A Multiversion Concurrent Object Model for Distributed and Multiuser Environments

Toshio Hirotsu
hirotsu@mt.cs.keio.ac.jp

Hiroko Fujii[†]
kanko@mt.cs.keio.ac.jp

Mario Tokoro
mario@mt.cs.keio.ac.jp

Department of Computer Science, Keio University
3-14-1 Hiyoshi, Kouhoku, Yokohama 223 JAPAN

Abstract

In this paper, we describe the design and implementation of an extended concurrent object model for distributed and multiuser systems called the Multiversion Concurrent Object (MCO). In this model, an object executes multiple methods concurrently within itself, but is handled as a concurrent object. Each MCO makes a copy of its own state, called a version, for executing a method that is not interleaved with other similar copies during execution. The results of concurrent execution are merged at termination of each method. The MCO increases programmability, because it does not require any synchronization statements inside its methods. Using this model for shared objects, we can construct a system in which each user can work freely without being negatively impacted by others. Finally, a prototype implementation of MCO and performance evaluations are presented.

1 Introduction

In distributed systems, many people work and share computational resources together. Modularity of objects is useful for managing such shared resources. Each object manages its own computational resources and provides services to users and other objects. When multiple users use shared objects, one user's access is often interdependent. If many users access multiple objects, the order in which access requests are processed at each object may differ from object to object. In order to preserve consistency, the concept of transactions was incorporated into objects. In this research, which includes work on Transaction Management Object (TMO)[3], Local Atomicity[8], and Cooperative Atomicity[6], concurrency inside an object was excluded. The concurrency inside objects, therefore, become too complex to detect the dependencies among operations, from which dependencies among transactions arise, and synchronization code had to be included in the object programs.

The original concurrent object model[9] is very simple and does not consider concurrency inside an object. A concurrent object serializes all requests in its message queue, and invokes them one by one. It is very easy to write programs because the object freely accesses internal variables without synchronization while executing methods, and the dependencies among methods can be clearly determined. Another benefit is that the concurrent object can define application-specific consistency criteria through the semantics of the object. This object model is not entirely suitable for applications in distributed and multiuser environments.

In the concurrent object model, an object can send messages to other objects when executing its method, but may block itself before receiving the reply. In distributed environments, this kind of blocking may last for a long periods because of the communication delay between objects and serialization delay in the receiver message queue. Moreover, this kind of firm serialization causes dependencies among requests. If some users abort already executed requests, all subsequently executed requests must be undone and redone to invalidate the effects of those aborted requests. These dependencies can be weakened by commutative relations among methods[7]. However, in general, it is difficult to define commutativity relations among methods, and it is necessary to evaluate the different orders in which methods can be executed to increase the commutativity relations [6]. One solution to these problems involves the use of fine-grained objects, and on increase in concurrency by running multiple objects. This solution is attractive, and may be suitable for core concepts of concurrent objects. The problem is that it would require the restructuring of systems, software, and programming techniques. Nested transactions[5] is another approach used to increase concurrency, in which the sending of multiple concurrent message (asynchronous sends) are emulated as subtransactions. This technique increases concurrency in the system, but the problem of blocking still remains. Response time does not improve for an object, because the invocation of a request is delayed until the end of the previous method, even if this technique is used with concurrent objects.

Furthermore in an environment in which multiple users work together, they sometimes collaborate to abort jobs(victims), and in other cases they make a contract with each other to ignore conflicts when not aborting jobs. This means that a job can be aborted

[†]She now belongs to Research and Development Center, Toshiba Corporation.

by a user request without relation to the object state, as that, the commutative relations among methods are changed dynamically depending on the situation.

In this paper, we propose an extended concurrent object model called MCO (Multiversion Concurrent Object). In this model, an object invokes multiple methods concurrently in order to utilize maximal concurrency, while multiple versions are introduced to an object to execute methods independently, avoiding the needs for synchronization statements within a method. Each MCO maintains the sequential specification when it replies to a request by reflecting the result of any version to the other versions. This model has three benefits. First, the object does not block the execution of requests while executing other requests, so a shorter response time can be obtained. Secondly, we can reduce the cost of aborting transactions by use of a tree structured version history. Thirdly, we can write programs that do not require synchronization statements within a method.

