データベースシステム 107－8
（1996．3．11）

ハードウェアによるデッドロック検出支援機構について

最所圭三
sai@is.aist-nara.ac.jp
奈良先端科学技術大学院大学
〒630-01 奈良県生駒市高山町8916-5

ロックを用いる並行処理制御方式では，デッドロック検出のオーバヘッドが大きい．特に，著者らが提案した多重待ち二相施錠方式では，デッドロック検出を他のロックを用いた並行処理制御方式よりも多用するため，このオーバヘッドが大きな問題になる．このため，多重待ち二相施錠方式のハードウェア化することによって並行処理制御のオーバヘッドを小さくすることを試みたが，ハードウェア量が非常に大きくなり，実現が困難であった．このため，今回，ロックを用いる並行処理制御方式で最もオーバヘッドが大きいデッドロック検出機構のみをハードウェア化することを考え，その設計を行なった．その際，ハードウェア内で並列処理できること，構造が簡単なことを設計目標にした．

Design of Hardware Deadlock Detection Mechanism

Keizo SAISHO
Nara Institute of Science and Technology
8916-5, Takayama-cho, Ikoma 630-01, Japan

Deadlock detection problem is one of the most serious problems when a locking mechanism is employed by a concurrency control mechanism. This problem is much more serious for the Multi-Wait Two-Phase Locking Mechanism, which is proposed by the authors, than other mechanisms, because the mechanism has to use deadlock detection too many times. To avoid this shortcoming, we designed a hardware to realize Multi-Wait Two-Phase Locking. Another problem is, however, appeared on the hardware. The problem is that the size of hardware becomes too large and it is hard to realize it.

In this paper, only deadlock detection mechanism, which is most costly part of the concurrency control mechanism with locking mechanism, is selected and designed in order to reduce the size of the hardware. The followings is the designing policies: (1) parallel execution in the hardware, and (2) simplified structure.

1 はじめに

データベースシステムにおいては，データアクセスと計算を並列実行して処理効率を向上させ，利用者へのサービスを公平に行なうために，複数のトランザクションを並行して処理する．この場合，各トランザクションが独立してデータにアクセスするとデータベースに矛盾を生じることがある．これを防ぐために行なう制御を並行処理制御と呼ぶ．これまで多くの並行処理制御方式が提案されているが，二相ロック方式 [1] と時刻印方式 [2] が代表的である．時刻印方式はデータの更新が少ない場合には高い性能を示すが更新が増えると後退復帰が急増し性能低下が大きい．これに対して，二相ロック方式では更新が多い場合でも安定した性能が得られ，現在最も良く使われている並行処理制御方式である．

ロックを用いる並行処理制御方式では，常にデッドロックが問題になる．ロックするデータの順序を決めることによりデッドロックを生じないようにするアルゴリズムもあるが，ロックするデータ集合が既知である，必要でないデータをロックすることがあるなどの欠点を持つ．デッドロックが生じている時には，デッドロックを構成しているトランザクションを検出し，その中の適当なものを後退復帰しなければならない．このため，デッドロックが発生しているかを検出しなければならないが，通常この処理にはそのとき扱っているトランザクションの数の自乗に比例する時間を必要とする．このオーバヘッドを避けるためにタイムアウトを用いる方法もあるが，デッドロックが生じていないにもかかわらずデッドロックと判定され後退復帰させられることがあり，高価な後退復帰が増加する．もちろん，この増加がデッドロック検出のオーバヘッドより小さければ全体の性能は向上する．

本研究においては，デッドロック検出をハードウェア化することによりデッドロック検出のオーバヘッドを小さくすることについて議論する．筆者らは並列トランザクションのための並行処理制御として提案した多重待ち二相施錠方式のオーバヘッドを軽減するために，そのハードウェア化を試みたが [3, 4]，ハードウェア量が非常に大きくなるため実現することが困難であった．ハードウェアで実行順序を管理するために，施錠情報 (どのトランザクションがどのデータをロックしているか) と待ち情報 (どのトランザクションがどのトランザクションを待っているか) をハードウェア内に持たせた．施錠情報のために，

