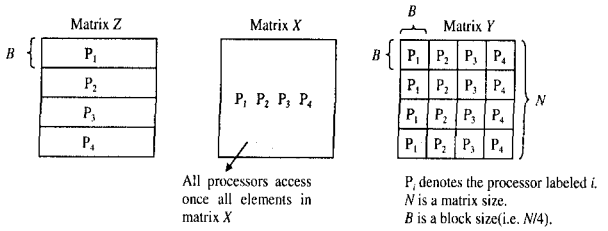

Fig. 2 Memory access patterns of a blocked matrix multiplication program.

interference in the cache.

Figure 2 shows a snapshot of the accesses to three matrices when four processors are used for parallel processing. From this figure, we know that each processor accesses the fixed locations in just two matrices Y and Z. Since these locations of two matrices Y and Z are reused B times within the iterations of loop, they cause good cache locality in this blocked program. On the other hand, since all elements of matrix X are accessed once during the overall execution time, they have not greatly impact upon cache performance.

3. Preemption-Safe Policy

In the blocked program using $B \times B$ blocks, since the blocks loaded into a cache memory are reused B times, the number of memory references is significantly reduced. However, when run on a multiprogrammed system, these reuses cannot be preserved, since many programs may be executing simultaneously, and each processor must be shared among multiple programs. In such environments, the frequent context switching can affect program cache behavior. If intervening programs wipe out a block of the blocked program before it is not fully reused, cache performance degrades due to the block displaced from the cache. To address this problem, we propose a preemption-safe policy to keep the advantage of the blocked algorithm even in a multiprogramming environment. The preemption-safe policy delays the context switching by the kernel scheduler per quantum until only a block of the blocked program is completely reused within a program.

To apply the preemption-safe policy in a multiprogrammed system, the operating system uses two extra variables per each process table (i.e., context table). One is used for denoting the *preemptable* or *unpreemptable* state of a process. Another one is used for representing the delay period of context switching. Besides the support of an operating system, blocked programs are also modified so that they inform the operating system of a start and end time of the computation for a block.

Figure 3 shows the control flows between a blocked program (i.e., blocked matrix multiplication) and an operating system in situations where the preemption-safe policy is used in a multiprogramming system. Be-

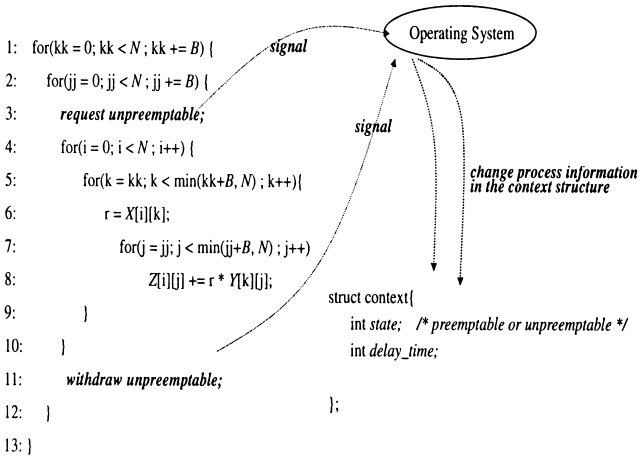

Fig. 3 Control flows of preemption-safe policy.

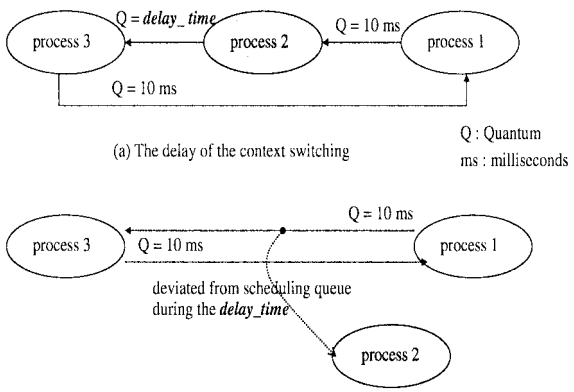

Fig. 4 Examples of process scheduling in preemption-safe policy.

