객체지향 종속 추적 및 체크포인팅(checkpointing)을 이용한 복구 가능한 분산 공유 메모리 시스템

김 재 훈[†]

요 약

메시지 전달 방식으로 노드간 통신을 하는 분산시스템의 고장허용을 위하여 메시지 저장과 체크포인팅에 관한 많은 연구가 이루어졌다. 복구 가능한 분산공유메모리 시스템에 대한 대부분의 연구 또한 메시지 전달 방식에서 사용되었던 방법을 채택하였다. 그러나, 메시지 전송시스템과 분산공유메모리 시스템의 근본적인 차이(함수전달(function shipping)과 데이터전달(data shipping)의 차이) 때문에 메시지 전달 시스템에서 사용되었던 방식이 분산공유메모리 시스템에 항상 적합하게 사용될 수 없다. 본 논문에서는 복구 가능한 분산공유메모리 시스템을 위하여 객체지향방법을 제안하였다. 프로세스간 종속 추적 대신 페이지간 종속 추적을 이용한 체크포인팅 및 복구 가능한 전략을 분산공유메모리 시스템에 적용하였다.

Recoverable Distributed Shared Memory Systems Using Object-Oriented Dependency Tracking and Checkpointing

Jai-Hoon Kim[†]

ABSTRACT

Many message logging and checkpointing schemes are proposed for fault tolerance in distributed systems in which nodes communicate by message passing. Most researches for recoverable distributed shared memory (DSM) also adopt similar schemes used in message passing systems. However, schemes used in message passing systems are not always appropriate to be directly used in DSM systems because the two systems, message passing systems and DSM systems, have different natures (function shipping and data shipping). Many modified schemes have been proposed for DSM systems to resolve these differences. In this paper, an object oriented approach is proposed for recoverable DSM. We present a new dependency tracking scheme between pages instead of processes. Based on this scheme, we propose new checkpointing and recovery schemes that can reduce overhead to make DSM recoverable.

1. Introduction

Message logging and checkpointing schemes are widely used for fault tolerance in distributed systems in which nodes communicate via message passing mechanism. Those schemes used in message passing system are also used for recoverable distributed shared memory (DSM). However, two primitives supporting distributed systems have different approaches. One uses message based on function shipping and the other uses shared memory based on data shipping. Message logging and checkpointing schemes are not efficient to be directly used in DSM systems. Thus, many modified schemes [11,12,13] have been proposed for recoverable DSM systems to

※ 본 연구는 아주대학교 1998년도 정착연구비 지원으로 진행되었음.
† 정 회 원 : 아주대학교 정보통신대학 정보 및 컴퓨터공학부 교수
논문접수 : 1998년 5월 25일, 심사완료 : 1998년 11월 30일

resolve the differences.

In message passing systems, dependencies between processes arise at each message transfer. When a process receives each message, the state of the process depends on the state of the sender of the message [14]. This dependency tracking mechanism is inefficient for DSM system because messages do not necessarily cause the receiver to depend on the sender in DSM systems. A process receiving a message for actual data transfer depends only on a process sending the message [12]. Dependencies can be further reduced by using release consistency [11] or lazy release consistency [13].

Related Works

Many recoverable DSM schemes have been presented in the literature. Some of them use stable storage (disk) to save recovery data [10,11,22,25], and others use main memory for checkpointing, replicating shared memory or logging the shared memory accesses [1,3,7,15,16,19,23,24]. However, only a few of the papers use an object oriented approach for a recoverable DSM.

Recoverable schemes for object oriented systems are proposed [9,18,20]. Object oriented approaches for recoverable schemes are proposed [6,21]. [6] presents an object-based recoverable scheme for implementing multiple transaction models. Objects (called segments) contain data and code. Multiple versions of an object are allowed to deal with failures. Data versions (shadows) are created in the implementation of various transaction models, and manipulated using the primitive operations of copying the contents of segment(s). [21] presents an object-oriented approach in implementing recoverable scheme to hide and separate the added complexity due to the inclusion of fault tolerance. Abstraction is used to hide fault-tolerance layers. Fault tolerance is added to an application by placing fault tolerance layers around instances of the data objects. Thus, the fault tolerance layers are relatively independent of the application program. [8] presents an event ordering mechanism in a distributed system based on shared objects.