データ数×トランザクション数

待ち情報のために，

トランザクション数×トランザクション数

に比例する領域が必要になった．扱えるデータ数に制限を加えれば，大量のデータを扱う応用には用いることができなくなる．これに対して，トランザクション数を制限しても，性能が多少低下するが，同時に扱えうトランザクション数が制限されるだけで，延べトランザクション数が制限されるわけではない．超高速のトランザクションマシンでも，瞬間的に扱っているトランザクション数はそれほど多くないと考えており，この制限はそれほど問題にならないと考えている．待ち情報を管理している部分でデッドロック検出を行なっている．このデッドロック検出機構のみ独立させてしまえば，他のロックを用いる並行処理制御方式にも応用できるという利点が出てくる．また，ハードウェア量が少なくなることにより，FPGA などのプログラマブルな論理素子を用いて，採用している並行処理制御機構に適した設計も可能となる．

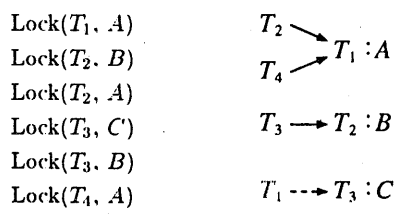
設計方針としては，ハードウェア内で並列処理できること，構造が簡単なこと，ハードウェアとホストが協調的に動作できることを目標とした．ハードウェア内で並列処理によって，最も時間のかかる処理でもトランザクション数に比例する時間で処理できる．

2 デッドロック検出

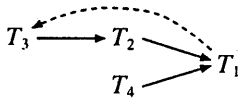
本節においては，ロックを用いた並行処理制御におけるデッドロック検出機構について説明する．

ロックを用いた並行処理制御では，トランザクションはデータを使用する前にロックし，使用後にアンロックする．このとき，ロックとアンロックを 2 つの段階に分離することにより直列可能性を満足させる方式が二相ロック方式である．

ロックされているデータに対してロックを要求すると，そのデータがアンロックされるまで待たされる．この時，現在ロックしているトランザクションと，新たにロックを要求したトランザクションとの間に待ち関係が生じる．この待ち関係をグラフにし



(a) ロックの系列　(a) データ上での待ち関係



(c) 待ちグラフ

図 1: デッドロックの検出

たものが待ちグラフである (図 1.c). 待ちグラフは，トランザクションを表すノードと，ロック要求を出しているトランザクションから，現在ロックしているトランザクションに向いた有向枝で構成される．図 1.a で Lock(T_i, A) はトランザクション T_i がデータ A にロックを要求していることを示す．データ毎に図 1.b で示すような待ち関係が形成され，これらを組み合わせることにより図 1.c で示すような待ちグラフが作られる．グラフでは，枝の先のトランザクションがアンロックしない限り，枝を出しているトランザクションはロックできない．ここで，待ちグラフで閉路が生じると，閉路を形成しているトランザクションは互いに相手がアンロックするのを待ち，デッドロックが生じる．図 1 で Lock(T_1, C) が新たに入ると図 1.b のデータ C で網掛けのトランザクションと点線の枝が加わり，図 1.c の待ちグラフには点線の枝が加わる．この結果待ちグラフに閉路を生じデッドロックになる．デッドロックの検出は待ちグラフ中の閉路の検出に置き換えられる．

3 ハードウェアデッドロック検出機構

ハードウェアの設計方針としては，1) ハードウェア内で並列処理できること，2) 構造が簡単なこと，3) ハードウェアとホストが協調的に動作できることを目標とした．

以下，ハードウェアで用いる向けデッドロック検出アルゴリズム，ハードウェア構成，ハードウェアの機能について説明する．

3.1 デッドロック検出アルゴリズム

ハードウェアの設計の前に，ここで用いるデッドロック検出アルゴリズムについて説明する．

待ちグラフ内の閉路検出は，自分が待たなければならないトランザクション集合に自分自身が含まれているかに置き換えることができる．そこで，各トランザクションに自分が待たなければならないトランザクション集合を持たせ，ロックやアンロックのときにその集合を更新することを考える．これによって，デッドロック検出はその集合の検査になる．以下，自分が待たなければならないトランザクション集合の更新のアルゴリズムを示す．

