システムソフトウエアと
オペレーティング・システム　60－29
マルチメディア通信と分散処理　61－29
（１９９３．　７．　９）

OSIディレクトリ情報ベース（DIB）のための高速名前解析処理方式

西山 智　　横田 英俊　　小花 貞夫　　浅見 徹　　鈴木 健二

国際電信電話株式会社　研究所

OSIディレクトリ情報ベース（DIB）の高速化にはディレクトリ操作の対象となるエントリを確定する処理（名前解析処理）の高速化が重要となる。従来の実装では操作で指定された提示名を基にディレクトリ情報木（DIT）の木の枝を1段毎に探索する名前解析処理方式を用いている。二次記憶上に格納されたDIBでこの方式を用いると、DITでのエントリ間の上位／下位に関する情報（DIT情報）を検索するために、a) 操作実行に必要なデータ（エントリやDIT情報）へのアクセス回数が増加する、b) アクセスするデータの局所性が低下するため、バッファのヒット率の低下からエントリに対する読み出し（または更新）が遅くなる、c) 名前解析処理性能がDITの形態（深さや各節での分岐数）に依存する、という問題がある。そこで本稿では識別名のハッシュ値をキーとしてエントリを格納し、指定された提示名のハッシュ値で検索することで直接に対象エントリを確定する名前解析方式を提案する。ここではハッシュ関数としてDESに基づくMACスキームを用いた関数を使用し、ハッシュ値の衝突の際には外部ハッシュ法を用いて対処している。また、エントリ格納のためのデータ構造としてはB-木を用いている。筆者らが開発済みの高速OSIディレクトリ用DBMS（ASSIST/D）上にこの名前解析処理方式を実装し、評価を通じて本方式の有効性を実証した。本方式により名前解析処理時間を $O(log(\text{エントリ件数}))$ に抑えることが可能となった。

High-Performance Name Resolution Method for OSI Directory Information Base (DIB)

Satoshi NISHIYAMA　　Hidetoshi YOKOTA　　Sadao OBANA
Thoru ASAMI　　Kenji SUZUKI

KDD R & D Laboratories
2-1-15, Ohara, Kamifukuoka-shi, Saitama 356, JAPAN

Name resolution (NR) method, a method to find an entry from OSI Directory information base (DIB) which matches the purported name specified in a Directory operation, is a key to improving the performance of DIB. DIB is modeled as a tree of entries, called Directory Information Tree (DIT), according to the Distingished Name (DN) given to each entry. The existing OSI Directory implementations use NR methods, which resolve the relative DNs (RDNs) in the purported name one by one, by navigating the DIT. This paper proposes a new NR method, which hashes the whole purported name and findes the entry stored in a B-tree with the hashed result as the key. The empirical result shows that the new NR method has asymptotic complexity of $O(log(n))$, where n is the number of entries stored in DIB, and performs much better than the existing NR methods, especially for large DIBs.

1. はじめに

通信相手の接続番号（アドレス）や通信能力等に関する情報を提供する通信支援データベースの代表的なものとして開放型システム間相互接続（OSI）ディレクトリ[1]がある。OSIディレクトリに格納されるエントリの集合であるディレクトリ情報ベース（DIB）は、各々のエントリを一意に識別するために付与された名前（識別名）に従ってディレクトリ情報木（DIT）と呼ばれる木構造にモデル化される。ディレクトリ操作は、指定された提示名（Purported Name）から操作対象のエントリを確定する処理（名前解析処理）とエントリに対する検索（または更新）処理の2段階で実行されるが、操作の高速化のためには名前解析処理の高速化が重要となる。

従来のDIBの実装[2, 3, 4, 5]は、OSIディレクトリの標準／勧告に示される論理的な名前解析処理の手続きをそのまま名前解析処理方式として実装している。この名前解析処理方式はDITにおけるエントリ間の上位／下位関係を表現する情報（以下、DIT情報と呼ぶ）を検索しながら提示名に含まれる相対識別名（RDN）を1段（あるいは複数段）毎に解析する。この方式では名前解析処理において常にDIT情報を検索することが必要となるため、二次記憶上に格納されたDIB上でこの方式を用いると、a) 操作実行のために必要なデータ（エントリまたはDIT情報）へのアクセス回数が増加する、b) アクセスするデータの局所性が低下し、主記憶上のバッファのヒット率が低くなることでエントリに対する読み出し（または更新）速度を低下させる、c) 名前解析処理の性能がDIBに格納されたDITの性質（深さや各節での分岐数）に依存する、という問題がある。

そこで本稿ではこれらの問題点を回避するため、DIT情報を検索する代わりに、提示名のハッシュ値をキーとしてB-木に格納されたエントリを検索することにより、直接に対象エントリを確定する名前解析方式を提案し、評価を通じてその有効性を示す。

