Strategies for Persistent Object Faulting: An Implementation in a Persistent C++ System *P3L*

（永続オブジェクトのフォールティング方式：永続C++システム *P3L* における実装）

鈴木慎司

①

Strategies for Persistent Object Faulting: An Implementation in a Persistent C++ System *P3L*

(永続オブジェクトのフォールティング方式: 永続 C++ システム *P3L* における実装)

鈴木 慎司

Acknowledgements

I would like to thank *Prof.* Masaru Kitsuregawa, my advisor, for the continued guidance on my research work and *Prof.* Miyuki Nakano for making my life as a graduate student at the Institute of the Industrial Science fruitful and enjoyable. My previous research work on parallel processing of satellite images stored in the geographical and environmental database convinced me of the importance of a persistent programming language. I would like to thank *Prof.* Mikio Takagi for his advice during that time and occasional home parties held for us students.

My appreciation also goes to my colleagues in the laboratory. *Dr.* Satoshi Hirano took time to discuss about issues, including but not limited to our research works. *Prof.* Masaya Nakayama helped me while I was trying to acquire familiarity with BSD Unix operating system. *Dr.* Minoru Nakamura, *Dr-to-be* Hiroyuki Tamura, Toshiyuki Nemoto and Kazuhiko Mogi shared their tips on programming, typesetting documents and system administration. *Stephen Davis* has offered his time in improving my conference papers under tight schedule. Takahiko Shintani helped me by taming a printer which liked to eat transparencies.

Without the freely available software mentioned in this thesis, this research could not have been completed. I would like to thank the authors of E, ESM, TEXAS, GCC, POVRAY and others that I failed to mention.

Finally, I would like to extend my gratitude to Huai Luu for the encouragements and the work in checking manuscripts of the thesis.

Contents

Chapter 1

Introduction

1.1 Background

Much research effort has been directed to the relational database (RDB) technology in the 70s and 80s and now commercial acceptance of RDB is booming. While RDBs meet the requirements from business applications such as banking, accounting and seat reservation systems, it has been pointed out that RDB does not well support so called *advanced applications* [1] [2]. Advanced applications are those which manipulate many complex or huge objects that are found in design applications and multi-media applications. Advanced applications encompass CAD/CAM tools, CASE tools, Hypertext and multi-media information systems, office information systems, knowledge bases, etc. The problem between these applications and RDB is termed *impedance mismatch*. In [1], the concept of impedance mismatch is illustrated by a figure like Figure 1.1. Another article[2] illustrates the same concept using Figure 1.2.

Virtually all programming languages used in building advanced applications support a construct called *reference* in order to model and represent certain relationships among data items[1] being managed. When a genuine reference in a programming language is

[1]Hereafter, the term *data item* and *object* are used interchangeably.

manipulated, the resolution of a relationship is usually very efficient because a reference often encodes location information of the associated data item. However, any relationships in RDBs is represented, in theory, through location independent values[2].

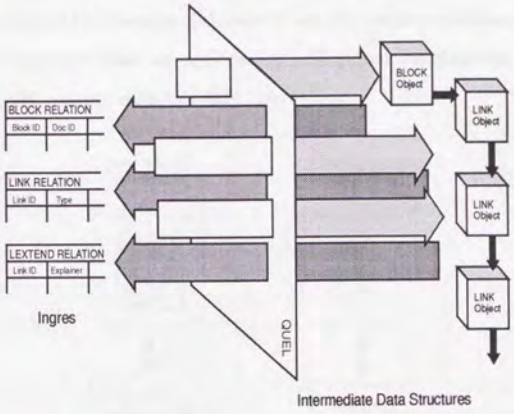


Figure 1.1: Impedance mismatch(1)

The difference in the constructs for representing relationship introduces both conceptual and performance problems in applications which utilize such programming languages in accordance with RDB. These problems are the results of the impedance mismatch. The conceptual problem is:

Due to the difference in the modeling of relationships, programmers must write code to shuffle and massage object representation when objects migrate between virtual memory space and a database system. This adversely affects programmer's productivity and reliability of the resulting products.

[2]Some RDBs provide back-door, for example, by providing keeping relation contiguous on a disk and by treating a certain type of value as an index into the contiguous region.

The above problem can be eliminated by providing a reusable software component for managing the conversion so that the software does not have to be rewritten for every new application. However, as stated previously, resolving a reference is likely a slow operation when its representation is based on a location independent value. Performance further degrades in practice because database is usually implemented as an independent process which is separate from an application. Required interprocess communication makes retrieving and storing objects costly operations.

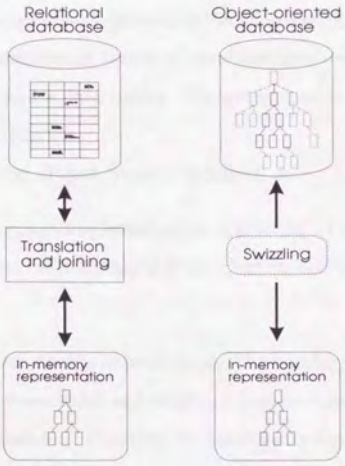


Figure 1.2: Impedance mismatch(2)

The approach presented in this thesis can be utilized to eliminate the conceptual gap, whether the underlining storage manager is relational or not. *P3L*, the system built through this thesis work makes little assumption about the architecture of underlining storage managers. This thesis introduces the concept and implementation techniques to automate the conversion. Much effort is spent in reducing the cost of checking whether

an object needs to be transferred from the persistent store to the memory space of an application.

1.2 Persistent Object Systems

In response to demands from advanced applications, many commercial vendors and research institutions came up with their implementations. They are termed object oriented database systems (OODB) and persistent object systems (POS). In this paper we consider that the term *persistent object system* has broader meaning than OODB in a sense that an OODB is more complete in terms of database functionality it provides, such as support for ad hoc query and a data model. The requirements for a system to be called OODB are enumerated in [3].

In this thesis a Persistent Object System is defined as:

> A programming language implementation consisting of compiler, runtime library and storage subsystem, which supports the creation and manipulation of persistent objects.

Now the term *persistent object* needs to be defined. The naming is biased towards the view of programming language researchers and implementors because, in database systems, all objects are inherently persistent. Contrary to database systems, objects are ephemeral in language systems meaning that the lifetime of an object created in an invocation of a program does not extend beyond the termination of the program. All useful information contained in created objects is lost when a program exits if no appropriate actions are taken. Traditionally the responsibility was placed on programmers. They have to write routines for recording information persistently on the persistent storage. That hints the impedance mismatch had existed before RDBs came out between languages and file systems. However, the performance problem introduced by interacting with file system is not as critical as with RDBs. As stated in the previous section, a RDB usually runs as

an independent process separated from an application. This would incur more cost in communication and data transfer as is illustrated in Figure 1.3.

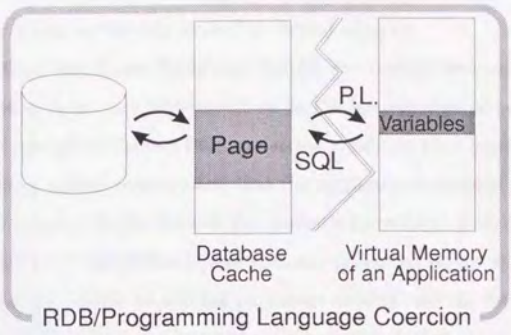


Figure 1.3: Interaction between RDB and an Application

When objects outlive the program invocation which created them, there is no need for any action to be taken for recording information persistently. It frees programmers from the tedious and error prone task. Persistent objects are objects which persist after a program terminates and which can be retrieved later on by the same or another program without intervention of programmers.

Due to its nature, POS can serve as a building block for database systems as well as applications which need database functions. Since *P3L* in any form does not yet support queries nor automatic maintenance of index structures, it is better positioned as a persistent object system than an OODB.

1.3 *P3L*

P3L is a persistent object system developed by the author at the Institute of Industrial Science, University of Tokyo. It is a persistent object system which consists of persistence-

aware compiler, runtime library and storage managers. The design goals of *P3L* are to be portable, to be compatible with C++ language, to reduce the amount of modification for having a normal C++ code to deal with persistent objects and to be efficient in terms of accessing objects which are already cached in virtual memory.

P3L[4] as well as the *Texas Persistent Store*[5] are notable among Persistent C++ systems in that they incur very little overhead by the introduction of persistent objects. That means it is one of the fastest POS under the condition that applications working set can entirely fit in virtual memory and that the applications perform intensive manipulation on those objects. To the best of the author's knowledge, *P3L*, *Texas Persistent Store*, *ObjectStore*[6] and *QuickStore*[7] are the only persistent object systems which use direct virtual memory pointer to address persistent objects. All the four have the same design goals in common. And the latter three take quite similar approach. That yields two different approaches to handling persistent objects which aim at improved performance at the said condition. All these employ rather eager pointer swizzling strategies. However, due to the difference in the eagerness and due to the use of memory protection hardware by the latter three, *P3L* and the others exhibit very different behavior in certain situations.

The contributions of this thesis are the proposal of the efficient pointer swizzling method and the verification that the software based scheme can well compete with the fast method which utilizes MMU. Another contribution is the conceptual clarification regarding pointer swizzling and object faulting strategies which lead to clear taxonomy of available strategies.

1.4 Organization of the Thesis

This chapter provided introduction to persistent object systems and persistent C++ system *P3L*.

Chapter 2 (Design Issues in Building Persistent Object Systems) enumerates necessary

building blocks in constructing persistent object systems and overviews their implementation techniques. Included among them are *object faulting* and *pointer swizzling*. Existing persistent object systems are surveyed and the design decisions on the issues are considered.

Chapter 3 (Object Faulting and Pointer Swizzling Strategies) first describes the concepts and then suggests that two notions, *reservation* and *residency* be considered when classifying existing pointer swizzling strategies in a coherent view. The proposed strategy is described both by placing it relative to other existing strategies and by following execution steps of an application. The proposed strategy is termed *pointer swizzling upon discovery*, in which references embedded within an object are partially swizzled at fault time and are fully swizzled later when they are detected by the program. It is shown that the proposed method lies between *pointer swizzling at fault time* and *pointer swizzling upon dereference* in terms of its eagerness for address reservation.

Chapter 4 (*P3L* Compiler Implementation) describes the *P3L* compiler which automatically inserts machine instructions for the reservation check. The compiler is based on Gnu C++ Compiler. The parts of the compiler for generating syntax tree and intermediate output code called *RTL* are modified. Further, the mechanism of generating *ancillary functions* and *runtime type information*, which become necessary to support the runtime system for implementing the pointer swizzling strategy, are examined.

Chapter 5 (Runtime Library and Storage Managers) looks at the exception handling system, the data structure used for maintaining mapping between object identifier and virtual memory address, and management of root objects and type identifier. Also the interface from *P3L* runtime to various storage managers, through which all storage managers are plugged to *P3L*, is presented with descriptions of the storage managers.

Chapter 6 (Performance Evaluation), evaluates the performance of *P3L* using OO1 benchmark which models behavior of design applications. It is shown that the proposed strategy for object faulting with pointer swizzling does quite well in the application scenario which the strategy is aimed at. Also, it is shown that the strategy is not panacea

that it performs less optimally under other conditions. The evaluation is augmented by another benchmark which picks up pathological examples in order to assess worst case overhead caused by the inclusion of reservation checking instructions. An application level benchmark is also included.

Chapter 7 (Conclusions) summarizes the contributions of this thesis, lays plan for future research and considers prospective application of the proposed strategy to other areas of computer science and engineering such as distributed computing and garbage collection.

Chapter 2

Design Issues in Building Persistent Object Systems

Sharing of large object base is becoming increasingly important to make complex information, in the form of objects, readily available to programmers. Should no support be provided for accessing such shared repository, programmers will end up writing and debugging daunting amount of code just to retrieve and store those objects. And that code tends to be reinvented every time one need to access new data of different types thus discouraging sharing of object between applications.

So it is desirable for a language to support an abstraction called *object persistence*. In a persistent programming language, persistent objects are kept in database or in operating system files but automatically retrieved when they are needed without programmer's intervention. Thus, to a programmer writing in a persistent programming language, objects seem to exist in a single level store.

Adding persistence to a language requires changes to the language itself, a compiler, runtime, and/or underlining hardware. The changes must be carefully balanced so that neither programmer's convenience nor computation efficiency is sacrificed.

What is essential for creating large shared object repository is a universal address space meaningful beyond the extent of virtual memory space in which they were created.

A persistent reference, introduced in section 2.1 must be used in order to refer persistent objects kept in database.

There are several design decisions to be made for building a persistent object system. This chapter enumerates them and describes the associated implementation techniques.

2.1 Representation of Persistent References

An important issue in designing a persistent object system is the choice of representation of reference and the mechanism through which the reference is resolved.

A reference in a language system implementation, which has not been designed ground up with persistence in mind, tends to be short and often it is a direct virtual memory pointer. That is because implementors make assumptions that the size of physical storage [1] is sufficiently large or that programmers manage the storage by themselves. As a consequence, resolving a reference is performed in less than a few machine instructions.

The assumptions do not hold when persistent objects are to be dealt with. The use of longer references is inevitable, longer in terms of the number of bits in a reference. The long reference needs to be independent of virtual memory locations, because virtual memory address assigned to an object is meaningful only in the application invocation where the particular object was created. Throughout the thesis *persistent reference* refers to this long reference which potentially points to a persistent object. A persistent reference may point to a volatile object as well and whether the reference itself is persistent or not is not assumed. Some implementations employ persistent reference whose structure is same as a virtual memory pointer. On the other hand, *volatile reference* means a reference to an object which is guaranteed not to be persistent, i.e. to exist on the physical storage.

Devising a design which meshes well a persistent reference and a volatile reference that a language implementation assumes is the challenge that persistent object system

[1] Physical storage is a storage area assigned to a virtual memory space. A physical storage usually comprises of RAM and backing store.

implementors must face. There are various strategies which can be taken depending on objectives of and constraints imposed on the design. Examples of questions on objectives and constraints are:

- How transparent the handling of persistent objects need to be to the programmers?
- What kind of performance needs to be maximized?
- Is building a special compiler feasible or the system should work with off-the-shelf compilers?
- Is the resulting system required to be portable across operating systems and CPU architectures?

2.2 Object Faulting

It should be noted that the persistent space covered by a persistent reference can be very large but an application may make use of only fractions of the space or may access only fractions of objects available in database.

Object faulting is an event triggered by an installed mechanism which directs transfer of an object from secondary store to physical storage. Object faulting happens as a result of a use of, or an anticipation of a use of, a reference to a persistent object. The mechanism of object faulting is one of the key factors which determine the characteristics of persistent object systems. In the next chapter we propose new notions of *reservation* and *residency* which hopefully help to build better understanding and classification of various object faulting mechanisms, especially those which use *pointer swizzling*.

Before transferring the contents of an object, a region in the virtual address space must be assigned to the object. The address assignment and the contents transfer may be carried out at the same time or contents transfer may be postponed until the first attempt to access the object contents is made by the program.

2.3 Pointer Swizzling: To swizzle or not to swizzle

One obvious approach to handling persistent object is to use a volatile reference which is long enough to account for large database size so that persistent reference can be structurally equal to a volatile reference. In a language system for a 64 bit processor, it is possible that a reference is 64 bit long. In that case, the size of the space covered by the reference is practically big enough to hold infinite number of objects. However, such an arrangement is not necessarily the best option considering that going to 64 bit makes either page-table-walk slower or the page size bigger. Also it can be disputed that silicon areas spent for handling 64 bit address can be better dedicated to extra cache or registers[8]. Another problem is that the use of 64 bit address increases the working set size of an application because more storage is needed to store a reference. It likely becomes a critical factor to performance when objects being manipulated are densely populated with references and the physical storage is rather limited. Another argument against simply using 64 bit address is that 64 bit address space is still not big enough. In order to allocate an identifier which is universally unique or to place redundant information in a reference so that use of a dangling pointer can be detected, pointers must be longer than 64 bit. For example, the Exodus storage manager use 96 bit OID.

