プログラミング－言語・基礎・実践－ 18－11
（1994. 7. 21）

単一ポインタ表現を持つ分散永続ヒープ上のGCの枠組

山本 耕平　猪原 茂和　益田 隆司

東京大学大学院 理学系研究科 情報科学専攻

要旨

永続的なデータをより容易に扱うために、永続・非永続データの透明性をアプリケーションに提供する永続オブジェクト管理システムが数多く研究されている。従来は、仮想空間の広さの制約から二次記憶と主記憶上で別々のポインタ表現が必要であり、それに伴う実行時のポインタ表現変換のコストが無視できなかった。今後一般的になると思われる64bit仮想空間を仮定することにより、単一のポインタ表現が可能となり、この変換コストは解消できる。一方、分散した永続ヒープには局所的なcompacting GCが必要不可欠であるが、従来の分散GCアルゴリズムは、64ビット空間の単一ポインタ表現下では効率的に動作しないことが分かった。本稿では、単一ポインタ表現を損なうことなく部分毎のcompacting GCを可能にするための枠組について述べる。この枠組の特徴は、単一ポインタ表現下でオブジェクトの移動を可能にするためのメタデータの導入と、トランザクションを利用して、非永続空間と永続空間との独立性を得ることによる言語独立な永続ヒープのGCの実現の2点である。

A Framework for Garbage Collection on a Distributed Persistent Heap with Uniform Representation of Pointers

Kouhei Yamamoto, Shigekazu Inohara and Takashi Masuda

Department of Information Science, Graduate School of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113

Abstract

A number of persistent object management systems have been developed to provide applications with transparency between persistent and transient objects. In existing systems, references to persistent objects are represented differently on the primary and secondary storage because of the limited size of virtual address spaces of the existing hardwares. This paper proposes a distributed persistent object management system on 64-bit address spaces that has the following three features: high performance at application run-time by uniform representation of pointers, independence from a specific programming language, local compacting garbage collection(GC) of a persistent heap. Because existing algorithms for local compacting GC do not fit uniform representation of pointers, we introduce meta-data for supporting compacting GC. Transactional updates of the persistent heap are integrated with GCs to garbage-collect the persistent heap without considering transient spaces whose type information is known only by a language.

1 はじめに

データベース応用を支援するために、永続的なデータを容易に操作することを可能にする永続オブジェクト管理システムが研究されている。永続オブジェクト管理システムは、二次記憶を型付けされた永続的なヒープとして管理し、ポインタを含む複雑な構造を持ちうるデータとして永続オブジェクトの概念をユーザーに提供する。永続オブジェクトを非永続オブジェクトと区別することなくアプリケーションプログラムが操作することを可能にするのが、永続オブジェクト管理システムの基本的機能である。また、永続ヒープのデータベース的性格から、永続オブジェクト管理システムは、永続オブジェクトへの操作をトランザクションとして管理し、さらに、分散永続ヒープのガベージコレクション(GC)を実現する。

本研究では、特に、(1)実行時性能を向上するために単一ポインタ表現を採り、(2)分散した二次記憶上に存在する永続ヒープを効率的にGCするため部分毎にcompacting GCを行ない、(3)複数言語のプラットフォームとするべく言語処理系に依存しない永続オブジェクト管理システムを提案する。これらの3つの目標を同時に満たすことは困難であり、従来これら全てを満たしたシステムはなかった。

既存の多くのシステムでは、分散永続ヒープ全体は32ビット仮想空間に入り切らないので、二次記憶上とメモリ上の2種類のアドレス空間を必要としていた。そのため、それぞれ対応するポインタ表現をアプリケーション実行時に変換する必要が生じ、オーバーヘッドを招いていた。また、実行時変換を行なうには、仮想記憶機構を利用したPointer Swizzling[7]と言語処理系のポインタ変換のコード生成に依存する方法の2つの実装が考えられるが、後者を選択し特定の言語処理系に依存したシステムも多かった。これに対し、今後普及すると考えられる64ビットプロセッサの仮想空間を仮定すれば、分散した永続ヒープ全体を一つの仮想空間にリニアにマップすることが可能となり、二次記憶上でもメモリ上でもオブジェクト間の参照を単一のポインタ表現で表せ、この結果、実行時の変換のためのオーバーヘッドも解消する。

