Two-Level Multi-Scan Scheduler Using Resource Partition Strategy by Loose Processor-Affinity

Jong-Moon Sohn and Gil-Yong Kim

Abstract

The performance of a shared memory multiprocessor system is very sensitive to process scheduling. We can enhance the performance of a whole system as well as of an individual process by taking the multiprocessor characteristics into account in the design of the process scheduler. In this paper, we propose a general purpose scheduler for a shared memory multiprocessor, called the Two-Level Multi-Scan (TLMS) process scheduler, that considers the processor affinity loosely and decreases the interference among multiple processors greatly. The TLMS scheduler is composed of a local scheduler at each processor and a semi-global scheduler that balances the load among processors. In particular, the semi-global scheduler tries to minimize priority inversion, which is an important factor of the system performance. The TLMS scheduler also tries to reduce the number of resources to be shared and improves the processor utilization. To meet these requirements, the semi-global scheduler interacts with the operation of the local scheduler when a need arises, thus the name is loose processor-affinity. We also show that the proposed scheduling technique can be extended for other types of resources making it a general purpose resource management queue.

I. Introduction

A medium-scale multiprocessor typically consists of a modest number of processors connected to main memory by a shared bus. This architecture is used for machines ranging from single-user workstations to enterprise-level computers, and provides a building block for large-scale shared memory multiprocessors[9]. But there is a limit on the number of processors that can be supported in this type of multiprocessor because of the interference among multiple processors. Generally, the bottleneck determines the range of the numbers of processors varying from 10 to 30.

The performance of a shared memory multiprocessor system is heavily dependent on the process scheduler. There are a number of requirements that the process scheduler should satisfy. Those requirements include the implementation of light weight process (LWP), processor-set, binding mechanisms of process to a specified processor, user-mode synchronization primitives, cache affinity mechanism. A conventional OS satisfies almost all requirements. Nevertheless, it is rare to have a general-purpose scheduler optimized for a multiprocessor.

In this paper, we propose an enhanced scheduling structure and policy. The focus of this paper is on the performance bottleneck of the operating system caused by competition for shared resources, which limits the scalability and throughput of the system. In particular, this paper describes a design and analysis of a scheduler called the Two-Level Multi-Scan(TLMS) process scheduler based on the modified processor affinity scheduling. The TLMS scheduler is a general purpose scheduler composed of a local and a semi-global scheduler. Thus we do not consider any relationship between processes. The local scheduler is based on the processor-affinity scheduling concept. Each processor has its own ready queue and associated processes to decrease the interference among the processors. The semi-global scheduler balances the load among processors semi-globally to prevent a potential degradation of processor utilization. It uses an algorithm called the Multi-Scan algorithm to prevent the priority inversion. To analyze the performance the TLMS scheduler, we measured the system scalability and the lock contention ratio while running several benchmarks. The results indicate that Two-Level Multi-Scan scheduler gives a nice trade-off between fairness(sharing) and resource utilization(no-sharing).

This paper is structured as follows: In Section 2, we describe previous works on multiprocessor scheduling. Section 3 gives a detailed explanation of the Two-Level Multi-Scan scheduler, and Section 4 describes the multiprocessor hardware used to implement the TLMS scheduler. In Section 5, we describe the measurement data for the scheduler, and discuss the results. Section 6 explains the applicability of the scheduling mechanism for other resources. Section 7 concludes the paper.

Manuscript received August 21, 1996; accepted April 8, 1997.

J. M. Sohn and G. Y. Kim are with Department of Computer Engineering, Pusan National University.

Ⅱ. Related Works

Processors are an expensive resource that is shared among processes. This sharing requires a balance between fairness and resource utilization that should be achieved through multiprocessing. To be effective, however, the multiprocessing overhead must be minimized. There are several potential sources of the overhead. These overheads include the context switching among applications, waiting time incurred by preemption, and various migration costs associated with moving a process from one processor to another[1].

The multiprocessor schedulers can be classified as either time multiplexing or space multiplexing. Typical schedulers are round robin(an unsynchronized time sharing), gang(a synchronized time sharing), and resource partition(space sharing) schedulers. It have been known that the synchronized time sharing and the space sharing are among the best performing strategies for multiprocessors[1].