記号の定義

T_i : トランザクション．

$Wdir_i$: T_i が直接待っているトランザクション集合で，T_i がロックを要求しているデータをロックしているトランザクションの集合になる．

$Wait_i$: T_i が待たなければならないトランザクション集合で以下のように定義される．

$$Wait_i \equiv Wdir_i \cup (\bigcup_{T_j \in Wdir_i} Wait_j)$$

***Wdir* への追加**

T_i があるデータに対しロックを要求したとき，すでにロックされている場合，$Wdir_i$ にその時ロックしているトランザクション集合を加えなければならない．その時 $Wait$ の定義により，$Wdir_i$ だけでなく，$Wait_i$ および $Wdir$ に T_i を含むトランザクションの $Wait$ も更新しなければならない．$Wdir_i$ に追加するトランザクション集合を $\boldsymbol{T}$ とする．

1. $Wdir_i$ に $\boldsymbol{T}$ を加える．
2. $\bigcup_{T_j \in Wdir_i} Wait_j$ を求める．
3. $Wait_i$ および $Wait_k$ ($T_i \in Wait_k$) に 2. で求めたトランザクション集合を加える．

***Wdir* からの削除**

アンロックの時にこの操作が必要になる．T_i がロックを要求しているデータを T_j がアンロックし，か

つ，他のデータ上で T_j を直接待っていない時に生じる．ここで，削除するものが集合になっていないのは，同時に複数のトランザクションがアンロックすることがないからである．

1. $Wdir_i$ から T_j を削除する．

2. 全てのトランザクションの $Wait$ を再構築する

2. で全てのトランザクションの $Wait$ を再構築するのは，$Wdir$ に含まれるトランザクションの $Wait$ に共通に含まれることがあることと，それを識別する情報を持たないからである．識別する情報を持たせないのは，ハードウェア量をできる限り小さくしたい，構造を簡単にしたいとの理由からである．また，$Wait$ を再構築しなければならないものは，実際には $Wait$ に T_j を含むものだけだが，後述するようにハードウェア内での処理を SIMD 的に処理することを考えており，全てのトランザクションの $Wait$ を再構築するほうがハードウェアの構造を簡単になると考えたためである．さらに，1 つのトランザクションが高々1 つデータでのみ待つ場合は，$Wait$ に T_j を含むものから $T_j \cup Wait_j$ を削除するだけでよいが，汎用性を優先して全てのトランザクションの $Wait$ を再構築することにした．

***Wait* の再構築**

以下のアルゴリズムによって再構築する．ここで $wait$ は作業用の集合である．

```
全ての Wdir を wait にコピーする
for all transactions Ti
  for all transactions Tj
    if (Ti ∈ waitj) then
      waitj = waitj ∪ waiti
全ての wait を Wait にする
```

外側のループで，トランザクションを 1 つづつ取り出し，内側のループで，その $wait$ を含めなければならないトランザクションの $wait$ に加えている．

次に，このアルゴリズムで $Wait$ を正しく再構築できることを示す．

$T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$ となるパスがあると仮定する．

1. $T_{i-1} \rightarrow T_i$ であるので，$wait_i$ を加える操作で $wait_i$ が $wait_{i-1}$ に加えられるので，この時点で，$wait_{i-1} \supseteq wait_i$ が成立する．

2. 1. 以降は，$wait_{i-1} \supseteq wait_i$ であるため，$wait_i$ に $wait_k$ が加えられる時には，$wait_{i-1}$ にも $wait_k$ が加えられる．

