Distributed-Thread Scheduling Methods for Reducing Page-Thrashing

Yoshiaki Sudo[1] Shigeo Suzuki Shigeki Shibayama
Media Technology Laboratory, Canon Inc.
Kawasaki-shi, Kanagawa 211, Japan
{sud,suzuki,bashi}@cis.canon.co.jp

Abstract

Although distributed threads on distributed shared memory (DSM) provide an easy programming model for distributed computer systems, it is not easy to build a high performance system with them, because a software DSM system is prone to page-thrashing. One way to reduce page-thrashing is to utilize thread migration, which leads to changes in page access patterns on DSM.

In this paper, we propose thread scheduling methods based upon page access information and discuss an analytical model for evaluating this information. Then, we describe our implementation of distributed threads, PARSEC (Parallel software environment for workstation cluster). Using user-level threads, PARSEC implements thread migration and thread scheduling based upon the page access information. We also measure the performance of some applications with these thread scheduling methods. These measurements indicate that the thread scheduling methods greatly reduce page-thrashing and improve total system performance.

1. Introduction

A distributed shared memory (DSM) system provides shared memory abstraction to a program running on multiple computers that do not physically share memory. Distributed threads on DSM provide an easy programming model for distributed computer systems, since the underlying DSM system takes care of communications and synchronization. However, the system tends to cause false-sharing, because a software DSM system uses a page as a unit of memory consistency. When distributed threads frequently access truly-shared or falsely-shared data, page access contention occurs, which results in heavy network traffic. This situation, called page-thrashing, is one of the most serious problems with DSM.

[1] Author's current address: Department of Computer Science, Princeton University, Princeton, NJ 08544, U.S.A., sud@cs.princeton.edu

This paper proposes thread scheduling methods for reducing page-thrashing and presents an analytical model of correlation between two threads based upon the page access information. We describe the design and implementation of the thread scheduling on our distributed thread execution environment called PARSEC [20, 19] (Parallel software environment for workstation cluster).

Many DSM systems [3, 9, 2, 14, 11] focus mainly on relaxed memory consistency models (e.g. weak consistency [6], release consistency [8], lazy release consistency [10], and entry consistency [2]) in order to solve the problem of page-thrashing. However, even if the system employs such a relaxed memory consistency model, it just lowers the risk, and numerous messages are still required to maintain consistency. Once page-thrashing occurs, the system performance drops immediately. That is, these systems still lack a means of reducing page-thrashing.

Page-thrashing in DSM results from overlapping working sets of threads. In order to reduce page-thrashing, we should ensure that threads with overlapping working sets are not running on different machines. This is our first thread scheduling policy. However, it is impossible to predict the accurate working set of a thread. Therefore, it is important that the system also provides a mechanism to stop page-thrashing, in case the prediction fails and page-thrashing does occur. Our second thread scheduling policy is to suspend threads that cause page-thrashing, which is analogous to the page-fault frequency algorithm [4].

This paper is organized as follows. Section 2 focuses on the thread scheduling methods proposed here. Section 3 gives an overview of PARSEC. Section 4 evaluates the performance of the thread scheduling methods. Section 5 discusses related work. Finally, our conclusions and future work are presented in Section 6.

2. Distributed-thread scheduling

In this section, we discuss the page access information, especially correlation between threads, using an analytical model, and then describe the details of the two thread

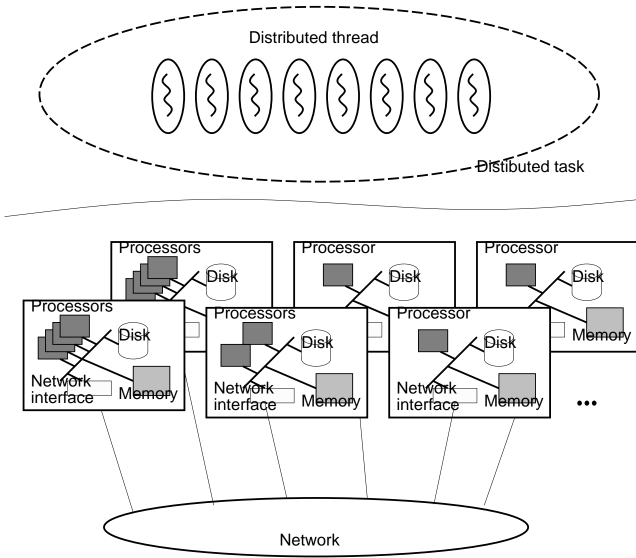

Figure 1. Distributed thread execution system model.