The round robin strategy is a static policy that time-shares processors among the processes in the system. This is a straightforward generalization of the uniprocessor round robin scheduler. The strategy is very useful for load balancing because we do not need to decide where a process should be run. However, since we do not know where a process will be executed, computing power can be wasted. It is because the process migration and preemption incur overheads, leading to performance penalties. This degrades the performance severely. An earlier study[2] shows the effect of the performance degradation incurred by the round robin scheduler. Most earlier multiprocessor operation systems had a single ready queue. This approach is very popular especially in UMA multiprocessors. A centralized ready queue can introduce a performance penalty of up to 99%, with 69% due to cache reload and 30% due to increased bus traffic and contention resulting from the affinity between a process and a processor[4].

Gang scheduling was introduced by Ousterhout in the context of the Medusa system[5]. The basic idea is to run all processes belonging to an application at the same time, so that processes do not need to be busy waiting for a resource held by a preemptive process. So, it coordinates a context switching across a number of processors so as to schedule a "gang" of interacting threads simultaneously. Gupta et al.[5] simulated gang scheduling on a cached bus-based machine and concluded that the gang scheduling is one of the best scheduling approaches, because the busy waiting is optimized[8]. However, they have the following disadvantages. First, the gang scheduling has poor cache performance. If there are several applications in the system, the machine must cycle through each of them, during which a substantial amount of cache state is cost[6]. Second, the utilization of a processor can be degraded if applications have a variable amount of parallelism, or if processes are not fairly assigned to time-slices of the computing power[1]. Third, the gang scheduling is not efficient to support fine-grain interactions: special hardware support is required to make interactions fast[8]. Lastly, whenever a processor allocation takes place, it is difficult to provide application with enough processors.

Processor sharing strategy does not share a processor among applications. Fig. 1 shows a processor set that consists of multiple processors. Once a processor set is fixed, it cannot execute a process of another processor set. The set may be dedicated to a group of processes for the fixed interval or for the entire duration of the running. If the set has only a single processor, we can avoid considerable overheads of context switching cost, waiting time, and migration cost. Unfortunately, to improve the performance by processor sharing strategy, there are a number of restrictions. First, to ensure fairness and utilize the processors effectively, the number of processors assigned to an application have to change when applications arrive or depart[6]. Second, the performance gain of a higher cache hit ratio is obtained when system has a large physical cache. Third, in case of an overload condition we cannot allocate processors to applications on demand. Lastly, even if a processor or a processor set is dedicated to a single application, all processes within the system may be related to each other because processes may share unpartable resources. Since processor scheduling has a strong relationship with other resources, we cannot eliminate the whole overhead. Zahorjan et al.[11] showed that dynamic resource partition is the best scheduling policy based on a simulation study of resource partition with variable parallelism. It is due to reallocation of unused processing power immediately.

The approach of McKenney et al.[13] to speed up a memory allocator is similar to that of TLMS. McKenney proposed a 4 layered allocator. The allocator consists of a per-CPU caching layer, a global layer, a coalesce-to-page layer and a coalesce-to-"vmblk" layer. Among these, the per-CPU caching layer supports high-speed allocation and deallocation in the most common case. Each CPU maintains a local cache of buffers. When a per-CPU cache is exhausted, it is replenished from the global layer. When it becomes to full, the excess is put back into the global layer. However, this approach is different from ours in that the latter considers the priority and eliminate the global resource pool. If a resource must be managed according to the priority, the McKenney's approach is not applicable.

Ⅲ. The Two-Level Multi-Scan Scheduler

Fig. 2 depicts the conceptional structure of the TLMS scheduler. The scheduler is made up of a high-level scheduler(a semi-global scheduler) and a low-level scheduler(a local scheduler). The local scheduler manages only the processes that have affinity, and is run at a associated processor. The semi-global scheduler balances the load among multiple processors to increase utilization

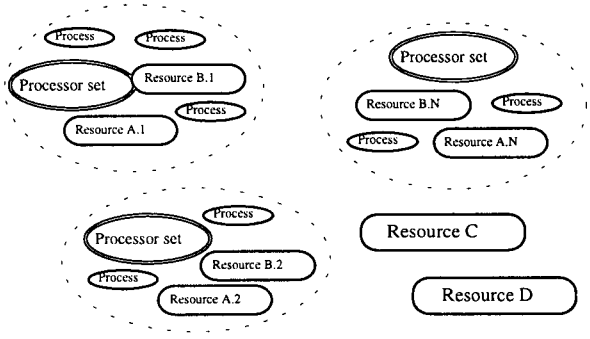


Fig. 1. Resource partition.