2. OSIディレクトリのDIB

OSIディレクトリの標準[1]で規定されるDIBの特徴を述べる。

2.1 DIBのデータ構造

DIBは、各エントリの名前管理（一意に識別できる名前を付与すること）のために、また、複数のディレクトリシステムエージェント（DSA）にまたがってDIBを分散管理できるようにするために、ディレクトリ情報木（DIT）と呼ばれる木構造で論理的に表現される（図1）。DIT木構造における木の節はエントリを、また木の枝はエントリ間の従属関係を表す。一つの節から出る木の枝にはそれぞれ他の枝と異なる名前（相対識別名：RDN）が付与される。木構造のルートから特定のエントリに至る相対識別名の並びは識別名（DN）と呼ばれ、そのエントリを一意に識別する。エントリは接続番号や通信能力等の属性の並びからなり、また各属性は任意の数の属性値を持つことができる。属性値の型は抽象構文

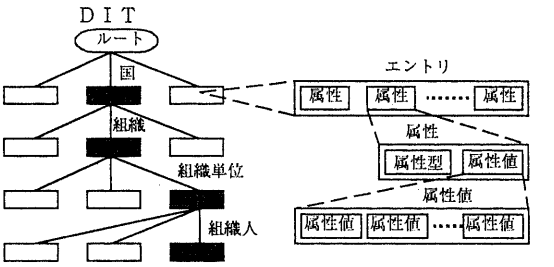

図1: ディレクトリ情報木（DIT）の構造

表1: DIBに対する操作

種別	操作名	操作の内容
検索	Read	エントリの情報の読み出し
	Compare	エントリの情報と指定値の比較
	Abandon	既に発行した操作の中止
	List	下位のエントリ一覧の表示
	Search	DIT部分木に対する条件検索
更新	AddEntry	エントリの追加
	RemoveEntry	エントリの削除
	ModifyEntry	エントリ内容の変更
	ModifyRDN	エントリの相対識別名の変更

記法1（ASN.1）[6]により規定される。

2.2 DIBに対する操作

DIBに対して、指定する識別名（提示名）からエントリの情報を読み出すRead操作、特定の条件に合致するエントリを検索し情報を読み出すSearch操作、エントリを追加するAddEntry操作等、表1に示す9種類の操作が定義されている。

3. 従来のDIB実装における名前解析処理方式の問題点

ディレクトリ操作は以下の2段階で実行される。

(1) 提示名から操作の対象となるエントリを確定する。

(2) エントリに対して検索、更新等の操作を行なう。

このうち (1) の段階は名前解析処理と呼ばれ、別名(Alias) の展開やDIBが複数のDSAに分散格納された場合の分散処理も含む。OSIディレクトリの標準／勧告では図2に示す名前解析の論理的な手続きを示しており、具体的な実現方式は実装に任されている。

従来のDIBの実装[3, 4, 5]は、図2に示す論理的な名前解析の手続きをそのまま処理方式として実装しており、エントリ内あるいは専用データとして格納されたエントリ間の上位／下位関係を示す情報（DIT情報）を、RDNの1段毎に検索して名前解析を行なっている。また[2]はRDNを複数段まとめて名前解析を行なうが、DIT情報を検索している点で図2の拡張と分類される。二次記憶上に格納されており主記憶上にバッファリングを行なっているDIBでは、これらの名前解析処理方式は、DIT情報を検索するために高速化の観点から以下のような問題点をもつ。

a) 操作実行のために必要なデータ（エントリまたはDIT情報）へのアクセス回数が増加する。

```
名前解析処理(提示名)
{
  CP = 提示名と最大限一致する名前コンテクスト;
  if (CP==NULL)
    上位参照の示すDSAに転送する;
  else
    ローカル名前解析処理(CP, 提示名);
}

ローカル名前解析処理(CP, 提示名)
{
  Matched = CP;
  while (1)
    {
      if (Matchedの指すエントリはalias)
        aliasを展開し再度名前解析処理を行う;
      if (Matched == 提示名)
        エントリを確定したので名前解析処理を終了;
      DIT情報を参照し
              次のRDNと一致を図る;
      switch (結果)
        {
          case エントリと一致:
            Matchedに RDNを追加;
            break;
          case 下位参照と一致:
            下位参照の指すDSAに転送して終了;
          case 一致せず:
            名前解析失敗;
        }
    }
}
```

図 2: 名前解析処理手続き

b) ランダムなディレクトリ操作を想定した場合、アクセスするデータの局所性が低下するため、エントリのみを読み出す（または更新する）場合と比較して主記憶上のバッファのヒット率が低くなる。結果として、名前解析処理後のエントリに対する読み出し（または更新）速度が低下する。

c) 名前解析処理の性能がDIBに格納されたDITの性質（深さや各節での分岐数）に依存する。即ち、RDNの1段（複数段）毎にDIT情報の検索を行なうので名前解析処理に要する時間は提示名に含まれるRDN数に従って増加する。また、各節での分岐数が多いDITに対しては、DIT情報に示された多数の枝からRDNに一致する枝を検索する必要があり、1段当りの解析速度が低下する。

