第6卷第11期
1995年11月

软 件 学 报
JOURNAL OF SOFTWARE

Vol. 6, No. 11
Nov. 1995

持久 OOPL 的存储管理模式*

陈 睿

(华南理工大学计算机系，广州 510641)

摘要 在持久环境(如持久 OOPL 和 OODBS)中，涉及到两类不同的物理存储器：内存和磁盘. 应开发隐藏物理细节的抽象存储器层次. 虚拟对象存储器(VOS)就是这种抽象存储器层次，在其上，持久对象和暂态对象的操作是同一的. 作者利用双拷贝模型实现了一个 VOS 管理系统. VOS 方法的优越性已通过 XDPC++的设计与实现得到了证明. XDPC++是基于 C++的持久 OOPL.

关键词 持久性，面向对象程序设计语言，面向对象数据库，存储管理.

数据库系统是多数计算机信息系统的核心子系统. 传统的数据库系统适用于商业事务. 但一些新的应用，例如 CAD、OA、多媒体系统等，都要求新的数据建模机构和新的数据库功能. 因而，有必要发展新一代数据库系统. 面向对象的数据库系统(OODBS)被认为是发展新一代数据库系统的一个很有希望的方向. 传统的数据库系统和程序设计语言是相互独立而发展的，这使得应用程序员面临着所谓的"阻抗失配"问题. 程序设计语言缺乏数据库功能，而数据库系统的语言又缺乏计算完备性. 这就需要将数据库系统和程序设计语言无缝地集成起来. 面向对象的数据库程序设计语言(OODBPL)正是这样一种集成系统. 发展 OODBPL 的一个方向是在 OOPL 中引入持久性，再利用持久 OOPL 作为数据库实现的基础[1,2]. 另一方向是在已有数据库系统的基础上发展持久 OOPL. 无论是哪种方向，持久 OOPL 都具有极端的重要性.

持久程序设计的一个关键问题是存储管理模式. 对传统的程序设计语言和数据库系统而言，存储管理模式都是单方面的. 在传统程序设计语言中，所有对象或值都是暂态的，因而只要考虑内存管理问题. 另一方面，传统的数据库系统将所有对象都看成是持久的，因而主要关心磁盘管理. 然而，在持久程序设计语言中，涉及到两类不同的物理存储器——内存和磁盘，因而内存管理问题和磁盘管理问题相互渗透. 现有的持久 OOPL 或 OODBS 都有自己的存储管理模式，它们在概念和实现的各方面都不尽相同. 但没有一种存储管理模式具有足够的抽象程度以使用户获得清楚的理解，使实现者方便地重用和修改存储管理系统. 为解决这一问题，本文提出了虚拟对象存储器(VOS)方法. 虚拟对象存储器是一个抽象存储器层

* 本文 1994－05－16 收到，1994－09－19 定稿

本课题部分工作由广东省自然科学基金支持. 作者陈睿，1968 年生，讲师，主要研究领域为面向对象程序设计，数据库系统.

本文通讯联系人：陈睿，广州 510641，华南理工大学计算机系

次,在其上,持久对象和暂态对象的操作是同一的.我们利用双拷贝模型实现了一个VOS管理系统.VOS方法的优越性已通过XDPC++的设计与实现得到了证明.XDPC++是基于C++的持久OOPL.

1 持久OOPL的存储管理模式

在传统程序设计语言中,数据通常是暂态的并存储在内存或虚拟存储器中,其存储管理模式主要处理下述问题:(1)寻址机制;(2)何时为数据分配和解配存储空间;(3)联编机制.关于寻址机制,提出了虚拟存储器方法,有效地扩大了寻址空间.联编(binding)机制主要负责将名字与地址相关联.对于第二个问题,有两种最重要的方法:(1)由程序设计语言定义名字作用域规则和生存期规则,依此采用栈式或堆式存储管理方法;(2)自动废址收集方法.