一方、分散永続ヒープを効率的にGCするためには、何らかの付加的情報が必要である。永続ヒープは二次記憶上に存在するので、二次記憶の利用効率改善のために、compactionを行ないディスクブロックのinternal fragmentationを解消する必要がある。また、分散永続ヒープを一括してGCするのは非効率であるので、ファイルという単位に分割し、それぞれ独立にGCしなくてはならない。ファイル単位でcompacting GCを行う際に問題になるのは、ファイル間ポインタだけで指されているオブジェクトを回収しない点と、オブジェクトを移動したときにそのオブジェクトを指しているファイル間ポインタを更新する点の2点を、ヒープ全体を走査することなく実装する所である。そのため、ファイル間ポインタで指されているオブジェクトを列挙した情報と、ポインタ更新を行なうための情報を、ファイル毎に取得できるようにする必要がある。

また、アプリケーション実行中に永続ヒープを正しくGCをするためには、アプリケーションの持つレジスタやスタックなど非永続的なメモリから永続オブジェクトを指すポインタも考慮しなくてはならないので、非永続空間の型情報が必要になる。しかし、非永続空間の型情報を言語処理系に依存することなく得ることは困難である。

本研究で提案する永続オブジェクト管理の枠組では、64ビット仮想空間上の単一ポインタ表現の環境下で、ファイル間ポインタを列挙するデータ構造Indirect Back Pointer(IBP)をファイル毎に作成することにより、ファイル毎のcompacting GCを実現する。さらに、永続ヒープへのアクセスに導入したトランザクションをGCにも利用することにより、言語処理系に依存することなくアプリケーション実行中に永続ヒープのGCを行なうことを可能にする。

アプリケーションがトランザクションを終了し永続ヒープへの更新を行なう際に、ファイル間参照を行なうポインタ群をシステムがIBPに登録する。GCの際に対象ファイルのIBPを参照することにより、ファイル間ポインタによって指されているオブジェクトを知ることができるので、オブジェクトを移動する時には対応する全てのポインタを更新することができる。さらに、IBPへの操作は、分散システム上での永続ヒープのGCに要求される局所性と並行性を備えている。

言語処理系に依存しないGCのために、同時実行中の他のトランザクションの影響を受けないというトランザクションの基本的な性質の一つを利用することにより、非永続空間の型付けを回避する。永続空間への参照が非永続空間に存在する可能性を取り除くことにより、永続空間のGCの際に、言語依存性を招く非永

続空間の型付けを必要としない。

2 本方式の概要

本研究の目的は、単一ポインタ表現による実行時性能の向上と、複数言語に容易に適用できる点である。その目的を崩さぬように、分散環境上での永続ヒープの性質から必要となるファイル毎の compacting GC を実装する点が問題となる。

2.1 部分毎の compacting GC のための Indirect Back Pointer の導入

ファイル毎の compacting GC を正しく実行するためには、ファイル間ポインタ全てを掌握して、ファイル間ポインタのみで指されているオブジェクトを回収せず、適切にポインタを更新しなくてはならない。しかし、GC 毎に永続ヒープ全体を走査することによってファイル間ポインタを列挙するのは、分散した永続ヒープの大きさを考えれば非現実的である。従来のシステムでは、この問題に対して、ファイル間のポインタを間接ポインタにすることにより解決していた。ファイル間ポインタによって指されるオブジェクトへのポインタを Incoming Reference Table(IRT)[2] と呼ばれるテーブルに登録し、ファイル間ポインタ自体は IRT のエントリへのポインタによって実現する。ファイル間ポインタによって指されているオブジェクトを参照するためには、IRT をたどって一段階の間接参照をしなくてはならないが、GC のときには IRT を rootset に加え、オブジェクトを移動する場合にも IRT のエントリを更新するだけと、局所的に正しく GC が行なえる。