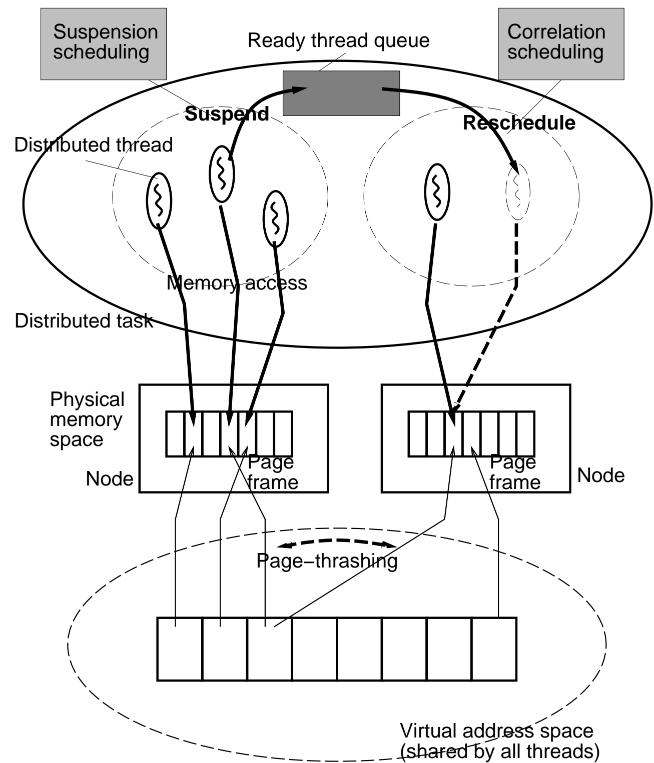

Figure 2. Two scheduling methods to reduce page-thrashing.

scheduling methods.

Figure 1 shows the system model of a typical distributed thread execution environment, where homogeneous machines are connected by a network.

Each task has an address space and multiple threads of control sharing the space. This programming model is called a distributed task/thread programming model and is easily implemented using DSM. The model makes it easy to write distributed programs and allows existing multiprocessor programs to run on the distributed environment without any modification.

Furthermore, if it has a thread migration mechanism, the system can dynamically move a thread to another machine in mid-execution. Using thread migration, the system is able to reschedule distributed threads to achieve high performance and/or load-distribution. To that end, it is important for the system to decrease the frequency of page-faults and reduce page-thrashing. Because thread migration changes the page access patterns of each machine, carefully considered thread scheduling can reduce page access contention and page-thrashing.

In order to reduce page access contention, the system collects the page access information on threads, and determines thread scheduling based upon this information. In this paper, we propose the following two thread scheduling methods, as shown in figure 2:

- correlation scheduling,
- suspension scheduling.

Correlation scheduling uses the correlation between threads, in order to prevent page-thrashing. It evaluates the correlation between two threads based upon their page access information recorded by the DSM server(s) that maintains consistency of DSM. This method aims to allocate strongly-correlated threads to the same machine and weakly-correlated ones to different machines.

On the other hand, suspension scheduling uses a page-fault frequency in order to stop page-thrashing. This method evaluates the page-fault frequency of threads, and if a thread frequently causes page-faults, the system suspends that thread.

2.1. Page access information

The DSM server maintains the consistency of shared memory by controlling the protection mode of DSM pages. The write-invalidate method (e.g. MESI protocol) is usually used as a consistency protocol. When the system maintains memory consistency with this protocol, the following page access information can be collected by the DSM server:

- source and destination threads of the page transfer,
- the page-fault frequency of a thread.

The former information is used for correlation scheduling. Because only fault accesses are perceived by the DSM server, not all the information about memory access can be collected, but such rough information is still useful for the scheduling. On the other hand, the latter information is used for suspension scheduling.

We present a simple analytical model for evaluating the correlation between two distributed threads. Let $D_i^p(t)$ be the access density function of thread i to page p (frequency of accesses to page p) at time t. When two threads share a page, they both have high correlation indices to that page. Therefore, the correlation between the two threads is given by the sum of the cross-correlation coefficients to all the shared pages. Thus, using the access density, correlation CA_{ij} between thread i and thread j is given by

$$CA_{ij} = \sum_{\forall p} \int_{T_1}^{T_2} D_i^p(t) D_j^p(t) dt. \quad (1)$$

Using CA_{ij}, we can also get the correlation between the thread and a machine. Correlation CN_i^n between thread i and machine n is given by

$$CN_i^n = \sum_{\forall j} \begin{cases} 0 & n_j = \emptyset \text{ (thread } j \text{ is not running.)} \\ CA_{ij} & n_j = n \\ -CA_{ij} & n_j \neq n \end{cases} \quad (2)$$

(where n_j is the number of the machine on which thread j is running.)