4. ハッシュによる名前解析処理方式の提案

3.章で示したように、従来の名前解析処理方式の問題点はDIT情報を検索することにより生じている。本章では、DIT情報を検索する代わりに、提示名をハッシュした結果をキーとして対象エントリを確定する名前解析処理方式を提案する。

以下ではまず対象とするDIBに関する前提条件を示し、次に、ハッシュによる名前解析処理方式の基本方針を示す。さらに本方式を実現するためのエントリの格納方式および名前解析処理手順を示す。

4.1 対象とするDIBに関する前提条件

ここでは汎用的なDIBを対象とするために以下に示す前提条件を設ける。

前提条件1)

標準／勧告に定めるように識別名として任意の木の深さ（RDN数）を許し、また1つのRDNにおける属性値の数、長さも制限しない。

前提条件2)

各エントリに付与される識別名は予め定まっているものではない。

前提条件3)

10^6件といった大量のエントリがDIBに格納されうる。従って、このDIBは二次記憶上に構築され、高速化のためにバッファリングを行なっている。

前提条件4)

格納されるエントリに対してランダムに操作要求が発行される。

さらにここでは、OSIディレクトリにおける分散処理の形態から、構築するDIBが一般性を失わない妥当な制限として以下の前提条件を設ける。

前提条件5)

分散処理のための知識（例えば下位参照、上位参照など。別名に関する知識も含める）の件数は格納されるエントリ件数と比較して相対的に少なく、主記憶上に格納できる件数である。

4.2 名前解析処理の高速化のための基本方針

高速化のための基本方針を以下に示す。

- 3.章の問題点a)、b)、c)の解決
  名前解析処理において、木構造のDIT情報を1段毎に検索する代わりに提示名全体をもとに直接エントリを確定することで、DIB上のデータへのアクセス回数を減少させ、またDITの形態（深さや各節での分岐数）に処理性能が影響されないようにする。
  結果としてRead操作、ModifyEntry操作といった単一エントリに対するディレクトリ操作の実行では、名前解析処理終了後のエントリに対する読み出し（または更新）でDIT情報をアクセスする必要がないため、操作実行に必要なデータの局所性が向上すると考えられる。
- 3.章の問題点b)に対する解決
  提示名をエントリの検索・格納用のキーとすると、4.1節の前提条件1)から不定長のキーとなり、効率的にエントリの格納が行なえない。このため提示名のハッシュ値をキーとしてエントリを格納する。
- 分散処理の実現
  提示名から直接エントリを検索した結果エントリが見つからない場合は、分散処理のための知識（別名に関する知識を含む、以下分散処理知識と呼ぶ）

表 2: データ構造の比較

	データ構造 1. 固定ハッシュ表	データ構造 2. 動的ハッシュ表	データ構造 3. B-木
格納効率	× 表のサイズの予測が困難	◎	○ B-木の格納効率 (≈ 0.69)
検索コスト	◎	◎	○ B-木の検索コスト
更新コスト	◎	× ハッシュ表の拡張時に大幅な処理が必要	○ B-木の更新コスト

を検索し、他の DSA に分散格納されているのか、提示名が誤っているのか判定する。4.1節の前提条件 5) からこの知識は主記憶に保持でき、高速な検索を実現することが可能である。

以下ではこの基本方針に基づき、具体的なエントリの格納方式と名前解析処理手順について詳細に述べる。

4.3 エントリの格納方式

識別名のハッシュ値をキーとしてエントリを格納する場合、1) エントリを格納するデータ構造、2) データ構造に適したハッシュ関数の選定、3) ハッシュ値の衝突の対処、について検討する必要がある。

4.3.1 エントリを格納するデータ構造

ハッシュを用いたエントリ格納のためのデータ構造として以下の選択肢が考えられる。

1. ハッシュ表を使用する。表のサイズは固定する。
2. ハッシュ表を使用する。表のサイズを動的に変更可能とする。(Dynamic Hashing[7]、Linear Hashing[8] 等の方法)
3. ハッシュ値をキーとして B-木等のアクセス手法で格納する。

ハッシュ値をキーとしてデータ構造に格納する場合、検索速度の低下を招くハッシュ値の衝突を極力避ける必要がある。ここではハッシュ値の衝突を極力避け、かつ 4.1節の前提条件 3) を満たす場合について、格納効率、検索コスト、ならびに更新コストに関してこれらのデータ構造を比較した結果を表 2に示す。ここではコストとしてデータ構造へのアクセス回数を想定している。また、アクセス手法の例として B-木を想定した。B-木を用いる場合 (データ構造 3.) では、検索、更新が操作に対して平均的なコストで実現でき、DIB の前提条件 3) に示す多数のエントリ件数に対応できる。さらに、エントリ件数に対して衝突が発生しないような十分大きなハッシュ値の値域を用いても格納効率があまり低下しない。従ってデータ構造として識別名のハッシュ値をキーとする B-木を用いることとする。

