分散並列オブジェクトデータベースシステム「出世魚」における Distributed Wait Depth Limited デッドロック回避法の実装

田村慶一[†] 金子邦彦[††] 牧之内顕文[††]

我々は，NOW（Network Of Workstations）上で稼働する分散並列オブジェクトデータベース「出世魚」の開発を行ってきた．分散デッドロックを解決する方法として「出世魚」に DWDL（Distributed Wait Depth Limited）法の実装を行った．我々は，通信に遅延があるために，DWDL 法の実装において課題があることに気付いた．本論文では，「出世魚」における DWDL 法の実装の報告と，課題の解決方法を報告する．

Implementation of Distributed Wait Depth Limited Deadlock Avoidance Method on Parallel Distributed Object Database System ShusseUo

Keiichi Tamura,[†] Kunihiko Kaneko[††] and Akifumi Makinouchi[††]

As a distributed parallel object database system, ShusseUo is built on NOW (Network Of Workstations). DWDL (Distributed Wait Depth Limited) is proposed as a method which solves distributed deadlock. We implemented DWDL on ShusseUo. We noticed problem that DWDL can not be implemented properly happens by delay communication. This paper is a report of implementation of DWDL on ShusseUo and describe how this problem is solved.

1. はじめに

我々は，NOW（Network Of Workstations）[2]上で稼働する分散並列オブジェクトデータベース「出世魚」[10]の開発を行ってきた．NOW とは，ネットワークによりつながれたワークステーションのことであり，Cluster Computing ともいう．「出世魚」の開発では，銀行の残高照会やインターネットビジネスなどでのトランザクション処理を，NOW を構成する複数のサイトを使って並行処理して，高い処理性能を得ることを目的としてきた[6]．「出世魚」は，NOW 上に分散されたデータベースを簡単に扱えるようにするための永続分散共有仮想メモリの機能を備えている[10]．

複数のサイトに分散されたデータを扱うようなトランザクションを並行実行させると，分散デッドロックが発生する．たとえば，サイト 1 に存在するトランザクション T1 が，サイト 2 に存在するトランザクション T2 のロック解放を待ち，同時に，T2 が T1 のロック解放を待つことがありうる．このような，複数のサイトのトランザクション間で発生するデッドロックを分散デッドロックという．分散デッドロックの解決は重要な研究課題である．

NOW は，サイトの追加が容易であるから，CPU 能力の不足は問題となりにくい．むしろ，NOW 上でのデータベース処理では，ディスク I/O やサイト間の通信が全体のボトルネックとなっていたり，ある特定のデータの集まりにアクセスが集中し，結果として，ロックを保持したままで多数のトランザクションが止まっていたりするような状況（ロックの競合）があると考えた．そのような状況では，CPU は最大限には使用されていない．そのような観点で分散デッドロックに関するいくつかの文献[4),8)]を調査した結果，「出世魚」での分散デッドロックの解決法として，DWDL（Distributed Wait Depth Limited）法[4]が最も適していると判断した（その理由は 2 章で説明する）．

DWDL 法による分散デッドロックの回避は，Local Wait For Graph（LWFG，詳細は 2 章で説明）といわれるデータ構造を利用して行われる．LWFG は，あるトランザクションが，別のトランザクションのロック解放を待っているという情報を，分散して各サイト

† 九州大学大学院システム情報科学府
Graduate School of Information Science and Electrical Engineering, Kyushu University
†† 九州大学大学院システム情報科学研究院
Graduate School of Information Science and Electrical Engineering, Kyushu University

に格納するためのデータ構造である．LWFG の情報のやりとりは，各サイト間の通信によって行われるが，通信には遅延があるために，DWDL 法の実装において次のような実際的課題があることに気付いた．

- あるサイト 1 でトランザクション T1 が終了すると同時に，別のサイト 2 で，T1 が獲得済みのロックを獲得しようとして，別のトランザクション T2 がロック衝突を起こしたとする．このようなときに，サイト 2 の LWFG から，T1 が取り除かれないようなことが起きることがある．
- 異なるサイト 2，3 で，異なるトランザクション T2，T3 が，トランザクション T1（サイト 1 上）が獲得済みのロックを獲得しようとして，同時にロック衝突を起こしたとする．このようなときに，本来は 1 回でよいのに，T1 が 2 回ロールバックされることがある．
- 異なるサイトで同時にロック衝突が起こると，本来の DWDL 法よりもロールバックが増える．

以上の課題があり，単純に実装したのでは，DWDL 法では正しくデッドロック回避できない．

従来，path pushing 法[1]などの分散デッドロック検出法では，通信の遅延によって phantom deadlock という問題が生じることが指摘されてきた．phantom deadlock とは，実際にはデッドロックではないものをデッドロックであると検出してしまう現象である[1]．しかし，DWDL 法での通信の遅延に関する研究は我々の知る限りない．

我々は，「出世魚」の実装において，トランザクションの状態（詳細は 4 章で説明）の情報をサーバ中に格納し，サーバ間での通信時に，この情報を利用することで，上記の問題を解決することにした．確かに正しく動作していることを確認するために，サイト数 16 の分散データベースを使って実験を行った．

本論文の構成は次のとおりである．2 章では DWDL 法の説明を行うと同時に，DWDL 法を選んだ理由を述べる．3 章では「出世魚」での DWDL 法の実装について説明する．4 章では DWDL 法での通信の遅延に関する課題と，その解決法を述べる．5 章では本実装の評価実験を報告し，6 章でまとめる．

2. DWDL 法

2.1 LWFG

ロッキングによる同時実行制御では，各トランザクションが，データにアクセスする前に，データのロックを獲得することで，全体のトランザクション間の同期をとる[1]．以下，トランザクションが出すロック獲得のための要求のことを，ロック要求と呼ぶことにする．トランザクションがロック要求を行ったとき，他のトランザクションがすでにロックを獲得していて，当該データのロックが獲得できないことがある．このことをロック衝突と呼ぶことにする．ロックには共有ロックと排他ロックの 2 種類がある．共有ロックと共有ロックではロック衝突が発生しないが，共有ロックと排他ロックではロック衝突が発生する．以下，2 つのトランザクションのロック衝突について，ロック要求を出した方のトランザクションのことを，ロック衝突を起こしたトランザクションと呼び，すでにロックを獲得していた方のトランザクションのことを，ロック衝突の原因となったトランザクションと呼ぶことにする．