Because the DSM server collects only page-fault information, the server must approximate the access density from the page-fault information. Our system obtains the correlation from the approximate access density.

The correlation between thread i and thread j is evaluated as follows. The system uses the page-fault frequency as an approximate density function on a page, and the page transfer frequency between threads shows the approximate value of the product of the two density functions. An aging function $Age(x, t)$ is introduced to age x (access density function) according to time t. When this aging function is used, the correlation is evaluated from the correlation in the past. Let $PT_{ij}^p(t)$ be the page transfer frequency on page p between thread i and thread j at time t. Thus, the approximate value of CA_{ij} is given by

$$CA_{ij} \simeq \sum_{\forall p} \int_{T_1}^{T_2} PT_{ij}^p(t) dt \quad (3)$$

$$\simeq \sum_{\forall p} \int_{0}^{T_1} Age(PT_{ij}^p(t), t) dt. \quad (4)$$

The system implements correlation scheduling using the correlation information derived from equations (2) and (4).

2.2. Correlation scheduling

In order to prevent page-thrashing, correlation scheduling uses the correlation between threads as follows. When correlated threads run on the same machine, page-thrashing does not occur between them. Therefore, it would be better if the system allocated such threads to the same machine. To achieve this scheduling policy, the system evaluates the correlation between threads using equation (2). When a machine becomes free, the system evaluates the correlations between this machine and every thread in the ready queue, and allocates the most correlated thread to the machine. Furthermore, when a thread is ready to run, the system also evaluates the correlation between that thread and every free machine, and the system allocates the thread to the most correlated machine.

2.3. Suspension scheduling

When two or more threads running on different machines continuously refer to the same page, page-thrashing occurs. The system has a thread suspension mechanism that suspends a thread in order to stop page-thrashing. The DSM server collects the page-fault frequency of threads. The system detects page-thrashing from the page-fault frequency and suspends the responsible thread using the thread suspension mechanism.

3. PARSEC

PARSEC is a distributed thread execution environment designed for high performance and/or dynamic load-distribution. It is implemented as the basic environment of the distributed system we have been developing. The goals of PARSEC are:

- to execute multi-threaded programs on the distributed system,
- to make distributed and cooperative programming easy,
- to build server programs with a distributed task/thread model,
- to achieve efficient dynamic load-distribution.

PARSEC implements the distributed task/thread model as a programming model and user-level threads upon kernel-level threads for both dynamic thread scheduling and dynamic load-distribution.

3.1. Design

Figure 3 shows the architecture of PARSEC built on the Mach 3.0 micro-kernel [1]. PARSEC consists of two kinds

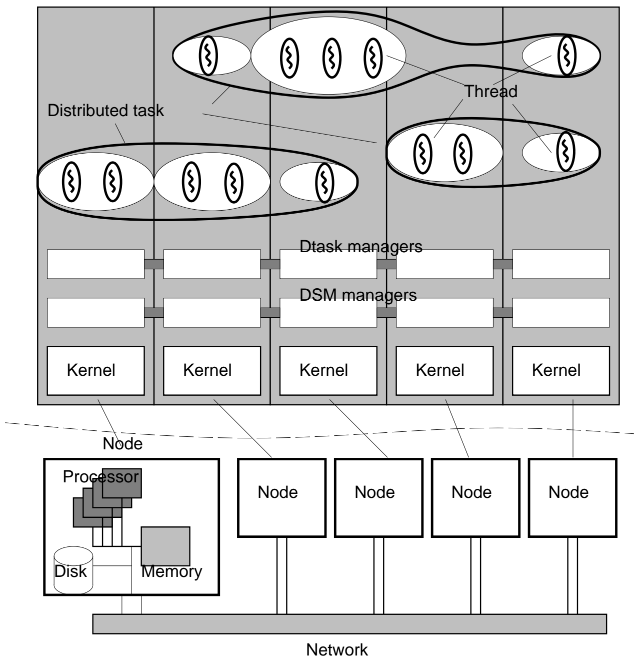

Figure 3. PARSEC organization.