4.3.2 ハッシュ関数の選定

エントリ格納のデータ構造として B-木を用いる場合、ハッシュ関数に対する条件として以下のものが挙げられる。

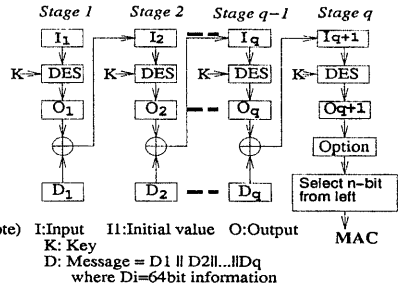

図 3: MAC スキーム (文献 [9] より引用)

(1) 不定長のバイト列を n バイト (4.1節前提条件 3) に示すエントリ数に対して十分大きな値域を表せるバイト数) に変換する関数であること

(2) ハッシュ値の衝突を少なくするためにハッシュ空間にランダムに分散する関数であること

特に条件 (2) はハッシュ値の衝突を避け検索効率を向上させるために重要である。これは、ランダムに分散する関数を用いることで、操作で指定される提示名の性質 (これは 4.1節の前提条件 2) から予めわからない) にかかわらず、ハッシュ値の衝突率は α = (格納エントリ件数/ハッシュ空間の大きさ) とみなすことができるからである。例えばハッシュ値の値域を 4 バイト、格納件数を 10^6件とすると $\alpha = 2.3 \times 10^{-4}$ となり、ハッシュ値の衝突を殆んど回避することができる。

ここでは、このような関数の 1 つとして DES(Data Encryption Standard) に基づくメッセージ認証 (MAC) スキーム[9] (図 3) を用いたハッシュ関数 (図 4) を用いることとした。ハッシュ関数への入力は、提示名に含まれる RDN の並びで、各 RDN は ASN.1 符号化されたものを用いる。同じ提示名に対するハッシュ結果の一意性を保証するために、RDN 中の符号化された属性値に対して各属性の定める正規化処理[2]を行なっている。また、AddEntry 操作等の実行では直接上位のエントリの存在確認が必要となる。さらに、ディレクトリの分散処理や別名 (Alias) の展開を実現する場合、ルートから指定された識別名までの間に含まれるエントリに対する分散処理知識を検索する必要がある。従って、図 4 では提示名に対するハッシュ値を得る際に、ルートからその提示名に至る上位エントリのハッシュ値のリストも同時に計算する。

4.3.3 ハッシュ値の衝突への対処

4.3.2節で述べたようにハッシュ関数の値域を 4 バイト程度に大きくとったとしてもハッシュ値の衝突は発生し得る。ハッシュ値の衝突への対処方法としては以下の 2 つが考えられる。

- 外部ハッシュ法
  複数件のエントリが同じハッシュ値を持つことを許す。ハッシュ値の衝突の際には、同じハッシュ値を持つエントリの集合を単位として B-木に格納する。

```
ハッシュ(識別名) /* RDN 数分のハッシュ値のリストを返す */
{
  const KEY = '01234567'; /* 適当な固定値 */
  初期値 = 'abcdefgh'; /* 適当な固定値 */

  識別名を正規化する;
  for (i=0,NextRDN=識別名の先頭の RDN; NextRDN!=NULL;
            i++, NextRDN=識別名の次の RDN)
    {
      Message = NextRDN をバイト列にしたもの;
         /* 8バイトの倍数に長さを合わせ不足分は0で充当 */
      Result[i] = MAC(KEY, 初期値,Message);
      初期値 = Result[i];
    }
  return(Result);
}
```

図 4: 使用するハッシュ関数

表 3: 外部ハッシュ法と内部ハッシュ法の比較

	外部ハッシュ法	内部ハッシュ法
検索コスト		
(成功)	$\approx (1+2\alpha)$	$\approx (1+\alpha)$
(失敗)	$\approx 2\alpha$	$\approx \alpha$
エントリ削除	エントリの集合からの削除。0件の集合になった場合集合を削除	エントリに削除した印を付与

注) B-木の葉以外のノードは主記憶上に存在することを仮定し、検索コストの単位は読み込んだエントリ件数。
αはハッシュ値の衝突率を示す ($\alpha \ll 1$ を仮定)

- 内部ハッシュ法
  もし、ハッシュ値が衝突した場合は、ハッシュ値の衝突が回避されるまで再ハッシュを繰り返し、必ず全てのエントリに異なるハッシュ値を持たせる。エントリ単位で B-木に格納する。