複数のトランザクションを並行実行した場合でも，トランザクションは並行実行している他のトランザクションの影響を受けず，その実行結果はトランザクションを何らかの順序で逐次処理した場合と一致しなければならない．このことをトランザクションの直列可能性という[5]．単純にデータのロックを獲得し，ロック解放を行うだけでは，トランザクションの直列可能性を保証できない．直列可能性の保証のためには，トランザクションは同時に複数のロックを保持しなければならない．直列可能性を保証するためのロッキングプロトコルとして 2 相ロッキングプロトコル[5),7]がある．

トランザクションが複数のロックを保持する場合，デッドロックが発生する．たとえば，トランザクション T1 が，別のトランザクション T2 のロック解放を待ち，T2 が T1 のロック解放を待ち，お互いにお互いのロック解放を待ち続けることがある．これをデッドロックという．

ロック衝突が起こると，ロック衝突を起こしたトランザクションは，当該データのロック解放を待ってからロックを獲得できるが，デッドロックを防ぐためにロールバックすることもある．ロールバックとは，1 度トランザクションがアボートし，トランザクションの行った処理結果がなくなった後で，トランザクションが再度，実行開始することをいう．

デッドロックが起きているか否かは Wait For Graph（WFG）で判定する[7]．WFG は，トランザクションがどのトランザクションのロック解放を待っているかを有向グラフで表現したもの[5]である．WFG に閉路があるとデッドロックである．WFG では，トランザクションを 1 つのノードで表現する．トランザクション Ti が Tj のロック解放を待っているということを，WFG のノード Ti からノード Tj への有向辺で表現

表 1 DWDL での場合分け
Table 1 The distinction of the case in DWDL.

		ロック衝突の原因となったトランザクション	
		ロック解放を待っている	ロック解放を待っていない
ロック衝突を起こしたトランザクション	ロック解放を待たれている	(A)	
	ロック解放を待たれていない	(B)	(C)

する．

分散データベースでは，Local Wait For Graph（LWFG）というデータ構造を使って，分散デッドロックを解決するのが一般的である．WFG は，データベースシステムで実行中の全トランザクションについての情報を 1 つのグラフで表現したものであるが，分散データベースでは，WFG をある特定のサイトに置くと，そのサイトがボトルネックになる[1]．LWFG は各サイトに置かれる．各々の LWFG は，当該サイトに存在するトランザクションを表すノードと，当該サイトに存在するトランザクションがロック解放を待っている他のサイトのトランザクションを表すノードおよび，それらのノード間の有向辺で構成されるグラフである．

DWDL 法では，拡張された LWFG を用いる．拡張された LWFG には，当該サイトに存在するトランザクションがロック解放するのを待っている他サイトのトランザクションのノードと，それに関する有向辺も含まれている．以下，本論文では拡張した LWFG を LWFG と呼ぶ．ロック衝突時とロック解放時に，LWFG の更新が行われる．たとえば，サイト 1 にトランザクション T1，サイト 2 にトランザクション T2 があるとき，T1 が T2 とロック衝突を起こしたとする．T1 がロック衝突を起こしたトランザクションであり，T2 がロック衝突の原因となったトランザクションである．このとき，サイト 1 の LWFG に T2 を表すノードを追加し，T1 を表すノードから T2 を表すノードへの有向辺を追加する．同時に，サイト 2 の LWFG に T1 を表すノードを追加し，T1 から T2 への有向辺を追加する．T2 がロックを解放すると，サイト 1 の LWFG から，ノード T2 と，T1 から T2 への有向辺が取り除かれると同時に，サイト 2 の LWFG からノード T1 と，T1 から T2 への有向辺が取り除かれる．

2.2 DWDL のメカニズム

DWDL 法では，LWFG の深さが 1 を超えたときに，トランザクションをロールバックする．その結果，LWFG の深さはつねに 1 以下になるのでデッドロックを回避できる．

DWDL 法は，ロールバックによって無駄になる仕事を減らす[4]ためにロールバックすべきトランザクションを注意深く選ぶ．ロールバックによって無駄になる仕事は，ロールバック時点までのトランザクションの長さ[3]と呼ばれ，WDL 法[3]や DWDL 法では，L(T)（T はトランザクション名）と書く．ロールバックすべきトランザクションの候補が複数あるとき，その決定に L(T) を用いる．L(T) の小さいトランザクションが選ばれるので，L(T) が大きいトランザクションはロールバックされない．

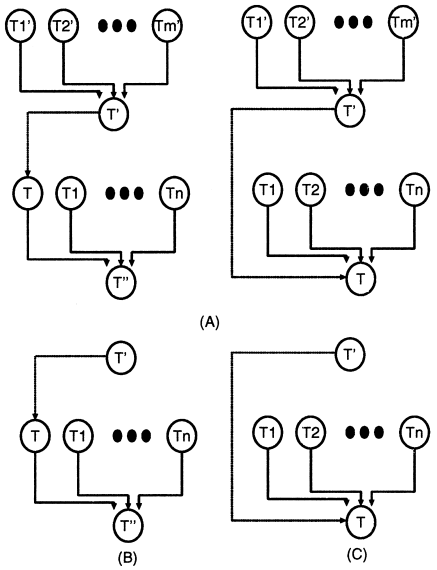

図 1 DWDL での場合分け
Fig. 1 The distinction of the case in DWDL.

トランザクションの長さ L(T) は，トランザクションによって使用された CPU，ディスク I/O，通信が関わっている．文献 3) では，L(T) はトランザクションが持っているロック数である．文献 4) では，L(T) の値は現在の時刻と開始時刻の差である．文献 4) では，実装に使用したデータベースシステムの制限により，L(T) の値として，トランザクションが持っているロック数が使用できず，トランザクションが持っているロック数に代わるものとして，現在の時刻と開始時刻の差を使用している．文献 3)，4) では，L(T) の値として，トランザクションが持っているロック数を用いるのが最も良いとしている．

DWDL 法では，ロック衝突が起こるたびに，ロッ

ク衝突を起こしたトランザクションが他のトランザクションからロック解放を待たれているかいないか，また，ロック衝突の原因となったトランザクションが他のトランザクションのロック解放を待っているかどうかで，**表 1** のように場合 (A)，(B)，(C) の 3 つの場合分けを行う．それぞれの場合を図示したのが**図 1** である．図 1 において，T'がロック衝突を起こしたトランザクション，T がロック衝突の原因となったトランザクションである．図 1 から分かるとおり，(A)，(B) の場合は LWFG の深さは 1 を超えている．

- T' を含む LWFG を調べ，T' へ向かう有向辺があれば，(A) である．さもなければ (B) かまたは (C) である．
- T を含む LWFG を調べ，T から出る有向辺があれば，(B) である．さもなければ (C) である．