In the next section, we describe the basic concepts of the MCO model, and show some short examples. In section 3, we describe the implementation of the MCO prototype, and evaluate its performance. In section 4, we discuss the benefits of our model, and offer our conclusions in the final section.

2 Multiversion Concurrent Object

In this section, we describe the basic concept of the Multiversion Concurrent Object, and the mechanism which preserves consistency within an object.

2.1 Basic Concepts

In order to use concurrent objects as shared objects among multiple users in distributed systems, we extended the concurrent object model to the *multiversion concurrent object (MCO)* model. When multiple users collaborate in distributed systems, one user's jobs will affect others. The result of a job may thus be propagated to others explicitly or implicitly. However, these outcomes will sometimes be undesirable. We need to keep a traces of user jobs to handle this situation. The concept of the transaction is useful for this. We consider a transaction to be a group of a user jobs, and it may be rolled back, aborted and redone to resolve any undesirable effects and preserve consistency among multiple objects. The Transaction Management Object(*TMO*), was introduced to deal with this consistency requirement among multiple objects[3]. Local Atomicity [8]and Cooperative Atomicity [6] have also been proposed for preserving consistency among multiple objects in distributed systems. In these research, the concurrency within an object was not dealt with, so Linearizability was one approach to resolving concurrency within an object. However, it preserves consistency by the use of lock mechanism inside an object during final analysis. Lock mechanisms are currently used in database systems to increase efficiency because they can be constructed from simple data structures and short operations. When applied to concurrent objects, they introduce complicated data structures and nested message sends. It is difficult to program synchronization

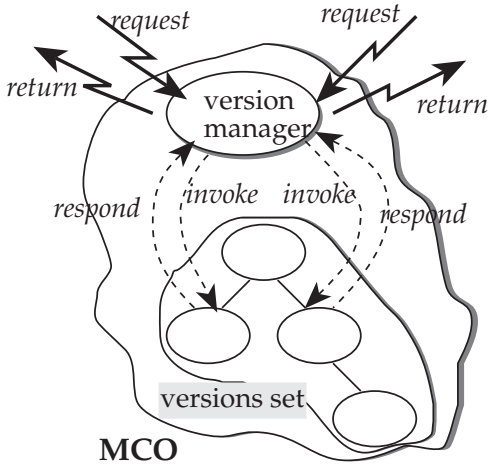


Figure 1: MCO Structure

mechanisms into methods to allow concurrent invocation for these kinds of complicated objects. Furthermore, in multiuser systems, users may abort many operations as they wish. In this paper, we propose the *Multiversion Concurrent Object(MCO)* model. This introduces a multiversion mechanism into a concurrent object in order to allow concurrent executions of the methods in an object.

The basic ideas behind MCO are concurrent invocation of multiple requests in an object, and automatic preservation of consistency among requests. When a request arrives at an MCO, the MCO creates a new version, which is a copy of the object's state, to execute that request. This means that multiple requests are not blocked in a queue and may be executed concurrently. After multiple concurrent executions, the method that finishes second is executed on a copied version of the version that finished execution first, before giving a result. This process is called *merging*, and if the effects of the execution and the re-execution are the same, the execution is called as *mergeable*. This merging process maintains the consistency inside an object. Executing the method again seems wasteful, but it is not. Each concurrent execution may prove useful later when checking commutativity among the requests. It is also useful if the previously finished method is aborted, as it can be considered as a recovery process that has been executed in advance. Furthermore, if we can define the two concurrent executions to be mergeable beforehand using the object specification (*mergeability*), the results of the methods that finish later can be returned before the re-execution.

2.2 Overview of MCO

The MCO consists of two main mechanisms called "version selection" and "merging." During version selection, a suitable version for execution is selected, and the consistency of an object is preserved during "merg-

