分散永続オブジェクト管理システム上のバージョン間衝突の解決*

1H－8

山本 耕平　原 政博　上原 敬太郎　宮澤 元　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

1　はじめに

ポインタを含むような複雑なデータ構造の永続化と共有を容易かつ効率よく行うために、永続オブジェクト管理システムが提案されている。これは、永続的なデータを永続オブジェクトとして抽象化し、永続オブジェクトのガーベッジコレクション(GC)や、アプリケーションプログラムが永続・非永続オブジェクトを区別することなく扱える透明性などを提供するものである。

このようなシステムでは、アプリーケーションの行うオブジェクト単位の操作の提供が主な目標ではあるが、永続オブジェクト全体の管理運用を考えると、永続オブジェクトの一定のまとまりであるファイル単位の操作を提供する必要もあると考えられる。アプリケーションに管理運用のための操作をまかせるとすると、永続オブジェクト全体としては様々なアプリケーションから共有されるので、各ファイルに応じて整備のためのプログラムを変更しなくてはならず、管理者の負担は増大し、アプリケーションのバグにより正常な管理が行なえない可能性を高めてしまう。このように、管理運用のための基本的な操作には、システムが提供する統一されたインタフェースが必要である。本稿では、永続オブジェクト管理システムにファイルシステム的な操作を導入することに伴う問題点とその解決について述べる。

2　ファイル操作導入の問題点と解決法

我々が研究している永続オブジェクト管理システムLucas Persistent Store(LPS)は、実行時オーバーヘッドが小さく、特定の言語処理系に依存しないことを目標に、単一仮想空間とSingle Level Storeを組み合わせ、永続オブジェクトを一意にアドレス付けしている。Opal[2]などの64ビットOSや、64ビットを想定した他の永続オブジェクト管理システム[3]でも同様の方法で低オーバーヘッドを実現している。しかし、一意なアドレス付けのため、永続オブジェクトの集合に対して一般的なファイルシステムとしての操作を適用する場合に、単純なファイルシステムと異なり、データ構造の意味、特にポインタの意味を保存する必要がある。例えば、ファイルの削除はdangling pointerを生じさせないようにする必要があるし、ファイルの複写はポインタの参照関係を保存する必要がある。データ構造を保証するために必要な処理をアプリケーションに期待することはできないので、このような操作はシステムで管理しなくてはならない。特に複写について見ていくと、その意味は幾通りも考えられるが、指定されたファイルだけを複写するshallow copyと、指定されたファイルを起点にファイル間ポインタの推移閉包を複写するdeep copyの2通りに大別できる。しかし、後者の場合、大量のデータを複写する必要が生じ、即時に複写・再配置したのでは操作にかかるオーバーヘッドと消費する二次記憶容量から効率的ではない。

本研究ではこの問題を、ファイルのバージョン化と、それにより発生するバージョン間衝突を解決するアルゴリズムを導入することにより、複写と再配置を可能な限り遅延させ解決する。LPSにおけるバージョン化とは、単一仮想空間の制約を緩め、同一アドレスに複数のファイルが存在し得るようにした上でcopy-on-writeを行ない、再配置を行なわずに仮想的に複写することである。具体的には、複写後のファイルを参照するアプリケーションには、copy-on-writeされている複写後のファイルを元のファイルの存在したアドレスにマップする。

バージョン化にともない発生するバージョン間衝突とは、バージョン化されたファイルへのポインタが参照、作成、複製された場合に、どちらのバージョンを参照しているのか不明であるという問題である。表面上は区別が困難でも、本来、最初にポインタが作成された時に参照していたバージョンを参照していると解釈できる。この解釈に従い、GCのために導入したデータ構造と現在マップしているファイル群を元に、ポインタ作成時に参照していたバージョンを選択するアルゴリズムを考案し、衝突問題を解決した。しかし、現在マップしているファイル内に両バージョンを参照するポインタが混在していると判定された場合には、意味的にも両バージョンを同時に参照し得るので、実際の複写と再配置を行う。

3　Indirect Back Pointer (IBP)

LPSのファイルは、GCのために導入したIBPと型情報、ファイルのページイメージの3つの情報で構成される。IBPとはファイル間のポインタを列挙するデータで、ファイル毎のcompacting GCと単一仮想空間による低実行オーバーヘッドを両立するためのものである[1]。具体的には、ファイル間ポインタのソース側に作られファイル間ポインタの所在を示すreference tableと、ファイル間ポインタのデスティネーション側に作られファイル間ポインタによって指されるオブジェクトを示すreferent tableから構成され(図1)、システムにより矛盾のないように管理される。

Arbitration among Versions on the Distributed Persistent Object Management System

YAMAMOTO Kouhei, HARA Masahiro, UEHARA Keitaro, MIYAZAWA Hajime, INOHARA Shigekazu, and MASUDA Takashi
The Department of Information Science, Graduate School of Science, the University of Tokyo