传统的数据库系统则主要关心磁盘管理.其存储管理模式负责进行数据库的物理组织,并借助索引、聚簇(clustering)、缓冲等技术来加快数据库存取的速度.通常认为OODBS应具备持久程序设计界面.在此情况下,应用程序运行期间将同时存在持久对象和暂态对象.据文献[3],这意味着持久程序设计的关键支撑技术是工作区管理或内存中对象的缓冲管理.我们认为,上述问题属于存储管理模式的范畴.

持久OOPL涉及到两类存储器:存储暂态对象的易变存储器(Volatile Storage),和存储持久对象的持久(或稳定)的存储器.物理上,易变存储器对应于内存或虚存,持久存储器对应于磁盘,这使得寻址、分配、解配和联编都复杂化.文献[4]中详述了持久程序设计语言中的联编机制.本文主要考虑寻址机制.先综述现有系统中采用的主要方法.

E[5]和O++[6]都在C++基础上扩充了持久性.E或O++的用户要操作多种指针.暂态对象位于内存中并持有内存地址,持久对象位于磁盘上并持有磁盘地址.正常的C++指针以内存地址表示,而E中的db类型指针或O++中的persistent指针以磁盘地址表示.O++提供了第三类指针——dual类型指针来沟通正常的C++指针和persistent指针.E和O++的方法意味着用户必须面对两个不同的地址空间,且相互间的映射不透明.

ObjectStore[7]的寻址方式与虚存寻址方式基本一致,只不过ObjectStore将数据库视作持久的虚地址空间.通常,ObjectStore的一个数据库是一个虚地地址空间中;若单个数据库的规模很大,在一个虚地址空间中放不下,则将该数据库划分为多个分区,每一分区对应一个虚地址空间.ObjectStore对C++进行了扩充,使之支持持久性,并提供了专门的编译器.在ObjectStore C++程序中,可以创建和删除持久对象,所有的持久对象都是跟特定的数据库相关联的.ObjectStore负责为每个数据库指定一个或多个持久的虚地址空间,并利用操作系统的虚拟存储器功能在适当的虚地址空间中为持久对象和暂态对象分配和解配空间.指向持久对象的指针和正常C++指针持相同格式.ObjectStore采取这种设计的动因是性能问题.文献[7]声称,指向持久对象的指针和正常C++指针的引用求值同速,但未说明在ObjectStore C++中,非持久对象指针能否与其它C++编译器下C++指针同样快地引用求值.ObjectStore C++的寻址机制可概括如下:

- 多个虚地址空间同时存在;
- 多个虚地址空间中的地址采用相同格式;

• ObjectStore 负责为每个对象选定合适的虚地址空间.

LOOM[8]本身不支持持久性,但 GemStone[9]和 Orion[10]都采用了 LOOM 的方法来构造存储管理系统. LOOM 采用了虚存思想,但却与通常的虚拟存储管理系统(如页式虚拟存储器)大相径庭. 在 LOOM 中,每个对象都被赋予一个系统定义的对象标识符. LOOM 维持了一张名为 ROT(Resident Object Table)的运行时表,每个驻留在内存中的对象的标识符都被记录在 ROT 中. 当使用一个对象时,LOOM 就查找 ROT,以确定该对象是否在内存中;若不在,则从辅助存储器中调入该对象将其标识符填入 ROT 中. 当内存空间已满,LOOM 就从内存中调出一些对象到辅助存储器上,并从 ROT 中删去相应的标识符. GemStone 和 Orion 采用了 LOOM 的方法并扩充了持久性. 在 GemStone 和 Orion 中对象在内存和数据库间调入调出. Orion 还采用了双缓冲器技术来加速对象的交换,即一个缓冲器用于数据库页,另一则用于内存格式对象. Gemstone 提供了类似 Smalltalk 的 DML—OPAL,它是计算完备的. 这两个系统的寻址机制比较高级:只涉及一个地址空间,每个地址是一个逻辑地址——对象标识符,而逻辑地址与物理地址之间的映射则是透明的.