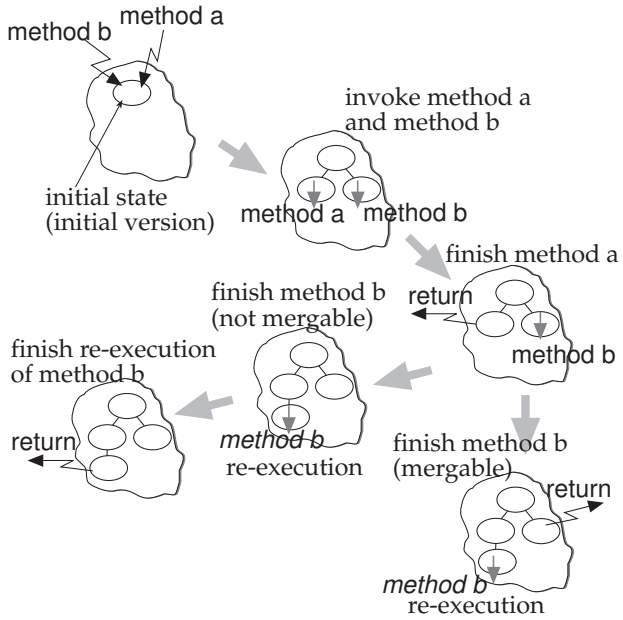


Figure 2: Method execution in an MCO

ing".

These mechanisms are achieved through two structures within the MCO, the version manager and the versions set (Figure 1). A version is a copy of the state of the object. Each version is assigned a thread to execute the method. The version manager receives requests, creates a suitable separate version for each request, and invokes the corresponding method to the request. The manager also maintains consistency when execution has finished in each version by re-executing another suitable version or returning the result. Thus, synchronization statements are not necessary inside the methods.

Each request is managed as follows (cf. Figure 2).

1. The version manager receives the request, selects a suitable version to handle the request, and creates a new version from a copy of the selected version to execute the corresponding method. This means that an MCO may invoke multiple methods concurrently within itself. This causes the versions to branch.

2. When the method assigned to the version is executed, it modifies the state of the version or sends messages to other objects.

3. When execution of the method is finished, the version returns the result to the version manager.

4. The version manager checks the execution result. If another method has not been invoked concurrently, it returns the return value of the method to the sender of the request as a response. If the MCO has invoked another method concurrently and its result is known as mergeable with respect to this result, it also returns the return value of the method to the sender and re-executes the latter method to make a linearized version history. In other cases, it creates a copy of the version that was invoked concurrently and finished first after its execution, and executes the method of the version that finished next in the newly copied version. After that, it returns the result as a response. We call this phase merging. The history that is composed of merged results of concurrent invocations (a branch of the version history), is called linearized version history.

2.3 Selecting a Version and Merging Versions

First, we define the *linearized version history*. The linearized version history is a linear tree of versions in the version tree on an MCO containing all unaborted executions in which the methods have already returned a result. We can maintain the linearized version history as follows.

Linearized version history

1. When an object is created, it contains an initial version. The initial version is a linearized version history.

2. Before an object returns the result of a method, the version for the method must be contained in the linearized versions.

3. The version that executes the aborted request is removed from the linearized versions.

Currently we use the tail version of this linearized version history as a parent of the new version for version selection. Of course, we can select another version as a parent if we can find a good criterion, and it results in a version tree which is suitable for that application.

Each request is executed in the version selected. However, some requests may cause concurrent execution of the methods in branched versions inside an MCO. The method that finished later is re-executed on the copy of the previously finished version. This process preserves the consistency inside an object and is called merging. Before we go any further, let us define the *concurrent invocation* on an MCO (Figure 3).

Concurrent invocation

When a method has finished execution, the relationship between the parent of the version for that method and the tail of the linearized version history decides the concurrency of the execution. If they are not the same, the terminated execution is called a concurrent invocation with respect to the executions between the parent and the tail of

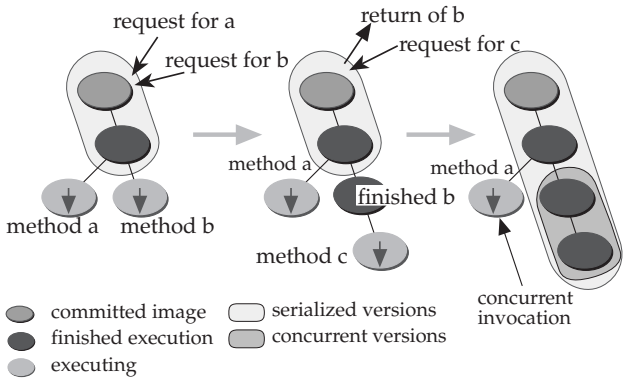


Figure 3: Concurrent invocation

the linearized version history. We call the sub-tree of the versions between the parent and the tail of the linearized version history, including the tail version, concurrent versions.