fore the computation for a block is started at line 3, the blocked program sends a signal to the operating system to request that it not be preempted. The operating system accepts this request and sets the *state* variable of context structure to *unpreemptable*. The kernel scheduler does not swap out the processes with the unpreemptable state until their state variables change to *preemptable*. When the computation for a block is finished at line 11, the program also generates a signal to withdraw its unpreemptable request. At this time, the operating system sets the *state* variable to *preemptable*, and records the unpreempted period to the *delay_time* variable and performs the context switching immediately for waited other programs. Thus, the value of the *delay_time* variable is processor time pre-used by the blocked program. This time should be compensated for guaranteeing the fair processor usage among all programs running together.

Figure 4 shows the operations in a run queue when three processes are run under the multiprogrammed system supported a preemption-safe policy. A kernel scheduler assigns a process to a processor in a round-robin fashion with a quantum length. The term Q denotes a quantum size. In this Figure, we assume that

the default value is 10 milliseconds except the unpreempted period, and that process 2 is a blocked program. As shown in Fig. 4 (a), the process 2 has an unpreemptable state and monopolizes a processor during the *delay_time*, which is time to complete the computation for a block. After completing all the computation for a block, the process 2 should withdraw its unpreemptable right. Figure 4 (b) shows that the process 2 is deviated from the run queue after passing an unpreempted state. In such situations, the kernel scheduler schedules only two processes (i.e., process 1 and process 3) under the round-robin manner with a default time quantum. After an elapse of any extra time the process 2 received, the kernel resets the delay_time variable, and relocates the process 2 into the run queue. From the operation of the preemption-safe policy, we know that the delay of context switching can help exploit the advantage of the blocked algorithm, and also that the compensation procedure for pre-used processor time ensures the fair processor usage for all programs running simultaneously.

However, using the preemption-safe policy invokes additional signals to request to an operating system, and needs kernel extensions as described in above. In particular, since the excessive delay of context switching can affect other programs' response times running together with blocked programs, the operating system should limit the delay period of context switching. If a process does not yield a processor within a small bounded period of time that is designated as a multiple of default quantum size (e.g., if a default quantum is 10 milliseconds, it can be 50 milliseconds, 100 milliseconds, and so on), the kernel scheduler should preempt it anyway and mark its context structure with this involuntary preemption. After that, this process suffers from the compensation procedure for pre-used processor time as described above. However, even if this process is preempted immediately, the reuse of a block till then will improve cache performance. When this process has its turn again after expiring the compensation period, the operating system sets again the *state* variable of its context structure to *unpreemptable*, since it was preempted forcibly without the voluntary withdrawal. Thus, this process can continue to execute the remaining portion for computing the block under the unpreemptable state, until it incurs the withdraw-unpreemptable signal or exhausts the bounded delay period again. In Sect. 5, we will measure the delay time under our benchmark programs. This experience may help the OS designer to designate the appropriate delay period in the system.

4. Simulation Environment

4.1 Simulation Technique

The proposed preemption-safe policy can be applied to both the uniprocessor and the multiprocessors. However, scientific programs using the blocked algorithm show data parallelism, since they perform identical operations on all data elements and these elements are assigned to various processors to parallelize the computation. Thus, we use the multiprocessors to exploit this parallelism.

Table 1 Timing values for cache coherency protocol.

Events (operations)	Penalties (cycles)
A write on a shared line (Invalidate signal)	3
A cache miss (The missed line is supplied by an another cache)	7
A cache miss (The missed line is supplied by the main memory)	22

In particular, we assume the shared-memory multiprocessors system with a shared bus (e.g., PC servers) as a machine chosen for this study. This system is widely used and commercialized for computing servers due to its low-cost high computing power and ease of use. These machines are also called UMA (Uniform Memory Access) machines, since access to a memory location via the bus takes the same amount of time regardless of which processor is performing the access and what memory location is being accessed. Cache coherency is maintained across processors through a variety of snooping and invalidation techniques. The simulated environment for this machine is described as follows.