PS—Algol[11]涉及到两个地址空间——虚地址空间和持久存储空间. 在虚地址空间中的对象以 LON(Local Object Numbers)为地址,在持久存储空间中的对象以 PID(Persistent IDentifiers)为地址. PS—Algol 的指针值或为一 PID,或为一 LON. 当对一个指针进行引用求值(dereferencing)操作时,首先对该指针施加一个软件检查,看它是否为一个 PID. 若是 PID,则运行时间系统查一个名为 PIDLAM 的运行时间表. 每一个 PIDLAM 表项都记录了内存中持久对象的 PID 和对应的 LON. 如果查到了指针所持的 PID,则转换为相应的 LON. 否则,运行时间系统从磁盘上调入具有该 PID 的对象并赋予一个合法的 LON,并更新 PIDLAM. 简言之,PS—Aogol 能够通过 PID 和 LON 之间的双向映射来同时操纵两个地址空间.

对于持久程序设计语言,上述方法有一些严重问题:(1)依赖于物理存储器;(2)为特定的语言特性而专门设计;(3)结构复杂;(4)缺乏可移植性和可维护性. 因而,必须对物理存储器进行一定的抽象. 为此,我们提出了虚拟对象存储器的思想.

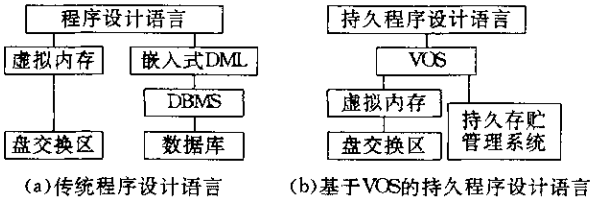
2　虚拟对象存储器(VOS)

虚拟对象存储器是建基于内存和磁盘上的一个抽象存储器层次. 提出 VOS 是基于这样一个原则的:持久性是对象的时态特性. 除了时态特性的差异外,持久对象和暂态对象之间不存在任何其它差别,因而,不论它们被存储在何处,它们应被同一地对待.

VOS 是指一个唯一的虚地址空间,在这个虚地址空间中每一对象位于某一地址. VOS 中的对象,无论是持久对象还是暂态对象,其寻址方式都是一样的. VOS 的优越性在于:(1)为用户提供了单一的存储器层次. 用户可以用相同的方法来操纵持久对象和暂态对象,而没有必要了解它们存储在哪里. (2)免除了实现者操作物理存储器的沉重负担.

回忆一下虚拟存储器的作用有助于更好地理解 VOS. 图 1 对传统程序设计语言和基于 VOS 的持久程序设计语言进行了比较.

为了支持 VOS 方法,必须有 VOS 管理系统. VOS 管理系统必须具备隐藏一切物理细



(a)传统程序设计语言 (b)基于VOS的持久程序设计语言

图1

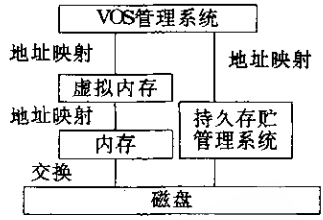


图2 VOS管理系统结构

节的寻址机制. VOS 在物理上是由内存(或虚拟存储器)和磁盘构成的,两种不同的物理存储器使用不同的寻址机制. 因而,VOS 寻址机制实际上是使用不同的物理寻址机制的一个统一界面. VOS 管理系统必须能够以透明的方式完成 VOS 地址与物理存储器地址之间的双向映射. 其次,VOS 系统必须负责物理存储器的组织. 当一个对象被创建时,VOS 系统应能在物理存储器中为它分配适当的空间;当一个对象被销毁时,VOS 应能回收和重用它所占用的空间. 如果有现成的物理存储管理系统,VOS 系统负责调用它们. 此外,同一对象在内存中和在持久存储空间中的格式可能不同,在这种情况下,当对象在内外存之间传递时 VOS 系统应负责格式转换. 图 2 表示一种可能的 VOS 系统结构.