When an execution of method m is found to exhibit concurrent invocation, the method should be re-executed in the version created from the tail of the linearized version history. If method m is known to be mergeable with the concurrent versions, the object can return the result of the execution before merging, and re-execute method m after the tail of the linearized version history. Otherwise, method m is re-executed first, and the object returns the result of the re-execution to the sender. In both cases, the version that re-executes method m becomes the tail of the linearized version history.

If the effects of the re-execution differ, there is one problem. The previous execution of a method may influence other objects by sending messages to them. If the influence of re-execution differs from the previous execution, it may cause inconsistency in other objects. This is undesirable, and the previous execution of m should be rolled back to erase its effects before the re-execution.

Next let us define mergeability, which decide the re-execution requirements on merging. We define three kinds of mergeabilities, strong mergeability, weak mergeability and reverse ordered mergeability. The term *strong mergeability* means that the concurrent executions can be merged without re-execution except for reconnecting the versions in the tree. The term *weak mergeability* means that the execution requires re-execution to compensate for the internal status but the result is replied before the re-execution. It is guaranteed that the reply will not change during the re-execution. The term *reverse ordered mergeability* means that an execution can be merged without re-execution if it is inserted before another execution. This mergeability is useful when a read-only operation finishes later. In order to describe these mergeabilities, we introduce some notations. First *m(v)* means that the method m is executed on the version whose parent is v, and the notation $m(v) \rightarrow v_m$ means that the result of the execution *m(v)* becomes v_m. *Int_Effect(m(v))* denotes the effect of the method m's execution on the internal status of the object. *Ext_Effect(m(v))* denotes the effect of the method m's execution to the external objects. The three mergeabilities are defined using these notations as followings.

Strong Mergeability

When two methods, m_1 and m_2, are executed concurrently on each version from version v, suppose that m_1 has finished and $m_1(v) \rightarrow v_{m1}$. If the following relations hold, method m_2 is strongly mergeable to m_1.

$$\begin{aligned} \mathit{Int_Effect}(m_2(v_{m_1})) &\equiv \mathit{Int_Effect}(m_2(v)) \\ \mathit{Ext_Effect}(m_2(v_{m_1})) &\equiv \mathit{Ext_Effect}(m_2(v)) \end{aligned}$$

Weak Mergeability

When two methods, m_1 and m_2, are executed concurrently on each version from version v, suppose that m_1 has finished and $m_1(v) \rightarrow v_{m1}$. If the following relations hold, the method m_2 is weakly mergeable to m_1.

$$\mathit{Ext_Effect}(m_2(v_{m_1})) \equiv \mathit{Ext_Effect}(m_2(v))$$

Reverse Ordered Mergeability

When the two methods, m_1 and m_2, are executed concurrently on each version from a version v, suppose that m_1 has finished and $m_1(v) \rightarrow v_{m1}$. After that, when the execution of m_2 has finished and $m_2(v) \rightarrow v_{m2}$ holds, method m_2 is reverse ordered (strong) mergeable to m_1, where the following relations hold.

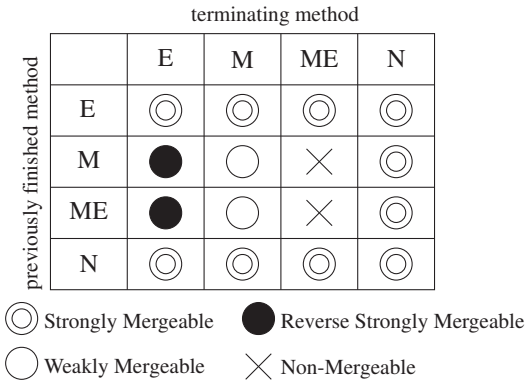
$$\begin{aligned} \mathit{Int_Effect}(m_1(v_{m_2})) &\equiv \mathit{Int_Effect}(m_1(v)) \\ \mathit{Ext_Effect}(m_1(v_{m_2})) &\equiv \mathit{Ext_Effect}(m_1(v)) \end{aligned}$$

Of course, we can also define reverse ordered weak mergeability, but it will cause cascading redo on the versions after the insertion point. This may change the result of the replied execution, and so we omitted this mergeability.