The simulators consist of a functional simulator that executes parallel programs and an architectural simulator that models the shared memory multiprocessors. We use an efficient program-driven simulator, MINT (Mips INTerpreter) [16] as a functional simulator. We construct an architectural simulator based on a shared-memory multiprocessor with eight processors and a shared-bus. Each processor is assumed to be a RISC processor with the same cache size and each instruction is executed in a single cycle except the memory reference. We assume that cache structure is 128 Kbytes direct-mapped with 16 bytes cache line size. The simulated cache coherency protocol is the write invalidation scheme [1]. 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 the timing values for the cache coherency protocol with the above microprocessors' parameters and 1 address cycle and 1 bus operation cycle.

The standard MINT provides facilities to run only one parallel program at a time, with each process permanently scheduled onto its own processor. Thus, we extended it to run multiple parallel programs at the same time and linked it with our scheduling module. We employ the *gang scheduling* as our basic scheduling module due to its easy implementation and less syn-

chronization overhead [9]. In this basic scheduling module, the number of runnable processes matches with the number of processors available, and all runnable processes of a program are scheduled to run on the processors at the same time. When a time slice ends, all running processes are preempted simultaneously, and all processes from a second program are scheduled for the next time slice. When the preempted processes have their next time slice, they are rescheduled onto the same processor to exploit the cache data loaded into the cache of the previous processor.

To construct the kernel scheduler involving the preemption-safe policy described in Sect. 3, we added its functional structure to the basic scheduling module. All processes invoked by the blocked program have the *unpreemptable* state during the period for reusing the blocks allocated to each process. The simulated kernel scheduler does not swap out the processes with the unpreemptable state until the state of all processes changes to *preemptable* as described in Sect. 3.

As mentioned above Sect. 3, whenever a block is executed, each processor incurs two signals to set or reset variables of its context structure. Since the handling of these signals needs the running of the kernel scheduler, it can result in cache pollution to the data already loaded into caches from matrices X, Y, and Z. However, as shown in Fig. 3, since the lines before the *request-preemptable* signal do not generate high cache locality, the cache pollution due to this signal does not affect cache performance. On the other hand, the codes between line 4 and line 10 result in high cache locality based on the reuse of $B \times B$ blocks in matrix Y. However, since the completely reused blocks within these lines do not reused in the next iterations, the cache pollution due to the *withdraw-preemptable* signal at the line 11 does not greatly affect cache performance. Thus, in the simulation, we do not consider the cache pollution due to the running of the kernel scheduler, since it has not impact upon cache locality induced by the reuse of $B \times B$ blocks in matrix Y, and because the kernel data for handling a signal are usually small in the operating system like UNIX.

In particular, since the programs using the blocked algorithm have compute-bound workloads, the cache pollution due to other OS services (i.e., system calls, page faults, and so on) is regarded as of little value. Mogul and Borg [8] also reported that a compute-bound workload generated almost exclusively involuntary context switches due to quantum elapses (i.e., Only fewer than 1% are caused by either system calls or page faults). Thus, our research concentrates on the problem of cache pollution caused by context switches due to quantum elapses.

In our simulation, a time slice (i.e., quantum) is assumed to be 10 milliseconds. Also, the preemption-safe policy needs additional signals between a program and an operating system as described in Sect. 3.

Table 2 Timing values for process scheduling.

Operations	Time (cycles)
Quantum(10 milliseconds)	2,500,000
Signal	2,700
Context Switch	26,425

Table 3 Benchmark programs and their data sets.

Program	Data sets
B-MM	three 256 × 256 matrices
B-LU	a 512 × 512 matrix
BS	131,072 sorting keys
MP3D	20,000 molecules, a 16 × 16 × 16 array
FFT	524,288 input points
OCEAN	a 512 × 512 grid, 25 two-dimension arrays

Small [12] reported that the time for treating a signal was 10.8 μseconds and the time for treating a context switch was 105.7 μseconds on a BSD UNIX operating system. From these considerations and the assumed clock rate (i.e., 250 MHz), the timing values used in process scheduling are shown in Table 2.