我们在双拷贝(dual—copy)模型的基础上开发了一个 VOS 管理系统. 下一节将讨论有关概念和实现.

3 基于双拷贝模型的 VOS 实现方法

众所周知,主存储器的存取效率远远高于辅助存储器. 因而,当计算一个对象时,这个对象应该在内存中. 虚拟存储器的交换技术(swapping)和 DBMS 的缓冲技术都是为了保证当前处理的对象驻留在内存. 在 VOS 中,持久对象存放在磁盘上,暂态对象存放在内存或虚拟存储器中,为了计算持久对象,VOS 系统必须透明地完成持久对象在内外存之间的传递. 这里,笔者提出了双拷贝模型:任何持久对象在被计算时都有两个拷贝,一个拷贝在内存中,一个拷贝在磁盘上. 磁盘拷贝可以永久存在,这保证了对象的持久性. 两个拷贝之间存在着透明的双向映射. 对于用户而言,所有的对象,不论是暂态的还是持久的,在被计算时都驻留在内存中. 这样,就形成了一个 VOS,这个 VOS 表面上就是内存.

3.1 双拷贝模型

如上所述,双拷贝模型可以作为 VOS 的实现方法. 但是,有一些问题必须解决:(1)何时为持久对象制备内存拷贝? (2)何时删除持久对象的内存拷贝? (3)如何在内存拷贝和磁盘拷贝中保持持久对象的引用结构? (4)如何保证持久对象的数据完整性?

3.1.1 对象活跃期

我们现在引入对象活跃期的概念. 对象活跃期(Object extent)是指对象驻留在内存或者位于虚拟存储器中的那段时间. 对于暂态对象而言,其活跃期正好就是它的生存期. 持久对象的活跃期则不同于其生存期,持久对象存储于磁盘上并能永远存在,但只在程序运行的特定时间段中才被调入内存(或虚拟存储器).

持久对象的内存拷贝在它的活跃期开始时被创建,在活跃期结束时被删除. VOS 不可能知道什么时候要对一个对象进行计算,因而定义对象活跃期不是 VOS 的责任. 对象活跃

期应由持久程序设计语言来定义.这表明,持久程序设计语言的设计者可以有在基于自动废址收集和基于作用域、生存期规则的运行时存储管理方法间进行选择的自由.

3.1.2　对象间的引用关系

在持久环境中,总共存在四种类型的对象间引用,如下所示:

引用类型	持久对象	暂态对象(被引用)
持久对象	R1	R2
暂态对象(引用)	R3	R4

其中,只有 R1 类型的引用关系才能被持久保存.设 O1 是一个持久对象,O2 是一个暂态对象.当 O2 被销毁后,O1 对 O2 的引用(R2 类型)是无意义的.如果 O2 引用 O1,这一引用(R3 类型)关系可以转换成 O2 对 O1 的内存拷贝的引用.这样,引用关系的类型就变成了 R4.R4 类型的引用关系在大多数传统程序设计语言中都能被有效和正确地处理.

持久引用(R1 类型)被表示为 PID(NIL 是一个特殊 PID,表示无意义的引用).任一持久对象都具有唯一的 PID.必须建立 PID 和内存(或虚存地址)的双向映射机制.类似于 PS-Algol 的方法(使用运行时表 PIDLAM)适用于这种要求,在笔者的实现方法中,使用了一个称作 PstList 的运行时表.以上的讨论非常明确地区分了持久对象和暂态对象.事实上,应该由持久程序设计语言来决定各个对象是持久的还是暂态的,而不是由 VOS 系统来决定对象的时态特性.语言的设计者也因而获得了选择对象持久性语义(如 O2[12]的引用可达语义或 O++的显式声明语义)的自由.