2.4 Classification of the methods

In order to reduce the cost of checking consistency, and to improve the response time of method execution, we grouped methods into classes. Each method is classified using the following terms: *"export"* (E), *"modify"* (M), and "none" (N). *"Export"* means that the internal state of the object is exported to other objects during method execution in a reply or request message. *"Modify"* means that the method modifies its own state. *"None"* means that the method causes no effects on the object's state. Table 1 shows how these methods are classified.

Table 1: Classification of the methods



When concurrently executed methods terminate, this table is used to determine whether the object should be re-executed before the return or after (mergeability). The combination of the concurrently executed methods decides the timing of the re-execution. The combination shows that the top listing methods are mergeable or non-mergeable to the left-hand methods. The combinations marked with a double circle are strongly mergeable, and they can return to the sender without re-execution. Those marked with a white circle are weakly mergeable, and that they can return to the sender before re-execution. The combinations marked with a black circle are reverse ordered mergeable, and that they can return to the sender after inserting the version previous to the concurrent versions. Those marked with a cross cannot be merged. This group of methods must be re-executed before the reply. These relations can be decided from the result of each execution. If knowledge of a method's classification is available before method invocation, the ME type methods should not be invoked because they may cause re-execution.

2.5 The Basic Action of the MCO

In this section, we describe how the MCO works. This algorithm also shows how the three types of mergeabilities are used. We assume that the effect of each message sent to other objects can be rolled back. This cost is not so expensive that the rollback is accomplished by cutting off the version from the tree. Problems with cascading rollback are discussed later.

MCO Algorithm

1. [**Version Selection**] When a request arrives, the MCO checks the tail of the linearized version history to see whether the re-execution after return by weak mergeability is running.
   (a) If such re-execution is running, version selection is delayed until the end of the re-execution while keeping the request in its message queue.
   (b) If it is not running, a new version is created as the copy of the tail of the linearized version history, and the request is assigned to the created version.
2. [**Method Execution**] The method corresponding to the request is executed on the assigned version.
3. [**End of the Execution**] When the execution of the method has finished, it is notified to the version manager. This notification corresponds to a merging request to the version manager, and the version manager keeps the request in its request queue.
4. [**Version Merging**] First, the version manager checks to see whether the re-execution after return by weak mergeability is running. If such re-execution is running, the MCO delays version merging. If it is not running, the version manager processes the version merging for each finished execution in its request queue one by one. The version merging process is described below.
   (a) If the concurrent versions is empty, the manager connects the merging version to the tail of the linearized versions, and returns the result to the sender.
   (b) If it is not empty, the manager checks whether the emerging version is mergeable to each execution in the concurrent versions.
      i. If the method executed on the merging version is strongly mergeable to each method executed in the concurrent versions, the manager connects the merging version to the tail of the linearized versions, and return the result to the sender.
      ii. If the method executed on the merging version is strongly or weakly mergeable to each method executed in the concurrent versions, the version manager returns the result to the sender first. After that, it re-executes the method that has been executed on the merging version to the tail of the linearized versions. If some messages are sent to other objects during the initial execution, the messages for erasing effects of these messages are sent before the re-execution.
      iii. If the method executed on the merging version is strongly mergeable in reverse order to the method executed on the top of the concurrent versions, the manager inserts the merging version before the concurrent versions, and returns the result to the sender.
      iv. If there is a method in the concurrent versions that is not mergeable with the method executed on the merging version, the manager crates a new version composes of the copy of the tail of the

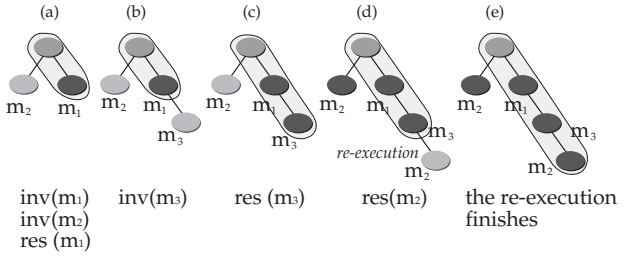


Figure 4: Execution Example

concurrent versions and re-executes the method that has been executed on the merging version on it. If some messages have been sent to other objects during the initial execution, the messages for erasing the effects of those messages are sent before the re-execution.