この IRT を用いた解法では、ファイル間ポインタとファイル内のポインタを永続オブジェクト管理システムが識別し、アプリケーション側からはこれらを区別なく扱えるようにしなくてはならない。しかし、2 種類のポインタを実行時に変換しなくてはならないので、32 ビット仮想空間での永続ヒープの実装と同じく、実行時オーバーヘッドが存在する。

そこで、本研究では、アプリケーション実行時に参照されるポインタは直接的なポインタとして表現し、ファイル毎の compacting GC を正しく実行するために GC が参照する情報は、別に作成することにした。これにより、実行時オーバーヘッドの問題は解消する。

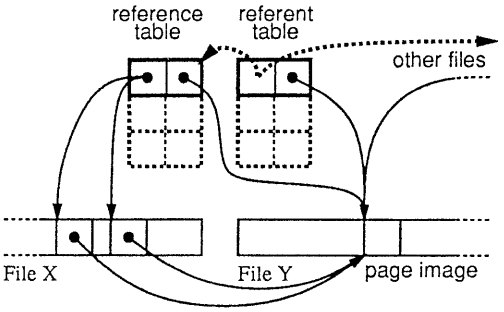

図 1: Indirect Back Pointer の構造

正しい GC を保証する付加的データ構造である Indirect Back Pointer は、ファイル間ポインタをインデックスとする 2 つのテーブルからなっている (図 1)。一つは、ファイル間参照のソース側ファイル (File X) に作られ、ファイル間ポインタの所在を格納するテーブル reference table であり、もう一つは、デスティネーションの側のファイル (File Y) に作られ、ソース側ファイル番号を格納するテーブル referent table である。トランザクションが commit する時に、システムが dirty page に関して reference table とページイメージを比較することにより、ファイル間ポインタの作成や更新を検出し、それに応じて reference table とファイル間ポインタのデスティネーション側の referent table を作成、更新する。

GC は、対象ファイルの referent table を rootset として加えることにより、ファイル間ポインタのみで指されているオブジェクトを回収することはない。また、ファイル間ポインタで指されているオブジェクトを移動した時には、referent table で示されるポインタのソース側ファイルに、移動後のアドレスを通知する。永続オブジェクト管理システムは、通知を受けたファイルの reference table を参照し、更新をポインタ本体に反映させる。このポインタの更新は、ファイル間ポインタを含むページがメモリ上に読み込まれるまで、reference table のみに留められ遅延される。ファイル全体より小さい reference table のみに更新を抑えることにより、ポインタ更新に局所性を与えることができ、更新のコストを低減することができる。この方式は基本的には、ポインタ更新のために back pointer を張ったと考えられ、同時に indirection を入れることにより、分散 GC に必要な局所性を実現している。

言語処理系に依存せずに実装できる IRT+Pointer

Swizzling方式とIBP方式の2つを比較してみると、実行時のポインタ変換が不要なのでIBP方式の方がアプリケーション実行時の性能が良い。GC時の性能を考えると、IRT方式はGC対象ファイルに対応したIRTのみを更新するだけでcompacting GCを行なえるので、完全に局所的であると言えるが、IBP方式はポインタ更新通知が必要なので完全に局所的ではない。しかし、GCと一般のアプリケーションの実行頻度を比較すれば、アプリケーションを実行する方が多いと予想できるので、IBPは妥当なトレードオフの結果と考えられる。

2.2 言語処理系に依存しないGCのためのトランザクションの利用

複数のユーザが永続ヒープを共有することは、しばしば複数の言語から永続ヒープを共有することにつながるが、既存の多くの永続オブジェクト管理システムは特定の言語に深く依存していたので、このような場合に対応が難しかった。