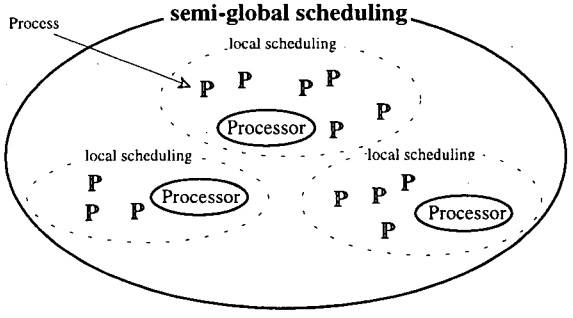


Fig. 2. The conceptional structure of the TLMS scheduler.

of processors. This two-level structure associates processors loosely. A local scheduler emphasizes the processor-partition and a global scheduler does the processor-sharing.

1. The Local Scheduler

The local scheduler is associated with each processor, and schedules the processes in the processor's ready queue. The high level scheduler(the semi-global scheduler) designates the affinity-processor for a process. The affinity-processor of a process is not changed until the next semi-global scheduling point.

A process has an affinity with a processor because of the data resident in the processor's cache from a previous dispatch[7]. The purpose of the earlier processor-affinity scheduling was to enhance the cache performance, and the focus was on how much we can reduce the cache miss ratio. There is an add-on benefit of the processor-affinity scheduling. Because the scheduler makes a process run only at the designated processor, a processor can be managed independent of other processors. Thus, we can reduce the synchronization cost and the interference among multiple processors.

The local scheduler belongs to the general operating system schedulers known as the round robin. Fig. 3 shows the ready queue of each processor and the process that has the affinity with the processor. The solid line represents the migration path of a process when the local scheduler is executed. The dotted line denotes the fact that a process can be migrated to an alternative ready queue by semi-global scheduling.

```
1   local_scheduler()
2   {
3       /* A new process is given by the semi-global scheduler */
4    loop:
5
6       if (there is a process in my run queue) {
7             lock my run queue;
8             schedule the process in a round robin way;
9             dequeue process and set flag value;
10            unlock my run queue;
11            execute the process;
12                  goto loop;
13      }
14      else {
15            invoke semi-global scheduler to get a process;
16            if (there isnt a process to execute)
17                  idle;     /* wake up by the interrupt */
18            else
19                  execute the process;
20            goto loop;
21      }
22  }
```

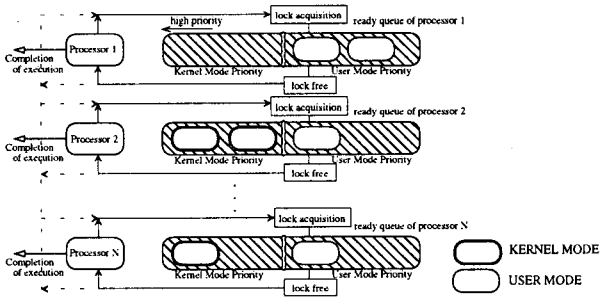


Fig. 3. Ready queue of TLMS scheduler.

2. The Semi-global Scheduler

While the space sharing by the local scheduler reduces the competition for shared resources, it does not use the computing power efficiently. As the autonomous scheduling is so much emphasized to reduce the interference among processors, the performance of the whole system is degraded due to the negligence of resource management in a system-wide aspect. The following problems can potentially occur when we emphasize only the space sharing. The first is that the local scheduler partitions the scheduling resources and performs the autonomous scheduling without considering the performance of the whole system. It cannot perform the load balancing because it uses only the ready queue of an associated processor. Therefore, the processor will become idle when its ready queue is empty while other processors are busy. Fairness is not guaranteed in such a system. The second

is the limitation of the UNIX dispatcher, which can be described as an idle processor seeking a process to run. Instead of an affinity-processor, sometimes it is necessary for a process to seek an idle processor[7].

The major functions of semi-global scheduler are the processor allocation for processes and the management of idle processors. Whenever a process is created, the processor allocation function is invoked. The function designates the next processor of the previously allocated one. It is just like the circular queue operation. The idle processor management function is invoked when the processor has no executable process. The semi-global scheduler refers the next ready queue in a circular manner if the ready queue has no process to schedule, that is, a process to be executed is brought from the other processor's ready queue. It is optimum if the selected process has the highest priority among the existing ones. But, choosing the highest priority is impractical. That is why the scheduler looks up all run queue globally.