   したがって，1. 以降は常に，$wait_{i-1} \supseteq wait_i$ が成立する．

3. 本アルゴリズムでは，1. は必ず発生する

最終的には $wait_{i-1} \supseteq wait_i$ となる．最後に $wait$ は $Wait$ にコピーされるので $Wait_{i-1} \supseteq Wait_i$ となる．この結果，

$$Wait_1 \supseteq Wait_2 \supseteq \ldots \supseteq Wait_n$$

が成立し，$Wait_1$ は $\{T_2, \ldots T_n\}$ を含む．全ての可能なパスでこれが成立するため，本アルゴリズムによって $Wait$ に含まれなければならないトランザクションは全て含まれる．

また，2. で加えられるトランザクション集合は，1. で加えたトランザクションからのパスが存在するので，必ずパスが存在する．これはパスが存在しないトランザクションを $Wait$ に含まないことを表す．

以上，本アルゴリズムによって $Wait$ を正しく再構築できる．

3.2 ハードウェア構成

本稿で提案する，ハードウェアデッドロック検出機構のブロック図を図 2に，各トランザクションの待ち情報を管理する部分の詳細を図 3 に示す．

ハードウェアで扱えるトランザクション数を N とする．ハードウェア内では i 番目のトランザクション (T_i) は i ($1 \leq i \leq N$) 番目のビット (LSB を 1 番目，MSB を N 番目のビットとする) に対応させる．これは，ハードウェア内でのトランザクション集合は N ビットのビットパターンで表すためである．この結果，集合和はビット毎の **or** 演算，集合から要素を削除する演算はビット毎の **not** 演算と **and** 演算の組み合わせ，集合積はビット毎の **and** 演算でそれぞれ実現できる．

はじめに，図 2 を説明する．

制御部　ホストとのインタフェースと待ち情報部の制御を行なう．制御機構は，ホストからのコマンドを解析するデコーダ，実行するシーケンサなどがから構成される．DATA1 と DATA2

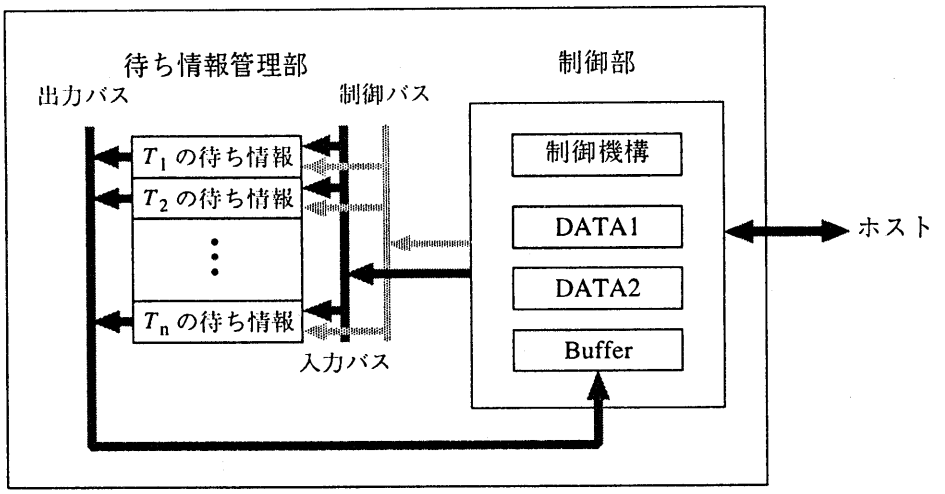


図 2: ハードウェア構成

は，コマンドの引数を保存する N ビットのレジスタである．また Buffer は，待ち情報管理部での処理の結果を保持する N ビットのレジスタである．

待ち情報管理部　トランザクション毎の待ち情報を管理する N 個の要素からなる．各要素は互いに通信を行なわない．詳細は図 3 の説明で行なう．

入力バス　制御部から待ち情報管理部にトランザクション集合を与えるためのバスである．

出力バス　待ち情報管理部での演算結果を制御部に返すためのバスである．バスには同時に複数の要素を出力できるが，その結果は or される．これにより，バス上で集合和を実現できる．