これら2つの対処方法の比較を表3に示す。表中の検索コストの見積りにおいては以下の仮定を行なった。

- B-木の葉でないノードは主記憶中にバッファされているものとする。
- 外部ハッシュ法ではエントリの集合単位で格納されているため、エントリの集合単位でのみ検索可能であるとする。
- コストは実際に読み出したエントリの件数とする。

ハッシュ値の値域が格納件数よりも十分大きく衝突率 $\alpha \ll 1$ である場合、検索コストは成功時も失敗時も内部ハッシュ法と外部ハッシュ法の差は殆んどない。エントリの削除要求に対しては、内部ハッシュ法では削除印をエントリにつけるだけでエントリは物理的には削除できない。従って、内部ハッシュ法にはエントリの追加、削除を繰り返すと格納領域の効率が低下するという問題があり、ここでは外部ハッシュ法を用いる。

4.4 名前解析処理手順

前節で述べたハッシュによるデータ格納方式を用いた名前解析処理手順を図5に示す。ローカル名前解析処理で提示名のハッシュ値をキーとしてエントリを検索したがエントリが存在しなかった場合は、他の DSA に格納されているかその提示名を持つエントリが本当に存在しないかを名前解析処理で決定する。このためには

```
名前解析処理(提示名)
{
  /* 途中のエントリに対するハッシュ値も
                          セットされる */
  ハッシュ値配列 = ハッシュ(提示名);
  /* 提示名に対応するハッシュ値 */
  ハッシュ値 =
      ハッシュ値配列 [提示名に含まれる RDN 数-1];
  ローカル名前解析処理(ハッシュ値, 提示名);
  if (結果がエントリ)
     return(そのエントリ);
  /* エントリが格納されていない場合
                    分散処理知識を調べる */
  for (i=提示名に含まれる RDN 数-2;i>=0;i--)
    {
      知識リスト = ハッシュ値配列 [i] をキーに
                  分散処理知識を検索;
      if (知識リストが存在)
         for (知識リスト中の全ての知識について)
           if (知識中の識別名が提示名の一部と一致)
             if (知識は Alias である)
               {
                 提示名を alias 展開する。
                 return(名前解析処理 (新しい提示名));
               }
             else
               return(その知識);
    }
  return(NotFound);
}
/* 提示名をハッシュした結果がハッシュ値に格納されている */
ローカル名前解析処理(ハッシュ値, 提示名)
{
  ハッシュ値をキーにエントリを格納する B-木を検索;
  if (検索結果==NULL)
    return(NotFound);
  for (検索結果に含まれる全てのエントリ)
    if (提示名==エントリ中の識別名)
       return(そのエントリ);
  return(NotFound);
}
```

図 5: ハッシュによる名前解析処理手順

提示名と最大限一致する識別名を持つ分散処理知識を見つける必要がある。本稿で示すようなハッシュ関数を用いた場合、ある提示名のハッシュ値とその提示名から RDN が1つ少ない名前のハッシュ値には相関関係はない。従って最大限一致する知識を検索するために、その提示名に含まれる RDN を1つずつ減少させた名前のハッシュ値をキーとして、RDN が含まれなくなるまであるいは合致する知識が見つかるまで、順番に分散処理知識を検索する。

5. 評価

提案したハッシュによる名前解析処理方式の有効性を評価するため、筆者らが開発済みの高速 OSI ディレクトリ用 DBMS ASSIST/D (Advanced Supervisary Supporting System for Integrated Services of Telecommunication / DIB)[5]に本方式に基づくソフトウェアを実装し、Read 操作の操作応答時間を測定した。評価に Read 操作を用いたのは、Read 操作は名前解析処理以外には確定したエントリの内容を読み出す

表 4: 測定条件

測定環境	Sun SPARC 670MP	
測定内容	同一マシン上の利用者プロセスでの操作をＤＢＭＳに送信してから結果を受信するまでの応答時間	
格納内容	国	属性：クラス、国名の 2 件 9 バイト
	組織	属性：クラス、組織名の 2 件 23 バイト
	組織単位	属性：組織単位名等 5 件約 60 バイト
	組織人	属性：通称等 6 件約 70 バイト
操作内容	Read	指定エントリの全属性の読み出し

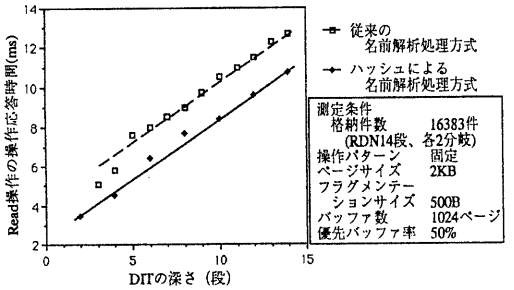

図 6: DIT の深さに対する Read 操作応答時間

処理だけで実現できることから、その操作応答時間特性は名前解析処理時間の特性を反映していると考えられるためである。今回の実装では、単一のディレクトリシステムエージェント（DSA）で DIB を構成した場合を想定し、分散処理を行なっていない。測定に用いた条件を表 4に示す。