*本研究の一部はIPAの独創的情報技術育成事業の支援を受けて行なわれています。

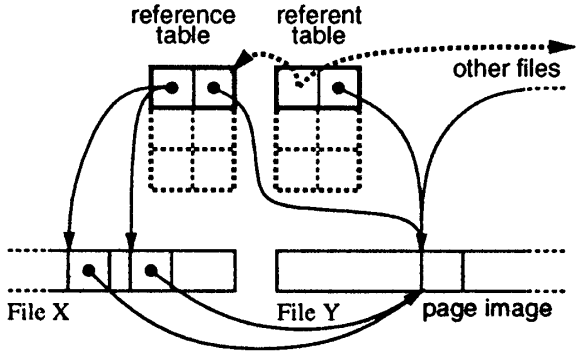

図 1: Indirect Back Pointer の構造

4 バージョン化

バージョン化は基本的にファイルの各要素を copy-on-write し、前のファイルと同一のアドレスにマップすることにより行なわれる。ページイメージと型情報は単純に copy-on-write すればよいが、IBP の複写には複写の意味を加味しなくてはならない。shallow copy の場合、新しくバージョン化されたファイルはまだ他のファイルから参照されていないと考えられるので、以前の referent table とは無関係に空の referent table を作成する。deep copy の場合、複写対象になっているファイルからの参照に限って複写し新しく構築する。reference table の複写はページイメージの copy-on-write と組にして遅延し、実際に書き込みが生じたページにだけ新しく reference table を作成する。これにより、reference table が示す参照先にバージョン化により参照の増えたことを通知するメッセージも遅延でき、複写の操作の分散環境における局所性が向上する。

一方、複写時に別のアドレスを割り当てておき単純に copy-on-write するという手法も考えられる。しかし、この方法ではページを読み込む毎にそのページのポインタを再配置する必要があるので、アプリケーションの実行効率を下げる。そのコストを 1 回限りにするために再配置後のページイメージを保存しておくとすると、参照される毎に共有されるページが減るので、二次記憶容量の効率も悪い。

5 バージョン間衝突の解決

バージョン間衝突の解決は、ファイル間ポインタを最初に作成した時にどのファイルから参照したのかが referent table に記録されている点がキーポイントである。衝突時には、それぞれのバージョンを参照しているファイル群と現在マップしているファイル群から、問題のポインタを作成した時にどちらのバージョンをマップしていたかがわかるので、現在マップされているファイルからポインタをたどって到達可能なバージョンを選択していく。マップしているファイルから両方のバージョンを参照し得る場合には、複写と再配置を行う。ただし、スタックやレジスタなどシステムが再配置できないところに両方のバージョンへのポインタが存在する可能性があるので、実行中のアプリケーションは再実行させる必要がある。LPS ではトランザクション単位で永続記憶へのアクセスを行なっているので、トランザクションをアボートさせ、再配置後に再実行させることができる。

既にマップしたバージョン (a) の別バージョン (a') を参照しているファイル群をマップさせないために、ファイルをマップする時には別バージョンを参照しているファイルをマップ禁止ファイル群として記録しておく。後でマップしようとするファイルがマップ禁止ファイル群に含まれていた場合、まだマップしていない方のバージョン (a') を再配置する。この場合は、再配置する a' へのポインタはまだ参照されていないので、再配置が終るまでトランザクションの実行を延期するだけでよい。

また、トランザクション終了時に新しいファイル間ポインタを IBP に登録する場合にも、ファイル間ポインタの指すアドレスがバージョン化されているとバージョン間衝突となる。この場合も前述の判定方法を適用し、新しいポインタがどのバージョンを指しているか決定不可能な場合トランザクションをアボートし再実行させる。

これらの衝突解消処理はマップ時とトランザクション終了時だけ発生し、一度マップしたファイルへの読み書きにオーバーヘッドはないので、判定にかかるコストはほぼ無視できる。性能上の問題点としては、実際の複写と再配置をしなくてはならない場合の処理にかかるオーバーヘッドである。マップ禁止ファイルに該当した場合、トランザクションは一旦停止して再配置後に再開するだけなので、即座に再配置する場合と比較してコストの増加はない。両バージョンを参照し得ると判定され、トランザクションを再実行した場合は、即座に再配置した場合より再実行の分だけ多くのコストがかかってしまう。しかし、この場合になるのは、派生したバージョンの両方を同時に参照しようとした場合であり、一般的には状況は両方のバージョンが大きく食い違ってきた時に起こると考えられる。共有部分が少なくなった場合には複写と再配置を行なう等の発見的手法を導入すれば、事前に別々のファイルとなっている可能性を高くすることができる。

6 まとめ

分散永続オブジェクト管理システムにファイルシステム的な操作である複写を導入した際に、バージョン化という効率的な実装方法と、それに伴うバージョン間衝突の解決方法について述べた。現在、LPS を Mach 3.0 上で実装中であり、今後は本機構を導入し、バージョン化による改善とトランザクションのアボートの危険性から再配置処理の閾値を調べていく予定である。

参考文献

[1] 山本耕平, 猪原茂和, 益田隆司. “単一ポインタ表現を持つ分散永続ヒープ上の GC の枠組.” SWoPP 琉球 '94, 第 18-1 巻, pp. 81–88, 7 月 1994 年.

[2] Jeff Chase, Hank Levy, Miche Baker Harvey, and Ed Lazowska. “Opal: A single address space system for 64-bit architecture.” In *Proceedings of the Third Workshop on Workstation Operating Systems*, Key Biscayne, Florida, April 1992.

[3] Alan Dearle, Gmail M.Shaw, and Stanley B. Zdonik, editors. “*Implementing Persistent Object Bases: Principles and Practice.*” Morgan Kaufmann Publishers, 1990.