Albeit the potential pitfalls, the use of flat 64 bit address space and a language system which assumes 64 bit reference can make implementation of a persistent object system simpler. Because the language system can handle persistent references in the same way as they handle volatile references, no conversions of references need to be performed. However, an arrangement must be made to detect access to non-resident objects and updates to persistent object need to be tracked for recovery and concurrency control purposes.

On 32 bit processors, a long persistent reference must be supported by a library or a compiler. In either way, a persistent reference is resolved in a different way than a 32 bit native reference is.

A library may define a special type(s) for persistent references and require the use of the type in dealings with persistent objects. An example of this approach is described in section 2.4.1. When it is possible to change the semantics of reference resolution through compiler modification or meta-level protocol, the notion of a persistent reference can be added to the language. Detailed explanation on this approach is given in section 2.4.2.

In contrast to the '*persistent reference being long all the way*' approach outlined above, there is an alternative way to deal with persistent references, which involves *pointer swizzling*. For defining the word *pointer swizzling*, an excerpt of an illuminating posting in a usenet news group 'comp.databases.object' is attached below, unmodified except for spelling changes.

> From: francis@cs.adelaide.edu.au (Francis Vaughan)
> Newsgroups: comp.databases.object
> Subject: Re: Swizzling
> Date: 23 Apr 1994 03:39:07 GMT
> Organization: Adelaide University, Computer Science
> Distribution: world
> Reply-To: francis@cs.adelaide.edu.au
> NNTP-Posting-Host: balthazar.cs.adelaide.edu.au
>
> Essentially swizzling is an implementation optimisation to help with the costs of translation. Objects logically refer to one another by some pointer, however often the representation of the pointer needs to be different depending upon where the object resides currently.
>
> [...]
>
> ** It is the overwriting of the OID with the VM address of the object that is known as swizzling. ** It probably came into use in the early '80s, but there is no clear evidence about who invented it. It probably was simultaneously invented by

number of groups. An idea whose time had come. Worse arguments abound as to the origin of the name, but they have already been canvassed without resolution on this newsgroup.

One remark I would like to make is that the result of swizzling does not necessarily have to be a virtual memory address. To summarize, pointer swizzling may be defined as an overwriting of a persistent reference in one format with a corresponding persistent reference in another format. The former can be longer than the latter or they may share the same structure but then they would have different semantics.

The benefits of employing pointer swizzling are as follows:

- Pointer resolution becomes more efficient.
- Because pointers can become shorter, working set may become smaller.
- Because pointers can become shorter, copying pointers may become faster.
- Because pointers can become shorter, CPU registers may be better utilized.

The primary purpose of pointer swizzling is to reduce the mean cost of locating the target object in virtual memory given its OID[2]. Thus all pointer swizzling methods are expected to have the first benefit. The efficiency includes reduced number of instructions and less cache/TLB flushes which is a byproduct of the compacted representation of references. It should also be noted that by converting a long reference to a short reference it becomes possible to capitalize, to certain degree, on a conventional language system when building a persistent object system.

The cost incurred by the introduction of pointer swizzling are:

- Unnecessary translation from OID to LID may be performed.
- The need to swizzle and later deswizzle a reference adds to the overhead which may not be amortized.

[2]The definitions of OID and LID are given in section 3.5.

- More read operations from the persistent store may be issued than is necessary.
- More virtual memory space may be consumed than is necessary.
- The need to detect whether translation must be performed or not adds to the overhead.

Each of these costs trades with others. For example, if all existing references are converted to swizzled form at a program startup, the program does not have to continuously run the check to see whether a translation must be performed, although all the other costs increase.

When dealing with huge persistent object base, the eager conversion described above is not a feasible option. It was assumed during the design and development of *P3L* that applications exhibit locality in the access to objects and that entire working set can fit in virtual memory. For these reasons, we do not give much consideration to the first and the second costs. Our primary focus is on reducing the last cost and in this thesis I propose a new pointer swizzling scheme, which provides unique trade off between the last three costs.

2.4 Making Pointers Smarter

Almost all C/C++ [12] compilers assume that a reference is represented by a direct virtual memory address, where the referent resides, for maximizing efficiency of program execution. However, when persistent objects are to be handled, it may become necessary to modify the way a reference is represented and resolved[3].

Altering the semantics of reference resolution may be likened to *making pointers smarter* because the new way of resolution performs more elaborated operation than just fetching an address into a register and accessing another memory location using the

[3]It is possible to avoid alteration of the underlining language system even if pointer swizzling is employed, as exemplified by the Texas Persistent Store.

value in the register. There are two major ways of making a pointer smarter in C++ context, *Smart Pointer* and compiler alteration. Other languages may offer other possibilities. For example, in Smalltalk, runtime system may be modified to place a hook on a failed method lookup for extending the semantics of reference resolution.

2.4.1 Smart Pointer

Smart pointer is a name, in C++ parlance, given to a class which embeds a reference. A smart pointer class [9] [10] [11] is defined in such a way that an instance of the class behaves like a native pointer except that more complex resolution may be performed under the hood upon dereference.

C++ allows overloading of operators. That is, a programmer can define the meaning of operators being applied to a user defined type. Most significant operators in the implementation of a persistent object system are the dereference operators (*, $\Rightarrow$) and the array subscript operator ([ ]). Being able to tailor the behavior of these operators applied to a smart pointer means that a programmer can implement arbitrary semantics for a reference resolution. Thus smart pointer approach may be used to implement an elaborate resolution in a 'long reference all the way' approach or to implement required conversion and checking in a 'pointer swizzling' approach. An example of a smart pointer class definition and its use is given in Figure 2.1 at the end of this chapter.

One of the benefits of using smart pointer is that it can be implemented at library level thus no compiler modification is required. That means a persistent object system build with smart pointer is likely more portable than one which depends on a particular compiler. Another benefit is that only definitions of a reference need to be changed, not their use. In the example of Figure 2.1, the definition of 'ip' is different from a pointer definition.

$$\text{Employee * ip;} \iff \text{Ref<Employee> ip;}$$

However the use of the reference, ip$\rightarrow$name, is identical in both cases. This would help

when converting programs written without consideration to persistence to those which can handle persistent objects. However, in other cases more work is required to convert existing program than changing pointer declarations. For example, a pointer may be tested to see whether it is NULL or not:

if(p) <statement>
while(p) <statement>

For the case above, comparison operator must be defined. Also, the use of smart pointer poses technical problems regarding type conversions such as :

derived * $\Rightarrow$ **base** *
const T * $\Leftrightarrow$ T *

Please refer to [11] for details.

2.4.2 Compiler Alteration

Another way to make a pointer smarter is to modify a compiler so that semantics of pointer dereference operation is changed. A dereference of a native pointer will be compiled into code sequence like the one below for a SPARC processor:

```
sethi %hi(_p),%g2
ld [%g2+%lo(_p)],%g2
ld [%g2],%g2
```

Of course this sequence can handle only virtual memory pointers. The sequence fetches only one machine word and more words must be fetched to handle a long persistent reference. When a persistent reference on physical storage is longer than virtual memory pointer, not only dereference operations but also copy operations require a different and

longer instruction sequence. Also algorithm for laying out structure members must account for the change in the size of the reference.

E[21] alters Gnu C++ Compiler in order to handle persistent references. An example instruction sequence for a dereference operation looks like this:

```
sethi %hi(_p),%o0
or %o0,%lo(_p),%o0
call __EMapAddr_NOSW,0
mov 0,%o1
ld [%o0],%o0
```

The apparent number of instructions is only about twice of those for a raw pointer. However, the number of instructions actually executed is much more due to the call into EPVM (E Persistent Virtual Machine). Copying a pointer, especially pushing a pointer onto the stack requires a long instruction sequence. Thus modifying compiler to take this approach does not equate with being more efficient. With that said, a compiler can assume more about dereferencing operations so that it has more opportunities for optimization. For example, multiple dereference of the same pointer in one *visit*[4] can be executed with only one address translation even if swizzling is not performed. The optimization is similar to CSE (Common Subexpression Elimination) which folds multiple calculations of the same expressions into one. When the library approach is taken, such optimization requires expensive inter-procedural optimization, if ever possible, because the compiler cannot make any assumption of what a library call for dereferencing does.

In *P3L*, Gnu C++ Compiler is slightly modified so that the semantics of copying of pointer is changed to account for persistent object access. Unlike *E*, a persistent reference in *P3L* is structurally identical to a raw virtual memory pointer. Also no distinction

[4] A *visit* is a set of access, which can be statically determined to be against a single object, executed in a basic block.

between a persistent reference and a volatile reference is made in *P3L*. That is, there is only one reference type, whose structure is equal to a raw pointer, and all pointers are treated as a persistent reference except when one is guaranteed to point to a volatile object through compile time analysis. Further explanation on the treatment of references in *P3L* is given in 3.5.

2.5 Update Detection

Persistent object systems require the detection of updates to an object for:

1. Transferring only updated objects back to the persistent store.

2. Concurrency control.

Because the number of objects accessed in a session of an application can be enormous, it is desirable to commit only updated objects back to the persistent store in order to reduce I/O bandwidth. When objects in database are accessed from applications concurrently, appropriate locks must be obtained for objects accessed or updated so that transactions issued concurrently from users can be guaranteed to interleave in a serializable order. Distinguishing read locks and write locks allows more interleaving of transactions, i.e. increase concurrency. Thus capability of deciding whether obtaining a read lock is enough or a write lock must be acquired is desirable.

There are three well-known ways of detecting updates. One is to use hardware memory protection via MMU. Decent processors are capable of prohibiting a write access to a specified region of memory and delivering a notification when an access violation is detected. Many operating systems provide mechanisms for fielding the notification up to an application program. With these facilities provided, applications can track updates entirely through hardware instead of issuing any extra instructions for that purpose. This means that updates made within a pre-compiled library which have been compiled by a non-persistent compiler can be tracked. The *Texas Persistent Store* adopts this method.

Another way is to again modify a compiler so that the generated instructions keep track of updates. A generated sequence for the expression 'x=y', where x and y are integers placed in static storage, would look like the one below.

```
sethi %hi(_x),%o0
sethi %hi(_y),%o1
ld [%o1+%lo(_y)],%o1
st %o1,[%o0+%lo(_x)]
or %o0,%lo(_x),%o0
call _track_update,0
mov 4,%o1
```

After copying a word from the location of 'y' to 'x', a call *track_update(&x, sizeof(x))* is made with the address and the size of the updated object. The last instruction is in the delay slot for the call instruction. Thus it executes before the branch to track_update function is made, passing the size 4 to the *_track_update()* function.

When applying this method to a C++ compiler, a subtle problem arises due to pointer arithmetics allowed in C++ and the representation of raw pointer geared towards efficiency of the dereference operation. In Figure 2.2, the function *clear()* clears the second element of integer array (*array[1]*) and integer member of an instance of class S (*m_i*). In passing arguments, *array+1* and *&sp→m_i*, a conventional compiler will calculate the address of the integer to be cleared. Thus, the sub-function *clear_int()* has no way to know, from the argument pointer alone, what is the start address of the object. That means either the presentation of a pointer must be changed, for example making a pointer to consist object identifier and an offset, a runtime data structure must be maintained with which to lookup the start address, or updates must be performed at different granularity. (An example of the second one is described in section 5.3.1). The first method requires significant changes to the entire part of the compiler and sacrifice efficiency on

the use of pointers. The second makes update operations rather expensive. The third place restriction on memory allocation and relocation. Locating the start address is a rather expensive operation which cannot be avoided even if memory access protection is utilized. The benefit of using MMU is that it reduces the overhead to once at the first update, and after marking objects in an accessed page, the write protection on the page can be lifted.

Another option for tracking updates is to leave the responsibility to the programmer by providing an API to notify an update with. Because a forgotten call to the API will likely result in a lost update, this method risk proper functioning of the system. However, because this method provides best portability over operating system platforms and compilers, many commercial OODBs take this approach. ODMG93[15] standard also adopts this. An example is provide below:

```
void f( Part * p, int age, const char * nm )
{
    p→m_age = age;
    strcpy( &p→name[0], nm );
    p→mark_dirty();
}
```

The call to the member function *mark_dirty()* marks the updates performed to the object in the function. This assumes that the member function is defined for the class *Part*. Providing this member function is usually achieved through inheriting from a *persistent base class*[5], a class which implements necessary functions for persistence related operations.

[5]The class to inherit from when declaring a class so that instances of the class can become persistent. ODMG[15] standard define the name of this class to be *Pobject*.

2.6 Runtime Type Information

A persistent object system requires runtime type information for various reasons such as locating references embedded within a particular object and determining the size of an object.

In the previous section, the *mark_dirty()* member function was introduced. The method may be implemented in the persistent base class. However, the size of an instance of a class derived from the base class cannot be determined in the member function defined for the base class. One way to solve the problem is to define a virtual member function *size_t size()* in the base class and to require each newly defined subclass to override this member function so that it returns the proper size. However, this would add to bookkeeping which a programmer must take care of when a new class is defined. Putting a wrong *size()* member function compromises a system validity in obfuscated way.

The other solution is to make use of runtime type information (RTTI). RTTI is description on types defined in a program. The description includes the size of an instance of each type, inheritance DAG of classes, the name and the type of each members of a class (type), etc.

The implementation of *mark_dirty()* can look into this information in order to decide the size of an instance for which the member function is called. The actual type of the instance (type identity) can be determined from the value of the *this* pointer passed to the *size()* function. Knowing the type identity allows to lookup the relevant size information in the runtime type information database. The benefit of this approach is that the *size()* function only needs to be defined once in the persistent base class thus this scheme is less intrusive to programmer. Another frequent use of RTTI in a persistent object system is for locating references within an object. When performing pointer swizzling within an object, this information is essential to decide which parts of the object need to be investigated.

RTTI is also useful for pretty printing or generating graphical presentation of the

contents of an object. For printing the value of an object, a programmer needs to define a member function such as the one shown below in the absence of runtime type information.

```
ostream& Person::operator << (ostream& o)
{
    o << "name:" << m_name << ", age:" << m_age << endl;
    return o;
}
```

This must be declared and defined for each user defined type. And it must be updated when a change is made to the class. By implementing the member function in the persistent base class and having the implementation look into RTTI, operator << becomes available to all persistent classes without any intervention of a programmer except deriving user defined classes from the base class. How to generate RTTI is an important issue and is discussed in section 4.3.4.

2.7 Orthogonal Persistence

Orthogonal Persistence is a characteristic of a persistent object system which can make any object persistent[16]. It is claimed that this is a desired characteristic because it makes programming with the system more comprehensive. Of course, there are disputes. A counter argument to orthogonal persistence is that persistence is a special behavior of an object thus it is better to require to inherit the behavior explicitly. A rebuttal to this counter argument is that persistence is meta-behavior, thus it should be transparent.

Use of a persistent base class ease an implementation and improves the portability of the implementation. However, if a persistent base class is utilized, a programmer needs to derive persistent classes from the persistent base class. This would compromise orthogonal persistence[17]. Introducing a special reference type for handling persistent

reference also compromise orthogonal persistence because an object which is pointed to by a volatile reference cannot be treated as a persistent object. The problems introduced by persistence un-orthogonality is described in [18].

2.8 What to make Persistent and what not

In designing a persistent programming language, a decision must be made on what kind of objects are allowed to be persistent and what not.