5.1 DIT 木の深さに対する性能

DIT の深さが変化した場合の名前解析処理性能を見るために、以下の測定を行なった。

使用する DIT

- RDN の深さが 14 段（1 段目が国、2 段目が組織、3 段目以降が組織単位）
- 2 段目以降の各段全てが 2 分岐

測定項目

- Read 操作 1 回あたりの操作応答時間
  （同一操作を繰り返し実行した場合の、DIT の最右端のエントリに対する操作応答時間と最左端のエントリに対する操作応答時間の平均）

パラメータ

- 操作指定する提示名に含まれる RDN 数を変化（2 ～14）

ここでは DIT 各節の分岐数の変化の影響を排除するため、完全な 2 分木を測定に使用し、またエントリ格納件数の変化による影響を排除するため DIB のエントリ件数を固定した上で同一の操作を繰り返し実行しバッファヒット率が 100%となるようにした。結果を図 6に示す。比較のため、図 6には同一条件下での従来の名前解析処理方式の性能[5]も併せて示す。

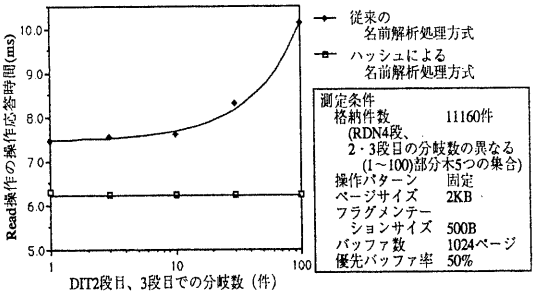

図 7: DIT の分岐数に対する Read 操作の応答時間

5.2 DIT での各節の分岐数に対する性能

DIT の各節での分岐数が変化した場合の名前解析処理性能を見るために、以下の評価を行なった。

使用する DIT

- RDN の深さが 4 段（国、組織、組織単位、組織人）
- 2 段目、3 段目で均等に分岐し、分岐数が異なる（100 分岐～1 分岐）5 つの部分木の集まりからなる DIT

測定項目

- Read 操作 1 回あたりの操作応答時間
  （指定する提示名は RDN4 段のものとし、同一操作を繰り返し実行した場合の、その部分木の最右端のエントリに対する操作応答時間と最左端のエントリに対する操作応答時間の平均）

パラメータ

- 分岐数の異なる部分木で測定（1 分岐～100 分岐）

ここでは DIT の深さの変化の影響を排除するため、提示名中の RDN 数を 4 段に固定した。またエントリ格納件数の変化による影響を排除するため DIB のエントリ件数を固定した上で同一の操作を繰り返し実行しバッファヒット率が 100%となるようにした。結果を図 7に示す。

5.3 データ格納件数に対する性能

データ格納件数が変化した場合の名前解析処理性能を見るために、以下の評価を行なった。

使用する DIT

- RDN の深さが 4 段（国、組織、組織単位、組織人）
- 2 段目以下が各段で均等に分岐

測定項目

- Read 操作 1 回あたりの操作応答時間およびバッファのヒット率
  （指定する提示名として RDN4 段のものからランダムに選択）

パラメータ

- 2 段目以下の各段での分岐数を変化（1 分岐～46 分岐）

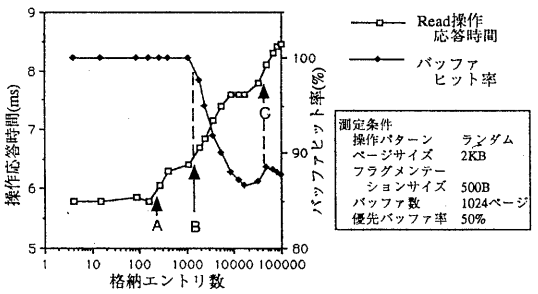

図 8: エントリ格納件数に対する Read 操作応答時間とバッファのヒット率の変化

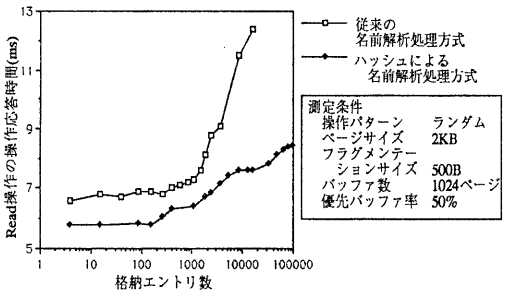

図 9: エントリ格納件数に対する Read 操作応答時間の従来方式との比較

ここでは DIT の深さの変化の影響を排除するため、提示名中の RDN 数を 4 段に固定した。また、バッファ性能も含めて測定するために操作で指定する提示名は 4 段目（最下段）のエントリからランダムに選択した。図 8に操作応答時間とバッファのヒット率を示す。また、従来の名前解析処理方式[5]と同一条件下での Read 操作の応答時間の比較を図 9に示す。