3.1.3　持久对象的调入和调出

持久对象的调入(pinning)是为持久对象创建内存拷贝的过程.调出(unpinning)是调入的逆过程,即用内存拷贝重写磁盘拷贝并删除内存拷贝的过程.持久对象的调入和调出分别发生在持久对象活跃期开始和结束的时刻.持久对象的内存拷贝和磁盘拷贝的格式不同,这是由 PID 与内存地址格式不同所造成的.因而,在调入和调出的过程中,必须进行格式转换.这就要求 VOS 系统了解对象的结构.在具体实现中,我们将持久对象的结构信息存放在其 PID 中.这些结构信息告诉 VOS 系统一持久对象中是否包含对其它对象的引用(即指针,表示为被引用对象的 PID)及这些指针的位置.

要保证持久对象一致性,VOS 系统必须确保任意持久对象在任意时刻只能有至多一个内存拷贝.在双拷贝模型中,施加于持久对象上的所有操作都作用于它的内存拷贝.那么,在调出时持久对象根据它的唯一的内存拷贝被更新,就可以保证持久对象的一致性.我们是通过 PstList 来保证内存拷贝唯一性的.任意持久对象,如果有内存拷贝,则它在 PstList 中有一个表项.PstList 的表项称作"伴生元"(Companion),伴生元中记录了持久对象的 PID 和其内存拷贝的内存地址.当调入一个持久对象时,就查 PstList,看是否有一个伴生元的 PID 等于待调入对象的 PID.如果没有,则调入该持久对象,同时创建一个伴生元,并将其插入 PstList 中.否则就终止调入过程.当调出一个持久对象时,从 PstList 中删除其伴生元.

3.1.4　持久对象间引用关系的处理

表示持久对象间引用关系的指针在磁盘拷贝中是 PID,而在内存拷贝中则是内存地址,因而在调入和调出时不但要进行格式转换,还必须映射持久对象间的引用关系.设 O1 和

O2是两个持久对象,O1引用了O2,则O1的磁盘拷贝中的特定位置上包含有O2的PID,而这一位置信息包含在O1的PID中.在这种物理实现中,VOS系统可以根据O1的PID很容易地确定O1对O2的引用关系.当调入O1时,运行时系统先将O1的磁盘拷贝读入内存,再根据O2的PID在PstList中查找O2的伴生元,取得O2的内存拷贝地址,然后将O1中包含的相应指针转换为O2的内存拷贝地址,即得到O1的内存拷贝.当调出O1时,运行时系统又要在PstList中查找一个伴生元,该伴生元包含的内存拷贝地址正好等于O1的内存拷贝所含的那个指针值(可能已改变,若未变,则将找到O2的伴生元).如找到,将O1的内存拷贝中所含的指针转换成该伴生元包含的PID,然后重写O1的磁盘拷贝.如果没有找到,就将O1所含的引用表记为无意义的(NIL).上述方法只在一种情况下有效,即O1所引用的持久对象都有内存拷贝.于是,双拷贝模型强制语言设计者遵守一条约束:持久对象O1的活跃期应包含在被O1直接或间接引用的任意持久对象的活跃期中.

3.2 基于双拷贝模型的VOS系统

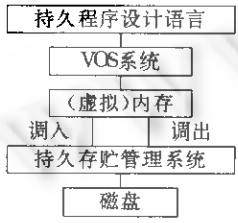


图3 基于双拷贝模型的VOS系统

在双拷贝模型中,用户(语言实现者和程序设计员)只需操纵内存(或虚拟存储器)就可以获得持久性支持.这样,就形成了一个抽象存储器,它表面上就是内存(或虚拟存储器).图3表示了基于双拷贝模型的VOS系统结构.

这种实现方法有下述好处:(1)虚拟存储器是很有帮助的,但并不必需;(2)获得了两个层次的独立性:向上独立于语言设计,向下独立于物理存储管理系统;(3)持久对象的内存拷贝与暂态对象没有任何差别;(4)提高了运行时效率;(5)性能问题集中于持久对象调入调出效率.