The options may be categorized into two groups. One is those which requires to specify an object to be persistent to allow the object to be alive beyond the lifetime of the creating program. The other group is to have the system automatically decide which object to keep alive. The detection of the live objects are performed using the same principle which garbage collectors follow. In a system of the latter group, a set of objects are specified to be *persistent roots* by being given with a name, for example. The system traverses the references embedded in objects transitively, starting at the persistent roots. Since a persistent root must be first accessed in order to access any object in the persistent store, not being reachable from any of the persistent roots means there is no possibility that the object will be accessed in the future. On the other hand, being reachable from one of the persistent roots at the time of transaction commit, i.e. when objects are written back to the persistent store, means the object must be written to the persistent store. The object is automatically elected since otherwise the same data structure cannot be recreated later when they are loaded again. The principle is called *persistence by reachability.*

At the first glance, the latter approach seems superior. However, there are problems in the approach especially if interaction with a language implementation is considered.

First, persistence by reachability assumes that references can be identified precisely [6]. This assumption does not hold in most off-the-shelf C/C++ compiler implementations

[6] A conservative estimation of reference as is found in [19] may be applied. However, retaining unnecessary storage may be more severe problem with regard to the persistent store where the exhaustion of

due to the existence of an anonymous union. When a union contains both a pointer member and a scalar value such as int, there is no way for a runtime system to tell whether the value stored in the union is a pointer or a scalar value.

Secondly, if the persistence by reachability rule is strictly followed, the enormous amount of garbage may be committed into the persistent store when there is a corrupted pointer in one of the persistent objects. The C/C++ language provides the programmer much freedom in the manipulation of pointers, which is convenient but potentially introduces the risk of corruption of data structure. Since a corruption is definitely a bug in the application, taking such programs into consideration may seem unjustifiable. But in practice, storage corruption together with storage leak seems to be the number one bugs found in C/C++ programs.

Thirdly, some persistent object systems maintain a set of objects called *extent*. Extent is a set of instances of a particular class and often becomes the target of a query. In this case, it means there is an access path to an object which does not go through a persistent root. Thus the idea of persistence by reachability does not apply well.

Lastly, judging persistence by reachability is not the right thing to do in some cases. For example, a result of computation by an application may be kept while the application is running so that the application does not need to perform the calculation again. In order to avoid permanently devoting storage space for the cached object, it may be more desirable just to discard the object and reset the references pointing to the object to the NULL value when committing it to the persistent store. Such a reference is termed a *weak reference*. The notion of the weak reference can be incorporated into the rule. However, a question on how to specify whether a reference is weak or not still remains. Systems with explicit persistence specification can handle this issue with less problems. References to an object which have not been specified to be persistent may be reset to NULL.

As for designs with explicit specifications of persistent objects, there are a few specifi-

storage (i.e. disk full) is a way of life.

cation methods. In some persistent object systems, all instances of classes deriving from the persistent base class are automatically made persistent. In others, an object is specified to be persistent at creation time. In *P3L*, objects are promoted to persistent objects during runtime when it becomes clear that they need to be persistent.

2.9 Examples of Persistent Object Systems

In this section, several persistent object systems are surveyed to review their approaches to design issues raised in the previous sections. Those with pointer swizzling will be further mentioned in section 3.5.

2.9.1 PS-Algol

The work on PS-Algol[13][14] pioneered the research on persistent object systems. In PS-Algol, pointer swizzling was performed between OID and LID, which are called PID (Persistent Identifier) and LON (Logical Object Number) respectively. PID and LON are of the same structure and they are distinguished from each other by the most significant bit. In some implementations of PS-Algol, LON was a pointer to an entry of a table called PIDLAM (PID to Local Address Map) and in others LON degenerated into a raw pointer.

To support pointer swizzling, objects are tagged by a header word so that they become self-describing or a pointer to a type description was inserted into the header so that the contents of the object can be interpreted by the type description. The compiler emits type description and it is stored in database as well as in applications. Thus runtime systems can locate pointers and also perform type-checks on the use of persistent objects.

2.9.2 Texas Persistent Store

The Texas Persistent Store swizzles[5][26] a long persistent reference to a raw pointer when it migrates into virtual memory storage. That means any reference in physical storage is a raw pointer as far as a compiler is concerned. However, since address space covered by a long persistent reference can be much larger than available virtual memory space, objects must be incrementally migrated from persistent space to virtual memory space as they are accessed. In order to achieve the incremental faulting, the Texas Persistent Store utilizes access protection through memory management hardware. The details are given in the next chapter. It utilizes memory management unit also for detecting updates. RTTI is extracted from debugging information attached to object files (.o), which are produced through compilation with -g flag turned on.

Because of these reasons, the Texas Persistent Store requires no modification to a compiler. Thus Texas Persistent Store is compatible with libraries which have been compiled without persistence in mind. Currently the runtime system is implemented on SunOS, Linux and OS/2 for the use with Gnu C++ compiler as well as CSet++ compiler on OS/2. Because a format of debugging information differs from a compiler to another, a different compiler needs explicit support.

2.9.3 E

E is a persistent programming language developed in the Exodus project [20][21][22][23]. It is an extended C++ with persistence, iterator and other database oriented features. It originally used long persistent references without any swizzling but newer implementations added pointer swizzling.

E exposes two types of reference: long persistent references and raw pointers. The type of pointer is determined by the type of the target object. For example,

```
dbint * p;
```

declares a persistent reference to an integer; It is allowed to assign the former to the latter. That means, a persistent reference is allowed to point to a volatile object. However, assignment in the reverse direction is not allowed. The type system consists of two mirrored universe (non-db and db). That is, all primitive types and type constructors such as 'struct' have a corresponding db-type such as 'dbint', 'dbchar', 'dbstruct'. *O++*[24] also introduces two kind of types by adding *persistent* type qualifier.

In the E implementations, both the translation of OID to a raw pointer and detection of updates are taken care of by the instructions emitted by the compiler. One implementation modifies *cfront* C++ translator and the others base on Gnu C++ compiler. RTTI is not required for swizzling pointers because references are swizzled as they are found by compiler-issued instructions, not in the runtime system, and size information is managed by the memory manager. The compiler utilizes its knowledge about the locations of references within objects when it generates swizzling instructions. As for deswizzling of swizzled pointers which is performed when the containing object is being passivated to the persistent store, the runtime system keeps bitmap of swizzled pointers. For these reasons E implementations can do without RTTI prepared separately.

2.9.4 LOOM

LOOM[25] was developed as an extension of storage system for Smalltalk-80 implementation which was using 16 bit *oops*[7]. An oop points to an entry in the object table which in turn points to the body of the corresponding object. Due to the shortness of oop, the number of objects manageable was very limited. LOOM was needed in order to increase the number of objects manageable as well as to expand the size of the heap.

A persistent reference is 32 bit long while a regular reference which is handled in the Smalltalk virtual machine is kept to 16 bit. Byte code compiler was not modified but the virtual machine was slightly modified in order to detect the first message send to a

[7]Object Oriented Pointer

short oop whose target is a special object called *leaf*. Another modification to the virtual machine was applied so that the virtual machine detects a special short pointer value called *lambda*. *Lambda* is a nil pointer, which stands for a persistent reference which has not yet been converted to valid LID. More detailed explanation is given in 3.4.3.

Smalltalk has a built-in capability to manage and handle RTTI by making classes themselves an object, an instance of class Class. Detecting updates poses no problem because all updates are performed via the virtual machine in the Smalltalk implementation and dirty bit was kept in each object table entry, which is easily accessible given an short Oop.

2.9.5 ODMG93 compliant Systems

ODMG (Object Database Management Group) is a consortium mostly consisting of vendors of object database. The group has come up with the standard to which applications may conform so that applications can be run under different object databases from different vendors[15]. Due to its nature as such, the standard is rather conservative, that is, useful or fanciful features of a particular vendor are not incorporated in the standard.

The standard specifies a persistent root class called *Pobject*, which serves as a persistent root class. *Mark_modified()* function is a member function of the class and is used to tell the runtime system about updates applied to a persistent object. The pointer resolution is performed through a templated smart pointer class Ref<*T*>. Runtime type information is generated from type definitions separated in a definition file written in ODL (Object Definition Language).

```
//                    A Smart Pointer Definition
template < class T >
class Ref {
    long long m_OID;
  public:
    Ref( const char * obj_name ) : m_OID( lookup_oid(obj_name) ) { }
    ...
    T * operator →() { return (T*) translate_oid( m_OID ); }
    T& operator *() { return *(T*) translate_oid( m_OID ); }
    T& operator [](int idx) { return ((T*) translate_oid( m_OID ))[idx]; }
};

//          Use of a Smart Pointer in a client program.
void f()
{
    BeginTransaction();
    Ref<Employee> ip("UTOKYO-PRESIDENT");
    cout ≪ ip→name ≪ endl;
    EndTranscation();
}
```

Figure 2.1: Definition of a Smart Pointer and A Use

```
void clear_int( int * p )
{
    * p = 0;
}

struct S {
    S * next;
    int m_i;
};

void clear(int * array, S * sp )
{
    clear_int( array + 1 );                    // &array[1]
    clear_int( &sp→m_i );
}
```

Figure 2.2: A problem with updating through a derived pointer

Chapter 3

Object Faulting and Pointer Swizzling Strategies

In this section, after several terms are defined with regard to object faulting and pointer swizzling, two object faulting strategies which do not employ any pointer swizzling strategy are reviewed. Then the benefits of pointer swizzling and dimensions which constitute the design space of pointer swizzling strategies are examined. In order to clarify the positioning of the pointer swizzling strategy employed by *P3L*, alternative strategies are classified based on the notions of *reservation* and *residency*. Finally, the swizzling and object faulting strategy developed for *P3L* is described along with the programming interface and the way the reservation check is performed.

3.1 Term Definitions

3.1.1 Physical Storage

To store an object in a virtual memory page, either RAM or backing store must be allocated for the page. Because these two are indistinguishable from the point of view of an application program, the combination of these storages is referred to by the term

physical storage in a sense that they are physical medium for storing bits and bytes. Note that both virtual memory space and physical storage, especially the latter, are scarce resources.

3.1.2 Persistent Reference

A persistent reference is a reference in the physical storage which refers to a potentially persistent object. More description is given in section 2.1.

3.1.3 Volatile Reference and LID

Many persistent object systems introduce a reference type with which persistent objects are referred to. In that case, a type of reference with which a persistent object is referred to differs from a type of reference with which a volatile object is referred to. Volatile objects are objects which do not outlive the program which created them. Thus volatile objects are always guaranteed to be present in physical storage. A volatile reference is a pointer known to point to a volatile object. Thus an identifier contained in a volatile object is a valid identifier which relates to a valid virtual memory address. Thus dereferencing a volatile reference is much more efficient than dereferencing a persistent pointer. No address translations and no checks are needed. Such an identifier is referred to with the term *LID* (local identifier). In C and C++ systems, LID is often a raw pointer, i.e. a direct virtual memory pointer. In other languages, it may take another form. For example, a LID may be a pointer to an object table entry, via which the virtual memory address of the target is obtained through indirection.

3.1.4 Reservation and Residency

LIDs are only valid in the process invocation where they were generated. If the structure of LID is different from the structure of an OID, an OID must be translated to a LID at

some point during a program execution for the CPU to access the contents of the target object.

Assigning a LID to a persistent object does not require that the object be made resident in physical storage. This means that transferring the object's contents into physical storage may be delayed later than the time a LID is assigned, although the loading needs to be completed before the contents can be accessed[1].

We propose two notions in order to clarify the two notions described above. By *reservation* we mean allocation of a LID for an object which is identified by a given OID. And by *residency*[2] we mean the existence of the contents of the object in the physical storage. A LID is assigned to an object as a result of a failed *reservation check* or reservation for an object may be performed when the first reference to the object migrates into the physical storage. The contents of an object is transferred to the physical storage as a result of a failed *residency check* or an object may be made resident when a LID is reserved for it. These implementations of the checks are explained in the later part of this chapter.

3.2 Faulting Strategies without Pointer Swizzling

In this section, faulting strategies without *pointer swizzling* is described. The first one is *persistent virtual memory* (PVM) which utilizes a single type of reference which serves for both persistent and volatile reference. Thus no reservation is performed at all.

[1]This is simply because it is physically impossible to access the content without bringing it into physical storage.

[2]The term 'residency checking' was first coined in [28]. Our definition gives more stringent meaning to 'residency'.

3.2.1 Persistent Virtual Memory

The simplest way of handling an object fault is to take advantage of memory management hardware for residency checking and not to introduce a local address space for accessing persistent objects. In this way reservation checks do not have to be performed because reservation does not make any sense.

The problems with a PVM implemented completely in hardware arise from the limitation in the size of the address space available and fragmentation within the space. Fragmentation in this context means the dispersion of spaces allocated to objects which an application touches. When sharing a single large addressing space among multiple applications, it may be impossible to cluster relevant objects in contiguous regions of the address space for a particular application. This leads to the problem of wasted memory for the page table with conventional page table structure. Also to guarantee a space big enough to handle vast amount of data, extension of address bits in CPU may be required. Extending address space means the number of steps in a page table walk increases or a page size becomes larger. And as discussed in section 2.3, 64 bit address space seems not big enough.

The problems can be obviated in two ways. One is to come up with fanciful hardware which avoids the said problems, e.g. long address bits, object memory, an inverted page table, etc[29] [30]. Another is to keep long OIDs but to assign a LID to an object so that the object can be accessed by translating the OID to the LID. The next sub-section introduces such a scheme.

3.2.2 Soft-Pointer Scheme

In this scheme, all OIDs are translated to a LID before being used to access the contents of the target object. The failure of this translation means reservation has not been made for the object. Residency checking may be performed either by hardware or software or eliminated by forcing residency when a LID is reserved for an object. However, forcing

residency earlier than the time of access provides little advantages since address translation must still be performed every time an access is made.

A soft-pointer is a long persistent reference which contains an OID in the same format as it is in the persistent store. When a compiler detects a use of a soft-pointer it emits a call instruction to a runtime library routine which probe the *ROT* (Resident Object Table) to translate the OID to the corresponding LID. ROT maintains the mapping between the OID of an object for which reservation is made and the LID assigned to it. A ROT is often organized as a hash table since spatial continuity of OIDs of accessed object cannot always be assumed. Some systems choose to use an ordinal page table organization assuming good clustering of objects, for example Mneme[28].

ROT is probed in order to translate an OID contained in a soft-pointer to a LID. Reservation checking is carried out implicitly by the probing. Failing to find a relevant entry triggers a reservation. When an object is made resident being transferred from a persistent storage, each reference embedded in the object may be assigned with a LID so that reservation checking can be eliminated. However, it is usually not worth doing, because assuming the completion of the reservation checking in the translation does not make the translation noticeably slower and assigning LIDs eagerly does not reduce the number of translations. Rather it risks unnecessary reservation.

When a LID is assigned to an object as a result of a failed translation, an implementation can choose whether to make the target object resident at this point or to defer the loading until later when the contents of the object are actually accessed. When deferred, residency checking needs to be present to detect a need to transfer the contents of the referent into physical storage.

This deferring, too, is not so beneficial because the result of translation will soon likely be used to access the contents of the target object considering that the running program already requested the translation of an OID to a LID. However, when large objects are dealt with and the translation is performed per object instead of per page, there are cases where deferring the transfer turns out to be a great advantage due to the reduced number

of I/Os performed.

Performing pagewise translation makes implementing partial faulting of a large object easier. Because a page can be made resident at the time of LID assignment without loading all the other pages which comprise the target object. The price to be paid is that the translation of a derived pointer is always necessary[3]. A derived pointer is a pointer which is obtained through an integer arithmetic operation (add or subtract) on another pointer. That is, provided an OID o, corresponding LID l, and specified byte-offset off into the contents of the object, neither the LID nor the reservation status of the byte at $o[off]$ cannot be determined from those of $l[0]$ [4]. Contrarily, when objectwise reservations are carried out, the LID corresponding to $o[off]$ is simply given by $l[off]$, saving costly translations.

