ELSEVIER

Journal of Systems Architecture 46 (2000) 1191–1204

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

A scheduling policy for preserving cache locality in a multiprogrammed system ☆

Inbum Jung *, Jongwoong Hyun, Joonwon Lee

Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 373-1, Kusong-Dong, Yusong-Ku, Taejon 305-701 South Korea

Received 17 December 1999; received in revised form 13 March 2000; accepted 20 May 2000

Abstract

In a multiprogrammed system, when the operating system switches contexts, in addition to the cost for handling the processes being swapped out and in, the cache performance of processors also can be affected. If frequent context switching replaces the data loaded into cache memory before they are completely reused, the programs suffer from cache misses due to the damage in cache locality. In particular, for the programs with good cache locality, such as blocked programs, a scheduling mechanism of keeping cache locality against context switching is essential to achieve good processor utilization. To solve this requirement, we propose a preemption-safe policy to exploit the cache locality of blocked programs in a multiprogrammed system. The proposed policy delays context switching until a block is fully reused, 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. © 2000 Published by Elsevier Science B.V. All rights reserved.

Keywords: Blocked algorithm; Multiprogrammed system; Cache locality; Context switching

1. Introduction

Parallel programs are often run on shared-memory multiprocessors due to usable programming environments and low-cost high performance systems. In these systems, each processor accesses memory locations via its cache memory to fetch the data. Cache memory reduces the speed difference between the fast processor and the slow main memory by holding regions of recently referenced main memory. Thus, cache misses incur memory access latencies in retrieving the corresponding data from the main memory. During this penalty period, processors must stall until the data arrive.

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 neither possible to build a cache that is

☆ 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.

* Corresponding author.

E-mail address: jib@calab.kaist.ac.kr (I. Jung).

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 [1–4]. 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.

When programs are run on a multiprogrammed system, multiple programs must share each processor. In such environments, an operating system performs the context switching, since many programs may be executed 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 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 context 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 by 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 executed 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 Section 5, we measure the performance of a preemption-safe policy on multiprogrammed shared-memory multiprocessors. Finally, related work is presented in Section 6 and we conclude in Section 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. 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.

Example 1. A naive matrix multiplication in C language

```
for(i = 0; i < N; i++){/* parallelized location */
    for(j = 0; j < N; j++){
        r = X[i][j];       /* register allocated */
        for (k = 0; k < N; k++)
            Z[i][k] += r * Y[j][k];
    }
}
```

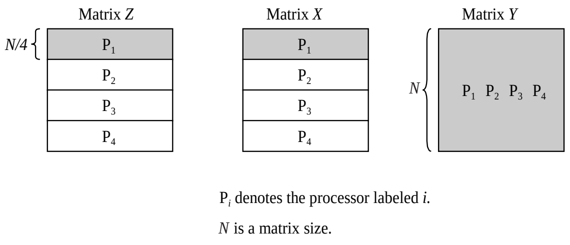


Fig. 1. Memory access patterns of a native matrix multiplication program.

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.

Fig. 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 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.

To avoid this phenomenon, a blocked algorithm can be exploited for some scientific applications [1,3]. 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. Example2 is a blocked matrix multiplication code.

Example 2. A blocked matrix multiplication in C language

```
for(kk = 0; kk < N; kk+ = B){
   for(jj = 0; jj < N; jj+ = B){
      for(i = 0; i < N; i++){
         for(k = kk; k < min(kk + B, N); k++){
            r = X[i][k];   /* register allocated */
            for(j = jj; j < min(jj + B, N); j++)
         Z[i][j]+ = r * Y[k][j];
            }
         }
      }
   }
```

At 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 are $2N^3/B + N^2$ if there is no address interference in the cache. To parallelize this blocked program, a block size B is used as a grain size for parallel processing. Thus, $B \times B$ blocks in matrix Y are assigned to each processor between line 1 and line 2. Fig. 2 shows a snapshot of the accesses to three matrices when four processors are used for parallel processing.

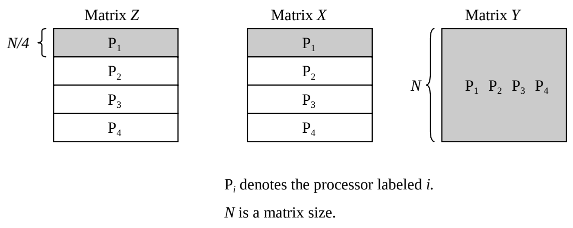


Fig. 2. Memory access patterns of a blocked matrix multiplication program.