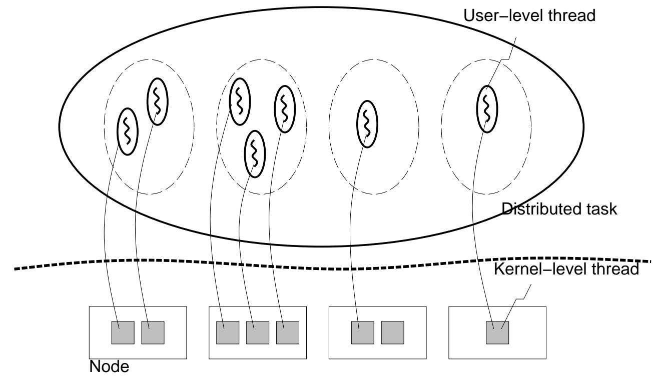

Figure 4. Distributed task with user-level threads.

of servers and a library: the DSM managers, the Dtask managers, and the D-threads library. There is one DSM manager and one Dtask manager per machine, and the D-threads library is linked to all PARSEC programs. All modules use MACH port and message mechanisms for machine local communication and a TCP/IP protocol with a network.

The DSM manager is a server managing DSM and it also collects the page access information that is used for the proposed scheduling methods. It provides sequential consistent shared memory for PARSEC application programs, and uses a write-invalidate protocol with external pager interfaces [17] to realize shared memory.

The Dtask manager manages creation and termination of distributed tasks, allocation of kernel-level threads to distributed tasks, and dynamic load-distribution described below. The managers communicate with each other and also distribute workload information of each machine for load-distribution.

The D-threads library is a user-level thread package for the distributed task/thread programming model. This library schedules user-level threads on kernel-level threads provided by the Dtask manager, and implements synchronization and conditional-waiting mechanisms. It is based on the Mach 3.0 C-threads library [5].

Figure 4 is a diagram of a distributed task with user-level threads. Implementing user-level threads over kernel-level threads has several benefits. The most important benefit is making thread migration possible. The context information about a user-level thread is saved in the user address space so that kernel-level threads on other machines can access the saved context information with the consistency being maintained within the task. When the system resumes the user-level thread on another machine, it only has to load the context into a kernel-level thread on that machine. As a result, the user-level thread migrates to that machine from the machine which the context of the user-level thread was saved on. In short, the system can migrate a distributed thread, even when the kernel does not have a thread migration mechanism.

Another important benefit is that dynamic kernel-level thread allocation can be achieved. Because user-level threads are separated from kernel-level threads, the system dynamically controls the number of kernel-level threads allocated to a distributed task. If the system decreases the number of kernel-level threads on a machine A and then increases the number on another machine B, user-level threads suspended on the machine A start to run on the machine B. As a result, workloads on the machine A can be moved to the machine B. These mechanisms play important roles in dynamic load-distribution.

Figure 5 shows the thread scheduling actions in a distributed task. There are two kinds of schedulers in a distributed task: the global scheduler and local schedulers. The global scheduler exists only on the machine where the task is created. It manages the number of free kernel-level threads on each machine and the user-level threads in the global ready queue. On the other hand, a local scheduler exists on each machine. It allocates user-level threads to kernel-level threads.

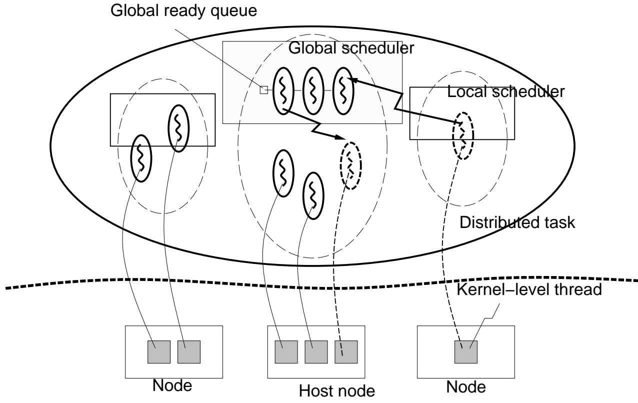

Figure 5. Thread scheduling in distributed task.

3.2. Page access information collected by the DSM managers

The DSM managers collect the page access information, as mentioned in the previous section. Two data structures are held in the DSM manager. One is used for correlation scheduling and the other for suspension scheduling.

The former data structure is a two-dimensional array, and each entry represents the correlation between two user-level threads. The DSM manger evaluates the correlation using equation (4). Each DSM manager keeps a record of page holder thread numbers whose access caused the page transfer to the local machine. When page data is transferred, the holder thread number is also transferred with the page data message. On the other hand, when a page-fault occurs, the DSM manager knows the thread number whose access caused the page-fault. Therefore, the DSM manger gets the source thread number and the destination thread number. Using the numbers as two-dimensional indexes, the server increases the correlation array entry value.