The other penalty imposed on the objectwise translation is quickly consumed LID space when object size is large and access to it is sparse. On the other hand, the pagewise translation is penalized because of its inability to pack many objects in a page when objects being dealt with are relatively small compared to the page size and they are sparsely accessed within containing pages.

Another tradeoff in the selection of translation granularity is the size of the translation table. When the mean size of objects is small, objectwise translation incurs higher storage overhead, because more mappings must be maintained. On the other hand, if the mean size is larger than a page, pagewise translation incurs higher storage overhead because the mapping must be maintained for each page. This tradeoff applies to all object faulting methods, including those with pointer swizzling, as well as to designs of a hardware memory management unit.

The considerations so far given in this section lead to four dimensions which collectively constitute the design space of persistent object faulting:

[3]A derived pointer is usually an interior pointer. Section 5.3.1.

[4]This is not true when an object is guaranteed to fit within a page completely.

- Residency checks: when and how.
- Reservation checks: when and how.
- Granularity of reservation (translation)
- Granularity of object transfer

The advantages in using the soft-pointer scheme are summarized as follows:

1. Large address space can be handled by a narrow address machine.
2. Objects can be discarded or written back to persistent store without finding and updating pointers which point to the object being evicted.
3. Objects can be easily relocated in virtual memory to remove fragmentation.

When an object is relocated within LID space, only the relevant entry in ROT has to be updated. The relocation removes fragmentation in free memory chunks. Note that relocation for removing page-external fragmentation is not necessary with pagewise translation because free blocks need not be contiguous. But that means page-internal fragmentation cannot be removed.

In the event of eviction, only the ROT entry corresponding to the evicted object needs to be dropped because the next attempt to translate the victim's OID will again pull the victim from the persistent store. Thus garbage collecting entries in ROT is not an issue[5] even when new object keeps migrating in and out of virtual memory.

The definition of soft-pointer schemes given above does not specify the mechanism through which OID to LID translation is enforced. Possible mechanisms have been discussed in section 2.4.

[5]See the next section for an explanation.

3.3 Pointer Swizzling

3.3.1 Benefits

Though the originator of the term is unknown, it is agreed that the technique of *pointer swizzling* was pioneered by the work on PS-Algol. Using a soft-pointer approach, an OID to LID translation must be performed every time an object is dereferenced by a pointer. This frequent translation can be reduced by remembering the result of a translation. The result may be remembered by overwriting a reference or by copying the result (i.e. the LID rather than the OID) to the destination. In the latter case, saving in the number of translation comes from the elimination of translation on the copied reference. This replacement of an OID in a reference with a LID is called *pointer swizzling.*

Because OID to LID translations are bypassed if swizzling is employed, the translation routine cannot bare whole responsibility of performing reservation checks unlike with soft-pointer schemes. Thus separate reservation checks must be implemented or the need for the checks must be removed.

Pointer swizzling becomes less effective in reducing the number of translations when pagewise translation is employed and the target object of a swizzled pointer is not guaranteed to fit within a page. This is because $LID + offset$ must be evaluated for different *offsets* in that case.

The primary advantage of pointer swizzling is the reduced number of translations. Another advantage of swizzling pointers is that it can reduce the size of a reference. Note that copying a reference is an operation performed very frequently, e.g. it occurs when pushing arguments onto the stack in preparation for a procedure call and when following a pointer chain. Reducing the size of pointer reduces the number of CPU cycles needed in such occasions as well as keeping pressure on register, TLB, cache and physical storage as low as possible.

In the case of C/C++, being able to reduce a long persistent reference to the size of

a raw pointer is also good for maintaining compatibility with compiled libraries. Instruction sequence emitted by a persistence-unaware compiler assumes all references be raw pointers. For these reasons, the use of a persistent reference which is longer than a raw pointer invites incompatibility with pre-compiled libraries.

For a persistent reference to be dereferenced, it needs to migrate onto physical storage first and then to a CPU register. Finally when a CPU register contains a virtual memory address, it can be dereferenced to fetch the target object through a machine instruction. An OID must be translated to a LID during the course. The reservation status of the target object must be checked at some point so that unswizzled references can be detected.

3.3.2 Chances for Reservation Checking

There are two well known chances for reservation check. They are *upon dereference* and *upon discovery*[6]. There is also another swizzling method which performs no reservation check. If each OID gets translated to a LID when the containing object is transferred into physical storage, there is no need to perform reservation checks afterwards. Because,then, every references fetched into a CPU register or into another memory location is guaranteed to contain a valid LID. This variety is called *pointer swizzling at fault time*[7]. However, the concept is applicable to objectwise LID assignment and contents transfer, provided access protection of object granularity is available, e.g. segmented memory architecture. Pointer swizzling strategies are described and classified in detail in section 3.4.

3.3.3 Other Dimensions

The timing of reservation checking is one dimension which constitute design space of pointer swizzling strategies. Other dimensions are:

[6]The term 'discovery' was coined in [22].

[7]The original proposal was made for pagewise LID assignment and contents transfer, thus the name *pointer swizzling at page fault time* was given to the scheme proposed in [26].

1. **When the first attempt to replace an OID is made.**

   It is perfectly legal to try to swizzle earlier than a reservation checking scheme mandates. Doing so may result in fewer OID translations being performed while it may result in unnecessary allocation of LID. Without a residency check mechanism implemented, the eager allocation of LID triggers I/O operations which may be wasted if the referent is never accessed.

2. **How physical storage and LIDs are reclaimed.**

   Since physical storage is a very limited resource, for applications which deal with large number of objects, it becomes necessary to recycle the storage. One way to reclaim the storage is to detach portion of the storage from the LID space to which they are attached.

   To detach a portion from physical storage, four steps are required:

   - A part of LID space (the virtual memory region) mapped onto this portion of physical storage must be set to such state that the next residency check on the region will cause an access fault. The residency check will bring the evicted objects onto the physical storage again. Until that happens, the storage becomes available to be attached to other virtual memory spaces.
   - LIDs in the portion of physical storage need to be swizzled back to OIDs.
   - The contents of the page must be transferred to persistent store.
   - A table must be updated to indicate that the physical storage is now free.

   The process depicted above is the opposite operation of the object transfer performed upon a failed residency checking.

   The other way is to reclaim LID space itself together with physical storage committed to the space. For this to work, all references on physical storage must be tracked so that objects being referred to can be kept intact without being evicted or refer-

ences to evicted objects can be deswizzled to an OID. The mechanisms available for this tracing are *reference counting*, *back pointer management* and *reference tracing*.

Reference counting is a memory management scheme where the number of references to each object, i.e. *reference count*, is maintained. When a new reference to an object is created, e.g. by copying another reference, the reference count for the object is incremented. The count is decremented when a reference to the object is destroyed. A reference count reaching zero means all references to the object cease to exist. The problem with reference counting is that it adds much to the cost of copying reference. Another problem is that objects which point to each other through cyclic reference cannot be detected to be dead although there is no chance that they are accessed. Overflow of the reference count must also be considered.

Through reference counting, LIDs which can be recycled are determined. However, locating references to a given object is impossible through reference counting. Thus one must give up eviction of objects to which there are references or device another way to keep track of pointers.

Instead of providing the capability of locating pointers, a back pointer list may be maintained. With this scheme, when a reference to an object is created, it is put into a list associated with the object. A dead object can be detected by an empty list. The problems are the storage overhead and high cost of maintaining the back pointer list. The performance further degrades because the use of this scheme precludes references being held in a register.

Reference tracing is used by garbage collectors, to which extensive research have been devoted[27]. It is known that tracing garbage collectors, especially copying collectors, are generally more efficient than reference counting. However when applied to usual C/C++ compiler, this scheme shares the same problem with reference counting that locating pointers (on stack) is difficult. Thus efficiently implementing forced object eviction to be performed with a conventional C/C++ compiler may

seem not easy. However, it is possible to identify references on the heap exactly by consulting RTTI. That means objects pointed to only by references in the heap can be forcibly evicted. This is similar to the action taken by the mostly copying garbage collector by Bartlett [42].

3. **Whether the method employs *narrowing* of OIDs.**
   If reservation is performed rather lazily, it becomes necessary to place an OID in a short reference in the physical storage. Thus a long OID may have to be represented by a short reference. Certain implementations ([25] [4]) introduce a special LID value for that purpose. The value is used to represent an OID to which has not been swizzled yet. The reduction of the length of persistent reference is termed *narrowing* in this paper and it is a variation of partial swizzling.

3.4 Survey of Pointer Swizzling Strategies

In section 2.9, several persistent object systems were introduced and many of them employ pointer swizzling in their implementations. In this chapter, various pointer swizzling strategies are classified and implementations which conform to each strategies are described in detail. As has been pointed out in section 3.3, there are three classes of lazy pointer swizzling methods. In this section, four swizzling methods are described which consist of the three mentioned and the most eager scheme.

In following the presentation below, please note that for the lazy swizzling strategies, the distinction can be best made in terms of when reservation check is performed. Since, as pointed out in section 3.3, trying to swizzle earlier than a reservation checking strategy mandates is often beneficial and many implementation do so. The reason that they were not given names such as 'reservation check upon discovery' is that the term is less well recognized than the term *pointer swizzling*.

3.4.1 Fully Eager Swizzling

The most naive approach for pointer swizzling is to convert all references within database at program startup. By doing so, there are no need to carry out reservation check during program execution because the program is guaranteed to see only valid LIDs. All persistent references must be made resident on physical storage in order for them to be overwritten with a LID. Scalar fields can be faulted in on demand basis if desired.

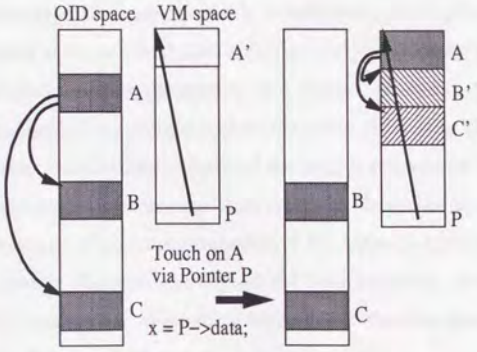
This scheme is less complicated because no modification to the compiler nor handling of a page fault is required. Yet, writing a software component which can be utilized for each new project without programmer's intervention is not trivial because C++ language standards do not support runtime type information and runtime type identification. Inheritance and templates help to make such reusable component easier to use, but still a programmer needs to write helper functions. Examples of such functions are a member function which returns a size of an instance and a member function which returns the set of offsets within an object where pointers are located. This issue is further discussed in section 4.3.4.

3.4.2 Pointer Swizzling at Fault Time

In the fully eager swizzling scheme, all references are converted to a corresponding LID at program startup. However, for a running program it does not make any difference whether a reference that it tries to fetch from memory has been converted in advance or the reference gets converted in the course of program execution as long as the conversion is transparently performed to the running program. The conversion itself and the detection of the need of conversion must be transparent, though. Otherwise the running program must be modified so that it performs the conversion and the check by itself. In comparison with the fully eager strategy, *pointer swizzling at page fault time* delays the swizzling of a reference until the page containing the reference is first accessed. The *Texas Persistent Store* utilizes memory protection through a memory management unit for detecting the

first access to a memory page.

At program startup, only those pages in which root objects reside are mapped into protected virtual memory space, whose contents are not set yet. The mapping is established through an API call similar to 'p3l_bind_root()' call described in section 5.4.2. The left side of Figure 3.1 illustrates the situation.



The object (A) pointed to by the root pointer is
mapped onto VM space. But the contents are still on disk.
When any portion of A is accessed, a page fault is triggered.
Then the exception handler loads the contents and maps
those objects (B,C) which are pointed to by pointers in A.

Figure 3.1: Reservation at Fault Time

The access protection on these memory pages is set to 'no-access' at first. Thus the first access to one of the pages triggers an access violation. The runtime system of the *Texas Persistent Store* catches the exception and lookup the persistent address which is associated with the accessed page. Then a page of physical storage is allocated and attached to the virtual memory page so that the contents of the accessed page can be transferred from the persistent store to the physical storage.

During the transfer, each reference in the page is swizzled. If a persistent page pointed to by a reference is already mapped into virtual memory space, the reference is swizzled with the virtual memory address. Otherwise, a new page is allocated in virtual memory space for the referent so that the pointer can be swizzled. The protection status of the page is set to 'no-access' and the correspondence between the persistent address of the page and the virtual memory page is remembered.

When the contents of the page is entirely transferred and all references are swizzled, the protection status of the accessed page is set to 'read-only'. As far as pointer swizzling is concerned, the status may be changed to 'full access'. However, the *Texas Persistent Store* uses memory protection also for update detection. It is after the first write to the page is detected that the protection status of the page is relieved to 'full access'.

The idea of swizzling pointer at page fault time was discovered and abandoned in late 70's during development of an implementation of PS-Algol. A system has been built on VMS operating system. However, because of the high frequency of faults due to scarce amount of memory and the cost of handling faults due to the slow processor speed at that time, the idea was abandoned [5].

The benefits of *pointer swizzling at fault time* stems from the fact that instruction sequence for the dereference operation does not have to be altered in order to deal with persistent objects. That means, a conventional compiler can be used for compilation and pre-compiled libraries can be freely linked and used against persistent objects. An example of such library call is 'memcpy()'. memcpy() copies the contents of a region of memory to another region. Copying a non-resident persistent object to another memory location properly works even if memcpy() does not account for persistence because the memory management unit watches for an access to the persistent object and fetches the object if necessary.

The problems inherent in the swizzling scheme are:

1. LID space is consumed aggressively.

2. Physical storage and main memory may also be consumed unnecessarily due to the eager allocation of LIDs if only page-wise access protection is available.

3. Fielding access faults incurs overhead.

4. An implementation is not portable to a new operating system which does not support setting memory access protection and notifying access faults to an application.

LID space is scarce resource compared to persistent address space and under certain situations it may be exhausted easily. For example, consider a B-tree index on a set of large image objects. A leaf page of the index may consist of pointers to an image object. If a leaf page contains 1024 pointers and an image object is 4MB bytes each, swizzling cannot be performed simply because there is not enough virtual memory space to map all objects into, unless address space larger than 32 bit is assumed.

The aggressive consumption of LID space is not a problem to many classes of applications because 10MB object is not a typical size of an object and it may be possible to devise a data structure with which no single page is filled entirely with pointers. However, the problem of excess consumption of physical storage caused by the eager allocation of LID and the lack of protection granularity finer than a page is more severe because physical storage is much limited resource and main memory is even more limited. The problem is illustrated in Figure 3.2. Suppose that the object R has been just loaded from the persistent store because the program accessed a part of it. The object R contains 3 pointers which point to the object A, B and C. The pointers must be swizzled at this point but will not be used in the future. Thus the part of the page committed for the object R is wasted by the object A.

This example assumes object-wise reservation instead of page-wise reservation. Object-wise reservation was chosen in this example in order to show that the problem is inherent to any variant of this swizzling scheme. In the case where page-wise LID allocation is performed, there is a possibility that the fragment will be occupied by objects completely unrelated to 'Object-R'. That may degrade the average utilization ratio further.

Fielding access fault is a relatively expensive operation under modern processor architecture for such reasons as cache flushes due to the execution of the fault handler, saving and restoration of processor contexts, etc. However, because the execution of the fault handler is often accompanied with corresponding I/O operations, which is a relatively slow operation, the cost of catching faults may not be significant.

Finally, this method cannot be implemented without support from operating system for setting memory protection status and for catching access faults. That affects portability of an implementation. However, most modern operating systems support such capabilities.