言語処理系に依存しない永続オブジェクト管理システムの実装の際に問題となるのは、GCのために非永続空間の型付けを行なう必要がある点である[1]。アプリケーション実行中にGCを行なう際には、アプリケーションのスタックやレジスタなど非永続空間に存在する永続オブジェクトへのポインタを考慮しなくてはならない。しばしばスタックやレジスタにポインタを退避し、そのようなポインタのみで指される永続オブジェクトが生じるが、仮に非永続空間を考慮しなかったとすると、そのポインタを他の永続オブジェクトへ格納する前にGCが起動され、生きている永続オブジェクトがガベージとして回収される危険性がある。さらに、compacting GCを考えると、全てのポインタを把握しなければ、オブジェクトの移動により更新されたポインタと更新されなかったポインタが混在し正常にアプリケーションを続行できなくなる。このため、非永続空間の型情報を用いて、非永続空間から永続オブジェクトを指すポインタ全てを掌握する必要がある。しかし、非永続空間のスタックやレジスタの型情報を言語処理系に依存することなく求めることは不可能である。

そこで、我々は、永続オブジェクトへのアクセスに導入したトランザクションの性質を利用することにより、非永続空間に存在し得る永続オブジェクトへのポインタを排除し、非永続空間の型付けの必要性を回避した。もともと、永続オブジェクトは複数のユーザーにより同時に共有されるので、何らかの並行性制御が必要である。永続オブジェクトへのアクセスをトランザクションとすることは自然で、既存の永続オブジェクト管理システムの多くもトランザクションを採用している。

実行中のアプリケーションは、トランザクション外を実行中の場合と、トランザクション内を実行中の場合に分けられるので、それぞれの場合について非永続空間の型付けが不必要なことを説明する。一般的にあるトランザクションで得た値は、別のトランザクションのcommitにより変更されているかも知れないので、次のトランザクションに持ち越すことに意味はない。これはポインタにも言えるので、トランザクション外を実行中のアプリケーションの非永続空間に、永続オブジェクトを指すポインタがあったとしても、GCはそれを無視して問題ない。プログラマ側から見れば、トランザクション外へ永続オブジェクトポインタを持ち出すことが禁止されているのだが、それはトランザクション元来の性質によるものなので正当化されている。

アプリケーションがトランザクション内を実行中だった場合には、永続オブジェクトへのポインタが非永続空間に存在するが、GC中の永続ヒープへのポインタを獲得し得るトランザクションをアボートし、そのトランザクションを再び始めから実行させることにより、そのトランザクションが持つ永続オブジェクトへのポインタを無効化する。GCにとってみれば、最後にcommitしたトランザクションの更新以後、永続オブジェクトへアクセスしているトランザクションは存在しないことになるので、実行中のトランザクションの非永続空間からのポインタを考慮することなく正しくGCできる。これは、トランザクションを実装するためのアルゴリズムには依存しない。

この方式により問題となるのは、アボートするトランザクション数がGCにより増加してしまう点であるが、GCの対象ファイルを選定することにより、アボートするトランザクションを減少させることができる。これは、永続ヒープのGCが、アドレス空間の不足からではなく、主に二次記憶容量の不足により起こるので、同じ二次記憶を共有しているファイルならばどれをGCしてもよいからである。

[1] conservative GCでcompactingすることはできないので、永続オブジェクトへの型付けは行なう。

3 Indirect Back Pointer への操作とその検証

本節では IBP への操作の詳細を述べ、それらを並行に実行できる点を検証する。

IBP への操作は大別して、トランザクションによる更新と、GC によるファイル間ポインタの更新に分けられる。トランザクションは commit 時に dirty page を走査して、ファイル間ポインタの変更を検出するが、この時 IBP に対して、2 つの操作が起こりうる。ファイル間ポインタがファイル内ポインタへと変更された際の reference table エントリの解放と、ファイル間ポインタが新しく作られた際の reference table エントリの作成である。ファイル間ポインタが別のファイル間ポインタへ更新された場合、解放と作成が連続して起こったと処理されるので、新しい操作は必要ない。一方、一般に GC アルゴリズムはオブジェクトの解放と移動の 2 つの操作で構成できるので、GC の時には IBP に対して 2 つの操作が起こり得る。オブジェクトの解放の際の reference table エントリの解放[2]と、オブジェクトの移動の際の reference table と referent table のポインタ更新の 2 つである。