Next, we give a simple example to illustrate the mechanisms incorporated in MCO. We consider the case when three requests m_1, m_2, and m_3 arrive at an MCO. Here, we don't use the classification of methods for the most generic situation, so all methods require re-execution. We also denote the event that expresses the invocation of the request r as *inv(r)* and completion of the request as *res(r)*. Let us consider the case when the invocation and completion of the requests occur in the order shown below.

$$\text{inv}(m_1)\ \text{inv}(m_2)\ \text{res}(m_1)\ \text{inv}(m_3)\ \text{res}(m_3)\ \text{res}(m_2)$$

We depict the transition of the version tree structure for version selection and merging in Figure 4. In this example, the response of method m_2 was delayed for some reason, and the execution of method m_3 has finished earlier. In Figure 4(e), method m_2 is re-executed to create a linear history, but this step is skipped if this method is strongly mergeable with all methods in the concurrent versions. If it is not strongly mergeable, the reply of this method can also return before re-execution, if the method is weakly mergeable.

3 Implementation

In this section, we describe the prototype implementation and preliminary evaluation of our system. We have implemented an MCO prototype using the SUN Lightweight Process Library (Sun LWP) under SPARC Stations running SunOS 4.1. Each object has at least two threads (or lightweight processes) that handle selection and merging. Each version in an object is assigned a thread at creation time, and it executes the assigned method using the memory area allocated for its version. We also implemented (single-threaded) concurrent objects for comparison.

First, we measured the overhead for version creation. In our current prototype system, the overhead for version creation is 280.7 microseconds. This is small enough to make other costs such as the synchronization of method invocations appear comparatively large. Next, we evaluated the difference between original concurrent objects and MCOs. The reason we compared these models is that the concurrent object model provides similar programmability with the MCO model. An explicit mutual exclusion mechanism, such as locking, may provide better performance, but we cannot compare programmability and performance.

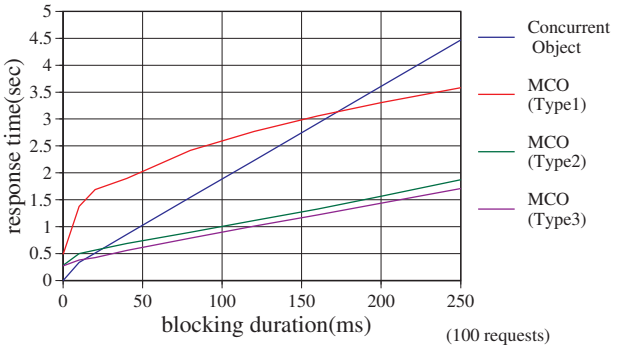


Figure 5: The relationship between blocking duration and response time

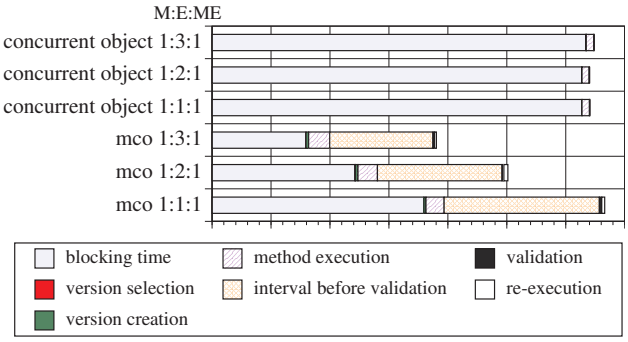


Figure 6: Break-down time of the response of a method

We measured the response time for an MCO method call. First we checked the relationship between the blocking time during a method and the response time. We implemented a kind of counter object for evaluation. It is assumed that the requests for three types of methods, ME, M and E arrive at the same rate on average, and ME method requests block for some duration by sending a message to other objects or to I/O.

Figure 5 depicts the transition of the average of the response time versus blocking during a method when all requests arrives at once. The graph annotated as type 1 shows that the MCO executes and re-executes its requests without restriction. That of type

2 shows that the MCO prevents multiple re-execution, and type 3 shows that the MCO prevents multiple re-execution and multiple invocation of ME methods whose executions will become wasteful. So MCO will be useful, if methods block for more that 180 milliseconds during execution. It shows better performance with some restriction on method invocation.