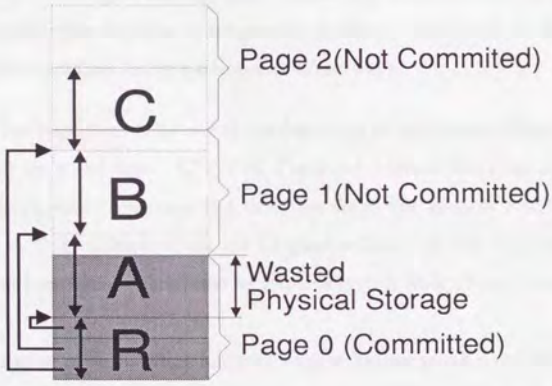


Figure 3.2: Wasted Physical Storage due to eager LID allocation

3.4.3 Pointer Swizzling upon Discovery

Pointer swizzling upon discovery is implemented in *P3L* and LOOM. Every reference is reservation checked when it is discovered. Thanks to the existence of the check, refer-

ences do not need to be swizzled upon the fault of the containing object unlike *pointer swizzling at fault time*. However, it poses no problem to swizzle references at fault time and actually it is advantageous when referents are resident or known to be accessed soon. Thus although *P3L* and LOOM perform a reservation check every time when a reference is discovered, it swizzles some references at fault time forcing the address reservation if doing so seems beneficial. The term *discovery* was coined in [22]. The paper defines a discovery as follows.

> The approach used by EPVM 2.0 is to swizzle pointers within objects as they are "discovered", i.e. when the location of the pointer becomes known. We shall call this type of swizzling **swizzling upon discovery**. A pointer within an object may be discovered when its value is assigned to another pointer, when it is involved in a comparison operation, or in a number of other ways.

The subtle but important difference in our definition of *pointer swizzling upon discovery* and theirs must be noted here. *EPVM* (E Persistent Virtual Machine) merely tries to swizzle a reference upon discovery and does not force the address reservation for the referent. That is, if the referent is not yet assigned with a LID, the swizzling is deferred. This means that reservation check must be implemented in later phase, for example, *upon dereference*.

The combination of an attempt to swizzle *upon discovery* and reservation check performed *upon dereference* found in *E* is similar to the combination of an attempt to swizzle *at fault time*[8] and reservation check performed *upon discovery* found in *P3L*. The difference is that *P3L* carries out both operations one step earlier than *E* does.

Reservation check can be implemented in either hardware or software. For example, quoting [5],

[8] Actually, 'upon object transfer' instead of 'at fault time' because deferred loading is not yet implemented in *P3L*.

The Moby address space system for LMI Lisp Machines used page-wise relocation, but pointers were swizzled on discovery (one at a time) using the hardware's support for tagging.

Unfortunately, the kind of hardware which is suitable for implementing the swizzling method is rare and LMI Lisp Machines has long deceased. However, the most widely used CPU, Intel x86 series of processors, have memory segmentation hardware which performs access checks on a memory segment when a segment selector is loaded into a segment register. Thus this hardware can be utilized to implement the swizzling scheme in hardware. However, the number of available segments is rather limited and ,worse, no readily available compilers support the pointer representation which consists of a segment selector and a 32 bit offset within the segment and no commonly used operating systems supports the execution of an application which contains such a pointer. Thus the use of the mechaism results in very non-portable implementation.

In many applications, a reference is more frequently dereferenced than it is fetched from memory because the pointer may be repeatedly dereferenced to access the same or different part of the referent. An exception is when a reference itself is compared with another reference instead of being dereferenced. That is, when an identity of an object is compared instead of its value.

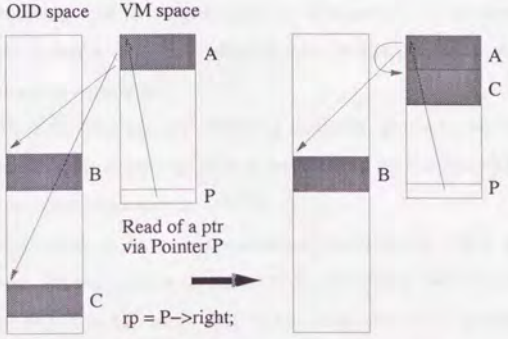
In the *P3L* implementation, the number of reservation check is further reduced by maintaining the following invariant.

> Pointers which reside on stack, on static storage or in a CPU register are all in swizzled form.

If this invariant is assumed, pointers fetched fetch a memory location on stack or in static storage need not be reservation checked. It is impossible that the invariant will be invalidated by normal execution of a program because, thanks to the reservation check installed, the running program will not copy a reference before it is being swizzled.

The only possibility of invalidation is left in an API call to *P3L* runtime library such as 'p3l_bind_root()'. However, the function loads a persistent object in the heap, partially swizzles embedded references and return the virtual memory address of the object. Thus it never allows a persistent reference creep onto stack or static storage. Other APIs do not return a persistent reference either.

Thus, only references in the heap are subject to reservation checks. That means the compiler needs to emit reservation checking instructions only for an expression which yield a pointer through a dereference of a pointer. Details are given in chapter 4.



The object (A) pointed to by the root pointer is read into VM memory at startup. When a compiler detects a pointer dereference which yields a pointer value, it inserts extra instructions for checking the value. If the pointer is in OID form then virtual memory space for the referent is allocated, the target is loaded, and the pointer is overwritten with the address.

Figure 3.3: Reservation upon Discovery

The swizzling method as implemented in *P3L* is illustrated in Figure 3.3. Like in Figure 3.1, traversal on a binary tree is assumed. Please note that, although the figure assumes that objects are made resident when a LID is assigned to it, it is possible to use

memory access protection to install residency checking for implementing delayed loading of object contents.

One of the benefits of this swizzling scheme is the moderate LID space consumption compared with *pointer swizzling at fault time*. A reference will not be swizzled unless the reference itself is accessed. However, as has been pointed out, a reference gets swizzled regardless if the referent is accessed. Compared with *pointer swizzling upon dereference*, the number of OID to LID translation is reduced because a reference is swizzled before being copied to other locations. The number of reservation check is also reduced.

The object-wise relocation combined with rather lazy allocation of LID space helps the *P3L* implementation achieve compact packing of objects to be accessed by a program. This characteristic makes a difference when objects in the persistent store are not well clustered as is shown in section 6.

Having described the two pointer swizzling methods, *pointer swizzling at fault time* and *pointer swizzling upon discovery*, now is the time to review the object faulting and pointer swizzling method employed in LOOM.

LOOM was developed as an enhancement to Smalltalk-80 which was using 16 bit *oops*[9] at that time. An oop points to an entry in the object table[10] which indirects to the content of the object in the *heap*. Due to the shortness of an identifier, the number of objects manageable was very limited. The heap was not plentiful either. LOOM was needed in order to increase the number of objects manageable as well as to expand the size of the heap.

In LOOM , when an object is faulted in, each reference in the object gets translated to a LID if so chosen. A LID is an index to an OTE, which is called a *short oop* in LOOM parlance. In case the target object has not yet been given a LID, a new entry (i.e. a new LID) is allocated from the object table and the OID of the target is remembered in the block of memory which is linked to the OTE. This proxy for a non-resident object is

[9]Object Oriented Pointer.

[10]OTE,Object Table Entry. The table also serves as a ROT.

called a *leaf*. Since a LID is assigned at fault time, no reservation check is needed as long as only leaves are involved.

The OID contained in the attached memory block makes it easy to locate the real contents of the object in a persistent store when it is needed. A message sent to a leaf is detected by the message dispatcher built into Smalltalk Virtual Machine and the target object gets transferred at this point. That is, LOOM carries out residency checks by software at message send. The process of object faulting using leaves is depicted in the left part of Figure 3.4.

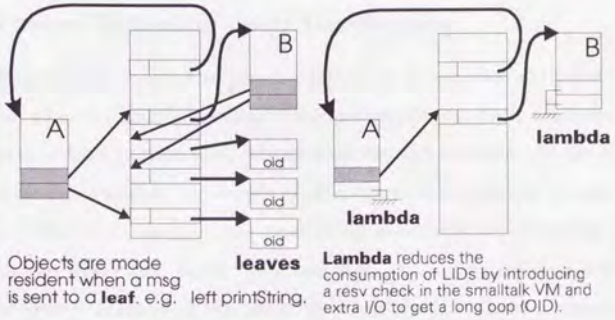


Figure 3.4: Leaf and Lambda in LOOM

LOOM has another mechanism for representing non-resident objects. With the use of leaves, LID assignment is carried out very eagerly. This makes the mechanism prone to exhaustion of OTEs. To alleviate this problem, a special short oop value named *lambda* was introduced. Lambda stands for a reference which has not been fully swizzled yet. When the VM fetches lambda from memory, it goes over to the image of the containing object in the persistent store to fetch the OID of the referent from the relevant field. Then it creates a leaf or a normal object. Reducing the consumption of heap and OTE through lambda incurs the cost of extra I/Os and extra check (reservation check) required in the

virtual machine. Use of leaves and lambda is illustrated in the right of Figure 3.4.

Physical memory can be reclaimed by contracting a regular object into a leaf object without scanning all the resident objects. In this case, only an update to OTEs is required. For reclaiming LIDs, each OTE is reference counted. Note that contracting resident objects into leaves will free some OTEs.

In summary LOOM tries to swizzle upon object fault and optionally narrows references through the use of lambda. Residency checks are carried out when a message is sent. Reservation checking is performed upon discovery also by software.

3.4.4 Pointer Swizzling upon Dereference

This swizzling method does not require a reference to be swizzled until just before it is dereferenced. It is the laziest swizzling strategy conceivable. Early implementations of PS-Algol seem to have implemented this strategy through software. As the implementation details are not available, the details of swizzling implementation are not clear. The speculation is that there was a virtual machine instruction for dereferencing a reference, carrying out the reservation check simultaneously. An implementation using a smart pointer may swizzle pointers at this point by overloading dereference operators on the smart pointer.

The idea is illustrated in Figure 3.5. Like in the case of *pointer swizzling upon discovery*, residency check is assumed to be not present. In this case, however, deferring transfer of object contents is not so beneficial if the amount of transfer is small because the referent is going to be accessed immediately after swizzling is performed. It is beneficial when a large object is partially accessed, though.

A software implementation of this scheme incurs high overhead in the number of CPU cycles because of frequent checks on a reference, though compile time optimization can reduce the number of checks. The number of OID to LID translation also increases because references are likely to be copied before they are swizzled. Each copy would require one

OID to LID translation once dereferenced.

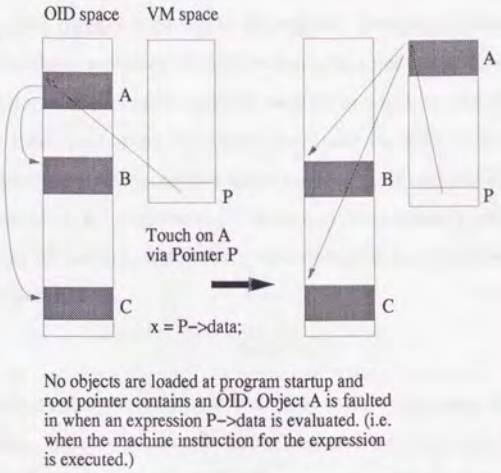


Figure 3.5: Reservation upon Dereference

Recently an implementation of this swizzling scheme which utilizes a memory management hardware on a stock processor was proposed [31]. In the proposed implementation, a dereference of a persistent reference is detected by MMU. References to an object not present in virtual memory space are partially swizzled to a pointer into an appropriate entry in a table. The entry describes the original OID and other info. And the memory protection status of a virtual memory region containing the table is set to no-access.

Unlike *pointer swizzling at fault time*, a pointer which caused an access fault does not contain a valid LID. Instead, it points to an entry of the *OidLid map* which is access protected. The fault handler can retrieve the OID of the target object based on the entry to which the memory access was made. This is the principle how the implementation works. However, there are a few tough problems to be solved.

First, a stock processor would unlikely provide the information, with which the register holding the persistent reference can be identified. If there is only one register that contains the fault address, that register is probably the culprit. However, in case more than one register contains the value, swizzling all register contents is not a correct operation because it is possible that some of the values may be an integer which accidentally matches the fault address. On CISC processors, no register may hold the fault address. The solution proposed for this problem is to designate a single register[11] and exclusively use this register when a dereference is to be performed. This would likely affect performance severely because interleaving of multiple dereference operation will be prohibited. For example, consider an expression like

$$*p + *q$$

In order to evaluate this expression, at least four memory accesses are required. That is, an access to fetch 'p', an access to fetch '*p', an access to fetch 'q', and an access to fetch '*q'. An optimizing compiler for a RISC processor would allocate one register, each for storing the value of 'p' or 'q', and emits instruction sequence which looks like the one below.

$r0 \leftarrow p$
$r1 \leftarrow q$
$r2 \leftarrow [r0]$
$r3 \leftarrow [r1]$
$r4 \leftarrow r2 + r3$

In this case, it is likely that the contents of 'p' is present in 'r0' when the third instruction is to be executed. On the other hand, under the constraint that a dedicated register must be used for a dereference operation, the interleaving exemplified above

[11] The register shall be referred to as 'deref_reg' for convenience in the remainder of this section.

become impossible, forcing the execution of the instruction for accessing '*p' stall for the delivery of the contents of 'p'.

Second, executing fault handler is expensive and naive implementation of this strategy would likely generate a lot of them. Their solution is to designate another register to remember the memory location from which a persistent address is fetched. By doing so, the source location can be swizzled as well as the contents of the 'deref_reg'. To further reduce the number of faults, it is proposed to keep a linked list of persistent references which could not be fully swizzled at fault time. The fault handler consults this list to swizzle those known references. Half (32 bit) of a 64 bit pointer is used for this chaining. The rest contains a degenerated OID, which is set to point to a corresponding entry in the *OidLid map*. Even with this enhancement, not all persistent references related to the access fault at hand can be found out and swizzled because the degenerated OID may have been copied to somewhere else after the persistent reference is loaded into physical storage. Updating the linked list at each copying is too costly and is not performed.

Third, a dereference operation on a persistent reference does not necessarily cause access fault on the right entry of *OidLid map*. That is because of a pointer arithmetic. Consider an expression, $p \rightarrow x$, which access the member 'x' of the structure pointed to by 'p'. Because the memory location where the value of the expression is calculated by adding the offset for 'x' to the value of 'p', the address the fault handler observe may be different from the address contained in 'p'. The handler needs to know 'how much' was added to 'p' in order to locate the correct entry. To account for this problem, another register is assigned to hold the offset. That means, the register must be zeroed out even if no offset calculation is necessary.

3.5 Pointer Swizzling in P3L

As described previously *P3L* employs the *pointer swizzling upon discovery* with *narrowing*[12]. By eagerly swizzling pointers compared to *pointer swizzling upon dereference*, every pointer migrating into a CPU register is guaranteed to be a valid virtual memory pointer. Thus the method eliminates the need to perform reservation check upon each dereference. As is shown in chapter 6, this results in very small overhead even in navigation intensive applications.

This chapter describes in detail the pointer swizzling method as is implemented in *P3L*. Also, the goals set for the design and implementation of the pointer swizzling and object faulting method are presented. And the way how they are handled in a running application is illustrated.

3.5.1 Goals

The design issues in building a persistent object system were discussed in chapter 2. The decisions made regarding those issues were driven by the objectives which were set for *P3L*. The objectives are:

1. To preserve as much compatibility as possible with C++ language.
2. To make the handling of persistent objects as transparent as possible to programmers.
3. To preserve as much compatibility as possible with available programming tools such as linkers and debuggers.
4. To make the language implementation as efficient as conventional C++ implementations that does not provide support for persistent objects.