The scheduler searches for a process "semi-globally". When the scheduler refers ready queues sequentially, the first available process can be moved to the idle processor. At that time, the most important feature is to prevent the priority-inversion. The cost of the priority inversion is expensive if the process is executed in the kernel mode. The priority inversion of a process executing in user mode does not affect the whole system's performance. The priority inversion of a kernel-mode process is severe because the process has connection with the system resources. The resources are prioritized by the order that interrupts should be handled. The processes executing in the kernel mode sleep and wake up according to that priority. This increased the utilization of the resources. Therefore, the priority inversion problem reduces the resource utilization because the lower priority process executing in the kernel mode can disturb the higher priority one.

In this paper, we scan the run queue multiple times. Each time a scan occurs, a range of scanning is specified. For example, if we scan the queue twice, the first scan can be for kernel mode and the second scan for both kernel mode and user mode. Fig. 3 shows this 2-scanned ready queue. If a context switch occurs on processor 1, the following procedure is needed to dispatch a new process. First, processor 1 scans its ready queue of kernel mode priority. Since processor 1 does not have kernel mode process, it investigates the ready queue of processor 2 with kernel priority range in next step. In this case, processor 1 gets a kernel mode process from the ready queue of processor 2. If there is no kernel mode process within all ready queues, processor 1 investigates all ready queues with kernel priority range and the first scan is finished. Because any processor can acquire kernel mode process during the first scan, the scheduler extends the scanning range to the kernel and user mode. Finally, processor 1 gets a new user mode process from its ready queue. Fig. 4 shows the data structure to support this operation. The solid box denotes private

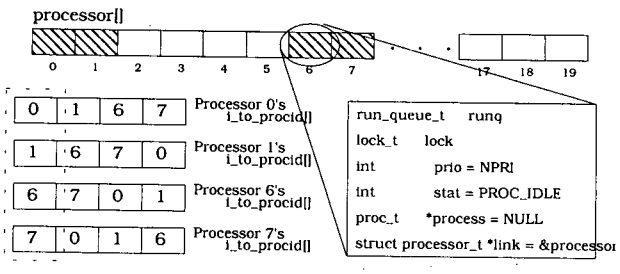


Fig. 4. Data structure for TLMS scheduler.

scheduling information of each processor. "runq" is private ready queue. "state" and "process" are for handoff scheduling. "i_to_procid" is a local memory array to speed up the indexing of processor. The following algorithm is used in the dispatcher of the semi-global scheduler with the 2-scan technique.

```
1   semiglobal_dispatcher()
2   {
3     search = 1, search_range = KERNEL_MODE;
4
5   second_search:
6     if (current processor's run queue count > 0) {
7       lock current run queue;
8       for (priority queues in search range) {
9         if (process is adequate) {
10          dequeue process and set flag value;
11          set local memory variable;
12          unlock current run queue;
13          return(p);
14        }
15      }
16      unlock current run queue;
17    }
18    for(each processor's run queue in system) {
19      if (this processor's run queue count > 0) {
20        lock this run queue;
21        for (priority queues in search range) {
22          if (process is adequate) {
23            dequeue process and set flag;
24            set local memory variable;
25            unlock this run queue;
26            return(p);
27          }
28        }
29        unlock this run queue;
30      }
31    }
32    if (search == 1) {
33      search++;
34      search_range = KERNEL_MODE | USER_MODE;
35      goto second_search;
36    }
37    return(NULL);
38  }
```

Ⅳ. The Environments of Experiment

This section outlines some of the features of our environments of experiment and of the benchmarks used. All of our results of experiment were measured in the operating system (a UNIX System V-based operating system) run on the TICOM shared memory multiprocessor. The TICOM hardware in Fig. 5 consists of a shared system bus, a system control, several processor boards, memory boards, I/O processor boards, and peripheral devices. It consists of ten 20 MHz MC68030 CPUs and main memory of 512 Mbytes. The operating system used is compatible with the UNIX System V. The TICOM with symmetrical multiprocessing feature can execute an interrupt, kernel and user code in any processor. It is parallelized to allow more than one processor to execute the kernel code at the same time. Locks ensure exclusive access to system resources. There are two types of locks in the operating system : semaphore lock and spin lock. And there is analogy between the TICOM process scheduler and the general round robin scheduler. This is why the TICOM process scheduler is derived from the process scheduler of the UNIX System V for a uniprocessor.