また、これらのエントリの操作により、対応する reference table と referent table に作成、解放、更新のメッセージが送信される。reference table の作成 (または解放) では、ポインタのデスティネーション側のファイルに referent table エントリの作成 (解放) メッセージが送られる (図 2の add-reft, free-reft)。referent table のポインタ更新では、ポインタのソース側ファイルに reference table のポインタ更新メッセージが送られる (図 2の update-ref)。ただし、4.2節で述べるように、複数のファイル間ポインタが同じオブジェクトを指す場合、add-reft はそのファイルからの最初のポインタが作成された場合のみであり、free-reft は最後のポインタが消滅した場合にのみ送出される。

複数の GC と複数のトランザクションを分散環境で並行実行した場合、全体として正しく動作するためには IBP へのメッセージ間で同期を取る必要がある。同期を考慮せずに並行実行した場合の問題点は 2 つある。第一に、あるファイル間ポインタのソース側から free-reft が送信され、同時にデスティネーション

[2]生きているオブジェクトは解放されないので、GC 対象ファイルの referent table エントリが直接解放されることはない。

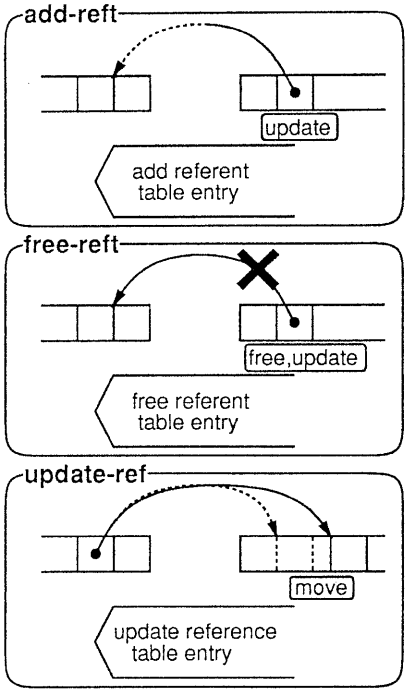

図 2: IBP 操作のためのメッセージ

側で update-ref が送信された場合に起こる、free-reft と update-ref の交錯である。free-reft には、ポインタによって指されているオブジェクトのアドレスが identity として指定されているが、free-reft を受ける側ではオブジェクトの移動によってそのアドレスが変更されているので、食い違いが生じる。この結果、解放されずにいつまでも残る referent table エントリができる上に、全く別の referent table エントリを解放してしまうかもしれない危険性がある。これは、本来不可分であるべき IBP への操作が、メッセージを用いて実現され、不可分ではないために起きている。実際、上記の場合でも、free-reft と update-ref が逐次的に送信されれば、その順番に関係なく正しく対処することができる。

第二に、update-ref と add-reft に関する問題であるが、これは IBP への操作が不可分であったとしても発生する。GC は既に存在する referent table を元に update-ref を送信するが、送信し終った後も、update-ref が届く前のポインタを参照したトランザクションの commit により、add-reft が GC 対象のファイルに送信されるかもしれない。遅れた add-reft