[12]This technique is also referred by the term *partial swizzling* in this thesis.

The compatibility with C++ language was sought because it is the most widely accepted object oriented programming language and both free and commercial implementation are abundant. Not altering the language too much was another goal because otherwise choosing the well-known language to start with does not make any sense. The aim was to make the benefits available to programmers without any retraining which is accompanied with a slow learning curve. Compatibility with existing tools was important otherwise programmer needs to learn a new tool set, and essential tools have to be reinvented .

Even with all the compatibilities and transparent handling of persistent object achieved, the implementation will be useless if programs run much slower in *P3L* than programs developed through conventional development tools. Most of persistent object systems available do not try to compete with non-persistent systems in terms of the efficiency of program execution when all objects visited are cached in the physical storage and the application is memory access intensive. Because of the length of reference and the way references are resolved in those persistent object systems, applications run much slower.

For many OODB vendors, efficiency under the said condition is not a significant concern. Whereas for the designers and implementors of *P3L*, *Texas Persistent Store* and *ObjectStore*, it is. An example of such opinion can be observed in a news article attached below:

```
From: dlw@odi.com (Dan Weinreb)
Newsgroups: comp.databases.object
Subject: Re: ODBMS Selection:  Versant vs. ObjectStore
Date: 14 Dec 94 16:27:10
Organization: Object Design Inc., Burlington, MA
Lines: 25
Message-ID: <DLW.94Dec14162710@butterball.odi.com>
References: <D01qFI.BuK@intertv.com> <ED.94Nov30215404@heinz.odi.com>
<DLA.94Dec14081948@heinz.odi.com> <1994Dec14.191921.20487@objy.com>
Reply-To: dlw@odi.com
```

NNTP-Posting-Host: butterball.odi.com
In-reply-to: urs@server.objy.com's message of Wed, 14 Dec 94 19:19:21 GMT

In article <1994Dec14.191921.20487@objy.com>
urs@server.objy.com (Urs Bertschinger) writes:

> My point is that, in my opinion (and, yes, it is an opinion), this speed advantage is not significant for most "real" applications (once again, I am not arguing that it does not exist).

Dave Andre has already said most of what I would say to reply to your posting. I'd just like to add that there are certainly real applications for which the speed of pointer dereferencing is very important, at least according to our customers. You may feel that a 33% or even 50% slowdown isn't important, and indeed in some situations it isn't, but in others, it is. For example, one of our prospective CAD customers told us flat out that he wanted to store such objects as "chip", "wire" (or even "piece of wire"), "pin" and so on as database objects, but when his system draws the circuit on the screen, if screen redisplay slows down by 10%, that would be considered unacceptable (i.e. they'd sooner not use a DBMS at all than suffer such a slowdown; I'm just telling you what I heard).

Certainly for some applications dereference speed is not a significant factor, and there are plenty of other aspects of an OODBMS whose speed can become the critical bottleneck. Indeed, even in the papers we have been citing, the authors mention quite a few other aspects of the the different architectures that affect performance.

Achieving the level of efficiency discussed here entails careful choice of the representation of persistent reference in the physical storage. Ideally the representation should match that of a raw pointer. That would make identifying pointers on stack precisely more difficult, which may prevent the reclamation of LID from being implemented. Despite of this potential problem, having the efficient representation of persistent references is vital. Because:

1. There are many applications whose maximum working set fits comfortably in the virtual memory space available.

2. One can fall back to the use of smart pointer, if the reclamation of LID is an issue.

3. For many applications, it is likely enough to evict object which is not referenced by any pointer on the stack. That means a type of conservative garbage collector[42], which does not mandate precise identification of pointers on the stack, can be utilized.

4. There is a compilation technique which allows pointers on stack and in registers to be precisely located [32].

The *Texas Persistent Store* and *ObjectStore* utilize access protection through a memory management unit to avoid the overhead of carrying out residency check in software. And reservation checking is completely eliminated by swizzling pointers at page fault time as is described in section 3.4. On the other hand, *P3L* reduce the number of reservation checks by carrying out reservation check rather eagerly, but less eagerly than the swizzling at fault time approach. Current implementation eliminates the need for residency checking by fetching an object when a LID is assigned to it. Planned improvements on the implementation includes the use of memory management hardware for residency checking. In the rest of this chapter, the swizzling method employed in *P3L* is described in detail.

3.5.2 Example Database

In order to illustrate the pointer swizzling method, retrieval of a persistent data structure is used as an example. The structure is a persistent binary tree, which consists of a root pointer and a tree structure. The tree in turn consists of elements of type 'Node', which is allocated in persistent heap space. The necessary class declarations are given in Figure 3.7. And the traversal code for picking up a matching node is shown in Figure 3.9.

3.5.3 Binding the Root Object

The application program must first connect to a storage manager and start transaction. The interface to these functions is described in chapter 5. After initialization is completed, an application gets at a root object by binding a pointer to it. A root object is a persistent object which has a name attached to it. A name is an array of character terminated by a null character. Separate applications share objects through a name. The binding is established through the call to *p3l_bind_root()*. The correspondence between the virtual memory address where the persistent object is loaded and the OID of the object is remembered in a table called *OidLid map*.

Alter the binding is established, the pointer 'tree_root' is initialized with the address where the copy of the persistent object named 'THE_ROOT_OF_TREE' is placed. Because the contents of this object is a reference to another object (an instance of 'Node' class which sits at the top of the persistent tree) and the object is not yet given a LID, the OID contained in the reference gets converted to a special LID value which is called *lambda*. In the current implementation of *P3L*, lambda is represented by the value (void *)(-1). When the OID is converted to a lambda, the memory location to which the lambda was written into is remembered with the associated OID in another table called *Addr2Oid map*. The situation at this point is depicted in Figure 3.6.

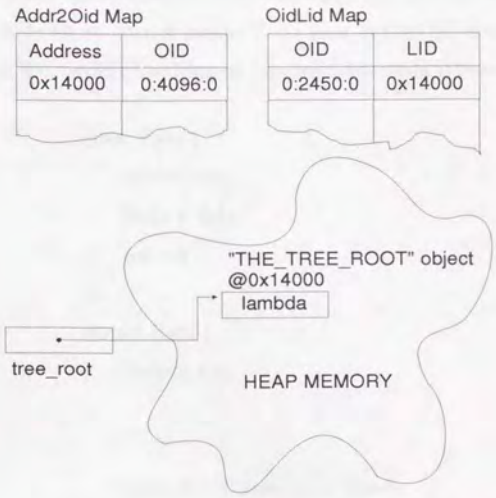


Figure 3.6: After p3l_bind_root() call.

3.5.4 Performing Swizzling

When evaluating the initializer for the variable 'np' in line 6, the compiler issues instructions for checking whether the value being fetched from memory is a lambda. If the check turns out positive, a function called *_p3L_sw_ptr()* is called with the address of the lambda passed as an argument. The called function *_p3L_sw_ptr* determines what the OID the lambda stands for by consulting *Addr2Oid map*. In this case, the OID of the object turns out to be 0:4096:0, which is a 96 bit value. The representation of an OID and a reference differ depending on the storage manager under whose control the object is created. The *P3L* Simple Storage Manager uses 96 bit reference, whose first word designates the file

volume and the second word represents the offset within the volume. The last word is an offset which accounts for an interior pointer[13]. As soon as the OID is retrieved, the table entry via which it was retrieved is dropped because the lambda is being removed.

```
class Node {
    Node * left;
    Node * right;
    int val;
};
struct Root {
    Node * top;
};
```

Figure 3.7: Definition of classes

Then *_p3l_sw_ptr()* function issues a call to a member function of the storage manager, which loads the contents of the object. During the transfer of the contents of the object the storage manager performs the narrowing operation if necessary. Because there exists no cyclic references in this example database, all persistent references in the loaded objects are replaced with a lambda and corresponding entries are added into *Addr2Oid map*. Since a Node instance has two outgoing references, the number of entries being added is two. Finally, the correspondence between the OID and the virtual address where the object has been transferred to is added to *OidLid map*. The situation at this point is depicted in Figure 3.8.

Just like 'tree_root→top' was reservation checked, 'av[1]' is reservation checked because it is a pointer value which is generated through dereference of another pointer. However, 'av[1]' contains a valid virtual memory address because it points to a volatile object. Thus

[13] An interior pointer is a pointer which points to any part of an object which is not at the top.

the check is performed but no faulting is triggered by this expression.

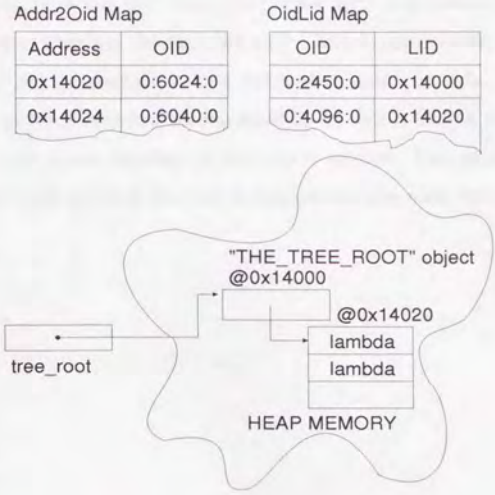


Figure 3.8: After the first object fault

3.5.5 Accessing Scalar Fields

Because of the pointer swizzling and associated object faulting described above, the pointer 'np' now contains a valid virtual memory address where the contents of the root node of the tree resides. The contents have already been transferred to the physical storage. Pointer fields may contain a lambda but the field which is not a pointer contains a correct value. Because of this arrangement the compiler issues no checking code for the expression 'np→val'. This is how *pointer swizzling upon discovery* can reduce the number of reservation check.

3.5.6 Object Faulting goes on

Now, depending on the value 'diff' either the pointer 'left' or the pointer 'right' is followed although the loop is exited in the case 'diff==0'. Both expressions regarding 'np' in line 11 and 12 are subject to reservation check and both contain a lambda. Thus regardless of the path taken, another swizzling accompanied by an object fault is performed in either line 11 or 12. Then a new iteration of the loop is entered. This swizzling and faulting process continues until a leaf of the tree is encountered or a node whose 'val' is matched by 'v'.

```
#include <stdlib.h>
#include "treedef.h"
static Root * tree_root;                                    // line 1
int main(int ac, char**av)                                  // line 2
{
    StorageManager * sm = create_ESM();                     // line 3
    sm→StartTransaction();                                  // line 4
    p3l_bind_root(&tree_root,sm,"THE_ROOT_OF_TREE");        // line 5
    Node * np = tree_root→top;                              // line 6
    int diff, v;                                            // line 7
    v = atoi( av[1] );                                      // line 8
    while( np ) {                                           // line 9
        diff = v - np→val;                                  // line 10
        if ( v<0 ) np = np→left;                            // line 11
        else if ( 0< v) np = np→right;                      // line 12
        else break;                                         // line 13
    }
    sm→CommitTransaction();
}
```

Figure 3.9: Tree Traversal Code

Chapter 4

P3L Compiler Implementation

In this chapter, the implementation of the *P3L* compiler is described. Two experimental implementation of *P3L* preceded the current *P3L* compiler implementation. Both adopted a translator approach for generating reservation checking instructions. The one being worked on now is a native compiler. The handling of generation of ancillary functions and the type information has changed from the initial compiler implementation to the current compiler implementation.

In the first section, implementation strategies are discussed, including a prospect which discourages adding language extensions.

4.1 Implementation Strategies and History

The most recent *P3L* implementation adopts *pointer swizzling upon discovery* through compiler modification. Gnu C++ compiler is modified so that the compiler generates required reservation checking instructions against a discovered pointer. Previous experimental implementations were tried as a translator based on PCC (Portable C Compiler) on BSD unix and a translator written from scratch. Unfortunately the former did not support new ANSI C syntax thus making compilation of some benchmark applications impossible. And neither of them supported compilation of C++ code. Theoretically,

they could have been extended to cope with C++. But C++ is a very complex language and even parsing C++ properly entails much effort and time. Also the stability and speed of the translator were not too good. Thus, the idea of the current implementation which bases on Gnu C++ compiler was conceived. The new compiler generates machine instructions directly without going through a C source file as intermediate output. The modifications made to the compiler is detailed in section 4.3.1 and 4.3.3. As a result, compilation is faster and the compiler is more stable.

The two translator implementations supported a special keyword *persistent* and a new syntax form, *binder*, for defining a persistent root. *Persistent* is a type qualifier which specifies the object declared or defined to be persistent. *Binder* specifies the name of the persistent object to which the variable qualified to be persistent is bound. For example, a definition of a persistent integer is made in the following way:

```
persistent int aNumber <= "anPersistentInteger";
```

This language extension has been abandoned in the current implementation with Gnu C++ Compiler in favor of portability of application programs and the same function can be implemented within the language standard. Thus an application written for *P3L* system can be compiled by any legitimate C++ compiler. And compiled programs run properly if a runtime system which supports *pointer swizzling at fault time* is provided as in the *Texas Persistent Store*. Otherwise, a special persistence-aware compiler, such as the *P3L* compiler, is required. To further facilitate porting of applications, construction of 'reservation check checker' is being considered. This checker parses source codes and picks up expressions which are subjected to reservation check. If any of such expression is not surrounded by a call to the reservation check function, the checker issues warning about it. Using such a checker, manually inserting reservation checking code becomes less error prone process and the written code can be compiled with any C++ compiler and run flawlessly. An expression checked in this way would look like:

ResvChk(p→next)→next = ResvChk(ResvChk(q→prev)→next);

Of course, the expression looks much simpler if a persistence aware compiler like the *P3L* compiler is available so that those calls to ResvChk() do not clutter source codes. Fortunately, Gnu C++ Compiler is a very portable system and runs on almost all flavors of Unix operating system in addition to OS/2, Microsoft DOS and Microsoft Windows NT. Thus portability of *P3L* applications can be maintained to a certain degree without embedding reservation checking code in applications. However, for various reasons not only technical but legal, the use of G++ compiler must often be avoided. The checker approach should serve in such occasions.

4.2 Overview of Gnu C++ Compiler

Gnu C++ Compiler is a C/C++/Objective-C compiler which is freely distributed under Gnu Public License. All the three compilers can be generated from a single distribution package.

4.2.1 Machine Description

GCC has a back-end which eases porting of the compiler to a new processor architecture. One can adopt the compiler to a new processor by writing a machine description instead of modifying the back-end itself. Machine description consists of a set of templates, which are used by the back-end to choose a suitable machine instruction sequence generated from a particular RTX, which is to be explained shortly. The following is an example of a template of machine description.

```
(define_insn "tstsi"
  [(set (cc0)
        (match_operand:SI 0 "general_operand" "rm"))]
  ""
  "*
```

```
(if(TARGET_68020 || !ADDRESS_REG_P (operands[0]))
    return \"tstl %0\";
 return \"cmpl #0,%0\";}")
```

The name "tstsi" indicates that this template can be used to generate an instruction(s) for testing a value of single integer against 0. When generating RTX from AST (Abstract Syntax Tree), to be described shortly, the compiler utilize the knowledge that this CPU can handle "tstsi" operation. The RTX bracketed by '[' and ']', specifies the RTX, to which this template can be applied to. The last slot is a C code fragment for generating instruction pattern given to assembler.

The above paragraph talks about the conversion performed at the lowest level in the latest phase of the compilation, that is, RTX to a machine instruction(s). In the next section, we look at higher level representation of a program from which RTXs are generated.

4.2.2 Abstract Syntax Tree

AST is a more abstract representation of a program than RTX. As the name implies, an AST represents a fragment of source code such as a variable declaration, a function definition, a statement, an expression. The compiler converts input file into ASTs function by function. Examples of AST are shown in Figure 4.1.