制御バス　制御部が待ち情報管理部に処理を指示するためのバスである．

次に，図 3 の待ち情報管理部の要素の詳細を説明する．

Wdir　2 節で説明した $Wdir$ を保持するレジスタ．

Wait　2 節で説明した $Wait$ を保持するレジスタ．

ID　自分が扱うトランザクションを示すもので，そのトランザクションのみの集合である．

演算器　次節で説明するハードウェアの機能を実現するための演算を行なう．

選択タグ　要素内の状態を保持するフラグで，出力ゲートの制御，$Wdir$ および $Wait$ への書き込みの制御を行なう．これらのフラグへの書き込みは，制御部から個々のフラグに指示が行なわれる．書き込む内容は，演算器の出力が空集合でなければ真に，それ以外は偽を書き込む．このフラグが真でかつ制御バス上から許可信号が入った時にゲートのオンにしたり，$Wdir$ や $Wait$ への書き込みが行なわれる．

出力ゲート　オンのときは演算器の出力をそのまま出力し，オフの時は空集合を出力する．

3.3　ハードウェアの機能

本節では，3.2 節で説明したハードウェアが備える機能について説明する．

1. setnum(n)
   現在使用している要素番号の最大値を与えるもので，reconfig の時の範囲を示す．制御部に保存される．

2. clear(i)
   $Wdir_i$ および $Wait_i$ を空にする．

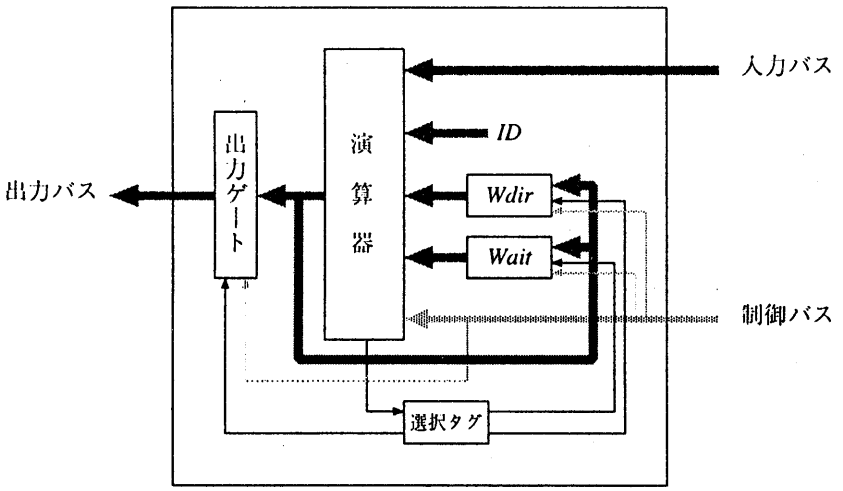


図 3: 待ち情報管理の詳細

3. add(i,$\boldsymbol{T}$)
   $Wdir_i$ に $\boldsymbol{T}$ を加え，関連する $Wait$ を更新する．太字にしているのは集合を表すためである．更新する $Wait$ は $Wait_i$ および $Wait_j$ ($T_i \in Wait_j$) である．また，更新は，

$$\bigcup_{T_k \in \boldsymbol{T}} Wait_k$$

   を加えることである．

4. delete($\boldsymbol{T}$, i)
   $Wdir_j$ ($T_j \in \boldsymbol{T}$) から T_i を削除し reconfig する．

5. delete_all(i)
   全ての $Wdir$ から T_i を削除し reconfig する．

6. reconfig
   全ての要素の $Wait$ を再構成する．

7. is_deadlock
   デッドロック検出を行なうもので，全ての要素に対し $Wait$ に ID が含まれていないかを検査するものである．

8. is_wait(i, $\boldsymbol{T}$)
   T_i が $\boldsymbol{T}$ を待たなければならないかを検査する．$Wait_i \cup \boldsymbol{T}$ が空でない場合は真を返し，それ以外は偽を返す．