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 do not have a great impacted on cache performance.

3. Preemption-safe policy

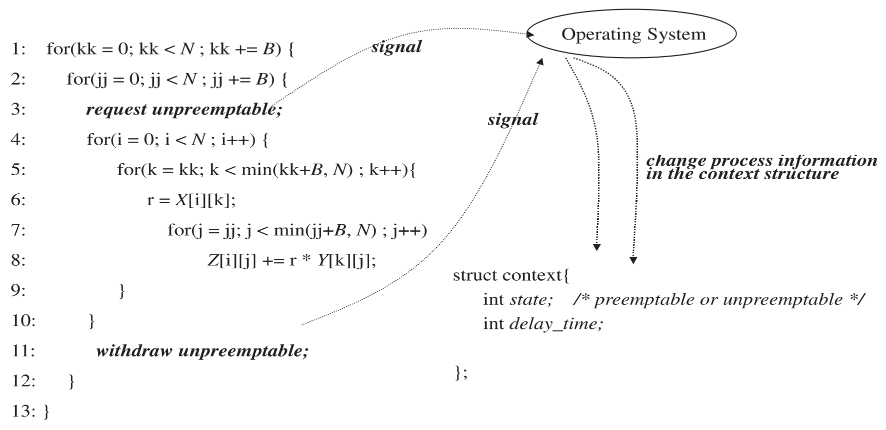
In the blocked program using $B \times B$ blocks, since the blocks loaded into a cache memory are reused B times, the number of memory references is significantly reduced. However, when run on a multiprogrammed system, the cache locality attributed to these reuses cannot be preserved, since many programs may be executed simultaneously, and 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. Thus, the proposed policy provides blocked programs with safe execution since they are not preempted during the computation of a block.

To apply the preemption-safe policy in a multiprogrammed system, the operating system uses two extra variables per 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 delayed 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.

Fig. 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. Before the computation for a block is started at line 3, the blocked program sends a signal to the operating system to request that it should 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 preused by the blocked program. This time should be compensated for guaranteeing the fair processor usage among all programs running together.



```
for(kk = 0; kk < N ; kk += B) {
    for(jj = 0; jj < N ; jj += B) {
        request unpreemptable;
        for(i = 0; i < N ; i++) {
            for(k = kk; k < min(kk+B, N) ; k++){
                r = X[i][k];
                    for(j = jj; j < min(jj+B, N) ; j++)
                        Z[i][j] += r * Y[k][j];
            }
        }
         withdraw unpreemptable;
    }
}
```

```
struct context{
    int state;    /* preemptable or unpreemptable */
    int delay_time;

};
```

Fig. 3. Control flows of preemption-safe policy.

Fig. 4 shows the operations in a run queue, when three processes are run under the multiprogrammed system supported by 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 ms except the unpreempted period, and that process 2 is a blocked program. As shown in Fig. 4(a), 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, process 2 should withdraw its unpreemptable right. Fig. 4(b) shows that 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 process 2 received, the kernel resets the delay_time variable, and relocates 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 preused 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 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 delayed period of context switching. If a process does not yield a processor within a small bounded period

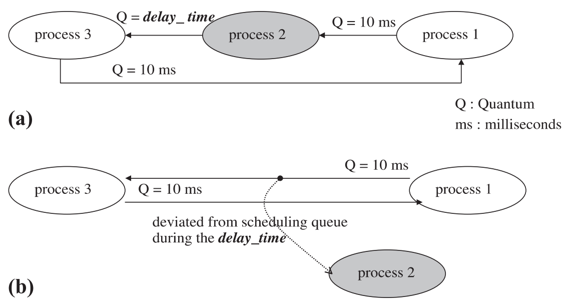


Fig. 4. Examples of process scheduling in preemption-safe policy: (a) the delay of the context switching; (b) the compensation for preused processor time.

of time, the kernel should preempt it anyway. In Section 5, we will look at the impact of delayed context switching in other programs running together with blocked programs.