The construction of an AST progresses in a bottom up fashion together with parsing. For example, when the production rule listed below is taken, an AST node for binary '+' operator is created through the call to *build_x_binary_op*. Each of the second and the third argument represents the AST which has been generated while respective *expr_no_commas* expression is parsed.

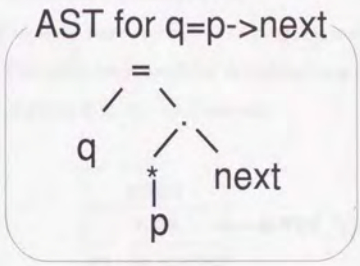


Figure 4.1: An example of an AST node.

```
expr_no_commas:
        ...
| expr_no_commas '+' expr_no_commas
{ $$ = build_x_binary_op ($2, $$, $3); }
```

The construction of '+' operator node proceeds as follows if the two operands are compatible with regard to '+' operation. If not, a node representing an erroneous AST is returned.

- A memory chunk is allocated from a memory pool, called *the new node* in the following.
- *Code* field in the new node field is initialized with 'PLUS_EXPR' for the purpose of identification.
- The pointers to the sub-AST are copied into *operands* field.
- The type of the new node is determined and set in the new node. If one of the two operands is a pointer and the other is int, for example, then the type of the new node picks up the type of the pointer.

Figure 4.2 illustrates the procedure sketched above.

The important thing to note here is that the compiler maintains type information for each nodes in AST. This aids very much in detecting expressions which need to be reservation checked as is explained in the next section.

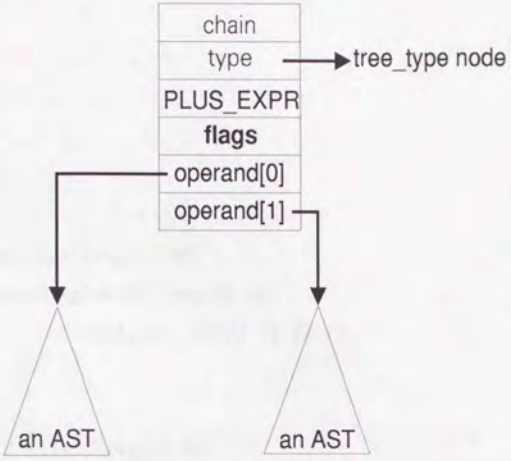


Figure 4.2: Constructing PLUS_EXPR Node in an AST.

4.2.3 Register Transfer Level Expression

The representation of AST is a little too high level for performing optimization and machine code generation on. Thus the compiler first converts ASTs into a sequence of RTX[1]. Then it performs optimization on RTX, assigns hard registers to pseudo registers, and finally converts RTXs into machine instructions following the guidance by the machine

[1]RTX is an abbreviation for Register Transfer level eXpressions.

description, which is described in section 4.2.1. A function definition and a sequence of RTX generated by compiling the function are shown below:

```
int foo(void)
{
    char * p;
    char * q;
    *p = q[4];
}

;; Function foo

 1:(insn 9 6 11 (set (reg:SI 68)
 2:        (mem:SI (plus:SI (reg:SI 65)
 3:                (const_int -4)))) -1 (nil)
 4:    (nil))
 5:
 6:(insn 11 9 13 (set (reg:SI 69)
 7:        (const_int 4)) -1 (nil)
 8:    (nil))
 9:
10:(insn 13 11 14 (set (reg:SI 71)
11:        (mem:SI (plus:SI (reg:SI 65)
12:                (const_int -8)))) -1 (nil)
13:    (nil))
14:
15:(insn 14 13 16 (set (reg:SI 70)
16:        (plus:SI (reg:SI 69)
17:            (reg:SI 71))) -1 (nil)
```

```
18:    (nil))
19:
20:(insn 16 14 18 (set (reg:SI 72)
21:        (reg:SI 70)) -1 (nil)
22:    (nil))
23:
24:(insn 18 16 19 (set (reg:QI 73)
25:        (mem/s:QI (reg:SI 72))) -1 (nil)
26:    (nil))
27:
28:(insn 19 18 21 (set (mem:QI (reg:SI 68))
29:        (reg:QI 73)) -1 (nil)
30:    (nil))
31:
```

RTX generation is performed assuming unlimited number of registers. Later when pseudo registers are assigned with a hard register, spill codes are inserted to account for the limitation in the number of available hard registers. 'reg:SI' stands for a register which can hold single precision integer while 'mem:SI' stands for a memory location pointed by the operand RTX. const_int -4 and -8 being added in the RTX starting at line 1 and 10 are offsets from the current frame pointer to the location where the variables 'p' and 'q' are stored on the stack.

4.3 Modification to GCC

The previous section is an overview of the organization of GCC. This section describes what modification were needed to make GCC *persistence aware*. There are two things to be taken care of. They are:

- Insertion of the reservation checking code.
- Generation of ancillary functions.

4.3.1 Insertion of reservation checking code

Pointer swizzling upon discovery requires the reservation checks be performed on every pointer which potentially contains reference to a persistent object. Because each pointer to a persistent root is initialized in the way described in section 3.5.3 and persistent objects are loaded into the heap only, the sole possibility of a lambda being discovered by a running program is through a memory fetch from the heap. Thus by screening lambdas upon those memory access and performing swizzling on them if needed, the invariant can be maintained that every pointer read from memory to be fed into a register or a slot on stack is all fully swizzled. The screening is performed by having the compiler adds extra instructions to those generated from an expression which involves a pointer dereferencing and yields a pointer or a pointer-containing aggregate value. This strategy may seem to pose problems with memory-indirect addressing modes in some CISC machines. However, it is not a problem with P3L compiler because it generates two separate RTXs for loading the value of a pointer and for dereferencing the pointer, and inserts another RTXs in the middle for reservation checking. Thus the compiler cannot combine a RTX for loading pointer and a RTX for dereferencing to generate a single load instruction with memory indirection.

The reservation checking code is inserted through the following 2 steps. While AST is being built, the construction of a node for one of the three operator ('*','*rightarrow*' and '[]') is monitored. If the first operand of these operators is a pointer or a structure which contains a pointer, then the node representing the operator is marked using one of the bits in the 'flag' field shown in Figure 4.2. The second step is performed while RTXs are generated from an AST. When the compiler sees a marked node, it first generates RTX for the first operand of the node (the left triangle in Figure 4.2, for example). The

RTX specifies a memory location. Using the RTX, the compiler adds a sequence of RTXs which is equivalent to RTXs expected from the following code fragment.

```
// In case '*pp' is a pointer
if( (*pp)==(T *)-1 ) *pp = __p3l_swizzle_ptr(pp);
// Note that 'pp' is a pointer to a pointer and
// 'T **' is the type of 'pp'.


// In case '*pp' is a pointer containing structure
__p3l_swizzle_struct( pp );
```

Different code generation strategies are taken depending on whether the type of the node representing the expression '*pp' is a pointer or a structure containing a pointer. That is, a pointer is reservation checked in-line while pointers within structures are reservation checked through a function call. The purpose of taking this strategy is the reduction of code size.

Marking a node in the parsing phase and then checking the marker in the RTX generation phase may seem redundant. However, it was needed to implement compiler optimization.

4.3.2 Optimizations

P3L compiler performs a few optimizations rather than following the above principle naively. They are:

- Pointers to functions are not reservation checked.
- A pointer, whose value is compared with 0 and immediately discarded, is not reservation checked.

- If a single member of an aggregate is extracted by .-operator, swizzling all pointers in the aggregate is avoided.

Since no dynamic linking of functions are performed at runtime, a pointer to a function within an object is assume to be always fully swizzled at fault time. Because of this, applications do not have to reservation check a pointer to function at runtime.

```
char ** find_empty_slot( char ** array, size_t size )
{
    int i;
    char **r = 0;
    for(i=0; i<size; ++i)
      if( array[i]==0 )
        { r = &array[i]; break; }
    return r;
}
```

Figure 4.3: Finding an empty slot in an array of pointers

Pointer swizzling at fault time and pointer swizzling upon discovery, require a pointer to contain a virtual memory address when its value is examined. This implies a pointer must be swizzled even if it is never dereferenced as long as its value is examined. The code fragment shown in Figure 4.3 iterates over elements of an array of pointers to find an entry which contains a null pointer. Since 'array[i]' in the expression 'array[i]==0' is a pointer value yielded by an array subscript operator, the expression 'array[i]' is subject to the reservation check. However, the information required here is whether the yielded pointer contains null pointer or not. A null OID is fully swizzled to NULL pointer at fault time and other non-null OIDs are either fully swizzled to a virtual memory address

or partially swizzled to a *lambda* which is not a NULL pointer. Thus whether a persistent reference is null or not can be determined without forcing swizzling, simply by checking the pointer. For this reason a pointer does not have to be fully swizzled upon discovery if the value of the expression is only compared with null pointer.

The last optimization in the list also concerns the removal of unnecessary swizzling of pointers. By following the above procedure for marking AST nodes, given a structure and a variable definition below,

```
struct BinaryTreeNode {
    BinaryTreeNode *  left;
    BinaryTreeNode * right;
    char           val[256];
} * p;
```

the expression '*p' is subject to the reservation check. The reservation check on the structure looks at both 'left' and 'right' data members to see whether they need to be fully swizzled. This is the right thing to do in an expression which uses the entire structure as a value, for example in an structure assignment. However, if either of the data members is extracted from the structure value (e.g. (*p).left), running reservation check and swizzling the data member which is not extracted turns out to be a waste of the cpu cycles. The *P3L* compiler optimizes this case by emitting reservation checking only on the used field.

The last two optimization is performed by discarding or hoisting the marked node as shown in Figure 4.4. The pair on the left shows what happens when a node of AST for the operator '==' is constructed. The right operand is checked to see whether it is value 0 or not. Since that is the case, the mark on the '*' node is discarded meaning the expression neither '*p' nor '*p==0' must be reservation checked. The pair on the right shows what happens when a node of AST for the operator '.' is constructed. At that point, the mark on the '*'-node is hoisted up to the '.' node. Since the type of the '.' node is a pointer, the mark is kept there. Otherwise it is discarded. The actual code fragments in the compiler

is shown below. Similarly the mark is removed from the operand of &-operator and the left side operand of assignment operator (=) since only l-value of the operands is required. This is a requirement for the system to work correctly, not for optimization.

```
/* for the '==' operator */
if(TREE_NEEDS_RC(op0)&&TREE_CODE(op1)==INTEGER_CST&&integer_zerop(op1))
  TREE_NEEDS_RC(op0) = 0;
if(TREE_NEEDS_RC(op1)&&TREE_CODE(op0)==INTEGER_CST&&integer_zerop(op0))
  TEE_NEEDS_RC(op1) = 0;

/* for the '.' operator */
TREE_NEEDS_RC(ref) =
  p3l_is_subject_to_rc( TREE_TYPE(field) ) && TREE_NEEDS_RC(datum);
TREE_NEEDS_RC(datum) = 0;
```

The modification made to GCC for the generation of reservation checking code is very little. It consists an additional few dozens lines to existing files (mainly in 'c-typeck.c') and two new files of about 500 lines.

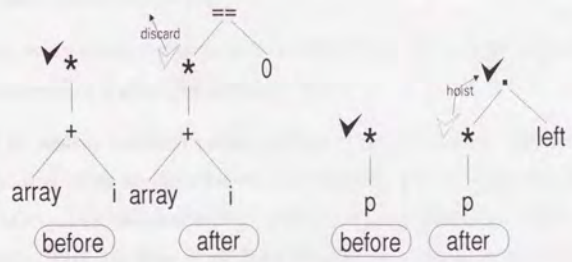


Figure 4.4: Discarded or Hoisted RC-mark

4.3.3 Generation of Ancillary Functions

For an application generated through compilation by the *P3L* compiler to run, a set of functions which are collectively referred to as *ancillary functions* must be generated by the compiler. As is discussed in section 4.3.4, leaving the responsibility of defining these functions to the programmer means reduced productivity and decreased reliability of the product.

_p3l_swizzle_struct() is one of the ancillary functions. Ancillary functions are specific to the type of an argument(s). In the case of this function, it performs the reservation check on pointers contained in the object whose address is passed in the argument. Depending on the type of the arguments, the function looks at different locations within the object. Another example is *p3l_bind_root()* function. The first argument to this function is a pointer to a pointer to the root object specified by the name given in the second argument. The type of the object varies.

The ancillary functions may be generated as implicitly defined functions emitted by the compiler. Or they may be generated using a language provided facility. The use of preprocessor macro and template functions are examples.

The strategy for ancillary function generation in the *P3L* compiler has evolved over time in the three steps listed below:

- At first, every ancillary functions were emitted by the compiler when a type definition is parsed or a priori for primitive types.
- The first scheme required rather elaborate modification to the compiler when a specification of an ancillary function is changed. This property was detrimental to make the system compatible with multiple storage managers. Thus in this implementation, only one kind of ancillary function was generated by the compiler. The function *_p3l_collect_ptr()* was emitted for each type defined in a program[2]. Other

[2]The functions are overloaded by their argument types. Thus there exist multiple versions of *_p3l_collect_ptr()* in a single program.

ancillary functions were defined as macros which embeds a call to *__p3l_collect_ptr()* function.

- The most current version of *P3L* generates ancillary functions through template function definition. Thus no compiler modification is applied in terms of the ancillary function generation.

The first two versions shared a problem, which results in larger code or programmer's drudgery. It is about avoiding multiply defining a single function. Since program modules may be compiled separately, it is possible that exactly same functions are generated in two or more modules. This may lead to an error in linking phase. The solution adopted was to restrict the scope of ancillary function definitions to the module in which they are defined, i.e. to add 'static' storage class to the definition. This would avoid the linker error with the expense of getting larger code. An alternative way of solving this problem is to give a programmer the control over instantiations, using a macro or a compiler pragma. In the second version, instantiation of functions, except *__p3l_collect_ptr()*, was controlled by the programmer. A similar problem arises from the need to determine the dynamic type of an object at fault time, which is described in section 4.3.4. For certain kinds of ancillary function, the *P3L* runtime must dispatch to the right version of an ancillary function given a dynamic type of an object (See section 4.3.4). As the type identity of persistent objects are determined by the name of their class, a table must be maintained to associate a type name to the corresponding *__p31_collect_ptr()* function. A similar problem arises regarding the generation of this table. However, making the storage class of the table static does not solve the problem this time because the *P3L* runtime library must have access to the table. The first two versions of the compiler did not support the table generation thus polymorphic persistent objects were not supported.

Below, the three incarnations of the ancillary function generation are described with their benefits and problems.

Direct Generation

In this version, each ancillary function for a specific type did not share code for locating references, returning its size, etc. Given the following declarations,

```
struct listnode {          |  struct treenode {
   listnode * next;        |     treenode *  left;
   int         val;        |     treenode * right;
};                         |  };
```

the following ancillary functions for reservation checking are generated.

```
/* one for 'listnode' class */
void __p3l_swizzle_struct( listnode * p ) {
    __p3l_swizzle_ptr( &p->next );
}
/* one for 'treenode' class */
void __p3l_swizzle_struct( treenode * p ) {
    __p3l_swizzle_ptr( &p-> left );
    __p3l_swizzle_ptr( &p->right );
}
```

The knowledge about the locations of references within an object is embedded in the functions. This would mean that the functions run faster since they do not execute interpreting type description as is done in the following two versions.