文献 3) と文献 4) では，ロールバックすべきトランザクションは次の規則で選ばれる．

(A) の場合 L(T') ≥ L(T) でかつすべての i について，L(T') ≥ L(Ti') であれば T をロールバックする．それ以外であれば T' をロールバックする．

(B) の場合 L(T) ≥ L(T") でかつ L(T) ≥ L(T') であれば T" をロールバックする．それ以外であれば T をロールバックする．

(C) の場合 LWFG の深さは 1 を超えないので何もしない．

2.3 関 連 研 究

LWFG を使った分散デッドロック検出法には path pushing 法[1]がある．一方，LWFG を使って適当なトランザクションをロールバックすることで，分散デッドロックの発生を回避する方法（以下，これを分散デッドロック回避法と呼ぶ）として，Wound Wait 法[8]，Wait Die 法[8]，DWDL 法[4]が知られている．

path pushing 法では，ロック衝突を起こしたトランザクション T について，T の存在するサイトの LWFG から T を含む部分を取り出して，他のサイトに送る．受け取ったサイトは，自サイトの LWFG を使って閉路を見つけると同時に，T を含む新しいグラフを作って他のサイトに送ることを繰り返す．閉路が見つかれば，デッドロックが検出されたことになり，閉路内のノードが表すトランザクションをロールバックすることでデッドロックを解決する．閉路ができているかどうかは，閉路の中のトランザクションが別々のサイトにあると，最悪すべてのサイト間の通信を行い，全サイトの LWFG を調べなければ分からない．

分散デッドロック回避法では，LWFG を調べ，分散デッドロックを引き起こすであろう可能性のあるノードを適宜選んで取り除く（つまりロールバックする）．分散デッドロック回避法では，閉路を見つけることをしないので，実際に分散デッドロックが発生していないにもかかわらず，トランザクションをロールバックするときがある．

ロックの競合が激しい場合，並行実行するトランザクションの数（Multi Programming Level; MPL）を変化させていったときの最大スループットは，Shared Nothing アーキテクチャのマシン 4 台によるシミュレーション実験では，Wound Wait 法，Wait Die 法や分散デッドロック検出法とも比べて，DWDL 法が最も良いことが示されている[4]．ロックの競合が激しいとは，トランザクションのロック解放を待っているトランザクションの数が MPL の大半を占めていることである．スループットは，単位時間あたりに処理されるトランザクション数のことで，データベースシステムの性能を測る 1 つの指標である[7]．ロックの競合が激しい環境下では，あるトランザクション T をロールバックすることで，T のトランザクションの長さ L(T) の分だけ無駄が出るものの，T を待っていた複数のトランザクションがロックを獲得できて，実行再開し，結果として，同時に実行されるトランザクションの数が増え CPU 使用率が向上する[9]．このように，ロックの競合が激しい環境下では，DWDL 法が他の手法と比べて良いスループットが得られることから，DWDL 法の有用性は高い．

3. 出世魚でのデッドロック回避

3.1 わ か し

「出世魚」はクライアントサーバアーキテクチャの分散並列オブジェクトデータベースシステムである．「出世魚」のサーバは，「わかし」と呼ばれ，分散透過な記憶空間（永続分散共有仮想メモリ[10]と呼ぶ）の機能を提供する．永続分散共有仮想メモリに格納されたデータは，永続性を持ち，複数のサイトから分散透過にアクセスできる．「出世魚」では，クライアント・サーバ間の通信は Remote Procedure Call（RPC）とシグナルを使って行われる．

永続分散共有仮想メモリは，ページ単位に分かれたデータ空間であり，ロックの単位はページである．「出世魚」のトランザクションは，厳格 2 相ロッキングプロトコル[1),5),7)]に従ってロックを行う．厳格 2 相ロッキングプロトコルでは，獲得したロックのロック解放は，トランザクションの終了時に行う（コミットとアボートとの 2 つのことを終了と呼ぶことにする）．厳格 2 相ロッキングプロトコルは直列可能性を保証でき

るが，デッドロックの解決が必要である．

「わかし」のクライアントとサーバ間の通信は，次の 7 種類である．トランザクションに対するシグナル通知は OS のシグナル機能を使って行う．クライアントであるトランザクションは，自サイトの「わかし」と通信する．

- lock（RPC）
  クライアントが，サーバに出すロック要求．返り値として OK，WAIT，ABORT のいずれかを受け取る．lock 要求のパラメータは，永続分散共有仮想メモリの ID，ページ番号，ロックモードである．
  - OK のとき，サーバに new lock 要求を出し，実行を続ける．
  - WAIT のとき，OS の sleep 機能を使ってサスペンド状態に入る．rollback シグナルか wake up シグナルの到着を待つ．
  - ABORT のとき，サーバに abort 要求を出した後で，トランザクションは再度，実行開始する．
- new lock（RPC）
  クライアントがロック獲得した後に，トランザクションの持っているロック数の値を更新するためにサーバに出す要求．
- begin（RPC）
  クライアントからサーバへの，トランザクション開始要求．begin 要求のパラメータは，クライアントのプロセス ID である．
- commit（RPC）
  クライアントからサーバへの，トランザクションコミット要求．
- abort（RPC）
  クライアントからサーバへの，トランザクションアボート要求．
- rollback（シグナル）
  デッドロック回避のためのロールバックが決定したことを知らせるシグナル．このシグナルを受け取ると，クライアントは，サーバに abort 要求を出した後で，トランザクションは再度，実行開始する．
- wake up（シグナル）
  ロック解放を待っているトランザクションに，ロック獲得ができたことを知らせるシグナル．このシグナルを受け取ると，クライアントは，サーバに new lock 要求を出し，実行再開する．

3.2 わかしのモジュール

「わかし」は各サイトで実行されるプロセスである．「わかし」は分散ロックマネージャ，デッドロック回避マネージャ，ストレージマネージャ（**図 2**）の 3 つのモジュールから構成される．

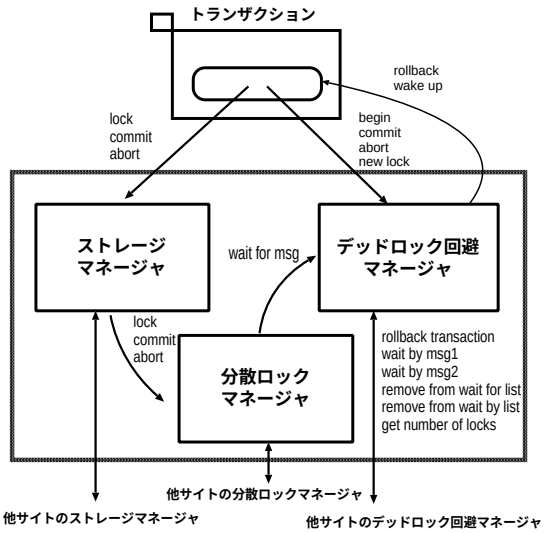

図 2 わかしのモジュールとトランザクション間の通信，モジュール間の通信
Fig. 2 Communication between modules of WAKASHI and transactions, and between modules.