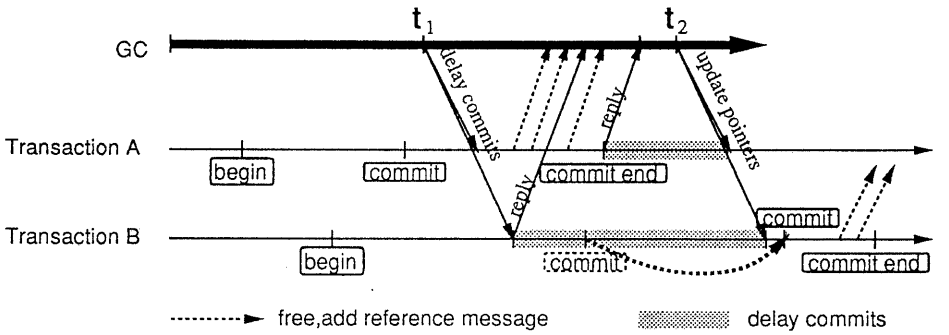

図 3: IBP への操作とメッセージの同期

に対して何も行なわないと、commit されたファイルに先の update-ref が届かないままのポインタが残される上に、GC 前のポインタに対応した referent table エントリが GC 後のファイルに作成され正しい挙動ができなくなる。しかし、add-reft が update-ref 適用後のポインタに対するものなのか、update-ref 適用後に対するものなのかは、一般的に判別不能である。これらの考察から、そもそも update-ref による更新はシステム全体で不可分となるように行なわなければならない。それに対して、まだデスティネーション側ファイルへ通知されていない新しいファイル間ポインタが commit 中のトランザクションに存在し、その新しいポインタが GC 中のファイルへのポインタであるかもしれないことが問題となっている[3]。

この 2 つの問題を解消するために、GC 終了時に two-phase commit 的な手順を踏んで、GC 対象ファイルへの全てのファイル間ポインタを掌握した後に、update-ref をまとめて送信するようにし、メッセージの交錯も起きないことを保証する。これを説明するために図 3 を例にとって説明する。GC は終了時 (t_1) に、GC 対象ファイルへポインタを持ち得るトランザクションに commit の遅延メッセージ delay commits を、GC 対象ファイルへポインタを持ち得る GC プロセスに free-reft の遅延を通知し、全ての返答が揃うまで待つ。delay commits を受け取ったトランザクションは、もし commit 中ならば commit が完了してから返答し (Transaction A)、そうでなければすぐに返答する。返答の後に commit を開始しようとした場合には、一連の手順が終るまで commit を延期する (Transaction B)。通知を受けた GC プロセスは、free-reft の発信と GC の終了を一連の手順が終るまで延期する。全ての返答がそろった時点 (t_2) で、それ以後 add-reft も free-reft も GC プロセスに届くことはないことが保証されるので、第一の問題点であった交錯は発生しない。また、その保証は全てのファイル間ポインタを掌握したことも示すので、第二の問題点も解決される。全ての返答が揃った後、update-ref をまとめて送信し、一連の手順が終了したことを延期中のトランザクションと GC に通知する。

この同期により、トランザクションの commit 同士、GC 同士、commit と GC が並行動作できることが保証される。

4 実装

本システムの構成、IBP、トランザクションの 3 つの点に関して具体的な実装について述べる。

4.1 システムの構成

本システムは、システム上に一つ存在する Central Server、各ノードに存在する Local Server、その配下の GC プロセスとインタフェースライブラリをリンクしたアプリケーションプログラムからなっている。Central Server はシステム全体で一意であるべき型情報の管理を行う。Local Server は各ノードに存在するファイルと、それに関するメタデータ (IBP と永続オブジェクトのアロケーション情報) を管理するとともに、アプリケーションプログラムとの通信によりトランザクションの管理も行う。

インタフェースライブラリはアプリケーションから Local Server へのインタフェースを提供する他、仮想記憶機構からのシグナルを検出し永続ヒープを実

[3] 前節で説明したように、commit 開始前のトランザクションはアボートされるので問題ない。

現する。アプリケーションインタフェースにはトランザクションの開始と終了、ファイル番号から永続オブジェクトへのアドレスの変換、永続オブジェクトの割り当てなどがある。永続オブジェクトの割り当て関数 `persistent_malloc` はマクロで定義され、引数 *type* にはコンパイラ内部の型のIDを取り出すプリミティブが適用される。

```
char *persistent_malloc(
        int file_id, int nitem, type )
```

ライブラリは得られた型IDを元に、アプリケーションプログラムのバイナリ中に埋め込まれた型情報を検索し、指定された型の構造、特にポインタの位置情報を得る。型のIDを取り出すプリミティブはGnu Cコンパイラに手を加え実現しており、最適化を行ったとしても型情報だけはバイナリ中に出力する改変も同時に施した。