The other data structure holds a number, called a page-fault-index, which indicates the page-fault frequency. There is one page-fault-index per user-level thread. When a user-level thread causes a page-fault, the DSM manager evaluates the interval of this thread's page-faults and increases the page-fault-index by the reciprocal of the interval. A large page-fault-index indicates page-thrashing.

Aging is performed by periodically multiplying data by a constant (< 1). PARSEC uses different constant values[1] for the two structures, because it is clear that the correlation information has a longer influence than the page-fault-index.

[1] PARSEC uses 0.97 every 2 second for the correlation and 0.5 every 2 second for the page-fault-index.

3.3. Scheduling implementation in the D-threads library

Correlation scheduling is implemented as follows:

1. When user-level thread i is ready to run, it is sent to the global scheduler.
2. If there are free kernel-level threads, the global scheduler evaluates CN_i^n using equation (2) for every machine n that has any free kernel-level threads, and then allocates the user-level thread to the kernel-level thread on the machine that has the maximum value of CN_i^n.
3. If no machine has free kernel-level threads, the user-level thread is put into the global ready queue.
4. If a kernel-level thread on machine n becomes free, the local scheduler on the machine send a request for user-level thread allocation. When the global scheduler receive the request, it evaluates CN_i^n for every user-level thread i in the global ready queue and allocates the user-level thread with the maximum value of CN_i^n to the kernel-level thread on machine n.

The detailed-flow of correlation scheduling is given in the appendix in figure 9.

Suspension scheduling is implemented as follows:

1. Whenever a thread causes a page-fault, the local DSM manager increases the page-fault-index of that thread. The DSM manager checks if the page-fault-index exceeds threshold τ for suspension scheduling. If so, the DSM manager instructs the local scheduler to suspend the thread by sending a suspend message.
2. When the local scheduler receives the message, the scheduler turns on the suspend request flag of the thread.
3. When the thread calls a routine of the user-level thread library, the suspend request flag is checked. If the suspend request flag is set, the user-level thread saves its own state and is suspended.
4. The suspended user-level thread is sent to the global scheduler and waits for rescheduling in the global ready queue.

4. Performance evaluation

The following measurements show the impact of the proposed thread scheduling methods on system performance. All the measurements were done on up to eight PCs (Pentium 120 MHz) connected through a 100-Mbps Fast-Ethernet (100 BASE-TX) HUB. Three application programs were

used for the measurements, Water-Nsquared, Fractal 1, and Fractal 2. These programs are multi-threaded and written for a multi-processor machine. These were not specially modified for PARSEC, but they were re-linked with the D-threads library.

4.1. Water-Nsquared

Water-Nsquared is an application program taken from the SPLASH–2 programs [22]. This application is an improved version of the Water program of the SPLASH programs [18]. Measurements with Water-Nsquared showed the page access pattern by an existing multi-threaded program; eight threads were used for the measurement.

This application evaluates the forces and potentials of water molecules in the liquid state. The main data structure used in the program is a large one-dimensional array called VAR. This array is divided into as many sub-arrays as the number of allocated threads. Because each sub-array is assigned to the corresponding thread number, correlation between neighbor threads is high. Thus, adjacent threads have stronger correlation.

Figure 6 shows the execution time of Water-Nsquared on two PCs with several scheduling methods: (a) with normal scheduling which uses the global ready queue as FIFO, (b) with suspension scheduling with a threshold ($\tau = 10$), (c) with suspension scheduling ($\tau = 30$), (d) correlation scheduling, (e) with both suspension scheduling ($\tau = 10$) and correlation scheduling, (f) with both suspension scheduling ($\tau = 30$) and correlation scheduling.

Suspension scheduling seems to have no effect, since page-thrashing does not occur frequently and does not continue for a long time. However, correlation scheduling has a strong influence, and system performance is improved by about 17%. The measurements show that the correlation between the threads is efficiently used for thread scheduling.

4.2. Fractal 1

Fractal 1 is an application program that draws a two-dimensional fractal image (512 × 512 pixels). These measurements were done on four PCs.

Figure 7 shows the execution time of Fractal 1 program measured with four scheduling methods: (a) with normal scheduling (same as above), (b) with suspension scheduling ($\tau = 10$), (c) with correlation scheduling, and (d) with both suspension scheduling ($\tau = 10$) and correlation scheduling.