In this paper, an object oriented approach is proposed for recoverable DSM. We present a new dependency tracking scheme between objects (pages) instead of procedures (processes). This paper shows that the object oriented approach for recoverable DSM systems can reduce overhead. For a recoverable DSM system, pages are more important objects to be recovered rather than processes as we discussed later. We can consider a page as an object thus the object-based approach can be easier to understand and implement the recoverable DSM than the conventional approach used in message passing systems.

2. Page Based Model

This section presents a model for recoverable DSM using object oriented approach. This model is based on the dependencies between the states of pages that result from data dependency between the pages. We assume that the DSM uses release consistency and dynamic distributed ownership analogous to Munin [5]. When variable X_j in page P_j is updated by referencing variable X_i in page P_i ($X_j = f(X_i)$) between acquire and release, there is data dependency from page P_i to page P_j, in our definition. The state of each page is represented by dependencies between the states of pages, and the state of the system is represented by a set of page states. The dependencies between the state of pages are analogous to those between the states of processes in [14].

Page State

Data updates in a page is divided into intervals, called a state interval of the page, by the propagation of updates at release. Each state interval of a page is identified by a sequential state interval index (or version). When a data dependency exists between pages P_i and P_j (Assume that P_i depends on P_j), the version of page P_i depends on that of page P_j. The state of a page is denoted by its current set of dependencies on all other pages. These dependencies are denoted by a dependency vector as follows :

$$\delta_i \langle \delta_1,\ \delta_2,\ \delta_3, \cdots,\ \delta_m \rangle,$$

where m is the total number of pages in the system. The dependency vector is maintained for each page. δ_j in page P_i's dependency vector is the latest version of page P_j on which page P_i currently depends, where $1 \le j \le m$.

Group Page State

In DSM systems using release consistency memory model, pages are updated at release instead of at every update of a page. When multiple pages are updated, this modification should be propagated to other nodes atomically. For instance, when node A updates page P_i and page P_j, and propagates this modification at release, node B should see all modifications of both pages or nothing. Because the granularity of the state model based on page state is too small and the modification of pages should be propagated to other nodes atomically, the model based on "page group" can reduce the complexity of dependency tracking by reducing the number of dependency and required memory space.

A page group consists of pages which are updated together at the same release. The size of a page group and the member of a page group are not fixed. A page group is created at release and maintained until it may not be used any more. "Group dependency vector" is defined as follows :

$$\Delta = \langle \Delta_1,\ \Delta_2,\ \Delta_3, \cdots,\ \Delta_n \rangle,$$

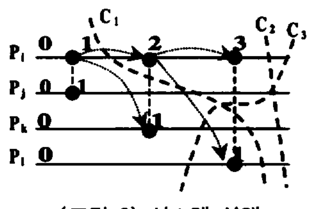
where n is the total number of page group in the system. Δ_j in page P_i's group dependency vector is the latest version of page P_y on which page P_x currently depends, where page P_x and page P_y can be any page in page groups P_i and P_j, respectively.

For example, consider the following application shown in (Fig. 1) using DSM.

Process A	Process B	Process C
acquire();	.	.
X_j = f(X_i);	.	.
X_i = g(X_i);	.	.
release();	.	.
	acquire();	.
	X_k = f(X_i);	.
	X_i = g(X_i);	.
	release();	.
		acquire();
		X_l = f(X_i);
		X_i = g(X_i);
		release();

(그림 1) DSM을 이용한 프로그램
(Fig. 1) Application using DSM

Assume that variables X_i, X_j, X_k, and X_l are in pages P_i, P_j, P_k, and P_l, respectively. Let $P_x(v)$ denote page P_x of version v in (Fig. 2). Dashed lines link pages in the same page group, arrow dotted lines denote dependency between pages, and bold dashed lines denote the system states which will be explained later. Now, we compute the group dependency vector for page $P_k(1)$. Page $P_k(1)$ is in the page group of $\{P_i(2), P_k(1)\}$. Page $P_i(2)$ does not depend on other pages except $P_i(1)$, and page $P_i(1)$ and page $P_j(1)$ are in the same page group. Page $P_k(1)$ also depends on page $P_i(1)$. Thus, the group dependency vector for page $P_k(1)$ is <2,1,1,N>, where N denotes no dependency.