3.2.1 ストレージマネージャ

永続分散共有仮想メモリのデータ転送と永続性に関する機能を持つ．クライアントから lock 要求を受け取ると，データ転送をページ単位で行う．クライアントから commit 要求を受け取ると，データをディスクに書き込む．クライアントから abort 要求を受け取ると，当該クライアントが更新したページを，トランザクション開始時点の内容に戻す．

3.2.2 分散ロックマネージャ

- lock 要求
  lock 要求をストレージマネージャを経由して受け取ると，他サイトの分散ロックマネージャと通信を行い，ロック獲得の処理を行う．返り値として，OK，WAIT，ABORT のいずれかを返す．OK はロック獲得できたことを示し，WAIT はロック衝突が起きた結果，クライアントは，ロック衝突の原因となったトランザクションのロック解放を待つことが決まったことを示し，ABORT はデッドロックを回避するためにロールバックすることが決まったこと示す．
- commit 要求，abort 要求
  commit 要求または abort 要求をストレージマネージャを経由して受け取ると，ロックを解放する．

3.2.3 デッドロック回避マネージャ

- new lock 要求
  トランザクションから new lock 要求を受け取ると，TCT（詳細は 3.3 節で説明）に格納されているトラ

ンザクションの持っているロック数の値を 1 足す．

- begin 要求
  トランザクションから，begin 要求を受け取ると，TCT，WFT と WBLT（詳細は 3.3 節で説明）にトランザクションを登録する．
- commit 要求，abort 要求
  トランザクション（T とする）から commit 要求，abort 要求を受け取ると，トランザクションの終了を待っていたトランザクション（T' とする）の実行再開を行う．T' が別のサイトに存在する場合，remove from wait for list を T の存在するサイトに出す．remove from wait for list を受け取ったデッドロック回避マネージャは T' の実行再開を行う．また，TCT と LWFG から T に関する情報を削除する．以上の一連の処理を終了処理と呼ぶ．
- wait for msg
  3.4 節で説明する手順で，デッドロック回避を行う．ロールバックすべきトランザクションに，rollback シグナルを送信する．トランザクションのプロセス ID を使ってシグナルを送信する．トランザクションのプロセス ID は後で説明する TCT に格納されている．

3.3 データ構造

デッドロック回避マネージャ内のデータ構造は，Wait For Table（WFT），WaitBy List Table（WBLT）と Transaction Control Table（TCT）の 3 つである．LWFG に関する情報は異なる形式で WFT と WBLT に格納される．

Transaction Control Table（TCT） 自サイトで実行中のトランザクションのプロセス ID，トランザクションの持っているロック数，トランザクションの状態（詳細は 4.2 節で説明）を格納するデータ構造である．プロセス ID はデッドロック回避マネージャが begin 要求を受け取ったときに登録される．

Wait For Table（WFT） 自サイトで実行中のトランザクションについて，各トランザクションがロック解放を待っているトランザクションの ID と，そのロック衝突情報が格納されている．ロック衝突情報は，ロック衝突が発生した永続分散共有仮想メモリの ID，ページ番号，そのページを排他ロックしようとしたのか共有ロックしようとしたのかという情報である．ロック衝突情報はトランザクションの実行再開時に使用する．トランザクションの ID とは，トランザクションを唯一識別するための ID である．このことを**図 3** に書いている．

Wait By List Table（WBLT） 自サイトのトランザクションについて，各トランザクションのロック解放を待っているトランザクションの ID と，そのトランザクションが持っているロック数が格納されている．このことを**図 4** に書いている．トランザクションが持っているロック数は，TCT と WBLT の両方に格納されている．

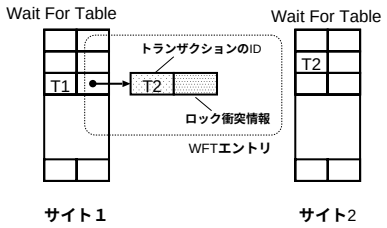

図 3 Wait For Table
Fig. 3 Wait For Table.

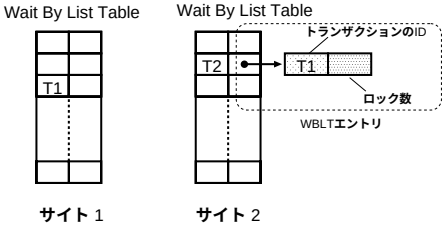

図 4 Wait By List Table
Fig. 4 Wait By List Table.

3.4 デッドロック回避メカニズム

TCT と WBLT を使って，トランザクションが持っているロック数が分かるので，我々は L(T) として，トランザクションが持っているロック数を用いる．ロック衝突が起こると，各サイトの「わかし」は互いに通信を行い，デッドロック回避を以下の手順で行う．

（1）分散ロックマネージャからデッドロック回避マネージャへの通信

分散ロックマネージャにおいてロック衝突が発生すると，分散ロックマネージャは，自サイトのデッドロック回避マネージャにロック衝突の通知メッセージである wait for msg を出す．wait for msg のパラメータは，ロック衝突を起こしたトランザクションの ID，ロック衝突の原因となったトランザクションの ID，ロック衝突情報である．

（2）デッドロック回避マネージャでの分散処理

wait for msg を受け取ったデッドロック回避マネージャは，ロック衝突の原因となったトランザクションが存在するサイトのデッドロック回避マネージャに wait by msg1 または wait by msg2 を出し，表 1 の (A)，(B)，(C) のどれであるかを判定し，ロールバックするトランザクションを決定する．

以下，ロック衝突を起こしたトランザクションを T’，ロック衝突の原因となったトランザクションを T とする．T’ の WBLT エントリと，T の WFT エントリとを調べて (A)，(B)，(C) のいずれであるかを判定する．

- T’ の WBLT エントリに要素が 1 つ以上あれば，(A) である．
- T’ の WBLT エントリがなくて，T の WFT エントリに要素が 1 つ以上あれば，(B) である．
- T’ の WBLT エントリと，T の WFT エントリの両方がなければ，(C) である．