4.2 Benchmark Programs

Table 3 shows six benchmark programs chosen for this study and their data sets used. All these programs are written in C language and use the synchronization and sharing primitives provided by the SGI's parallel macros package. The programs using the blocked algorithm are B-MM (Blocked Matrix Multiplication) [4] and B-LU(Blocked LU Decomposition) [17]. Other programs are BS (Bitonic Sorting) [4], MP3D [11], FFT [17] and OCEAN [17]. In choosing the non-blocked programs, we tried to include the programs of various characteristics. Their characteristics are already well-studied in many previous works [11], [17]. In our simulation, since we are more interested in the performance of blocked programs under the preemption-safe policy, the data sets of all programs shown in Table 3 are established so that two blocked programs are finished earlier than other programs.

To parallelize the computation, we use the coarsest grain size in all programs due to its less overhead for handling a grain queue, which is driven by dividing the data size described in Table 3 by the number of processors (i.e., eight processors we assumed in the above subsection). Thus, the grain size of the B-MM program is a 32×32 block and that of the B-LU program is a 64×64 block, since blocked programs use the block size B as a grain size for parallel processing.

5. Performance

To evaluate the performance of a preemption-safe policy in a multiprogrammed system, we also perform blocked programs under a preemption policy that is based on the basic scheduling module described in Sect. 4.1. This policy uses the round-robin manner

with a typical default time quantum of 10 milliseconds. On the other hand, except for the unpreempted periods, the preemption-safe policy also uses a default time quantum like the preemption policy. We use the term *programming level* to represent the number of programs executing concurrently on a multiprogrammed system. The higher the programming level is used, the more programs are executed concurrently and the more likely it is that programs may suffer from the cache pollution due to context switching.

5.1 Single Blocked Program

- B-MM Program

Figure 5 (a) shows the performances of the B-MM program on both the preemption policy and the preemption-safe policy. The range of programming levels is from level 2 to level 5. Table 4 shows the lists of the programs used at each programming level. The performances under the preemption-safe policy are better than those under the preemption policy in all programming levels. For example, the preemption-safe policy shows the improved execution time of about 25.5% at level 2 and about 12.5% at level 5, when compared to the preemption policy. These improvements result from a decrease in cache misses as shown in Fig. 5 (b). For example, cache misses decrease about 32% at level 2 and about 10.7% at level 5. From these results, we know that the decrease in execution time by using the preemption-safe policy correlates perfectly with the decrease in cache misses. However, in both policies, the higher programming levels use, the more cache misses occur. The reason is that the amount of cache pollution between context switching increases as the more programs are run at the same time. However, the preemption-safe policy could prevent only a $B \times B$ block from becoming the pollution due to context switching.

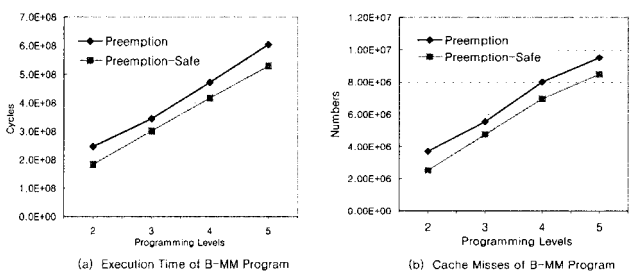

Fig. 5 B-MM program: Performance under various programming levels.