6. 考察

6.1 提案方式の性能について

(1)DIT 木の深さに対する性能

- 図 6からハッシュによる名前解析処理方式は従来の方式と比較して、全ての RDN 数の場合で約 2 ミリ秒高速であった。これは 3.章の問題点 a) で示した DIT 情報へのアクセスを回避していたためと考えられる。また、これはバッファが 100%ヒットしているという従来の方式に有利な条件であり、実際のランダムな操作要求のもとではバッファのヒット率が従来方式より向上するためさらに性能の差がひらくと思われる。
- 図 6におけるグラフの傾きから、RDN が 1 段増加した場合の操作応答時間の増加はおよそ 0.6 ミリ秒/段である。使用した DES に基づく MAC スキームを用いたハッシュ関数については、DES の処理速度が約 35 マイクロ秒/8 バイトであり、1 つの RDN が ASN.1 で 32 バイトに符号化されていることから、RDN の 1 段当りのハッシュ速度は約 0.15 ミリ秒と想定される。従って 3.章の問題点 c) で述べた DIT の深さによる名前解析処理性能については、RDN の 1 段あたりで約 0.15 ミリ秒の処理時間の増加に抑えられた。操作応答時間の残りの増加 0.45 ミリ秒は RDN 数の増加に従って、ASN.1 符号化／復号処理時間が増加しているためと考えられる。

(2) 分岐数の多い DIT に対する性能

- 図 7からハッシュによる名前解析処理方式の性能は DIT の各段の分岐数には依存しないことが示され、本方式が 3.章の問題点 c) の DIT の分岐数の変化に対する解決になっている。OSI ディレクトリのユニバーサルパーソナル通信（UPT）等のネームサーバへの適用[10]では、UPT 加入者を識別するための UPT 番号を、3 ないし 4 つの RDN に分割して識別名と対応付ける方法が提案されている。この方法では DIB は、例えば加入者数のエントリを格納し、3 段ないし 4 段の深さで分岐数の多い DIT となることが想定される。図 7の評価結果は、本稿で提案した方式は、UPT 等のネームサーバとして分岐数の多い DIT を持つ DIB を構築した場合にも従来の方式と比較して適していることを示している。

(3) エントリ格納件数に対する性能

- 図 9からわかるように、従来の名前解析処理では格納されるエントリ件数が増加すると、DIT 情報へのアクセス増加によるバッファヒット率の低下により操作応答時間が $O(log(\text{エントリ件数}))$ より悪い傾向を示す。一方、本稿で提案したハッシュによる名前解析処理方式の特性は $O(log(\text{エントリ件数}))$ であった。このことは、ハッシュによる名前解析処理方式を用いれば、例えば 10^6件といったエントリを格納し、10^3ページといったエントリ件数に対して比較的少ないバッファ数しかとれない場合でも、アクセスするデータの局所性の増加により高速な操作応答時間が得られることを示しており、本方式が 3.章の問題点 b) の解決になっていることを示している。
- 図 8の **A** 点、**C** 点で操作応答時間が急に増加しているのはこれらの点で B-木の深さが 1 段深くなったためと思われる。**C** 点でのバッファのヒット率が向上しているのも B-木の深さの増加による。また、**B** 点で操作応答時間が急に増加したのはバッファのヒット率の低下によると思われる。

(4) 分散処理性能について

- 4.4節で述べたようにローカルな名前解析処理でエントリが存在しなかった場合、分散処理を行なうために、最悪の場合提示名に含まれる RDN 段数-1 回分散処理知識を検索する必要がある。本稿で提案した名前解析処理方式では分散処理知識を DIT 情報とは独立に格納することができる。また、4.1

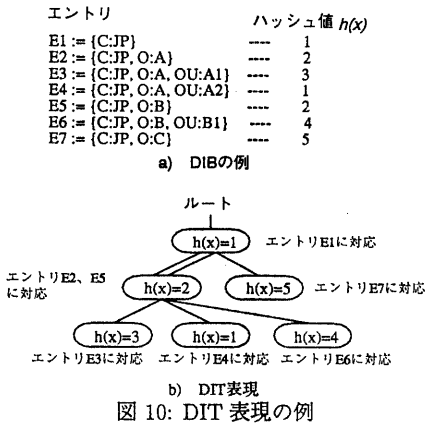

図 10: DIT 表現の例

節の前提条件 5) から分散処理知識は主記憶に格納できる程度の件数と想定している。従って、分散処理知識をハッシュ表等の適切なデータ構造で主記憶上に格納することで、分散処理知識を RDN 段数-1 回主記憶上で検索する処理時間を、ローカル名前解析処理に必要な時間と比較して十分小さくすることができる。従って、ローカル名前解析処理でエントリが存在しなかった場合も名前解析処理性能を維持できると考えられる。