今回C言語に対するアプリケーションインタフェースは、Gnu Cコンパイラを改変することにより実装したが、このように、ライブラリと永続オブジェクトの型を定義する手段さえ準備すれば、言語処理系のコード生成には何ら手を加えずに、容易に本システムを他の言語へ対応させることができる。

複数の言語のプラットフォームにするという観点から、言語特有のクラスタリングやGCアルゴリズムを適用できる余地を作るために、GCと永続オブジェクトの割り当て関数はファイル毎に指定できるようになっている。ただし、GCプロセスと永続オブジェクトの割り当て関数は空き領域に関する情報を共有しなければならないので、このGCプロセスと割り当て関数は一組として管理されている。永続オブジェクトの割り当て関数は、ファイル毎に別関数にするため、ダイナミックリンクライブラリの形態をとり、インタフェースライブラリによって、必要に応じリンクされる。この関数で該当ファイル中の永続オブジェクトの空き領域管理を行い、永続オブジェクトの割り当てられるべきアドレスをライブラリに通知する。インタフェースライブラリは、これに従い永続オブジェクトのアドレスと型を記録する。GCはLocal Server上では行わず、別プロセスとして実行される。Local Serverを長時間ブロックされないようにする他に、GC自体が不完全である場合にも他のファイルの運用に支障をきたさないためでもある。さらに、システム全体の保護の為には、IBPやオブジェクトのアロケーション情報も保護されなくてはならないので、GCプロセスやインタフェースライブラリは更新を要求するのみで、これらのデータの更新は全てLocal Serverが行う。

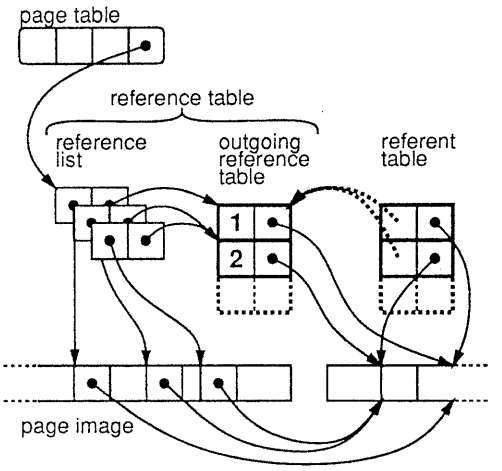

図4: IBPの詳細

4.2 IBPの実現

IBPのreference tableは、実現上2.1節で述べたものより複雑な構造を持っている。ファイル間ポインタ更新メッセージが到着した際に、reference tableのみに更新を留め局所性を出しているが、そのためにはファイル間ポインタを鍵としてreference tableを検索し、該当エントリ中のポインタを更新する必要がある。また、ページをメモリに読み込む際には、遅延されたファイル間ポインタの更新をページイメージ中のポインタに反映させなければならないので、読み込むページに含まれるreference tableエントリを検索する必要がある。このように、reference tableはファイル間ポインタの値とその所在の2つの値を鍵として検索されるので、ポインタの値で検索できるoutgoing reference tableと、ページテーブルからたどれるreference listの2つの要素から構成される(図4)。これにより、ポインタ更新メッセージが到着した際にはoutgoing reference tableに対して検索と更新を行ない、ページを読み込む時にはページテーブルからreference listをたどりoutgoing reference table中のポインタとページイメージ中のポインタを比較し、更新する。

さらに、outgoing reference tableでファイル間ポインタのreference countを行なうことにより、最初のファイル間ポインタが作成された場合と最後のファ

イル間ポインタがなくなった場合にのみ referent table エントリの作成と解放メッセージをデスティネーション側のファイルへ送出することが可能になる。この結果、同一ファイルから同一オブジェクトへのファイル間ポインタが複数存在する場合の効率化が計れる。