An image plane drawn by this program is divided into 4 × 4 blocks. Each thread of the program draws one block. Because four threads drawing same line have strong correlation, correlation scheduling should be effective to prevent page-thrashing. However, the performance with only correlation scheduling is poor, because there are little opportu-

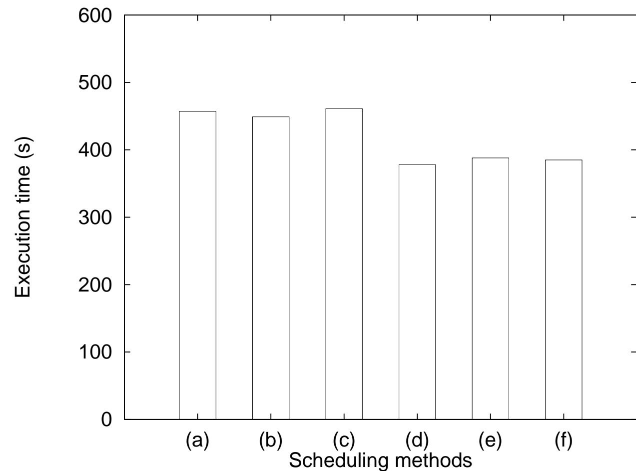

Figure 6. Execution time of Water-Nsquared: (a) with normal scheduling, (b) with suspension scheduling ($\tau = 10$), (c) with suspension scheduling ($\tau = 30$), (d) with correlation scheduling, (e) with both suspension scheduling ($\tau = 10$) and correlation scheduling, (f) with both suspension scheduling ($\tau = 30$) and correlation scheduling.

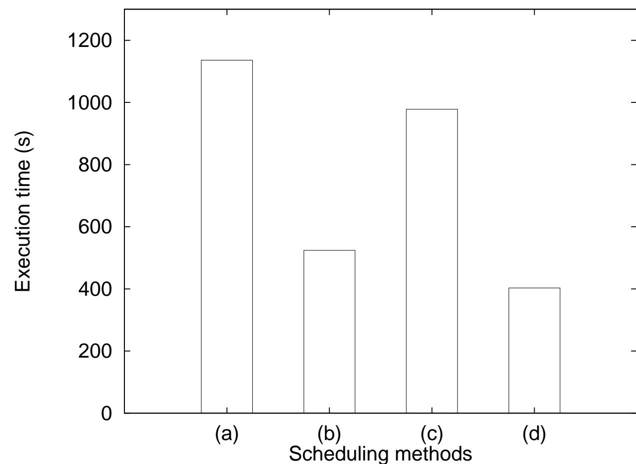

Figure 7. Execution time of Fractal 1: (a) with normal scheduling, (b) with suspension scheduling, (c) with correlation scheduling, and (d) with both suspension scheduling and correlation scheduling.

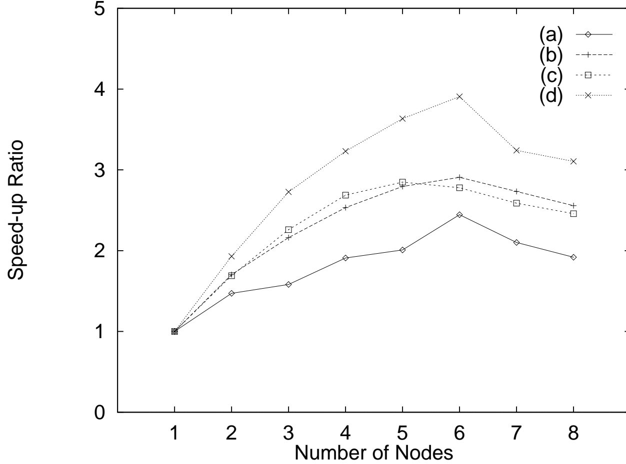

Figure 8. Speed-up ratio of Fractal 2: (a) program (1) with normal scheduling, (b) program (1) with suspension scheduling, (c) program (1) with both suspension scheduling and correlation scheduling, and (d) program (2) with normal scheduling.

nity to reschedule threads. The opportunity is important to make good use of the correlation information. The suspension scheduling method suspends and reschedules the thread causing page-thrashing. Therefore, the overhead of page-thrashing is reduced only using both correlation scheduling and suspension scheduling. That is also because frequent rescheduling of the thread can make the collected correlation information more accurate.