(a) T’ の WFT エントリに，要素（T の ID，ロック衝突情報）を追加する．

(b) T の存在するサイトのデッドロック回避マネージャに，wait by msg1 または wait by msg2 を出し，返事を待つ．T’ が，他のトランザクションから終了を待たれているならば，wait by msg2，それ以外ならば，wait by msg1 を出す．これらメッセージのパラメータは，T’ の ID，T の ID と T’ のロック数である．

(c) wait by msg1 を受け取ったデッドロック回避マネージャは，T の WBLT エントリに要素（T’ の ID，T’ のロック数）を追加する．次に T の WFT を調べる．T の WFT エントリがないならば (C) で，T の WFT エントリに要素が 1 つ以上あれば (B) である．(B) であれば，(B) の場合の規則に従ってロールバックするトランザクションを決定する．wait by msg1 の返事を受け取ったデッドロック回避マネージャは，返り値 WAIT を分散ロックマネージャに返す．

(d) wait by msg2 を受け取ったデッドロック回避マネージャは，T の WBLT エントリに，要素（T’ の ID，T’ のロック数）を追加する．wait by msg2 の返事を受け取ったデッドロック回避マネージャは，(A) の場合の規則に従ってロールバックすべきトランザクションを決定する．T が他のサイトに存在する場合，get number of locks を使って T が持っているロック数を TCT から獲得する．get number of locks は他のサイトに存在するトランザクションのロック数を求めるときに使用する．T’ がロールバックするトランザクションとして選ばれたとき，wait for msg の返り値として ABORT を返す．T がロールバックするトランザクションとして選ばれたとき，T が存在するサイトのデッドロック回避マネージャに rollback transaction を出し，wait for msg の返り値として WAIT を返す．

(e) rollback transaction を受け取ったデッドロック回避マネージャは，TCT を使って，トランザクションのプロセス ID を獲得する．プロセス ID を使って，トランザクションに rollback シグナルを送信する．

（3）分散ロックマネージャでの処理

デッドロック回避マネージャから WAIT か ABORT を受け取る．WAIT を受け取ると，WAIT をストレージマネージャを経由してトランザクションに返す．ABORT を受け取ると，ストレージマネージャを経由してトランザクションに ABORT を返す．

（4）具体例

トランザクション T1 がサイト 1 に，T2 がサイト 2 に，T3 がサイト 3 に，T4 がサイト 4 に存在し，すでに，T1 が T2 のロック解放を待っていて，T4 が T3 のロック解放を待っている．T2 が T3 とロック衝突を起こした場合を例にあげてサーバ間の挙動を説明する．

（a） トランザクション T2 はサイト 2 の分散ロックマネージャに lock 要求を出す（ロック衝突が発生）．

（b） サイト 2 の分散ロックマネージャは，wait for msg をデッドロック回避マネージャに出す．

（c） サイト 2 のデッドロック回避マネージャは，wait for msg を受け取り，T2 の WFT エントリに，要素（T3 の ID，ロック衝突情報）を追加する．

（d） サイト 2 の TCT より T2 の持っているロック数を調べる．T2 はすでに T1 から終了を待たれている．よってサイト 3 のデッドロック回避マネージャに wait by msg2 を出す．

（e） サイト 3 のデッドロック回避マネージャは，wait by msg2 を受け取り，T3 の WBLT エントリに要素（T2 の ID，T2 のロック数）を追加する．

（f） サイト 2 のデッドロック回避マネージャは，wait by msg2 の返事を受け取り，(A) の場合の規則に従ってロールバックするトランザクションの決定を行うために，まず，T2 のロック数を TCT から獲得する．次に T3 のロック数を獲得するために，サイト 3 のデッドロック回避マネージャに get number of locks を出す．T2 を待っているトランザクションのロック数の最大値を T2 の WBLT エントリから獲得する．T3 がロールバックすることが決定した場合，サイト 3 に rollback transaction を出し，返り値として WAIT を返す．T2 がロールバックすることが決定した場合，ABORT を返り値として返す．

4. 通信の遅延

4.1 クライアント・サーバ間のメッセージ

通信の遅延があるために，トランザクションがサーバに要求を出し，返事を待っている間に rollback シグナルあるいは wake up シグナルが届くことがある．サーバに lock 要求または new lock 要求を出している間は，これらシグナルを受け取らない（受け取ったシグナルはブロックされる）ように，OS のシグナル機能を使って，ページフォールトハンドラのシグナルマスクを前もって設定する．返事を受け取ってからブロックされていたシグナルがあれば，まず rollback シグナルを受け取るようにシグナルマスクを設定する．次に，wake up シグナルを受け取るようにシグナルマスクを設定する．ただし，new lock 要求のときは wake up シグナルがブロックされていることはない．

commit 要求と abort 要求を出し，返事を待っている間に，rollback シグナルが届くことがある．トランザクションは，commit 要求または abort 要求を出す前に，シグナルをブロックせずに無効にするようにシグナルマスクを設定する．

4.2 分散サーバ間のメッセージ

ロック衝突を起こしたトランザクション T' とロック衝突の原因となったトランザクション T が別サイトのとき，通信の遅延があるので T' が存在するサイトの WFT の更新と，T の存在するサイトの WBLT の更新は厳密には同時に行われない．同様に，あるトランザクション T が終了したとき，T を含む複数の WFT と WBLT の更新も同時には行われない．このことで，1 章であげた 3 つの問題点〔順に問題（イ），（ロ），（ハ）と呼ぶ〕が起きる．以下，3.4 節（4）での例を用いて説明する．

問題（イ） wait by msg2 がサイト 3 に届いたとき，たまたま T3 から commit 要求または abort 要求を受け取ったデッドロック回避マネージャが，終了処理を行っている．このとき，T3 のロック解放を待っている T4 を実行再開するために，サイト 3 からサイト 4 のデッドロック回避マネージャに remove from wait for list を出して，返事を待っていることがある．返事が来る前に，wait by msg2 によってサイト 3 の WBLT エントリに要素（T2，T2 のロック数）を追加すると，サイト 3 からサイト 2 に remove from wait for list が出されず，サイト 2 の WFT エントリから T3 が取り除かれない．