(그림 2) 시스템 상태
(Fig. 2) System States

System State

A system state is a set of page states. A system state is denoted by an m×m dependency matrix.

$$D = \begin{bmatrix} \Delta_{11} & \Delta_{12} & \Delta_{13} & \cdots & \Delta_{1m} \\ \Delta_{21} & \Delta_{22} & \Delta_{23} & \cdots & \Delta_{2m} \\ \Delta_{31} & \Delta_{32} & \Delta_{33} & \cdots & \Delta_{3m} \\ & & \vdots & & \\ \Delta_{m1} & \Delta_{m2} & \Delta_{m3} & & \Delta_{mm} \end{bmatrix}$$

where row i is the group dependency vector for page P_i. If two pages P_i and P_j are in the same page group, then row i is equivalent to row j.

Consistent System State

A system state is consistent if an observer can see the state at some instant during the execution of the system from its initial state to the current state, regardless of the relative speeds of the involved processes [14]. We can apply the definition of the consistent system state to our page based system state. If D is some system state, D is consistent if and only if $\forall i, j [\Delta_{ji} \leq \Delta_{ii}]$.

In (Fig. 2), bold dashed lines show some examples of system states in page based models. The state of a page is observed where the bold dashed line intersects the line of a page. System state C_1 is not consistent because Δ_{31} and Δ_{41} are greater than Δ_{11}. System state C_3 is not consistent because Δ_{14} is greater than Δ_{44}. However, system state C_2 is consistent.

$$C_1 = \begin{bmatrix} 1 & 1 & N & N \\ 1 & 1 & N & N \\ 2 & 1 & 1 & N \\ 3 & 1 & 1 & 1 \end{bmatrix} \quad C_2 = \begin{bmatrix} 3 & 1 & 1 & 1 \\ 1 & 1 & N & N \\ 2 & 1 & 1 & N \\ 3 & 1 & 1 & 1 \end{bmatrix} \quad C_3 = \begin{bmatrix} 3 & 1 & 1 & 1 \\ 1 & 1 & N & N \\ 2 & 1 & 1 & N \\ N & N & N & 0 \end{bmatrix}$$

Checkpointing

When a page is allocated and initialized, its initial state is checkpointed. A page can be checkpointed by saving on a stable storage [14] or maintaining at least two copies on two different nodes for each page [17]. Each checkpoint is maintained until it is no longer needed for recovery.

The following definition in a page based system is analogous to the definition in a process based system [14] :

- The predicate *logged(i, σ)* is true if and only if the diff (the modification of a page), that started version σ of page P_i is logged.
- The effective checkpoint for a version σ of some page P_i is the checkpoint for page P_i with the largest version ε such that $\varepsilon \leq \sigma$.
- State interval σ of page P_i is stable if and only if $\forall \alpha, \varepsilon < \alpha \leq \sigma$ [*logged(i, α)*], where ε is the version of page P_i recorded in the effective checkpoint for version σ.

A page may be checkpointed at any release. When multiple pages are updates and propagated, these pages are checkpointed atomically.

Recoverable System States

A system state is recoverable if and only if all element page versions are stable as well as the resulting system state is consistent.

3. Design

Our design is based on release consistency memory model. We assume that all shared memory accesses occur in a "transaction" basis. A transaction denotes a sequence of operations -- namely, *acquire*, *shared memory access*, and *release*.

Page Table Structure

A page table is maintained for each page and has the following information :

- addr : page address.
- version : version of page (incremented at each

update on release).

- vector[m] : dependency vector.
- gvector[m] : group dependency vector.
- twin : used for making diff (difference between old version and new version of a page [5]).
- ckp[Max_ckp] : checkpoints addresses, where Max _ckp is the maximum number of checkpoints which are maintained for each page.
- ckp0 : checkpoint address. It can be used with diff[] instead of ckp[].
- diff[Max_ckp-1] : address of diff's which is difference from ckp0.