4. Simulation environment

4.1. Simulator design and simulated platform

The simulation environment consists 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) [5] as a functional simulator. The MINT supplies a memory-reference generator and simulation libraries. The memory reference generator executes a program on several processors and sends an event to the architectural 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 [6]. 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 synchronization overhead [7]. 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 Section 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 Section 3.

In our simulation, a time slice (i.e., quantum) is assumed to be 10 ms. Also, the preemption-safe policy needs additional signals between a program and an operating system as described in Section 3. Small [8] reported that the time for treating a signal was 10.8 μs and the time for treating a context switch was 105.7 μs 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.

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

Table 2
Timing values for process scheduling

Operations	Time (cycles)
Quantum (10 ms)	2,500,000
Signal	2700
Context switch	26,425

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 blocked matrix multiplication (B-MM) [9] and blocked LU decomposition (B-LU) [11]. Other programs are bitonic sorting (BS) [9], MP3D [10], FFT [11] and OCEAN [11]. In choosing the nonblocked programs, we tried to include the programs of various characteristics. Their characteristics are already well-studied in many previous works [10,11]. 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 Section 4.1). 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. Table 4 provides the basic characteristics about the programs when each program is run individually without interruption with eight processors.

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 $16 \times 16 \times 16$ array
FFT	524,288 input points
OCEAN	a 512×512 grid, 25 two-dimension arrays

Table 4
General statistics for benchmark programs

Program	Total cycles ($\times 10^8$)	Cache misses ($\times 10^6$)
B-MM	1.11	1.3
B-LU	3.77	6.4
BS	19.8	39.3
MP3D	20.1	8.1
FFT	13.4	25.3
OCEAN	16.1	6.2

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 Section 4.1. This policy uses the round-robin manner with a typical default time quantum of 10 ms. 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 executed concurrently on a multiprogrammed system. The higher the programming level used, the more the programs 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

5.1.1. B-MM program

Fig. 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 5 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 the programming levels used, the more the cache misses occur. The reason is that the amount of cache pollution between context switching increases as 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.

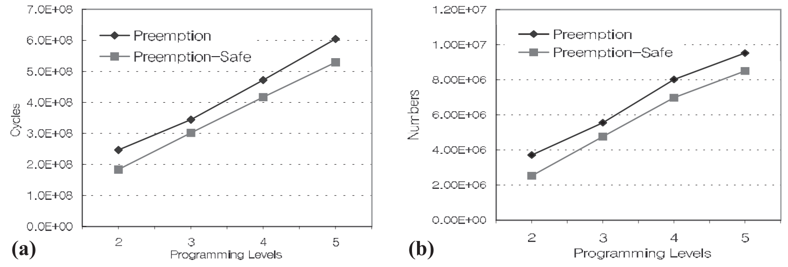


Fig. 5. B-MM program – performance under various programming levels: (a) execution time of B-MM program; (b) cache misses of B-MM program.

Table 5
Lists of programs running with a 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

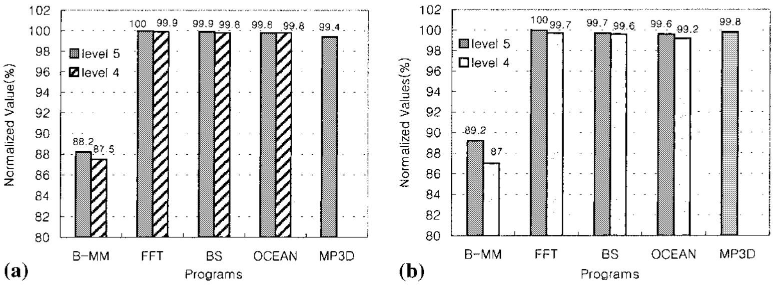


Fig. 6. Performance of B-MM program and other programs: (a) execution time of all programs; (b) number of cache misses on all programs.

Fig. 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 of computing a block, the average number of delayed context switches is three or four times (i.e., 30 or 40 ms). These delayed 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 preused time is exhausted. Thus, other programs can be provided with the fairness in terms of processor usage.

Fig. 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.