Table 4 Lists of programs running together with B-MM program.

programming levels	programs
level 2	B-MM, FFT
level 3	B-MM, FFT, BS
level 4	B-MM, FFT, BS, OCEAN
level 5	B-MM, FFT, BS, OCEAN, MP3D

Figure 6 (a) shows the execution times of all programs running under programming levels 4 and 5 with the preemption-safe policy. The results are normalized to the execution times under the preemption policy. This figure shows that other programs except the B-MM program are not greatly affected in preemption-safe policy, despite arbitrary delays on the part of their execution. This result can be explained as follows. We measured the number of delayed context switches when the B-MM program is located in the unpreemptable state for computing a block, since the excessive delay of context switching may increase the waiting time of other programs. However, during the period computing a block, the average number of delayed context switches is 3 or 4 times (i.e., 30 milliseconds or 40 milliseconds). These delay periods do not greatly affect the execution of other programs. Moreover, after the B-MM program exploits an unpreemptable state, the kernel scheduler does not permit the running of the B-MM program until its pre-used time is exhausted. Thus, other programs can be provided with the fairness in terms of processor usage.

Figure 6 (b) shows the cache misses when programs are run under programming levels 4 and 5 using the preemption-safe policy. These results are normalized to the cache misses under the preemption policy. As shown in this figure, the preemption-safe policy produces the reduced cache misses in the B-MM program, and also it decreases the cache misses of other programs very little. Little reduction in the cache misses of other programs is due to the fact that after the B-MM program is done early, the remaining programs may undergo the reduced cache pollution between their context switching.

- B-LU Program

Figure 7 shows the execution times and cache misses of the B-LU program obtained for two policies across a range of programming levels. Table 5 shows the list of the programs used at each programming level. Like the B-MM program, the B-LU program shows better performance in the preemption-safe policy because the number of cache misses is decreased in all programming levels. For example, when running at level 5, the preemption-safe policy shows about 13.3% improvement in the execution time due to the reduced cache

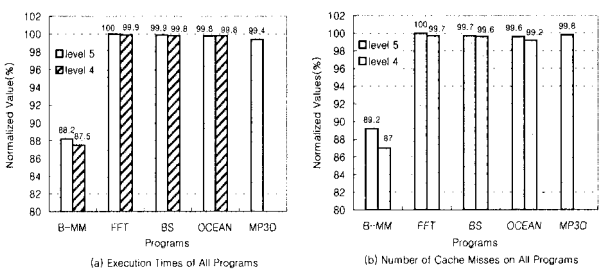

Fig. 6 Performance of B-MM program and other programs.

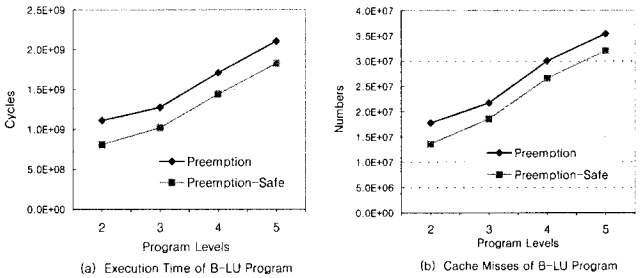

Fig. 7 B-LU program: Performance under various programming levels.

Table 5 Lists of programs running together with B-LU program.

programming levels	programs
level 2	B-LU, FFT
level 3	B-LU, FFT, BS
level 4	B-LU, FFT, BS, OCEAN
level 5	B-LU, FFT, BS, OCEAN, MP3D

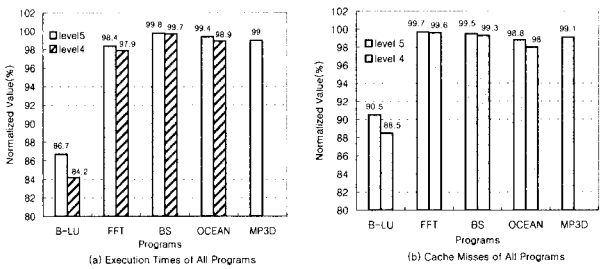

Fig. 8 Performance of B-LU program and other programs.

misses of about 9.5%.

Figure 8 shows the execution times and the cache misses when programs are run under at programming levels 4 and 5 using the preemption-safe policy. These results are normalized to those obtained for the preemption policy. The preemption-safe policy results in the better execution time due to the reduced cache misses in all programs. Like in the B-MM program, we measured the number of delayed context switches when the B-LU program is on the unpreemptable state. However, since the average number of delayed context switches is 4 or 5 times (i.e., 40 milliseconds or 50 milliseconds), these delays do not affect the execution of non-blocked programs as shown in Fig. 8 (a).