双拷贝模型只要求上层的语言机构正确地定义对象活跃期,告知对象的时态特性和结构信息.与现有的持久存储模式相比,VOS方法对持久程序设计语言具有更高的独立性:不论语言是采用引用可达的持久性语义还是显式声明的持久性语义,是采用堆栈式存储管理模式还是自动废址收集,是静态类型化的还是动态类型化的,是静态联编还是动态联编,都可以以VOS为基本环境.因而,VOS方法能适应多变的语言设计要求.对于下层的物理存储管理也没强加任何限制,只要能完成持久对象的调入调出,只要构造适当的接口,VOS系统就能通过调用物理存储管理系统来有效地工作.因而,一些行之有效的技术,如虚拟存储器技术、数据缓冲、存储保护等都能被直接和自然地加以利用.

4 VOS方法在XDPC++设计和实现中的应用

XDPC++是支持类型完全和类型正交的对象持久性[2]的C++[13]语言超集.其编译器已在VOS系统的基础上开发成功.XDPC++的设计和实现证明了VOS方法的优越性.

4.1 XDPC++的主要语言特性

我们在XDPC++中加入了3个关键字:persistent,pnew和pdelete.persistent是一个存储类说明符,它可以出现在存储类说明符auto可以出现的任何地方.它用于声明持久对象,并遵循C++的作用域规则.例如:persistent T *pt; // T指任意类型,声明了一个T类型指针pt,它本身是持久的,可指向持久的或暂态的对象.persistent T t,声明了一个T

类型持久对象.

pnew 是一个用于动态创建持久对象的 XDPC++算符,其用法与 C++算符 new 一样.例如,语句:pt=pnew T;创建了一个 T 类型的持久对象,pt 指向该对象.

pdelete 永远地删除由 pnew 创建的持久对象.例如:pdelete pt;

4.2 活跃期规则

XDPC++定义了一组活跃期规则,以便能正确地调用 VOS 系统.XDPC++的活跃期规则包括:(1)暂态对象的活跃期就是它的生存期,由 C++的生存期规则定义;(2)任意块 B 中用 persistent 静态创建的局部持久对象的活跃期在程序运行到它的声明位置开始、程序退出块 B 时结束;(3)在任意块外用 persistent 静态创建的全局持久对象的活跃期是程序运行的整个期间;(4)·用 pnew 动态创建的持久对象的活跃期从程序运行到 pnew 时开始,到程序运行完时结束.

上述活跃期称为"语法活跃期".另外还存在一种"引用活跃期":(1)如果 O1 和 O2 是两个持久对象,O1 引用了 O2,而当 O1 的语法活跃期开始时 O2 并不在活跃期中,则对 O2 强加一个"引用活跃期",O2 的引用活跃期等于 O1 的语法活跃期.(2)由 pnew 创建的持久对象的活跃期(语法活跃期活跃期或引用活跃期)在对它实施 pdelete 操作时结束.

为了支持上述活跃期规则,我们要使用一些运行时表.在一 XDPC++程序开始运行时,要创建一个名为 GlobalExtList 和一个名为 PnewExtList 的运行时表.所有全局的持久对象,及被它们直接或间接引用的持久对象的伴元被插入到 GlobalExtList 中.当遇到一个 pnew 操作时,被创建对象的伴元被插入到 PnewExtList 中.当程序进入一个块时,要创建一个名为 LocalExtList 的运行时表,在此块中声明的所有持久对象及被它们直接或间接引用的持久对象的伴元被插入到该 LocalExtList 中.当程序退出一个块时,所有伴生元在该块 LocalExtList 中的对象被调出,并更新 PstList,销毁此块的 LocalExtList.程序运行结束时,对 GlobalExtList 和 PnewExtList 进行同样操作.显然,若程序正在块 Bi 之中运行,且 B1,…Bj 是 Bi 的外层块,则有:

$$PstList = GlobalExtList \cup PnewExtList \cup LocalExtList\ of\ B1 \cup \cdots \cup LocalExtList\ of\ Bi$$