Dependency Tracking

Whenever a modification of a page is propagated at release, the version for a copy of a page in the receiving node is increased by one. From the acquire, the versions of all pages being read are stored in the table called "version_table[]". If page P_j of version v_j is read, version_table[j] is set to v_j. The first read of the recent version for each page causes version_table[] to be updated. On a release, vector[m] of the page table for the updated pages are set to version_table[] only for the elements associated with non-null elements of version_table[]. If vector[] is <1,2,N,3> and version_table[] is <N,3, N,3>, then vector[] becomes <1,3,N,3>. gvector[] is obtained from vector[]'s of the pages in the same page group.

Checkpointing

We can checkpoint a page by saving the content of the page on a stable storage or maintaining at least two copies on two different nodes for each page. For the latter case, ckp[] is used for the pointer of the page content checkpointed. [17] shows that guranteeing at least two copies for each page requires small overhead in many applications because DSM has more than one copy for each page in many cases (especially in update-based protocol). Thus, checkpointing overhead will be small. If the size of diff between the two versions (the last checkpointed version and current checkpointed version) is small, then incremental checkpointing by maintaining diff[] is efficient. This is similar to incremental checkpointing presented in [4]. However, our scheme saves the modified part of page (diff) on memory of at least two different nodes while [4] saves whole the content of modified pages on a stable storage.

4. Performance Comparisons

A page based algorithm has advantages over a process based algorithm. We can reduce checkpoint overheads as well as the number of checkpoints by reducing dependency.

Reduce Checkpoint Overhead

In process based message passing systems, saving process states on a stable storage is the main issues on checkpointing. In addition to process state, shared memory is also needed to be included in the checkpointing for recoverable DSM systems. Each page of the shared memory is saved on a stable storage by the process which has an ownership of the page, or the backup copy for each page is maintained.

When a server is waiting for a request, the major part of a process local state (the stack and control registers) have the same values and so their contents can be easily rebuilt by re-executing the initialization part [2]. In page based DSM systems, each process can be considered as a computational server which computes its own part of shared memory in a loop. When a process is at the start of the loop, the process local state has the same values (registers and stack) or can be restored (local variables). Thus, their contents can be easily rebuilt by one time checkpointing at the first iteration or re-executing the initial routine. As long as the shared memory space is saved, in page based systems, processes can restart by re-initializing a process local state (registers, stack, and local variables) with the saved shared memory.

By observing typical applications, periodical check-pointing for process local state is not necessary if shared memory is saved.

```
// A typical application //
main()       // executed by master node
{
   initialize();          // application initialize
   init_shm();            // shared memory initialize
   fork_thread(compute);
          // fork remote thread to execute compute()
   compute();             // compute
}
compute()                 // execute by remote thread
{
   while (more jobs) {  // repeat until FINISH
      if (first iteration)  // if first iteration,
         checkpoint();  // take a checkpoint for process
                          local state
      read_shm();       // read shared memory
      calculation();    // calculation
      write_shm();      // write shared memory
      synch();          // synchronization (e.g., barrier,
                          lock-unlock)
   }
}
```

As shown in the above typical application, a process can restart after failure with (1) the shared memory saved by checkpointing in a page based scheme and (2) process local state saved by check-pointing, which is necessary at the first iteration only. There are two typical types of applications. One uses barrier for synchronization and the other uses lock/unlock.

The following code sor_compute is the compute() routine SOR application which uses barrier for synchronization.

```
sor_compute()
{
   while (iteration--) {
     for (row=start_row; row <= end_row; row++)
        for (col=start_col; col <= end_col; col++)
          newmat[row,col] = (mat[row-1,col] + mat[row,col-1]
                      + mat[row+1,col] + mat[row,col+1]) / 4;
     mat[*,*] = newmat[*,*];
     barrier();
   }
}
```

In SOR application, only one variable, iteration, is changed in each start of the while loop. Variable iteration can be easily restored because other processes have the same values of iteration by barrier() synchronization. Thus, a process in a failed node can restart from the start of the loop once shared memory is saved.