As we increasing the number of processors, we measured overall performance using the macro benchmark called AIM III. The AIM III can measure the system performance according to the application areas. It prescribes 31 types of performance test and 8 application areas. Each application area has its own workload profile. Thus, we can simulate the specified environment by writing the application areas and the simulated user number. For example, if we write as "word processing--60% and database management--40%", the AIM III generates the 60% of the word processing load and the 40% of the database management load. At completion, the AIM III reports the measured performance data. The measurements can be used to estimate the performance and the expandability under the real environments, and also to compare with other different systems. The measured data are the

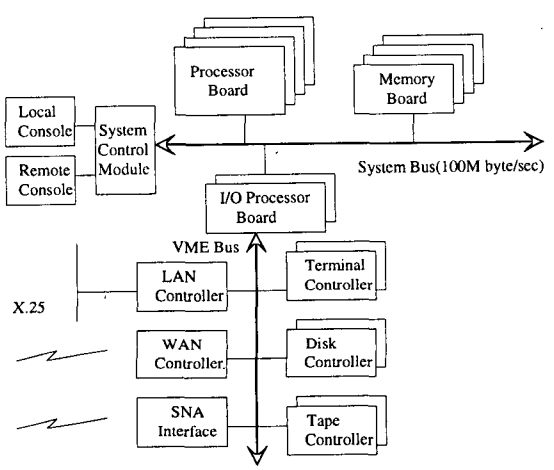


Fig. 5. The TICOM multiprocessor system.

number of jobs executed per unit time(job/min), user throughput (job/sec/user), and elapsed time(sec).

The micro benchmarks were also used to observe the specific functional behavior of the kernel and to stress the specific locks. Those consist of several programs[10]. We discuss the only three pertained to stress file reading(Read), context switching(Cswtich), and process fork and exec(Fexec). The Read benchmark measures the performance of file buffer cache by opening the file and reading the file repeatedly. It accesses the disk and allocates buffer cache. It is an I/O-bound job. We use the Cswitch benchmark to measure context switching cost by spawning a child process and by passing one byte between the parent and child using two pipes. Whenever they send the data to each other, two context switchings are involved. The Fexec benchmark repeatedly spawns a child process, executes the exec system call, and exits. It measures the cost of process creation and destruction, virtual memory subsystem, context switching, and reading some text from the disk to the memory.

Ⅴ. Analysis

The performance metrics are the lock miss ratio and the number of virtual users. The lock miss ratios are the ratios of unsuccessful attempts to total attempts in acquiring locks. It was measured while running the AIM III. The number of virtual users is a convenient metric. The AIM generates a unit of workload per virtual single user. The number of virtual users represents the maximum workload that the system can sustain.

The lock miss ratios of the TICOM scheduler are shown in Fig. 6. The "Original kernel" and "Enhanced kernel" in Fig. 6 are plots of the lock miss ratios for both schedulers of them. In case of comparing with the lock miss ratios of the two schedulers, the local scheduler solves the competition for the single shared lock. That is, as the local scheduler does not share the ready queue and the scheduling code, we can reduce the synchronization cost for the shared resource. The more important fact is that the TLMS scheduler increases the expandability of the system. The graph of the "Original kernel" is similar to that of the logarithms. The lock miss ratios increase dramatically when we increase the number of processor from 4 processors to 6, and increases incrementally thereafter. In the system with 10 processors, the lock becomes saturated. According to the results of the kernel profiling, 20% of the processing power is wasted due to the busy waiting. On the other side, the graph of the "Enhanced kernel" almost runs parallel to the horizontal axis. This proves that the processor-affinity concept of the local scheduler distributes the lock contention to each processors ready queue and each processor uses its own ready queue possible. The plots show that in the TLMS scheduler the lock saturation and the overhead caused by the lock miss does not occur. In other words, the TLMS scheduler allows the maximum utilization of cpu power.

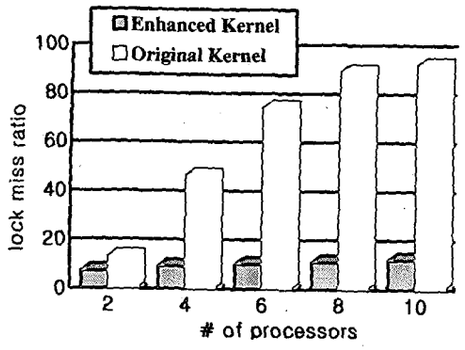


Fig. 6. The miss ratios of scheduler lock.