問題（ロ） サイト 2 で T3 をロールバックすると決定したと同時に，たまたま他のサイト S で T3 のロック解放を待っているトランザクションがあり，サイト S でも T3 のロールバックを決定する．たまたまサイト S からの rollback transaction がサイト 2 からの rollback transaction よりも先にサイト 3 に届き，T3 に rollback シグナルが送信される．rollback シグナルを受け取った T3 はサーバに abort 要求を出す．この間に，サイト 2 からの rollback transaction がサイト 3 に届くと，T3 が終了処理中であれば，サイト 3 の TCT にまだ T3 に関する情報が登録されているため，プロセス ID を使って，T3 に rollback シグナルが送信される．4.1 節で述べたように，commit 要求または abort 要求を出しているトランザクションは，rollback シグナルを無視する．ただし，ロールバックしたトランザクションは再度，開始時点に戻り，シグナルを受け取れるようにシグナルマスクを設定する．この設定後，rollback シグナルが届くことがある．このとき，実行開始前のトランザクションへの rollback シグナルを受け取ってしまう．つまり，トランザクションに複数回の rollback シグナルが届く．

問題（ハ）（1）wait by msg2 をサイト 3 に出している途中に，デッドロック回避マネージャにトランザクション T2 に対する rollback transaction が届くことがある．または，（2）wait by msg2 をサイト 3 に出している途中に，デッドロック回避マネージャに wait by msg1 または wait by msg2 が届くことがある（ただし，T2 がロック衝突の原因となったトランザクションとして）．（1），（2）は同じ問題を生じる．以下，（2）を使って問題を説明する．wait by msg1（ロック衝突を起こしたトランザクションは T5）を受け取ったサイト 2 のデッドロック回避マネージャは，T2 の WFT エントリを調べると，要素（T3 の ID，T3 のロック数）があるので，表 1 の (B) であると判断し，ロールバックするトランザクションの決定（a）を行い，ロールバックすべきトランザクションとして T2 または T3 を選ぶ．この後，wait by msg2 の返事を受け取ったサイト 2 のデッドロック回避マネージャは，表 1 の (A) に従ってロールバックするトランザクションを決定（b）を行い，ロールバックすべきトランザクションとして T2 または T3 を選ぶ．（a）の決定ですでに，サイト 2 の LWFG の深さが 1 以下になる．つまり，（b）の決定は必要ない．また，（a）の決定がなかったとしても，（b）の決定だけで，サイト 2 の LWFG の深さが 1 以下になる．（a），（b）の決定はどちらかがあればよいが，2 つの決定がこの例では必ず行われる．よって，本来の DWDL 法よりもロールバックするトランザクションが増える．

表 2 トランザクションの状態
Table 2 State of transaction.

状態名	定義
RUNNING	begin 要求を受け取った
WAIT FOR READY	ロック衝突を起こしたトランザクションとして wait for msg をデッドロック回避マネージャで処理中
BLOCKED	デッドロック回避マネージャにおいてロック解放を待つことが決定した
COMMIT READY	commit 要求を受け取って，デッドロック回避マネージャで終了処理を行っている
ABORT READY	abort 要求を受け取ってデッドロック回避マネージャで終了処理行っている または，rollback シグナルを送信済み

T3 が終了処理であることが分かれば，wait by msg2 は T3 の WBLT エントリに要素を挿入するのをとりやめ，T3 が終了処理中であることを示す返り値を返し，T2 の WFT エントリから T3 を取り除くようにすることで，問題（イ）を解決することができる．一度，rollback シグナルを送信したトランザクションには，rollback シグナルを送信しないようにすることで，問題（ロ）を解決することができる．T2 が wait by msg2 をサイト 1 に送信中であることが分かれば，wait by msg1 を受け取ったデッドロック回避マネージャは T2 の WBLT エントリに要素（T5 の ID，T5 のロック数）を追加するだけで，ロールバックするトランザクションを決定しないでよいことが分かる．(a) の決定を行わないようになるため，問題（ハ）を解決することができる．

デッドロック回避マネージャが begin 要求，commit 要求，abort 要求，wait for msg を受け取ったときと wait for msg を受け取った後にロック衝突を起こしたトランザクションがどうなったか（ロールバックが決定したか，待つようになったのか）をデッドロック回避マネージャが TCT にトランザクションの状態（**表 2**）として記録する．

デッドロック回避マネージャは begin 要求を受け取ると，トランザクションの状態を RUNNING に設定する．wait for msg を受け取ったデッドロック回避マネージャはロック衝突を起こしたトランザクションの TCT のトランザクションの状態を WAIT FOR READY に設定する．wait for msg の処理の中で，ロック衝突の原因となったトランザクションのロック解放を待つことが決定すると，TCT のトランザクションの状態を BLOCKED に設定する．デッドロック回避マネージャはトランザクションから commit 要求を受け取ると，TCT のトランザクションの状態を COMMIT READY に設定する．トランザクションの実行再開が決定すると，BLOCKED から RUNNING に設定する．デッドロック回避マネージャはトランザクションから abort 要求を受け取ったときと，wait for msg の返り値として ABORT を返したとき，TCT のトランザクションの状態を ABORT READY に設定する．トランザクションに rollback シグナルを送信したときも，将来，abort 要求がトランザクションから来ることが決定しているため，TCT のトランザクションの状態を ABORT READY に設定する．

「出世魚」において，通信の遅延があると問題を引き起こすサーバ間のメッセージは rollback transaction, wait for msg，wait by msg1，wait by msg2 の 4 つである．通信の遅延があると問題を引き起こすメッセージは，トランザクションの状態を使い次の 4 つの場合分けを行うことで，問題（イ),（ロ),（ハ）を解決する．

- メッセージを受け取ったとき，別の処理を行わなければならない場合
- メッセージを受け取ったとき，処理を行う必要がない場合
- メッセージの返事を受け取ったとき，別の処理を行う必要がある場合
- メッセージの返事を受け取ったとき，処理を行う必要がない場合

wait by msg1 または wait by msg2 の処理では WBLT エントリに要素を追加するが，ロック衝突の原因となったトランザクション T の状態が RUNNING かまたは BLOCKED であればそのままこの処理を行う．しかしながら，T の状態が ABORT READY または COMMIT READY であれば，終了処理中であるので WBLT エントリに要素を追加してはならないことが判断できる．これにより，終了処理中のトランザクションの WBLT に要素が追加されることを防ぐ．これは，メッセージを受け取ったとき，別の処理を行わなければならない場合にあたり，問題（イ）を解決している．また，このとき，wait by msg1 または wait by msg2 の返り値として，T が終了処理中であることを示す返り値を返す．この返り値を受け取った wait by msg1 または wait by msg2 は，トランザクションの状態が WAIT FOR READY であることを確認し，WFT エントリから T を含む要素を削除する．これ