6.2 DIT 情報の必要性と表現方法について

本稿で提案したハッシュによる名前解析処理では DIT でのエントリ間の上位／下位関係を表す DIT 情報は不要であるが、DIT 情報は複数エントリに対する操作 (List 操作、Search 操作) や DIT 木構造を更新する操作 (AddEntry 操作) 等で、あるエントリの直下あるいは部分木に含まれるエントリを見つけるためにも用いられる。従って本稿で提案する名前解析処理方式に従っても、DIT 情報を維持する必要がある。アクセスするデータの局所性を増加させるためには格納効率を向上することが重要であるが、そのためには、DIT 情報は識別名ではなく図 10に示すように提示名のハッシュ値をキーとする DIT 表現とするのが望ましい。外部ハッシュ法を用いた場合、同じハッシュ値を持つエントリが存在し得る。従って DIT 情報を図 10に示すようにエントリ間の枝の重複を許したネットワーク構造として表現する。

検索時はハッシュ値がユニークでないためこの構造を検索した結果は重複を含んでいる。図 10の例ではエントリ E2 以下の全部分木をこのネットワーク構造で求めるとハッシュ値の集合 $\{1,2,3,4\}$ が得られ、対応するエントリの集合として $\{E1, E2, E3, E4, E5, E6\}$ が得られる。この集合に含まれるエントリ各々を実際に読んで識別名を指定された検索範囲と比較することで本来の結果である $\{E2, E3, E4\}$ が得られる。

7. むすび

本稿では、DIB 上でディレクトリ操作を高速に実行するために、従来の研究で用いられている DIT 情報を 1 段毎に探索する方式ではなく、提示名のハッシュ値をキーとして直接操作対象のエントリを確定する名前解析処理方式を提案した。ここではハッシュ関数として、DES に基づく MAC スキームを用いた関数を使用し、ハッシュ値の衝突は外部ハッシュ法によって対処している。このハッシュ値をキーとして B-木にエントリを格納している。この方式では、二次記憶上に DIB が格納されている場合、名前解析時に DIT 情報を検索しないため、1)DIB 上のデータへのアクセス回数が減少し、名前解析処理が高速化される、2) アクセスするデータの局所性が増加しバッファのヒット率の増加のため名前解析処理後のエントリに対する読み出し (または更新) 速度が向上する、また 3) 名前解析処理性能が DIT の形態に影響されない。特に、エントリ格納件数に対する処理性能がエントリ件数に対して十分小さなバッファ数の場合でも $O(log(\text{エントリ件数}))$ であり、10^6件といった多数のエントリを格納する大規模な DIB 上でも高速にディレクトリ操作が実行できる。最後に、日頃御指導頂く KDD 研究所 浦野所長、眞家次長に感謝します。

参考文献

[1] ISO/IEC 9594 1-8: Information Processing Systems – Open Systems Interconnection – The Directory (1988).

[2] 小花 他: リレーショナルアプローチによる OSI ディレクトリの DIB (ディレクトリ情報ベース) の実装と評価, 情報処理学会論文誌 Vol.32, No.11, pp.1488-1497, (1991).

[3] 中川路 他: OSI ディレクトリシステムにおける DIB (ディレクトリ情報ベース) のオブジェクト指向による実現, 情報処理学会論文誌 Vol.32, No.3, pp.304-313 (1991).

[4] Robbins, C.J., et. al.: The ISO Developement Environment: User's Manual Vol.5:QUIPU, (1989).

[5] 西山 他: 拡張可能 DBMS 構築技法に基づく高速 OSI ディレクトリ専用 DBMS の設計と評価, 情報処理学会論文誌 Vol.34, No.6, (1993).

[6] ISO 8824,8825: Information Processing Systems – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1) / Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), (1987).

[7] Mullen, J.K.: Unified Dynamic Hashing, Proc. of 10th VLDB, pp.473-480, (1984).

[8] Litwin, W.: Linear hasing: a new tool for file andd table addressing, Proc. of 6th VLDB, pp.224-232, (1980).

[9] ISO/IEC 9797: Data integrity mechanism using a cryptographic check function employing a block cipher algorithm, (1989).

[10] 小花 他: ユニバーサルパーソナル通信 (UPT) への OSI ディレクトリの適用と評価, 電子情報通信学会論文誌 Vol. J74-B-1, pp. 959-970 (1991).