The process scheduler is executed by waking up the process waiting for I/O service completion as well as expiring the quantum. The process waiting for I/O service completion is activated by vsema(V semaphore operation). Fig. 7 depicts the execution steps of waking up a process. According to our measurements, the execution of the interrupt handler has delayed for 0.1msec or more to acquire the scheduler lock in an earlier implementation of the scheduler. Thus we need to highly parallelize the scheduling code if possible.

Fig. 8 shows the number of virtual users(the maximum load). This graph shows that the reduction of the synchronization cost does not solve the whole problem. The graph of the "Enhanced kernel (2)" shows the performance gain of the TLMS scheduler with Multiple-Scan(2-scan). The reduction of the competition for shared resources directly affects the performance of the system. The "Enhanced kernel (1)" of Fig. 8 shows the performance of the TLMS scheduler without Multi-Scan. Because the TLMS scheduler is performed without Multi-Scan, the "Enhanced kernel (1)" does not prevent the priority inversion. The wasted time by the priority inversion offsets the performance improvement, and the TLMS scheduler without Multi-Scan shows lower performance than the original scheduler. In this reason, we have measured the performance of 2, 4, and 6 processors. The graph of Enhanced Kernel(1) indicates that the performance improvement of a multiprocessor system needs to consider the global utilization and the fairness.

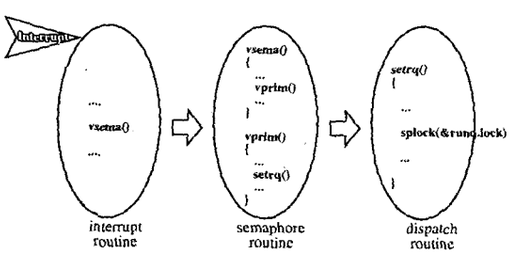


Fig. 7. The execution stream of scheduling code by I/O completion.

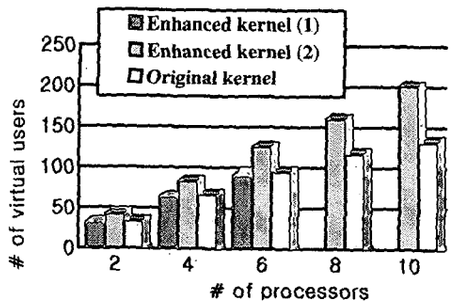


Fig. 8. The system scalability represented by the number of the virtual user.

The Multi-Scan cannot prevent the priority inversion completely. The priority inversion still occurs because the scanning is performed within the definite range. We have measured the performance of the 2-scan and 3-scan techniques in order to understand the effect of the priority inversion. This experiments were performed on the TICOM with 4 processors. First, we have measured the data gained from the micro benchmarks. The numbers in Table 1 are the fraction of the 2-scan technique's elapsed time. Since the Fexec and Read processes are related to disk interrupts, the 3-scan of the Fexec and Read have better performance than that of the 2-scan. In the case of the Read (extreme I/O bound-program), the performance gain is about 20%. Thus, it is clear that the priority inversion is critical to the I/O-bound process. The Cswitch gets a little longer elapsed time when the 3-scan technique is used. The performance loss is 3%. This effect may be due to the repeated search for the ready-queue. Second, the elapsed time is measured using the macro benchmark in Table 2. The 5, 10, and 30 of Table 2 are the virtual users that the macro benchmark performed. Because the context switching rate is much higher than the file reading, overall performance of the two-scan technique is not degraded by the priority inversion.

Table 1. The elapsed time ratios of 3-Scan technique to 2-Scan technique.

	Original	2-Scan Technique	3-Scan Technique
Cswitch	1.4	1	1.03
Fexec	1.3	1	0.99
Read	1.38	1	0.78

Table 2. The elapsed time(sec).

	2-Scan	3-Scan
5	208	216
10	394	409
30	1194	1193

VI. The Adaptability of Two-Level Multi-Scan Policy

Earlier researches[2, 10] have recommended parallelization of the data structure and algorithm managing the resource. But, parallelization of resources have been mostly dependent on the skill of the engineer because there is no general policy to apply. We also propose the two-level multi-scan scheduler as the general resource management policy on multiprocessors.