5.2 Multiple Blocked Programs

To measure the impact on the performance of preemption-safe policy when running multiple blocked programs, we run two blocked programs concurrently at programming level 6. Figure 9 shows the execution times and cache misses of all programs including two blocked programs when they are run under the preemption-safe policy. All results are normalized to those obtained for the preemption policy.

The Fig. 9 (a) shows that the preemption-safe policy improves not only the execution times of two blocked programs but also those of other programs. However, the improving rates in blocked programs are lower than those observed under programming level 4 or 5 shown in Fig. 6 (a) and Fig. 8 (a). The reason is that the higher programming level is used, the more data are displaced from the cache memory between context switches. In particular, as the programming level is increased, the rate of cache data replacement in the preemption policy is higher than that in the preemption-safe policy. Thus, the more cache misses in the preemption policy result in the more execution time, and the normalized values are diminished as the programming level is increased.

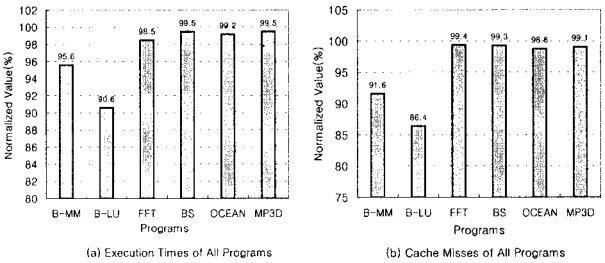

Fig. 9 Performance of programs on programming level 6.

As shown in Fig. 9 (b), the preemption-safe policy results in the reduced cache misses even in non-blocked programs. As explained in the above subsections, this result is because after two blocked programs are finished early, the remaining programs suffer less cache pollution between their context switching than when six programs are run concurrently. Moreover, since the early completion of two blocked programs increases other programs' processor usage, it also has a beneficial impact on their execution time.

5.3 Overhead

When the preemption-safe policy is exploited, kernel extensions are needed to support additional signals (i.e., *request* and *withdraw* of *unpreemptable*). The handling of these new signals within the operating system is the major overhead affecting on the execution time. For Example, for the UNIX operating system, the kernel extensions can be simply done through both allocating two new signal numbers to the current signal list and making the corresponding signal handler routines.

As mentioned in Sect. 3, the use of signals incurs the time for handling them between the kernel and the program. With the UNIX operating system, Small [12] reported that 10.8 μseconds are needed for treating a signal such as either setting or resetting kernel variables. Thus, in our simulation, we already reflected these time overheads as shown in Table 2. While these overheads degrade the execution time of programs, the decrease in cache misses due to the preemption-safe policy highly improves the execution time. To compare the

gain from the improved cache performance with the loss from time overheads due to signals, we illustrate an inequality as follows:

$$G - P \times N \times O > 0$$

G : improved execution time
P : number of processes
N : number of signals from each process
O : time overhead due to a signal call

Only if the left side of this inequality is more than zero, the proposed preemption-safe policy is valuable. For example, in situations where blocked programs are run under programming level 5 that are shown in the Fig. 5 and Fig. 7, time overheads due to signals are 345,600 cycles (i.e., P : 8 processors, N : 8 blocks $\times$ 2 signals, O : 2,700 cycles that is mentioned Table 2). On the other hand, the improved execution times due to the reduced cache misses are 75,309,351 cycles in the B-MM program and 280,148,974 cycles in the B-LU program. Thus, since the signal overheads are merely 0.1–0.4% as compared to the improved execution times, the preemption-safe policy is worthwhile technique.

If the time overhead for handling a signal call is greater than that of our system assumed above, specific instructions can be used to ascertain the request and the withdrawal of unpreemptable state. These instructions can reduce the run-time overhead due to the current signal handling procedure, but it needs the hardware mechanism to detect them and the compiler's help to arrange them.