4.3. Fractal 2

Fractal 2 is another fractal drawing program (1024×1024 pixels). One program (1) creates four threads per machine, where each thread draws one image line at a time. A variable stores the line number that has already been assigned. When a thread completes drawing an image line, it exclusively increases the variable. Because two lines belong to one page, two threads access the same page at one time, so page-thrashing occurs as a result. For comparison, we also used another program (2) assigning two image lines to a thread at a time. This did not cause page-thrashing by false-sharing.

Figure 8 shows the speed-up ratio of Fractal 2 program (1) measured with three scheduling methods and Fractal 2 program (2): (a) program (1) with normal scheduling (same as above), (b) program (1) with suspension scheduling with a threshold ($\tau = 10$), (c) program (1) with both suspension scheduling ($\tau = 10$) and correlation scheduling, and (d) program (2) with normal scheduling (same as above).

The difference between (a) and (d) shows the influence of page-thrashing caused by false-sharing of image lines. Because (b) and (c) use suspension scheduling, our attempt to reduce page-thrashing succeeds and the time overhead due to page-thrashing shown in (a) is reduced by up to 59%.

Comparing (b) and (c), we find that correlation scheduling has no effect. Correlation scheduling is not successful, because all thread have fluctuating correlation with each other and the correlation prediction fails.

4.4. Performance results

These measurements show that the proposed scheduling methods are very effective. For the existing multi-threaded programs, threads of the program often have strong correlation. Thus, correlation scheduling is effective to prevent page-thrashing. Furthermore, if the threads have no correlation, evenly distributed correlation or fluctuating correlation and correlation scheduling do not work well; in that case, suspension scheduling operates efficiently to stop page-thrashing.

Our implementation of the two thread scheduling methods uses non-preemptive user-level threads. Suspend-able points exist only at the entries of the thread library functions. Because of this restriction, the system cannot immediately stop threads, even when the system knows that page-thrashing is occurring. Hence, a preemptive thread implementation would perform better.

5. Related work

A number of software and hardware DSM systems have been developed. One early DSM system is IVY [15]. Because it uses sequential consistency [12] and a write-invalidate protocol, IVY is prone to page-thrashing. It depends on the programmer laying out the data structures so that page-thrashing is reduced. Mirage [7] first attempts to avoid page-thrashing by locking a page for a certain $\triangle$ time-window. However, since a long time-window leads to a long memory access delay and a short time-window leads to page-thrashing, it is difficult to determine $\triangle$. PARSEC uses thread scheduling to reduce page-thrashing and does not cause such a memory access delay.

Munin [3], TreadMarks [9], and Midway [2] are software DSM systems that use relaxed memory consistency models. Although the relaxed memory consistency models do not directly attempt to prevent page-thrashing, they have the effect of reducing page-thrashing. However, these systems still lack a method of reducing page-thrashing. Thus, if our thread scheduling can be applied to them, then page-thrashing will be further reduced.

The DASH [13] multiprocessor is a hardware DSM system. It implements release consistency in hardware using a directory-based protocol [14]. In DASH, the consistency

unit is a cache block. The larger the cache block is, the more "ping-pongs" (like page-thrashing) occur. If the system has a hardware mechanism to inform a thread scheduler of page/cache block access information, our thread scheduling methods can be applied to the hardware DSM system and will work effectively to reduce "ping-pongs".

The FLASH [11] multiprocessor implements DSM in software, but the cache consistency mechanism is executed by a special processor - MAGIC (Memory and General Interconnection Controller). A mechanism to collect page/cache block access information can be easily integrated with MAGIC and the proposed thread scheduling methods can also be applied to FLASH.

A technique similar to the proposed correlation scheduling method of PARSEC was proposed in [21, 16]. The scheduling called "affinity scheduling" allocates a process to a processor that already has data cached in the local cache. The affinity scheduling focuses only on the affinity between a process and the local cache. But our correlation scheduling focuses on the correlation between threads, because the main purpose of our scheduling is to reduce page-thrashing between correlated threads.

The page-fault frequency (PFF) algorithm [4] is a page replacement algorithm. It is similar to suspension scheduling; both use the page-fault frequency. The PFF algorithm determines whether a page should be replaced and/or a process should be swapped out based upon the page-fault frequency, while suspension scheduling determines only whether a thread should be suspended based upon the page-fault frequency.