Figure 6 shows the average breakdown time between the arrival of the message and its return. All measurements represent the result of average response times a hundred requests arrive at once. In this figure, "M:E:ME=1:1:1" denotes that out of the one hundred method calls, the ratio of each types is 1:1:1. In this case, almost the entire concurrent object time is used for blocking synchronization before execution. In MCO, we can see that the overhead required for synchronization before version creation and merging also takes up a large part of the execution time, and the cost for version creation and merging is not as high.

4 Discussion

MCO requires some executions with extra functionality in comparison with the concurrent object model. First, we consider the effectiveness of these executions, and whether this model is realistic or not. Next we compare our model with other previous works, and discuss the differences. Finally, we consider the problems that need to be resolved and as future plans.

First, we focus on concurrent invocations. If an unlimited number of processors can be allocated for an MCO, this model will be effective, because the object will be able to execute all concurrent invocations in parallel. However, when there is only a limited number of processors for an MCO, is MCO effective? It seems that an MCO works in the same way as a concurrent object on a single processor, but actually it does not. During the execution of a method, some events such as the sending of messages or waiting for I/O, prevent execution from continuing. Multiple concurrent invocations can actually be executed concurrently during this blocking. This means that the MCO does not work in the same fashion as a concurrent object. It can be considered that an MCO uses this blocking time, unused in the concurrent object, to execute other methods. The concurrent invocation seems to improve response time, and the result from the prototype system shows this.

If each re-execution incurs the same overhead as an execution before merging, the MCO has no benefit except that it has prepared for recovery by pre-execution of the recovery process and increased commutativity relations. In fact, re-executions are cheaper than the previous execution of the method, because the MCO does not require actual transfer of messages during re-execution. If factors that are exported externally from the object, such as the receivers of the messages, arguments, and return values, are different upon re-execution from the ones during the previous execution, the MCO should rollback the previous execution. Based on this, an MCO should maintain a record of all exported information contained in messages sent and all accepted information from replies received, and compare the information for each send with the corresponding information recorded during re-execution. When they are identical, the accepted information during the previous execution can be used for the reply of the message to be sent. Thus, MCO can be effective in distributed systems. Furthermore, the sub-trees of versions will be useful at recovery time. One problem with MCO is that it requires a good deal of storage space to work. However, some techniques such as copy-on-write will weaken the effects of this problem.

Another merit of concurrent invocation is deadlock avoidance. In the concurrent object model, the objects can deadlock easily by sending messages to each other. In the MCO model, deadlock does not occur in this case. Deadlock is detected as an inconsistency among objects, but the commutativity relation decreases the occurrence of such inconsistency.

Next, we discuss the difference between our work and some previous works. The execution history of the MCO after merging is the same as that for concurrent objects. This means that an MCO can be handled as a concurrent object during programming. But MCO does not serialize its executions before invocation. It takes a very optimistic approach in which the MCO invokes methods concurrently without waiting for the termination of running methods. Due to concurrent invocation, the response time for each method will be shortened because shorter jobs are not blocked by longer jobs, but some extra abortion will occur to avoid undesirable effects. In order to handle the interactions among multiple users, the MCO execution mechanism of can be effective if some extra abortion is allowed to occur. When users collaborate with others, one user may invalidate his or her actions if it causes negative effects on others. We think that our object model is effective in handling interactions, and that extra abortion is necessary for interaction and is acceptable. Another benefit of MCO is that a user can select the unit of dependency. In a concurrent object, the dependency relation is a relation among methods. But in an MCO, a user can choose to deal with the dependency between methods or between transactions by selecting one policy during version selection. If objects handle the dependency only between methods, dependency between methods may cause inconsistencies in transactions. In MCO, we can choose the unit of the dependency best suited to the characteristics of the object.

Our approach is a kind of optimistic scheduling. There is some research about optimistic concurrency control [4, 1]. In these works, consistency among transactions is checked based on read/write sets of data. The harmful interleaves are checked by validation based on the read/write conflicts. In these approaches, the nested message sent from an object was not handled. Our MCO approach considers this. The other difference between the previous approaches and ours, was that previous optimistic controls are based on read/write conflicts. Our MCO uses mergeability that can decide the influence of the executions using semantic information.