5.1.2. B-LU program

Fig. 7 shows the execution times and cache misses of the B-LU program obtained for two policies across a range of programming levels. Table 6 shows the lists 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

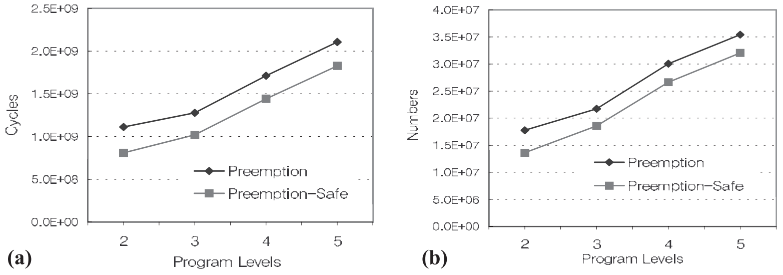


Fig. 7. B-LU program – performance under various programming levels: (a) execution time of B-LU program; (b) cache misses of B-LU program.

Table 6
Lists of programs running with a 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

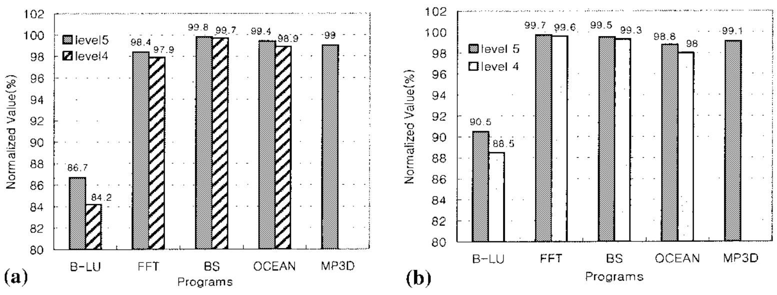


Fig. 8. Performance of B-LU program and other programs: (a) execution times of all programs; (b) cache misses of all programs.

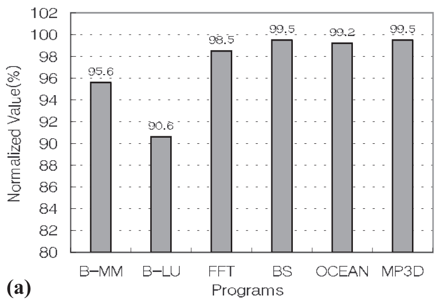
because the number of cache misses is decreased in all programming levels. For example, when running at level 5, the preemption-safe policy shows about 13.3% improvement in execution time due to the reduced cache misses of about 9.5%.

Fig. 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 four or five times (i.e., 40 or 50 ms), these delays do not affect the execution of non-blocked programs as shown in Fig. 8(a).

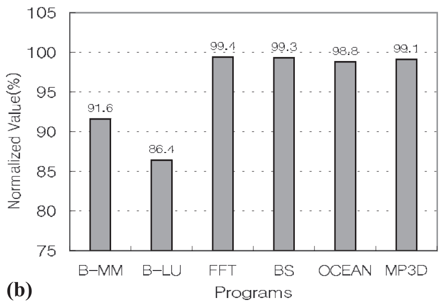
5.2. Multiple blocked programs

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. Fig. 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.

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 Figs. 6(a) and 8(a). The reason is that the higher the programming level used, the more the 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, more cache misses in the preemption policy result in more execution time, and the normalized values are diminished as the programming level is increased.



(a)



(b)

Fig. 9. Performance of programs on programming level 6: (a) execution times of all programs; (b) cache misses of all programs.

As shown in Fig. 9(b), the preemption-safe policy results in the reduced cache misses even in nonblocked 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.

6. Related work

Previous research on blocking has been done on evaluating the performance of blocked algorithm [1,2]. Lam [1] 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. Lebeck [2] suggested a visualized tool called cache profiling (CPROF) that classified cache misses at the source line and data structure level. In particular, this study showed that blocking technique can eliminate cache capacity misses by processing data in portions that fit in the cache. 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 [12–14]. In particular, Mogul and Borg [14] 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 for the interaction between scheduling strategies and cache performance of programs. To improve the cache utilization for a given scheduling policy, cache affinity scheduling has been introduced to exploit cache footprints [15–17]. Squillante [16] pointed out that the effect of the cache affinity scheduling depended on the program characteristics via an analytic model for cache memory operations. On the other hand, Vaswani [17] reported that the processor affinity had only a weak influence on the choice of scheduling discipline even for a multiprogramming system. These previous 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 the previous cache affinity scheduling, even if the process is rescheduled into the same processor, intervening processes may have overwritten some or all of the cache data. However, our proposed scheduling not only exploited the advantage of the cache affinity scheduling based on the running of the same processor, but also utilized the intrinsic locality of blocked programs through the limited delay of context switching.