A Two-Level policy has been applied to the management of the free buffer cache and free incore inode. In the case of free buffer, its list is structured as Fig. 9. Each processor has its private free buffer list. This structure is similar to that of the private ready queue. The free buffer list is managed as follows.

1) Free buffers are allocated to the head of the free buffer lists when the system boots up. Each free buffer has an affinity with a designated free list.

2) The free buffer belonging to a free list cannot change affinity free list.

3) Each processor scans only its free buffer when it needs a free buffer.

4) Only if its free buffer is empty, the processor references the free lists of other processors in a circular manner and borrows a free buffer from one of the free lists.

5) The free buffer borrowed from another processor is returned to its affinity processor after use.

Fig. 10 shows the performance and scalability of the TLMS policy. "Enhanced kernel(2)" is the 2-scan TLMS process scheduler's data as described before. And, "Enhanced kernel(3)" is the graph of the 2-scan TLMS process scheduler with the Two- Level (TL) buffer management policy. When we compare the two graphs, the TL buffer management policy enhanced the scalability as well as performance. The slope(average 0.98) of "Enhanced kernel(3)" is steeper than that(average 0.95) of "Enhanced kernel(2)".

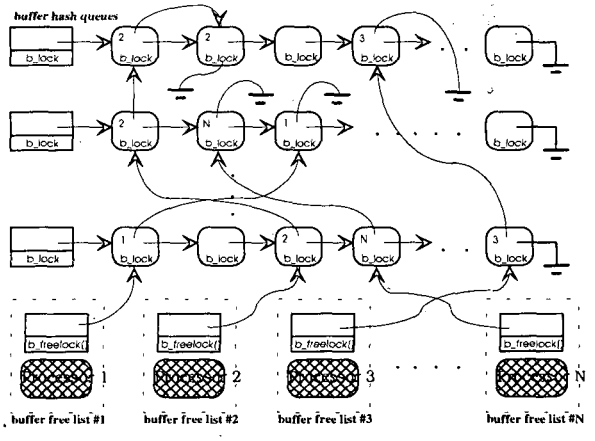


Fig. 9. Data structure of buffer cache.

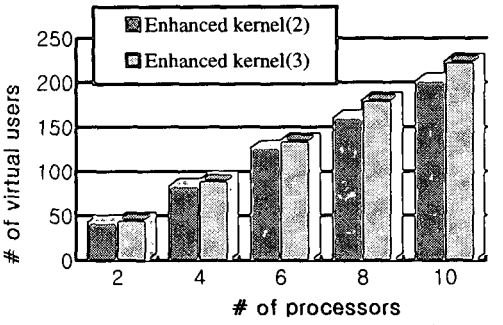


Fig. 10. The system scalability comparison between TLMS scheduler and TLMS scheduler + TL free buffer management.

According to the previous experiments and results, we can constitute the Affinity-Group(AG) that allows parallelized access to the resources on a general-purpose multiprocessor. AG consists of one processor and the parts of the several resources like Fig. 1. We can divide the partable resources and give each processor. There is no doubt that we can apply the TLMS technique to the partable resources. The part of the resource is allocated to the processor when the system starts up. The Affinity-Group policy consists of a 2-phase management procedure. In the first phase, an Affinity-Group has a relation with a processor. The resources of the AG are locally managed. The processor uses only its own Affinity-Group's resources. Each resource can employ the existing management algorithm without modification because the individual Affinity-Group is like uniprocessor environment. In the second phase, we prevent the decrease of the resource utilization. If the autonomous scheduling is so much emphasized in order to reduce the interference among processors, the performance of the whole system becomes low owing to negligence of the resource management in the system-wide aspect. To ensure fairness, sharing the AG of another processor should be performed only when there is nothing to use in its AG. This procedure goes with the scanning the AGs of other processors in order to choose the appropriate resource. If a resource has the priority, we scan the resource pool several times(Mutli-Scan). Since the priority inversion is very severe, it is very important.

VII. Conclusion

The measurements obtained from the benchmarks indicate that the TLMS scheduler gives an acceptable performance on a multiprocessor system by decreasing the competition for shared resources, and that the system can speed up linearly as the number of processors increases. In our environments of experiment, the TICOM system with 10 processors, the lock miss ratios for the scheduler lock were reduced from 92% to 12% and showed about 50% performance gain.