6. Related Works

Previous research on blocking has been done on evaluating the performance of blocked algorithm [5], [6]. Lam [5] experimented with a matrix multiplication using the blocked algorithm under the various cache structures. This study calculated the optimal block size based on given cache parameters that could avoid self address interference. However, these previous studies about blocking technique did not involve in a multiprogramming environment running several other parallel programs.

Several studies have analyzed the cache performance under a multiprogramming system [2], [8]. In particular, Mogul and Borg [8] experimented with the effect of context switches on cache performance in a multiprogramming system. They classified simulated programs as three types: a timesharing system with a few intensive users, a compute-bound load with a couple of larger programs, a repetitive client-server interaction program. According to the classified types, they divided the causes of context switches into system calls, page faults or scheduler and measured the costs of context switches for each program type on the basis of the divided causes. As a result, this study reported that the cache-performance costs of a context switch are greater than all other context-switch costs.

Several previous studies have been reported the interaction between scheduling strategies and cache performance of programs [13], [15]. These studies for cache affinity only focused on the footprints on the cache memory remaining after context switching, they did not consider the preservation of cache locality in a multiprogramming system.

When using multiprogramming, besides the overhead of cache pollution, synchronization primitives can substantially degrade the performance of parallel programs. If the processor that is to set the variable is preempted, the processors that are running but waiting for the variable to be set will waste processor cycles. Much previous work has been reported for operating system scheduling policies and synchronization primitives [3], [14], [18]. In particular, Kontothanassis [3] investigated synchronization algorithms used to avoid preempting processes with an active lock. When a process with a lock variable enters a critical section, it requests the operating system not to preempt it until it leaves the critical section. This non-preemption policy for locking mechanisms allowed the system to avoid the useless work induced by other running processes stalled while waiting for the lock variable occupied by a preempted process.

The blocked algorithm makes good cache locality for programs via source-code changes. If this locality suffers from context switching in a multiprogramming environment, the anticipated cache performance cannot be obtained. In this paper, we proposed the preemption-safe policy to preserve blocked programs' cache locality in a multiprogramming system. In the previous study for effective synchronization algorithms in a multiprogramming system, this non-preemption policy was used to avoid the preemption of the process with an active lock [3]. On the other hand, we used this policy to preserve the cache locality of programs.

7. Conclusion

In this paper, we have considered the performance of the blocked algorithm on a multiprogrammed system. The blocked algorithm improves cache performance by increasing the locality of memory references. To apply the blocked algorithm, the existing programs are modified to make the reused blocks, and these reused blocks result in the enhanced cache locality. However, in a multiprogrammed system, the blocks loaded into a cache memory can be polluted by the context switching. To address this phenomenon, we proposed a preemption-safe policy to keep the advantage of a blocked algorithm even in a multiprogrammed system. This proposed policy delayed context switching until the block loaded into a cache memory was fully reused.

Thus, this method gave blocked programs safe execution since they were not preempted during the period computing a block. Also, this method guaranteed the fair usage of processor time via compensating the processor time pre-used by blocked programs.

Simulation results showed that the preemption-safe policy improved the performance of blocked programs due to the reduced cache misses. In particular, since the delay period of context switching was short, it did not affect other programs' response time. Moreover, since early finished blocked programs caused the less cache pollution and the more processor utilization for the remaining programs, the overall system performance was also improved.

The preemption-safe policy needs the operating system's support and the program's modification. However, for the programs with good cache locality, such as blocked programs, exploiting the preemption-safe policy is worthwhile if the delay period of context switching does not severely affected programs' response time, because this policy can enhance the overall throughput for a given system.