は，メッセージの返事を受け取ったとき，別の処理を行う必要がある場合にあたり，問題（イ）を解決している．

wait by msg1 を受け取ったときに，T が，WAIT FOR READY であれば，表 1 の (B) であってもロールバックするトランザクションの決定を行わない．これは，メッセージを受け取ったとき，別の処理を行わなければならない場合にあたり，問題（ハ）を解決している．

rollback transaction を受け取ったデッドロック回避マネージャは，ロールバックするトランザクションの状態を TCT を使って調べる．トランザクションの状態が，COMMIT READY または ABORT READY 以外ならば，トランザクションに rollback シグナルを送信する．COMMIT READY または ABORT READY であれば終了処理中であるので，処理内容として何も行わずに処理を終了する．これにより，複数回，トランザクションに rollback シグナルを送信することを防ぐ．これは，メッセージを受け取ったとき，処理を行う必要がない場合にあたり，問題（ロ）を解決している．

wait by msg2 の返事を受け取ったデッドロック回避マネージャは，トランザクションの状態が ABORT READY であれば，表 1 の (A) に従いロールバックするトランザクションを決定せずに，WAIT を返す．これは，メッセージの返事を受け取ったとき，処理を行う必要がない場合にあたり，問題（ハ）を解決している．

5. 実　　験

5.1 目　　的

4 章で説明した解決法が確かに正しいことを確認するために，次の 2 つの実験を行った．

（1）各場合の発生数の計測　実験用データベースを使って，4.2 節で説明した 4 つの場合（上から順に場合 1，2，3，4 とする）が数多く発生するような状況を作って実験を行い，各々の場合の発生回数を計測する．各場合は，サーバ間の通信の遅延により必要となる場合分けであるため，サイト数と MPL が大きくなると発生回数が多くなる．

（2）解決法の正しさを証明する実験　DWDL 法を「出世魚」に実装する際の 3 つの問題（1 章を参照；上から順に問題（イ），（ロ），（ハ）とする）を解決しているかどうか確認する実験を行う．確認するためのコードをデッドロック回避マネージャに埋め込む．

- 問題（イ）が解決できているかを確認するために，デッドロック回避マネージャは commit 要求を受け取ったときの，WBLT エントリの内容 A と，commit 要求の返事をトランザクションに返す前の WBLT エントリの内容 B を比べる．もし，B の中のトランザクションの ID に A にはないものが登録されていないか確認する．
- 問題（ロ）が解決できているかを確認するために，rollback シグナルを送信する部分で，同じトランザクションに 2 回以上シグナルを送っていないか確認する．
- 問題（ハ）が解決できているかを確認するために，DWDL 法によってロック衝突を起こしたトランザクションがロールバックされるとき，そのトランザクションの WLT エントリ，WBLT エントリがあることを確認する．

同時にスループット，ロールバックするトランザクションの比率とロック衝突の回数がどのような傾向になるか調べるために測定を行った．

5.2 実験用データベースとプログラム

実験用データベースは，1 種類のクラス（2 つの属性値を持つ）の多数のインスタンスからなるオブジェクトデータベースである．

- オブジェクトの数（1 つのオブジェクトのサイズ 136 バイト）793600 個 + 25600 個 = 合計 819200 個
- 管理領域を含めデータベースのサイズは 421 M バイト．
- データベースを 10 個に分割する．
- 分割した各データベースのオブジェクトの数 79360 個 + 2560 個 = 81920 個．アクセスを偏らせるために，このうち 2560 個のオブジェクトにアクセスの 20%を集中させ，残りの 79360 個のオブジェクトにアクセスの 80%を行う．
- データベースは Sun Microsystems ULTRA5_10（**表 3**）の計算機に置く．

トランザクションは 1，2，4，8，16 台の計算機で並行実行される．トランザクションは，32 個のオブジェクトにアクセスする．50%の確率でオブジェクトを更新する．90%の確率で分割したデータベースの 1 つにアクセスする．MPL を 80 まで変化させ 3 分間計測する．使用した環境を表 3 に示す．

5.3 結　　果

各場合の発生回数　**図 5** に各場合の発生回数の結果をグラフで示す．サイト数が大きくなるほど，場合 1 と場合 3 の発生回数が大きくなっている．当然，サイト数 1 では 1 回も発生していない．また，ロック衝突

表 3 実験環境
Table 3 Environment of experimentation.

項目	内容
計算機 16 台	Sun Microsystems ULTRA5（CPU UltraSPARC IIi 270 MHz，メモリ 128 M バイト，ディスク 22 GB Seek Time: 9.5 ms 7200 rpm buffer: 512 KB）
計算機 1 台	Sun Microsystems ULTRA5_10（CPU UltraSPARC IIi 440 MHz，メモリ 1024 M バイト，ディスク 34 GB Seek Time: 9 ms 7200 rpm buffer: 2048 KB × 6 RAID 5）
OS	Solaris 8
ネットワーク	100 M イーサネット–スイッチングハブ

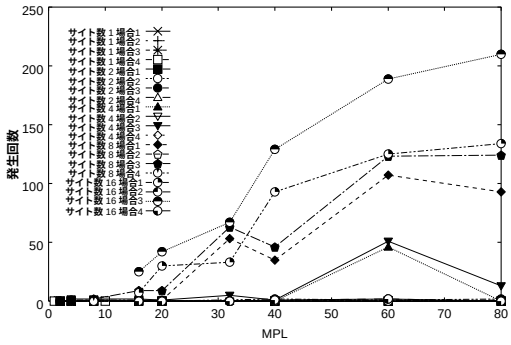

図 5 各場合の発生回数
Fig. 5 Number of the occurrences of each case.

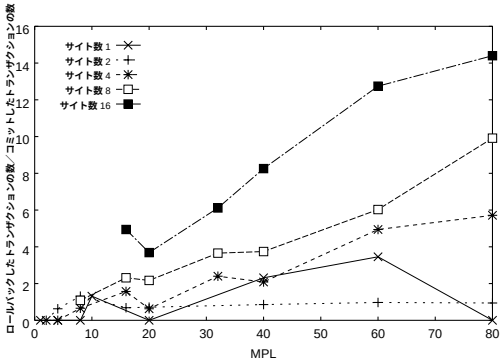

図 7 ロールバックの比率
Fig. 7 Rate of rollback.