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 [18–20]. In particular, Kontothanassis [20] 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 nonpreemption 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 nonpreemption policy was used to avoid the preemption of the process with an active lock [20]. 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 of computing a block. Also, this method guaranteed the fair usage of processor time via compensating the processor time preused 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 delayed 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 delayed period of context switching does not severely affected programs' response time, because this policy can enhance the overall throughput for a given system.

Acknowledgements

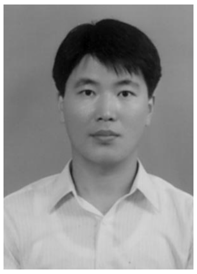
This work was supported in part by National Research Laboratory Program funded by Ministry of Science and Technology and university S/W research center program by Ministry of Information and Communication, Republic of Korea.

References

[1] M.S. Lam, E.E. Rothberg, M.E. Wolf, The cache performance and optimizations of blocked algorithms, in: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1991, pp. 63–74.
[2] A.R. Lebeck, D.A. Wood, Cache profiling and the SPEC benchmarks: a case study, IEEE Computer 27 (10) (1994) 15–26.
[3] D.A. Patterson, J.L. Hennessy, Computer a Quantitative Approach, second ed., Morgan Kaufmann, Los Altos, CA, 1996.
[4] I. Jung, J. Lee, Techniques for improving the cache performance in parallel applications, Eleventh IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'99), 1999, pp. 597–602.
[5] J.E. Veenstra, R.J. Fowler, MINT: a front end for efficient simulation of shared-memory multiprocessors, in: Proceeding of Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 1994, pp. 201–207.
[6] J. Archibald, J.-L. Baer, Cache coherence protocols: evaluation using a multiprocessors simulation model, ACM Transactions on Computer Systems 4 (4) (1986) 273–298.
[7] J.K. Ousterhout, Scheduling techniques for concurrent systems, in: Proceedings of the Third International Conference on Distributed Computing Systems, 1982, pp. 22–30.
[8] C. Small, 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.
[9] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing (Design and Analysis of Algorithms), The Benjamin/Cummings, Menlo Park, CA, 1994.
[10] J.P. Singh, W.D. Weber, A. Gupta, SPLASH: stanford parallel applications for shared-memory, Computer Architecture News 20 (1) (1992) 5–44.
[11] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, in: Proceedings of the 22nd Annual International Symposium on Computer Architecture, 1995, pp. 24–25.
[12] A. Agarwal, J. Hennessy, M. Horowitz, Cache performance of operating system and multiprogramming, ACM Transactions on Computer Systems 6 (4) (1988) 393–431.
[13] C.B. Stunkel, W.K. Fuchs, TRAPEDS: producing traces for multicomputers via execution driven simulation, in: Proceedings of ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 1989, pp. 70–78.
[14] J.C. Mogul, A. Borg, The effect of context switches on cache performance, in: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1991, pp. 75–84.
[15] J. Torrellas, A. Tucker, A. Gupta, Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors, Journal of Parallel and Distributed Computing 24 (2) (1995) 139–151.
[16] M.S. Squillante, E.D. Lazowska, Using processor-cache affinity information in shared-memory multiprocessor scheduling, IEEE Transactions on Parallel and Distributed Systems 4 (2) (1993) 131–143.
[17] R. Vaswani, J. Zahorjan, The implication of cache affinity on processor scheduling for multiprogrammed, shared-memory multiprocessors, in: Proceedings of the 13th ACM Symposium on Operating System Principles, 1991, pp. 26–40.
[18] J. Zahorjan, E.D. Lazowska, D.L. Eager, Spinning versus blocking in parallel systems with uncertainty, in: Proceedings of the International Seminar on Performance of Distributed and Parallel Systems, 1988, pp. 455–472.
[19] A. Tucker, A. Gupta, Process control and scheduling issues for multiprogrammed multiprocessors, in: Proceedings of the 12th ACM Symposium on Operating System Principles, 1989, pp. 159–166.
[20] L.I. Kontothanassis, R.W. Wisniewski, M.L. Scott, Scheduler-conscious synchronization, ACM Transactions on Computer Systems 15 (1) (1997) 3–40.

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, 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 M.S. 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., 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, 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.