9. is_waited(i, $\boldsymbol{T}$)
   $\boldsymbol{T}$ に含まれるトランザクションが T_i を待たなければならないかを検査する．

$$T_i \in \bigcup_{T_k \in \boldsymbol{T}} Wait_k$$

   が成立すれば真を返し，それ以外は偽を返す．

次に add, reconfig を実現するために制御部での処理および待ち情報管理部への命令の系列を示す．誌面の制限で 2 つの機能のみ説明するが他の機能も簡単に実現できる．

ここで IBUS および OBUS をそれぞれ入力バスおよび出力バスとする．また，第 1 引数は DATA1，第 2 引数は DATA2 にそれぞれ格納されているとする．IBUS への出力するデータ，演算器への指示，更新するフラグの種類，ゲート制御や書き込み制御の項目 (OBUF, $Wdir$, $Wait$ に指示するフラグをそれぞれ _OBUF, _$Wdir$, _$Wait$ とする) の前にそれぞれ `D:`, `O:`, `F:`, `W:`をつける．また，制御部内での内部処理を `I:`とする．

[add]

1. $Wdir_i$ および $Wait_i$ の選択
   `D:` DATA1
   `O:` ID and IBUS
   `F:` _$Wdir$, _$Wait$

2. $Wdir_i$ に $\boldsymbol{T}$ を追加
   D: DATA2
   O: *Wdir* or IBUS
   W: *Wdir*
3. $T_j \in \boldsymbol{T}$ となる $Wait_j$ を選択
   D: DATA2
   O: *ID* and IBUS
   F: _OBUF
4. 3. で選択した *Wait* の和を求める
   O: *Wait*
   W: OBUF
   I: OBUF を Buffer に書き込む
5. $Wait_i$ に 4. で求めた集合を追加
   D: Buffer
   O: *Wait* or IBUS
   W: *Wait*
6. $T_i \in Wait_j$ となる $Wait_j$ を選択
   D: DATA1
   O: *Wait* and IBUS
   F: _*Wait*
7. 6. で選択した *Wait* に 4. で求めた集合を追加
   D: Buffer
   O: *Wait* or IBUS
   W: *Wait*

[reconfig]

1. 全ての要素の選択，およびループの初期化
   D: 全てのビットを 1
   O: *ID* and IBUS
   F: _*Wait*
   I: ループカウンタを 1 にする
2. 全ての要素の *Wait* を *Wdir* にする
   O: *Wdir*
   W: *Wait*
3. $Wait_i$ を選択
   D: ループカウンタに対応するビットのみ 1
   O: *ID* and IBUS
   F: _OBUF
4. $Wait_i$ を Buffer にコピー
   O: *Wait*
   W: _OBUF
   I: OBUF を Buffer に書き込む
5. $T_i \in Wait_j$ となる $Waitj$ を選択
   ループカウンタの更新
   D: ループカウンタに対応するビットのみ 1
   O: *Wait* and IBUS
   F: _*Wait*
   I: ループカウンタを 1 つ増やす
6. 5. で選択した *Wait* に $Wait_i$ を追加
   ループの終了の検査
   D: Buffer
   O: *Wait* or IBUS
   W: _*Wait*
   I: ループカウンタが n より小さければ
   3. へ行く

reconfig 以外の機能は数クロックで実現できる．また，reconfig も n に比例する時間で実現できる．

4 ロックプリミティブの実現

本節では，3.3 節で述べたハードウェアの機能を用いた，ロックおよびアンロック操作を示す．ここで，1 つのトランザクションに対し，同時に複数のデータのロック待ちを許し，さらにロックは複数のモードを持つものとする．また，ロックのたびにデッドロック検査を行なうものとする．以下，T_i がデータ A に対してロック/アンロック要求を出すものとして説明する．

[ロック]