References

[1] J. Archibald and J.-L. Baer, "Cache coherence protocols: Evaluation using a multiprocessors simulation model," ACM Transaction on Computer Systems, vol.4, no.4, pp.273–298, Nov. 1986.
[2] A. Gupta, A. Tucker, and S. Urushibara, "The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications," Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer System, pp.120–132, May 1991.
[3] L.I. Kontothanassis, R.W. Wisniewski, and M. L. Scott, "Scheduler-conscious synchronization," ACM Trans. Computer Systems, vol.15, no.1, pp.3–40, Feb. 1997.
[4] 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.
[5] M.S. Lam, E.E. Rothberg, and M.E. Wolf, "The cache performance and optimizations of blocked algorithms," Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.63–74, April 1991.
[6] A.R. Lebeck and D.A. Wood, "Cache profiling and the SPEC benchmarks: A case study," IEEE Computer, vol.27, no.10, pp.15–26, Oct. 1994.
[7] C. McCann, R. Vaswani, and J. Zahorjan, "A dynamic processor allocation strategy for multiprogrammed, shared memory multiprocessors," ACM Trans. Computer Systems, vol.11, no.2, pp.146–178, May 1993.
[8] J.C. Mogul and A. Borg, "The effect of context switches on cache performance," Proeedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.75–84, Oct. 1991.
[9] J.K. Ousterhout, "Scheduling techniques for concurrent systems," Proceedings of the 3rd International Conference on Distributed Computing Systems, pp.22–30, Oct. 1982.
[10] D.A. Patterson and J.L. Hennessy, Computer A Quantitative Approach, 2nd ed., Morgan Kaufmann Publishers, Inc., 1996.
[11] J.P. Singh, W.D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory," Computer Architecture News, vol.20, no.1, pp.5–44, March 1992.
[12] C. Small and M. Seltzer, "Scheduler activations on BSD: Sharing thread management between kernel and application," Technical Report TR-31-95, Department of Computer Science, Harvard University, 1995.
[13] M.S. Squillante and E.D. Lazowska, "Using processor-cache affinity information in shared-memory multiprocessor scheduling," IEEE Trans. Parallel and Distributed Systems, vol.4, no.2, pp.131–143, Feb. 1993.
[14] A. Tucker and A. Gupta, "Process control and scheduling issues for multiprogrammed multiprocessors," Proceedings of the 12th ACM Symposium on Operating System Principles, pp.159–166, Dec. 1989.
[15] R. Vaswani and J. Zahorjan, "The implication of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors," Proceedings of the 13th ACM Symposium on Operating System Principles, pp.26–40, Oct. 1991.
[16] J.E. Veenstra and R.J. Fowler, "MINT: A front end for efficient simulation of shared-memory multiprocessors," Proceeding of 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp.201–207, Jan. 1994.
[17] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," Proceedings of the 22th Annual International Symposium on Computer Architecture, pp.24–25, June 1995.
[18] J. Zahorjan, E.D. Lazowska, and D. L. Eager, "Spinning versus blocking in parallel systems with uncertainty," Proceedings of the International Seminar on Performance of Distributed and Parallel Systems, pp.455–472, Dec. 1988.

Inbum Jung received the B.S. degree in electronics engineering from Korea University, in 1985, and the M.S. degree in information & communication engineering from Korea Advanced Institute of Science (KAIST), in 1994. He is currently working towards the Ph.D degree in computer science from KAIST. From 1984 to 1995, he was with the Computer System Division of Samsung Electronics Co., Ltd., Korea. His research interests include operating systems, computer architectures, parallel processing, cluster computing and multimedia systems.

Jongwoong Hyun received the B.S. degree from Korea University, in 1986 and the MS. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), in 1998. He is currently a Ph.D. student working in the computer architecture group at KAIST. From 1986 to 1998, he was with Computer System Division of Samsung Electronics Do. Ltd., Korea. His research interests include computer architecture, cluster computing, and Web server.

Joonwon Lee received the B.S. degree from Seoul National University, in 1983 and the M.S. and Ph.D. degrees from the College of Computing, Georgia Institute of Technology, in 1990 and 1991, respectively. From 1983 to 1986, he was with Yugong Ltd., and from 1991 to 1992, he was with IBM research centers where he was involved in developing a scalable shared memory multiprocessors. He is currently a faculty member at KAIST. He was a recipient of Windows NT source code. His research interests include operating systems, computer architectures, parallel processing, cluster computing and Web server.