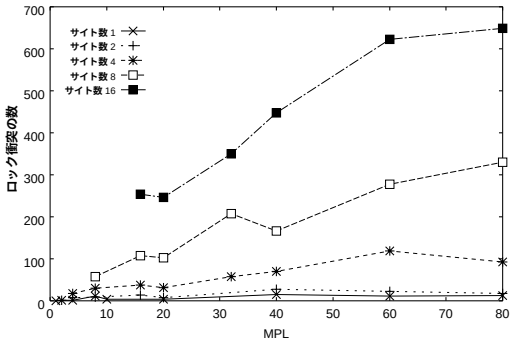

図 6 ロック衝突の数
Fig. 6 Number of conflicts.

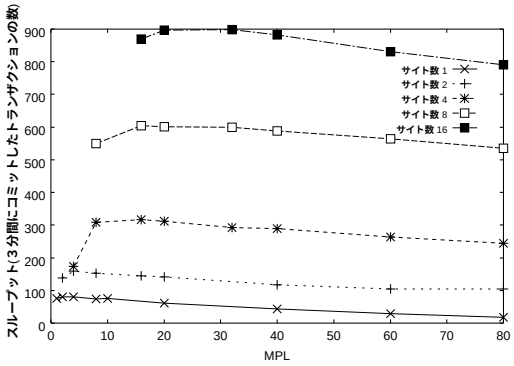

図 8 スループット
Fig. 8 Throughput.

の数（**図 6**）と，場合の発生回数は似たグラフになった．これは，ロック衝突が多く起きているほど，サーバ間のメッセージ数が多くなるためである．また，場合 2，4 はときどき 1，2 回起きる程度であった．

解決法の正しさを証明する実験 サイト数 16，MPL80 で実験を行った．耐久試験を行い，サーバにエラーメッセージが表示されないことを確認した．問題（イ),(ロ),（ハ）を確かに解決できている．

スループット，ロールバックの比率とロック衝突の総数 ロールバックの比率（**図 7**）は MPL が大きくなるほど大きくなっているが，スループット（**図 8**）はさほど落ち込んではいない．スループットとロールバックの比率，ロック衝突の数から評価できる性能評価はこれからの課題としたい．

6. むすびに

「出世魚」に分散デッドロック回避法である DWDL 法の実装を行った．今回，DWDL 法を実装を行うときの 3 つの問題を解決を行った．3 つの問題は，サーバ間の通信の遅延により発生する．我々は，「出世魚」において通信の遅延によって影響を受けるサーバ間のメッセージを，トランザクションの状態によって処理を分けることで，3 つの問題を解決した．実験によってサイト数と MPL が増えるほど，場合分けが必要に

なる回数が増えることを確認し，今回の実装の有効性を確かめることができた．加えて，サイト数16の実験によって，3つの問題を解決していることを確認した．

トランザクションがコミットまたはアボートしたとき，そのトランザクションのロック解放を複数のトランザクションが待っていたとき，その中のTがロックを獲得すると，他のトランザクションはTのロック解放を待つようになる．このTの選び方を注意深く行わないと，L(T)が大きなものがロールバックされる確率が多くなるという問題があり，性能向上のために課題として取り組んでいる．

謝辞 本研究の一部は，文部省科学研究費補助金（課題番号：10308012）の援助を受けている．また，実験データベースの製作において助言をくださいました九州大学大学院システム情報科学府の金泰勇氏に深く感謝します．

参 考 文 献

1) Bernstein, P.A., Hadzilacos, V. and Goodman, N.: *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Reading MA (1987).
2) Culler, D.E., Arpaci-Dusseau, A., Arpaci-Dusseau, R., Chun, B., Lumetta, S., Mainwaring, A., Martin, R., Yoshikawa, C. and Wong, F.: Parallel Computing on the Berkeley NOW, *9th Joint Symposium on Parallel Processing* (1997).
3) Franaszek, P.A., Robinson, J.T. and Thomasian, A.: Wait Depth Limited Concurrency Control, *Proc. 7th International Conference on Data Engineering*, Kobe, Japan, pp.92–101, IEEE Computer Society (1991).
4) Franaszek, P.A., Robinson, J.T. and Thomasian, A.: Distributed Concurrency Control Based on Limited Wait-Depth, *Proc. 12th International Conference on Distributed Computing Systems*, Yokohama, Japan, IEEE Computer Society (1992).
5) Gray, J. and Reuter, A.: *Transaction Processing: Concept and Facilities*, Morgan-Kaufmann, San Mateo, CA (1992).
6) Jin, T., Kaneko, K. and Makinouchi, A.: A Distributed Transaction Coordinator based on a Network Of Workstations with Distributed Shared Virtual Memory, *Proc. Invited Session of System Cybernetic Information (SCI) 2000*, Florida (2000).
7) Korth, H.F. and Simlberschatz, A.: *DATABASE SYSTEM CONCEPTS*, McGraw-Hill Book Co. (1991).
8) Rosenkrantz, D.J., Stearns, R.E. and Lewis, P.M.: System Level Concurrency Control for Distributed Database Systems, *TODS*, Vol.3, No.2, pp.178–198 (1978).
9) Thomasian, A.: A Performance Comparison of Locking Methods with Limited Wait Depth, *TKDE*, Vol.9, No.3, pp.421–434 (1997).
10) Yu, G., Kaneko, K., Bai, G. and Makinouchi, A.: Transaction Management for a Distributed Object Storage System WAKASHI – Design, Implementation and Performance, *Proc. 12th International Conference on Data Engineering*, New Orleans, Louisiana, Su, S.Y.W. (Ed.), pp.460–468, IEEE Computer Society (1996).

(平成12年 6 月20日受付)
(平成12年10月11日採録)

（担当編集委員 宝珍 輝尚）

田村 慶一（学生会員）
1998年九州大学工学部情報工学科卒業．2000年同大学大学院システム情報科学研究科知能システム学専攻修士課程修了．同年同大学院システム情報科学府知能システム学専攻博士後期課程入学．

金子 邦彦（正会員）
1990年九州大学工学部情報工学科卒業．1995年同大学大学院博士後期課程修了，同助手を経て，1999年九州大学大学院システム情報科学研究院助教授．情報処理学会，電子情報通信学会，ACM，IEEE Computer Society各会員．

牧之内顕文（正会員）
1967年京都大学工学部電子工学科卒業．1970年グルノーブル大学理学部応用数学科Docteur-Ingénieur取得．(株）富士通，(株）富士通研究所，九州大学工学部教授を経て，1996年九州大学大学院システム情報科学研究科教授．現在，同大学院システム情報科学研究院教授．電子情報通信学会，ACM，IEEE Computer Society，人工知能学会各会員．