4.3 トランザクション

2.2節でトランザクションが非永続空間に持つファイル間ポインタをGCが考慮する必要をなくすために、ファイル間ポインタを持ち得るトランザクションをアボートさせることを説明したが、これは outgoing reference table の更新を検査することにより実現する。本システムでは snapshot validation を用いた optimistic アルゴリズム [4] によりトランザクションを実装しているが、これはページ毎にタイムスタンプを設けることにより実現する。通常の snapshot validation では、トランザクション終了時の validation フェーズで、メモリ上にページを読み込んだ時のタイムスタンプの値と現在の値を比較し、全てのページに関して一致した場合に commit できると判断する。しかし、これだけでは、GC からのメッセージにより outgoing reference table へ更新があった場合でもトランザクションが commit してしまい、GC によって更新されなかったポインタが永続ヒープに書き込まれる恐れがある。

そこで、validation フェーズでタイムスタンプを比較するだけでなく、reference list をたどり outging reference table にあるファイル間ポインタとポインタの実体を比較し、メッセージによる更新がなかった場合のみ commit するとした。これにより、GC 中のファイルへのポインタを持ち得るトランザクションをアボートさせることを IBP のみで可能にしている。

5 関連研究

永続オブジェクト管理システムの初期の研究に、3種類のアドレス空間を用い S-Algol 上に実装した PS-Algol[1] がある。C++ の拡張である E[5] は二次記憶上とメモリ上の2種類のポインタを変換するコードを言語処理系に生成させることにより、永続・非永続の透明性を実装した。Eos[3] は言語 Leos により永続・非永続の透明性を提供し、forwarder, IRT, ORT を用いた on-the-fly GC,copying GC を実現している。以上のシステムは言語処理系に依存していたが、Cricket[6] は、複数の言語の storage system とするべく、Mach の外部ページャ上で実装された。本研究と同じく単一ポインタ表現を用いているが、storage system として開発されたので GC の実装は考慮されていない。

6 まとめと現状

IBP とトランザクションの GC への利用という枠組により、実行時性能がよく、複数の言語のプラットフォームになることができ、かつ、compacting GC 可能な分散永続オブジェクト管理システムが構築できることについて述べた。現状は、SPARC Station 上でローカルバージョンの実装中であり、C 言語で 9,000 行ほどになっている。トランザクションの管理、仮想記憶機構の利用、IBP の作成や更新などの基本的な部分は実装済みで、今後 GC プロセスと Local Server のインタフェースを固め、GC プロセスを実装する予定である。

参考文献

[1] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison, "An Approach to Persistent Programming," *The Computer Journal*, vol. 26, pp. 360–365, Nov. 1983.

[2] P. B. Bishop, *Computer Systems with a Very Large Address Space and Garbage Collection.* PhD thesis, Massachusetts Institute of Technology Laboratory for Computer Science, May 1977. Technical report MIT/LCS/TR-178.

[3] O. Gruber, L. Amsaleg, L. Daynès, and P. Valduriez, "Eos, an Environment for Object-based Systems," in *Proceedings of the 25th Anniversary Hawaii International Conference on Systems Science*, pp. 757–768, IEEE Computer Society, Jan. 1992.

[4] U. Prädel, G. Schlageter, and R. Unland, "Redesign of Optimistic Methods: Improving Performance and Applicability," in *Proceedings of the 2nd International Conference on Data Engineering*, pp. 466–473, IEEE Computer Society Press, Feb. 1986.

[5] J. E. Richardson and M. J. Carey, "Persistence in the E Language: Issues and Implementation," *Software: Practice and Experience*, vol. 19, pp. 1115–1150, Dec. 1989.

[6] E. Shekita and M. Zwilling, "Cricket: A Mapped, Persistent Object Store," in *Implementing Persistent Object Bases: Principles and Practice*, pp. 89–102, Morgan Kaufmann Publishers, 1990.

[7] P. R. Wilson, "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware," *ACM SIGARCH Computer Architecture News*, vol. 19, pp. 6–13, June 1991.