6. Conclusions

This paper discussed a DSM system specially designed for executing multi-threaded programs without any modification. We addressed the influence of thread scheduling based upon page access information in a distributed thread execution environment. We proposed two distributed-thread scheduling methods, correlation scheduling and suspension scheduling. These scheduling methods aim, in particular, at reducing page-thrashing. Correlation scheduling uses the correlation between threads of a task and allocates correlated threads to the same machine. Suspension scheduling tries to suspend threads that cause page-thrashing in order to stop page-thrashing.

PARSEC is a distributed thread execution environment that we have implemented. Its main features are dynamic load-distribution and efficient execution of distributed threads using the user-level threads. Our performance evaluations with PARSEC indicate that the proposed distributed-thread scheduling methods are efficient at reducing the page-fault frequency and reduce page-thrashing. These evaluations were done using some typical scientific applications, Water-Nsquared and two Fractal drawing programs. With Water-Nsquared, it became clear that correlation scheduling decreased the number of messages caused by page-thrashing. On the other hand, with Fractal 2 program, we found that suspension scheduling effectively stopped page-thrashing. The overhead caused by page-thrashing was reduced by up to 59%. Furthermore, these measurements made it clear that suspension scheduling played an important role to collect accurate correlation information and in result it increased efficiency of correlation scheduling.

We plan to continue working with PARSEC to evaluate the performance with more applications. And we will implement load-distribution on PARSEC and evaluate its performance.

Acknowledgments

We would like to thank our colleagues, Masahiko Yoshimoto and Kenjiro Cho who contributed to discussions in the early stages of this work, Yuqun Chen for his careful reading of this paper. Finally, we would like to thank the program committee and the other anonymous referees for insightful comments.

References

[1] M. Acceta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Technical Conference*, pages 93–112, 1986.

[2] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. Technical Report CMU-CS 93–119, School of Computer Science, Carnegie-Mellon University, 1993.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, 1990.

[4] W. Chu and H. Opderbeck. Program behavior and the page-fault-frequency replacement algorithm. *Computer*, 9(11):29–38, 1976.

[5] E. C. Cooper and R. P. Draves. C threads. Technical Report CMU-CS 88–154, School of Computer Science, Carnegie-Mellon University, 1988.

[6] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[7] B. D. Fleisch and G. J. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, 1989.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event

ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[9] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: distributed shared memory on standard workstations and operating systems. In *1994 Winter Usenix Conference*, pages 114–131, 1994.

[10] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, 1992.

[11] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, 1994.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C–28(9):690–691, 1979.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, 1990.

[14] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[15] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[16] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, 1993.

[17] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architecture. *IEEE Transactions on Computers*, C–37(8):896–907, 1986.

[18] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, 1992.

[19] Y. Sudo, S. Suzuki, K. Cho, and S. Shibayama. Thread scheduling methods of a distributed thread execution environment on distributed shared memory. In *Technical reports of IEICE, CPSY 95–63*, pages 103–110, 1995. (in Japanese).

[20] Y. Sudo, S. Suzuki, M. Yoshimoto, and S. Shibayama. The design and evaluation of a distributed thread execution environment on distributed shared memory. In *IPSJ SIG Notes, 94–OS–65*, pages 161–168, 1994. (in Japanese).

[21] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 26–40, 1991.

[22] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

Appendix: Correlation scheduling details

rescheduling user thread i

```
int CA[][];
if( free kernel thread exists on any machine ){
  int target machine;
  int max sum of correlation = MINIMUM of INTEGER;
  for( each machine n that has free kernel thread ){
    int CN = 0;
    for( each user thread j except user thread i ){
      if( user thread j is running on machine n )
        CN += (CA[i][j] + CA[j][i]);
      else
      if( user thread j is running on other machine )
        CN -= (CA[i][j] + CA[j][i]);
    }
    if( max sum of correlation < CN ){
      max sum of correlation = CN;
      target machine = n;
    }
  }
  send user thread i to target machine and restart;
}
else
  enter user thread i into global ready queue;
```

user thread request from machine n

```
int CA[][];
if( length of global ready queue > 0 ){
  int target user thread;
  int max sum of correlation = MINIMUM of INTEGER;
  for( each user thread i in global ready queue ){
    int CN = 0;
    for( each user thread j except user thread i ){
      if( user thread j is running on machine n )
        CN += (CA[i][j] + CA[j][i]);
      else
      if( user thread j is running on other machine )
        CN -= (CA[i][j] + CA[j][i]);
    }
    if( max sum of correlation < CN ){
      max sum of correlation = CN;
      target user thread = i;
    }
  }
  send target user thread to machine n and restart it;
}
else
  increase the number of free kernel threads on machine n;
```

Figure 9. Pseudo codes of correlation scheduling.