This paper shows that the processor-affinity scheduling can be used to reduce the competition for shared resources, and the

semi-global scheduling can be used to increase the utilization of resources on the system-wide aspect. The Multi-Scan is important to resolve the priority inversion problem because the global view increases the resource utilization effectively. In multiprocessor environments, it is desirable that the resource schedulers(process scheduler, buffer management, etc.) should provide a method of resource partition(local scheduling) and resources sharing(global or semi-global scheduling).

Since our study was measurement-oriented, there are many possible extensions to this work. As the cache speed and size increase, and the number of processors increases, simulation methods are needed to analyze the limitation of the benefits of processor-affinity scheduling. The TLMS scheduler developed in this research can be applied to other resource management policies to increase the system performance and scalability.

References

[1] Mark Crovella, Prakash Das, Czarek Dubnicki, Tomas LeBlanc, Evangelos Markatos, "Multiprogramming on Multiprocessors", The University of Rochester, Technical Report 385, 1991.

[2] Sang Lyul Min, Gil Yong Kim, "Bottleneck Analysis of a Multiprocessor UNIX Kernel", Journal of The Korea Information Science Society, Vol. 20, No. 10, Oct. 1993.

[3] S. Leutenegger, "Issues in multiprogrammed multiprocessor scheduling", Ph.D. thesis, University of Wisconsin/Madison, August 1990, Technical Report 954.

[4] M. S. Squillante, "Issues in Shared-Memory Multiprocessor Scheduling: A Performance Evaluation", Technical Report 90-10-04, University of Washington, Oct. 1990.

[5] A. Gupta, A. Tucker, 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 Measurements and Modeling of Computer Systems, May 1991.

[6] A. Tucker, A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessor", Proceedings of 1987 International Conference on Parallel Processing, pp. 245-254, August 1989.

[7] M. Devarakonda, A. Mukherjee, "Issues in Implementation of Cache-Affinity Scheduling", Proceedings of Winter 1991 USENIX Conf. 1992, pp. 345-358, Jan. 1992.

[8] Dror G. Feitelson, Larry Rudolph, "Gang Scheduling Performance benefits for Fine-Grain Synchronization", Journal of Parallel and Distributed Computing, Vol. 16, Num. 4, pp. 306-318, Dec. 1992.

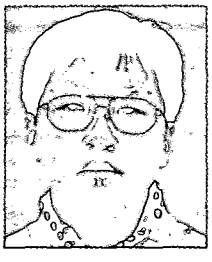
[9] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "Design of Scalable Shared-Memory Multiprocessors: The DSH Approach", Proceedings of 35th IEEE Comp. Soc. Intl. Conf. -- COMPON 90, Feb. 1990.

[10] J. H. Hartman, J. K. Ousterhaut, "The Performance Measurement of a Multiprocessor Sprite Kernel", Proceedings of Summer 1990 USENIX Conference, pp. 279-287, June 1990.

[11] J. Zahorjan, C. McCann, "Processor Scheduling in Shared Memory Multiprocessors", Proceedings of 1990 ACM SIGMETRICS Conference on Measurements and Modelling of Computer Systems, pp. 214-225, May 1990.

[12] Paul E. McKenney, Jack Slingwine, "Efficient Kernel Memory Allocation on Shared-Memory Multiprocessors", Proceedings of the Winter 1993 USENIX Conference, pp. 295-306, Jan. 1993.

[13] Ben Verghese, Scott Devine, Anoop Gupta and Mendel Rosenblum, "OS Support for Improving Data Locality on CC-NUMA Compute Servers", Stanford University, CSL-TR-96-688, Feb. 1996.



Jong-Moon Sohn received his B.S. and M.S degrees in computer engineering from Pusan National University, Korea in 1991 and 1993, respectively. He is currently work ing toward the Ph.D. degree in computer engineering at Pusan National University. His research interests include real-time scheduling, operating system, distributed multi-media system.

Gil-Yong Kim received his M.S. and Ph. D. degree in computer engineering from Seoul National University, Seoul Korea in 1983 and 1988, respectively. He is currently an associate professor of Computer Engineering Department at Pusan National University, Pusan, Korea, since 1988. Before joining the P.N.U., he was a researcher at the GoldStar Semiconductor Co. from 1983 to 1986. His research interests include parallel and distributed systems, real-time systems, and multimedia systems. He is a member of IEEE and ACM.