Qsort (Quick Sort) is an example of application using lock/unlock. qsort_compute is the compute routine for Qsort application.

```
qsort_compute()
{
   while (unfinish) {

      lock(Que);          // mutual exclusion for Que
      task1 = dequeue(Que);
      unlock(Que);

      lock(pid);
      task2 = sort(buf, task1);
      unlock(pid);        // for update propagation

      lock(Que);          // mutual exclusion for Que
      enqueue(Que, task2);
      unlock(Que);
   }
}
```

In Qsort application, no local variable is changed at the start of the while loop. Thus, a process of a failed node can restart from the start of the loop. However, we need a little modification for dependency tracking. We need to allocate variables task1 and task2 in each node as shared memory because Que and buf depend on each other via temporary variables task1 and task2. (Alternatively, we can modify application to update Que and buf atomically.)

Reduce Dependency

In process based systems, dependency occurs on data transfer. Thus, checkpoint is taken at release or acquire. However, in page based system, data transfer does not necessarily mean occurring dependency. Let's consider following four cases :

1. No process dependency; no page dependency : When process A reads and writes page P_i, it is updated by itself. Hence, no dependency occurs.
2. Process dependency; no page dependency : When process A reads and writes page P_i updated by process B, dependency occurs in a process based system. However, in a page based system, dependency does not occur. Even though process B rolls back to the previous state of updating page P_i, the page based system state is consistent because there is no data dependency between the pages. Process B can restart by initializing process local state. In most cases, the content of page P_i updated by process B will not be lost because the other copy of page P_i exists in the node of process A. If the content of page P_i updated by process B is lost, process B (or other process) will re-execute and updated page P_i as before. Consider the following scenario in Qsort application. Now, we consider Que only (buf will be considered in the next case) : (1) Process B updates Que; (2) Process A in other node reads and updates Que (We assume that Que is contained in a page). There are process dependencies (process A depends on process B) due to Que. However, no page dependency exists due to Que if Que can be contained in one page.
3. No process dependency; page dependency : When process A reads page P_i updated by process A itself and writes page P_j (content of page P_j depends on the content of page P_i), data dependency occurs between pages P_i and P_j in page based system only. However, we can ignore this dependency (1) if a checkpoints is taken in page group basis (instead of individual page), and page P_i and page P_j are in the same page group, or (2) if page P_i and page P_j are checkpointed together. For an example, in Qsort application, Que and buf depend on each other after a process executes the while loop. However, we can ignore this dependency if Que and buf are checkpointed together.
4. Process dependency; page dependency : When process A reads page P_i updated by process B and writes page P_j (content of page P_j depends on the content of page P_i), dependency occurs in both a process based system and a page based system.

Because the number of dependencies is reduced by avoiding dependencies in case 2, our dependency tracking scheme based on pages is more efficient than that based on process as in [11,12,14]. (More advanced recoverable system state can be found by using our scheme.)

5. Conclusion

A new dependency tracking scheme between pages instead of processes has been proposed. Based on this scheme, we have proposed new checkpointing and recovery schemes which can reduce overhead to make DSM recoverable. We have shown the cases of applications in which our object-based scheme can reduce checkpointing overhead and/or the number of dependencies. Our object oriented approach for recoverable DSM systems is easier to understand and implement than other approaches that have been used in message passing systems. Futher study is needed to verify the advantages of our object-based approach for recoverable DSM by simulation or implementation of our scheme.

References

[1] M. Banatre, A. Gefflaut, and C. Morin, "Tolerating node failures in cache only memory architectures," Tech. Rep. 853, INRIA, 1994.

[2] M. Banatre, P. Heng, G. Muller, N. Peyrouze, and B. Rochat, "An experience in the design of a reliable object based system," in Proc. of the International Conference on Parallel and Distributed Information Systems, pp.187-190, Jan. 1993.

[3] L. Brown and J. Wu, "Dynamic snooping in a fault-tolerant distributed shared memory," in Proc. of the 14th International Conference on Distributed Computing Systems, pp.218-226, June 1994.

[4] G. Cabillic, G. Muller, and I. Puaut, "The Performance of Consistent Checkpointing in Distributed Shared Memory Systems," in Proc. of the 14th Symp. on Reliable Distributed Systems, pp. 96-105, Sept. 1995.

[5] J.B. Carter, "Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency," Ph.D. dissertation, Rice University, Sept. 1993.

[6] M. Chelliah and M. Ahamad, "System mechanisms for distributed object-based fault-tolerant computing," Fault-Tolerant Parallel and Distributed Systems, pp.234-241, 1995.