4.3 对象间引用语义

如果类定义中含有指针类型(*)或引用类型(&)成员,就可导致对象间引用.在 XDPC++中,只需考虑一种情况,即持久对象 o1 引用了其它对象.我们定义了如下的引用语义:

(1)持久对象 o1 对另一对象 o2 的引用是有意义的,当且仅当 o2 是持久的且 o1 的活跃期包含在 o2 的活跃期中;

(2)如果持久对象 o1 引用了另一对象 o2,但上述条件不成立,则在调入和调出 o1 时,这一引用被当作对 NIL 的引用.NIL 是一预定义的 PID,表示任意非持久对象.

遵照如上语义,程序员可避免无意义的引用.

4.4 XDPC++的实现

在 VOS 系统的基础上,我们很容易地实现了 XDPC++.首先开发了名为 XKERNEL 的运行时系统,它负责利用 VOS 系统控制活越期和联编.接着开发了预编译器 XCP,它将 XDPC++程序翻译成 C++代码.XCP 负责生成类型描述符(描述每一类型的实例结构)供 VOS 系统使用,将含有 3 个 XDPC++关键字的语句转换为正确的 XKERNEL 调用.

5 结 语

虚拟对象存储器是一个抽象、通用和简单的存储器层次.在其上,可以用同一的方式来操纵时态特性不同的对象.本文将持久OOPL存储管理模式划分为3方面问题,理清和简化了VOS研究的思路,获得了向上相对于高层语言构造,向下相对于物理存储管理技术的独立性.我们还提出了用于VOS实现的双拷贝模型.VOS方法的优越性已通过XDPC++的设计与实现得到了证明.

参考文献

1 Weger P. Learning the language. Byte, March 1989,14(3):245—253.
2 Atkinson M P, Buneman O P. Types and persistence in database programming languages. ACM Computing Surveys, June 1987,19(2):105—190.
3 Kim W. Object—oriented databases: definition and research directions. IEEE Trans. on Knowledge and Data Engineering, September, 1990,2(3):327—341.
4 Morrison R *et al*. Bindings in persistent programming languages. SIGPLAN Notices, April 1988,23(4):27—34.
5 Richardson J E, Carey M J. Persistence in the E language: issues and implementation. Software Practice and Experience, December 1989,19(12):1115—1150.
6 Agrawal R, Gehani N H. ODE(object database and environment): the language and the data model. ACM—SIGMOD 1989 International Conference on Management of Data,1989. 36—45.
7 Lamb C *et al*. The objectstore database system. Communications of the ACM, October 1991,34(10):50—63.
8 Kaehler T. Virtual memory for an object—oriented language. Byte, August 1981. 378—387.
9 Butterworth P, Otis A, Stein J. The gemstone object database management system. Communications of the ACM, October 1991,34(10):64—77.
10 Kim W *et al*. Architecture of the orion neixt—generation database system. IEEE Trans. on Knowledge and Deta Engineering, March 1990,2(1):68—86.
11 Atkinson M P. An approach to persistent programming. The Computer J., 1983,26(4):360—365.
12 Deux O *et al*. The O2 system. Communications of the ACM, October 1991,34(10):34—48.
13 Stroustrup B. The C++ programming language. Addison—Wesley, 1986.

STORAGE MANAGEMENT SCHEMES FOR PERSISTENT OOPLs

Chen Rui

(*Department of Computer Science and Engineering, South China University of Technology, Guangzhou* 510641)

Abstract In persistent environments (e.g., persistent OOPLs, OODBSs), twodifferent kind of physical storage——memory and disk are involved. An abstract storage level should be developed to hide physical details. The virtual object storage(VOS) proposed in this paper is such an abstract storage level on which persistent objects and transient objects can be manipulated uniformly. The author has developed the dual—copy model for the VOS implementation. And the benefits of the VOS approach for persistent OOPL design and implementation are illustrated by the experiences with XDPC++, a persistent OOPL based on C++.

Key words Persistence, OOPL, OODB, storage management.