In the Weihl's and Nakajima's work, concurrency inside an object was prohibited. This may cause long

periods of blocking in other transactions until the reply is returned. Our approach does not cause this kind of blocking, but may cause some extra abortion on rollback of message sending. We think these problems are a tradeoff. The basic idea of our work is that the object tries to execute requests as fast as possible, and deals with inconsistencies later. We believe that this style of computation will be important when handling interaction among multiple users, and that such extra abortion will be reduced through user negotiation. A commutativity relation is basically defined from the specification of the object in previous work. In the Multiversion Atomic Object proposed by Nakajima[6], beyond the commutativity in the specification, the re-execution of the requests was used when the execution was found to conflict with its specification. We consider that it is very difficult to define the commutativity of an object from its specification especially if the method execution influences other objects (such as message sending). Re-execution will be required in usual. In our approach, the cost of checking the commutativity is reduced, because we can use the result from previously executed concurrent invocation during the commutativity check. Furthermore, our approach is applicable when commutativity relations change dynamically, for example, where users decide commutativity from the result of executions.

Linearizability [2] handles concurrency inside an object, but synchronization statements, such as locking, are needed in programs to maintain consistency. In our work on MCO, multiple versions are used to isolate the multiple executions, and consistency is automatically preserved by merging. Programmers, thus, do not have to concern themselves with consistency among the concurrent executions.

Some research on transaction mechanisms has focused on the consistent order among multiple transactions that comprises the bulk of processing. Commutativity relations among the operations on each object was used to relax the order of constraints. In MCO, we focused on the consistency among concurrent multiple executions on an object, and we introduced mergeability to decide the constraints among multiple executions of methods. Mergeability is therefore a little different from commutativity because of this difference in purpose. Commutativity decides which of two methods can execute in what order. Mergeability decides whether an execution can be serialized before or after another concurrent execution. It can be considered as a weaker constraints than commutativity. Combination with some scheme of atomic transaction is required to maintain the total atomicity of the system. We consider that this is not so difficult because the MCO provides a linear history that is compatible with atomic transaction schemes.

We have not used the specification of the object or the characteristics of the application to determine the commutativity. For example, if we can detect methods that always form dependency relations at compile time, the number of concurrent invocations can be decreased. In addition, some mechanism to define the mergeability will be required.

5 Conclusion

In this paper, we proposed a new object model called MCO (Multiversion Concurrent Object) for distributed and multiuser environments. MCO utilizes the concurrency inside an object aggressively and is suitable for sharing objects among multiple users. The basic mechanisms in MCO are concurrent invocation and merging. Multiple versions of an object are introduced to exclude synchronization statements from inside methods, and the consistency in an object is preserved automatically by merging. MCO is handled as a concurrent object in its program. MCO has three benefits over concurrent objects. First, a requests are not blocked by other methods during execution. Secondly, it is can handle an increase in the number of requests. Finally, we can make programs without synchronization statements. We proved these benefits by implementing of a prototype and measuring its performance. We believe that this aggressive approach to the use of concurrency inside an object is useful, but the tradeoffs involved between problems, such as the frequency and the cost of aborting need to be further investigated.

References

[1] P. Goyal, T.S. Narayanan, and F. Sadri. Concurrency Control for Object Base. *Information Systems*, 18(3):167 – 180, Apr. 1993.

[2] Maurice P. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Language System*, 15(5):745–770, Nov. 1993.

[3] Toshio Hirotsu and Mario Tokoro. Object-Oriented Transaction Support for Distributed Peresistent Objects. In *Proceedings of the* 2nd *IWOOOS*, pages 13–25. IEEE, Sep. 1992.

[4] H.T. Kung and J.T. Robinson. On Optimistic Method for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213 – 226, June 1981.

[5] J. Eliot B. Moss. *Nested Transaction: An approach to reliable distributed computing.* MIT Press, 1985.

[6] Tatsuo Nakajima. *Atomicity in Multiversion Atomic Object.* PhD thesis, Keio University, Jan. 1990.

[7] William E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transaction on Computer*, 37(12):1488–1505, Dec. 1988.

[8] William E. Weihl. Local Atomicity Properties:Modular Concurrency Control for Abstract Data Types. *ACM Transactions on Programming Language System*, 11(2):249–283, Apr. 1989.

[9] Akinori Yonezawa and Mario Tokoro, editors. *Object Oriented Concurrent Programming.* MIT Press, 1987.