[7] T. Fuchi and M. Tokoro, "A mechanism for recoverable shared virtual memory," University of Tokyo (manuscript), 1994.

[8] L. Gunaseelan and R. J. LeBlanc, Jr., "Event Ordering in a Shared Memory Distributed System," in Proc. of the 13th Int'l Conf. on Distributed Computing Systems, pp.256-263, May. 1993.

[9] B. Irlenbusch and J. Kaiser, "Towards a Resilient Shared Memory Concept for Distributed Persistent Object Systems," in Proc. of the 28th Hawaii Int'l Conf. on System Sciences, pp.675-684, Jan. 1995.

[10] G. Janakiraman and Y. Tamir, "Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputer," Proc. of the 13th Symposium on Reliable Distributed Systems, pp.42-51, Oct. 1994.

[11] B. Janssens and W.K. Fuchs, "Relaxing consistency in recoverable distributed shared memory," in Proc. of the 23rd International Symposium on Fault-Tolerant Computing, pp.155-163, June 1993.

[12] B. Janssens and W.K. Fuchs, "Reducing interprocessor dependence in recoverable distributed shared memory," in Proc. of the 13th Symposium on Reliable Distributed Systems, pp.34-41, Oct., 1994.

[13] B. Janssens and W.K. Fuchs, "Ensuring Correct Rollback Recovery in Distributed Shared Memory Systems," Journal of Parallel and Distributed Computing, Vol.29, pp.211-218, Sept. 1995.

[14] D. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," Journal of Algorithms, Vol.11, pp.462-491, 1990.

[15] S. Kanthadai and J. L. Welch, "Implementation of recoverable distributed shared memory by logging writes," in Proc. of the 16th International Conference on Distributed Computing Systems, May 1996.

[16] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut, "A recoverable distributed shared memory integrating coherence and recoverability," in Proc. of the 25th International Symposium on Fault-Tolerant Computing, pp.289-298, June 1995.

[17] J.-H. Kim and N. H. Vaidya, "Single fault-tolerant distributed shared memory using competitive update," Microprocessors and Microsystems, Vol.21, pp.183-196, Dec. 1997.

[18] L. Lin and M. Ahamad, "Checkpointing and rollback-recovery in distributed object based systems," in Proc. 20th Int. Symp. on Fault-Tolerant Computing, pp.97-104, 1990.

[19] N. Neves, M. Castro, and P. Guedes, "A checkpoint protocol for an entry consistent shared memory system," in Proc. of the 13th Annual ACM

Symposium on Principles of Distributed Computing, pp.121-129, Aug. 1994.

[20] P. Sousa et al., "Orthogonal persistence in a heterogeneous distributed object-oriented environment," tech. rep., IST-INESC, Lisboa, Portugal.

[21] R. Acree et al., "An object-oriented approach for implementing algorithm-based fault tolerance," in Proc. of the International Phoenix Conference on Computers and Communications, pp.210-216, 1993.

[22] G. Richard and M. Singhal, "Using logging and asynchronous checkpointing to implement recoverable distributed shared memory," in Proc. of the 12th Symposium on Reliable Distributed Systems, pp.58-67, Oct. 1993.

[23] M. Stumm and S. Zhou, "Fault tolerant distributed shared memory algorithms," in Proc. of the International Conference on Parallel and Distributed Processing, pp.719-724, Dec. 1990.

[24] O. Theel and B. Fleisch, "Design and analysis of highly available and scalable coherence protocols for distributed shared memory systems using stochastic modeling," in Proc. of the International Conference on Parallel Processing, Vol.I, pp.126-130, Aug. 1995.

[25] K.-L. Wu and W.K. Fuchs, "Recoverable distributed shared virtual memory : Memory coherence and storage structures," in Proc. of the 19th International Symposium on Fault-Tolerant Computing, pp.520-527, June 1989.

김 재 훈

e-mail : jaikim@madang.ajou.ac.kr

1984년 서울대학교 제어계측공학과 (학사)

1993년 Indiana University, Computer Science(석사)

1997년 Texas A&M University, Computer Science(공학박사)

1998년~현재 아주대학교 정보 및 컴퓨터공학부 조교수

관심분야 : 분산시스템, 실시간시스템