- A を $\boldsymbol{T}$ が競合するモードでロック中:
  この要求でデッドロックを生じないか検査
  → is_waited(i, $\boldsymbol{T}$) をハードウェアに発行
  - 生じる場合:
    T_i か $\boldsymbol{T}$ に含まれデッドロックの要因になっているトランザクションを後退復帰
  - 生じない場合:
    add(i, $\boldsymbol{T}$) をハードウェアに発行
    T_i を A のロック待ちキューに接続
- A がロックされていないか競合しない場合:
  - $\boldsymbol{T}'$ が A をロック待ちの場合:
    T_i が $\boldsymbol{T}'$ を待っていないか検査
    → is_wait(i, $\boldsymbol{T}'$) をハードウェアに発行
    * 待っている場合:
      T_i を A のロック待ちキューに接続
    * 待っていない場合:
      T_i にロックさせる
      $\boldsymbol{T}'$ に含まれ T_i とロックモードが競合するもの (T_j) について，add(j,{T_i}) をハードウェアに発行 (add の処理は

短いのでホスト側を待たせることはないと考えている)

- A をロック待ちのトランザクションがない場合: T_i にロックさせる

[アンロック]

1. 1つのトランザクションに同時に複数のデータのロック待ちを許すので，A をロック待ちのトランザクションが，他のデータ上でも T_i を待っている可能性がある．このため，A をロック待ちのトランザクションで T_i と競合するトランザクションの全てに対し，他のデータで T_i を待っていないトランザクション集合 ($\boldsymbol{T}$) を作る．

   1つのトランザクションが1つのデータでのみロック待ちするのであれば，A で T_i と競合するトランザクションが $\boldsymbol{T}$ になるので単純になる．

2. delete($\boldsymbol{T}$, i) をハードウェアに発行

3. A をロック待ちのトランザクションで優先順位の高いものから1つづつ選択し，

   A がロックされていないか競合しない $\wedge$ ロックの結果デッドロックを生じない

   を満足していればロックさせる．

   ロックの結果デッドロックを生じないかどうかの検査は，ロックの時と同様に is_wait を発行する．

   2. の delete の処理はハードウェアで高速に実現できるといっても，n のオーダであり無視できないくらいに時間がかかることがある．このため，最初の is_wait を発行するときに待ちが生じることがある．これに対しては，A をロック待ちのトランザクション全てに対して，ロックの条件の第1項のみを先に検査して，候補を絞り込むなどの処理を行なうなどによって待ち時間を短くできることもある．

5 まとめ

以上，ハードウェアによるデッドロック検出機構を設計し，ロック/アンロックにおける使い方を説明した．

デッドロック検出の部分のみハードウェア化することにより，ハードウェア量を小さくすることができた．現在の集積度でも1チップで数100のトランザクションを扱えると見積もっている．1チップに集積することにより 3.3 節で示した処理の各ステップを1クロックで実行でき，高速動作が可能である．

ハードウェア内で，処理中のトランザクション数に比例する並列度で処理することによって，最も時間がかかる処理である，全てのトランザクションの $Wait$ を再構築する処理を，トランザクション数に比例する時間で実現できるようになった．また，待ち情報管理部の構成要素が全く同一で互いに独立しているためチップに余裕が出たときの拡張が非常に容易である．

今後は，1) VHDL や NTT の SFL 等を用いて論理設計，2) FPGA を用いてプロトタイピングを行なうことによる，ハードウェア量の見積りおよび性能評価，3) ホスト側のプログラムの開発等を予定している．

参考文献

[1] K.P.Eswaren, J.N.Gray, R.A.Lorie and I.L.Traiger, "The Notion of Consistency and Predicate Locks in a Database System," CACM, Vol.19, No.11 pp.624-633, 1976.

[2] P.A.Bernstein and N.Goodman, "Timestamp-based Algorithms for Concurrency Control in Distributed Database Systems," Proc. of VLDB, pp.285-300, 1980.

[3] K.Saisho and Y.Kambayashi, "Multi-Wait Two-Phase Locking Mechanism and its Hardware Implementation," Proc. of 5th Intl. Workshop on Database Machines, pp.198-211, 1987.

[4] 最所, 上林: 並列トランザクションのための並行処理制御について, 信学技報, CPSY90 − 40, 1990.