Generation through Macros

Generating a different ancillary function for a different argument type is beneficial in terms of the execution speed but it is disadvantageous in terms of code size. Thus a method which use macros utilizing the information obtained by *__p3l_collect_ptr()* function was implemented. For example, given the macro definition below,

```
#define DEF_SWIZZLE_STRUCT(T)  \
    void __p3l_swizzle_struct( T * p ) {       \
        REF_SET  s;                            \
        __p3l_collect_ptr( &s,p );             \
        __p3l_generic_swizzle( (void *)p, &s );\
    }
```

inserting DEF_SWIZZLE(Person) to a source code generates the ancillary function for the type *Person*. The *__p3l_generic_swizzle()* is a runtime library function, which accepts the description of locations of references and performs the reservation check on them. There exists only a single version of the function. This contributes to the reduction of code size because generated ancillary functions are smaller and they are explicitly generated only once per program by the programmer. However, it was felt that the need of explicit instantiation was a retrogression. Also, the use of macros make debugging a nightmare. It was understood that exploiting template function definition will delegate the responsibility of implementing an efficient instantiation mechanism to a compiler writer. However, there was one problem for which no solutions were found. That is, it was felt that generating a type identity within a template function was impossible without the use of the runtime type identification mechanism put in the draft ANSI C++ standard (i.e. *typeid()* operator). Unfortunately, G++ does not implement the operator yet and adding the functionality did not seem trivial. In case of a macro, treating macro parameter as a string (i.e. #T) suffices.

Generation through Templates

A template function is characterized by the *template* keyword and type parameters. The following is an example of a template function.

```
template < class T > T max(T& a, T& b) { return a<b ? b : a; }
```

This provides generic definition of the function *max()* for arbitrary type *T*. A template function definition itself does not generate any code. A template function is implicitly materialized when a compiler sees the need to, usually through the use of the function. For example, the expression "i = max(127, i);" triggers the generation of *int max(int&a, int&b)* function. A generation of a specific version of a template function for a certain type is termed *template instantiation*. Different C++ compilers employ different instantiation mechanism. However, they all provide the control over the instantiation. Most of them allow explicit instantiations too.

In the current version of the *P3L* compiler, a template function is defined for swizzling structures (reservation checking structures).

```
void __p3l_sw( T * p ) {
   __p3l_generic_sw( TexasWrapper<T>::texas_unique_class_name,
                     &TexasWrapper<T>::texas_unique_cached_type_id,
                     p);
}
```

Because of the modification made to the GCC compiler, a call to *_p3l_sw()* is generate when the compiler sees a code like $*s = *t$ where s and t are a pointer to a structure with embedded references. This triggers the instantiation of the template function.

Interesting are the first two arguments to the runtime library function _p3l_generic_sw(). *TexasWrapper* is a template class which has *texas_unique_class_name* and *texas_unique_ cached_typ* as its static data members. The use of the latter member is described in section 5.4. The type of the former is *char ** and it is a pointer to the name of *T*. Use of this wrapper class provides a way to map the type used for an instantiation to the type name. The only remaining piece is how to initialize the data member with the correct type name.

C++ decorates function and class names with the types of arguments and the type of the return value in case of a function, when they are emitted to an assembler source. Thus a decorated name looks like:

```
__t12TexasWrapper1ZP7Connect$texas_unique_class_name
```

From this, it is known that *TexasWrapper* class was instantiated with regard to the *Connect* class.

After compilation of a module, the object module is inspected by a utility called *pnewfind* to find symbols which look like the above one. The set of those names found in the object module is put in a file associated with the module. Before linking, those files are merged by *tnamemap* and the following declaration is generated for each type found. XXX is assumed to be one of the names of the types.

```
char *
TexasWrapper< XXX >::texas_unique_class_name = "XXX";
unsigned int
TexasWrapper< XXX >::texas_unique_cached_type_id = (unsigned int) -1;
unsigned int
TexasWrapper< XXX >::texas_unique_cached_hash_val = (unsigned int) -1;
```

This mechanism for obtaining a type name within a template function was devised and utilities were developed by the implementors of Texas Persistent Store. The current *P3L* implementation borrows them.

4.3.4 Runtime Type Identification and Type Information

Runtime Type Identification

The *P3L* runtime system must be able to tell the dynamic type of a persistent object in order to handle object faulting properly. Figure 4.5 illustrates the reason why it must. The expression *p→m_father* in the body of the function may trigger an object fault. Since the apparent type of the expression is *Person **, calling *_p3l_fetch_obj(Person * p)* seems to suffice when the reservation check fails. As stated previously, however, the actual type of the object being pointed by *p→m_father* may be different from *Person*

due to the polymorphic type compatibility rule. Thus an application written using *P3L* must specify the type of an object supplying its type name. Section 5.4 details the issue. The need for explicit specification of an object type will be elided once the runtime type identification (RTTI) is incorporated into the C++ compiler. With RTTI supported, given a pointer to a type *T*, the name of the type name can be obtained through the expression *typeid(*T).name()* .

```
struct Person {
    Person * m_father;
    Person * m_mother;
    char * m_name[16];
    int age, salary;
};
struct Programmer : public Person {...}
struct Teacher : public Person {...}
int is_fater_called_jeff(Person * p)
{
  return !strcmp(p→m_father→m_name, "JEFF");
}
```

Figure 4.5: Polymorphic type and RTTI

Runtime Type Information

Runtime Type Information is meta information about characteristic of types, which is made available to a program at runtime. By utilizing type information, the running program can access its data abstractly, i.e. it can enumerate name and type of data members in a structure, calculate the size of an instance, obtain printable name of the

type of an object, etc. Although the current implementation of *P3L* maintains information on the pointer locations only, availability of a rich type information aids in improving programmer's productivity. Most of the other object oriented language such as Eiffel, Smalltalk provides the information through a standard library with cooperation from the compiler. Unfortunately there is no standard on runtime type information in C++. Thus the GCC compiler had to be modified to add this capability initially and later debugging information emitted by the compiler is utilized to extract runtime type information from.

The function *_p3l_collect_ptr()*, which is automatically emitted by the compiler for each type defined, may be considered to be very primitive type information.

On the other hand, the current *P3L* implementation extracts the same information from the debugging information attached to an application using utilities provided in the Texas Persistent Store distribution. Debugging information is produced by specifying -g flag when executing the compiler. Debugging information is description on functions, variables and types used by a program, using which a debugger can, for example, locate the position of a variable given its name, access it, and print the contents of the object in human readable format. Figure 4.6 taken from the Texas Persistent Store distribution illustrates the flow of meta data in the Texas Persistent Store and *P3L*. *.td is the type information for each object module, which are to be combined into *.tdf at the linking phase.

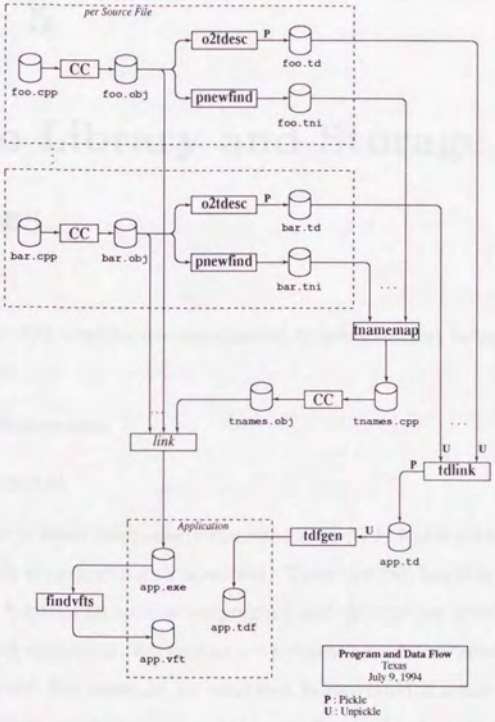


Figure 4.6: Flow of Data (from the Texas PS distribution)

Chapter 5

Runtime Library and Storage Managers

An interface to the *P3L* runtime system exposed to programmers belongs to either of these two categories:

1. Transaction Management
2. Object Management

The set of interface to these components constitutes an API, Application Programming Interface, of the P3L programming environment. These are call-based interfaces. There is another interface between an application program and the runtime system. That is, the interface for handling exceptions. Exceptions are notifications of error conditions detected by the runtime library. For example, an exception is generated if a non-existing object is requested from database. A conventional strategy for notifying an application of such erroneous condition is to encode the error status in the value returned to the application from the callee. However, in the context of P3L, some runtime library functions are called by instructions implicitly emitted by the compiler. This means, the function calls corresponding to the emitted instructions do not appear in the program text. Thus there is no way to implement the conventional strategy of checking for errors resulting from

those calls. Thus exception handling is a crucial part of the design of the *P3L* runtime system.

Other components of the runtime system are also presented in this chapter. One of these components governs the mapping between a persistent identifier and a virtual memory address assigned to it in addition to the mapping between a virtual memory address of a lambda and the persistent identifier the lambda is representing. Although the components are invisible to programmers, they bear significant importance. At the end of this chapter, the storage managers interfaced to the runtime system are reviewed.

5.1 Transaction Management

In the *P3L* system, updates to persistent objects are performed atomically. That is, either all of the updates in a transaction are recorded in the persistent storage or else none of them are ever permanently recorded. A transaction is a sequence of read and write operations which are explicitly blanketed in a section of code. The section is dynamically marked by calling APIs, not by a statically defined code block. Both functions are member functions of the class StorageManager. If and only if a transaction reaches a completion status and the updates performed during the transaction do not conflict with those performed, the effect of updates are permanently reflected in the database.

An application must declare the start of updates by calling *StartTransaction()*. The start of transaction is not implicit, that is, an application must explicitly call the *StartTransaction()* function in order to start manipulating persistent objects. The function is a member function of the class StorageManager. Similarly, an application must tell the runtime system of the end of a transaction through a call to another function. The name of the function is *CommitTransaction()*, which is also a member function of the class StorageManager. If the application wants its updates cancelled, it should call the *AbortTransaction()* member function of the same class. Updates made to database in a transaction will take effect if the call to CommitTransaction() completes successfully.

A transaction is aborted if the application running the transaction calls *AbortTransaction()* or conflict in access to items in the database is detected by the runtime system. The semantics of a transaction require either (1) undoing the effect of updates when a transaction aborts or (2) not updating the permanent copy of the database [1]at all until the transaction successfully commits. The Exodus Storage Manager adopts WAL (Write Ahead Logging)[33]. With WAL, updates are logged with a corresponding *before-image* so that updates can be undone during a rollback operation. A commit operation completes as soon as all log records for the transaction are forced into the log. The Texas Persistent Store also adopts WAL. However, no before-image are logged in the Texas Persistent Store. This is possible for the Texas Persistent Store because an access to a single page is allowed only for a single transaction and no updates to the permanent copy are performed until the transaction commits. In some cases, such as abrupt power failure or a system crash, an application terminates before it calls either *CommitTransaction()* or *AbortTransaction()*. In such an event, the storage system must perform a recovery operation, which is similar to rollback except that dirty pages are not available either in volatile store or in stable storage.

When more than one transaction is running, the order of read and write operations to the database from different transactions must be coordinated so that the consistency of the database is preserved. The general approach is to ensure that interleaved operations from different transaction forms a serializable history. A serializable history is a sequence of read and write operations which can be produced by serial execution of participating transactions. The task of ensuring a serializable execution is called *concurrency control*. The *P3L* runtime does not implement its own concurrency control scheme, but rather delegates the responsibility to an underlining storage manager. A storage manager can implement its own concurrency control method. Also, it is legitimate for a storage manager to implement no concurrency control mechanism if applications do not require it.

[1]Part of the stable storage apart from the log area.

Implementing transaction atomicity is optional as well although both storage managers available now implement it.

An operation which violates serializable history may be detected during a transaction long before the transaction attempts to commit. The implication of this is that a transaction abort may occur anytime during program execution as a result of accessing a persistent object. The conventional method of error checking and handling is not applicable in this situation because an access to a persistent objects occurs implicitly, i.e. because a programmer has not written I/O operations explicitly in the program. Thus P3L provides support for a primitive[2] exception handling mechanism, which is described in the next section. A programmer associates a handler (called an *exception handler*) to a specified block of program. Transaction aborts detected during the execution of that block will be signaled by the runtime as an error and control will be passed to the exception handler.

5.2 Exception Handling

5.2.1 Rationale

The runtime library of P3L signals transaction abort through the use of an exception handling mechanism. Due to the availability of this mechanism, an application does not have to repeatedly check the completion status of library calls. This is especially important for the API which is called implicitly as a result of failed reservation checks. Consider the code fragment in Figure 5.1. The assignment expression marked with the comment may cause object fault if the target object is not resident.

If the target object is assigned a proper address and made resident, the yielded value of the expression is the address. However, the process of making the target resident may

[2]Draft ANSI C++ standard defines a standard exception handling and it requires that destructors be called regarding objects created on stack when the stack frames are rewound due to an exception.

fail due to a transaction conflict or network failure, for example. In that case, what value should the expression yield? Returning NULL forces the iteration loop to terminate incorrectly. The P3L system solves this problem through the introduction of an exception handling system. In preparation for exceptional conditions such as network failure, an application installs an *exception handler* for a block of code during which an exception may occur. If the P3L runtime system detects an exception condition while executing, the runtime system looks for an exception handler installed to handle that exception and passes control to that exception handler. If a handler is not installed, the transaction is aborted and the entire application terminates.

```
struct LinkedListNode {
    LinkedListNode * m_next;
    int m_value;
};

int sum_up( LinkedListNode * p )
{
    int v = 0;
    while( p ) {
        v += p→m_value;
        p = p→next;                    // may cause an object fault
    }
    return v;
}
```

Figure 5.1: An expression which may cause an object fault

```
void show_sum(void) {                    // assume the transaction already started.
    int v = 0;
    int volatile retry = 0;
    LinkedListNode * p;
    do {
        TRY {
            p3l_bind_root( &p, theSM, "LinkedListRoot" );
            v = sum_up( p ); retry = 0;
        } CATCH( SmException,e ) {
            if( e.m_sm_errno==SmException::xact_abort ) {
                theSM→StartTransaction();
                retry = 1;
            } else THROW_LAST();
        } END_CATCH
    } while( retry );
    CString sv; sv.itoa( v );
    MessageBox(NULL, "Sum of values", sv, MB_OK);
}
```

Figure 5.2: An exception handler installed around a code block.

Figure 5.2 shows an example of an exception handler installed around the call to the function described above. Because an exception may be raised during the execution of the call to the function *sum_up*, the statement bracketed *TRY* and *CATCH* is protected by the exception handler bracketed *CATCH* and *END_CATCH*. When any exception which is an instance of the class *SmException* is raised during the execution of the protected block, control is passed to the exception handler. The handler examines the exact cause

of the exception by looking at the *m_sm_errno* data member. If the exception is cause by a notification of transaction abort from the storage manager, the handler sets the variable *retry* to 1 so that the operation in the protected block is retried. Otherwise, the exception is re-thrown and an enclosing exception handler is sought.

5.2.2 Implementation

Managing Stack of Contexts

The exception handling system is implemented using the setjmp/longjmp functions in the standard C library. Because exception blocks can be dynamically nested, a stack of jmp_buf must be maintained. The top of the stack is maintained in a variable called *exc_exception_frame*, which is a pointer to an instance of the *EXC_LINK* class. The *EXC_LINK* class has three members, *m_myself*, *m_up* and *m_jbuf*. The *m_myself* member variable is initialized to point recursively to the containing object so that corruption of the object can be detected. Processor context is saved in *m_jbuf* through a call to the setjmp routine when a block protected by an exception handler is entered. A stack of *EXC_LINK* instances is maintained by means of the *m_up* data member. The organization of the stack is depicted in Figure 5.3. When an exception is raised, the runtime system looks at the *exc_exception_frame* global variable, locates the *EXC_LINK* instance containing the most recently saved context and rewinds the execution up to the context.

Several macros, shown in Figure 5.4 and 5.5 are provided for performing the operations described above. Those in Figure 5.4 are for throwing exceptions by an application. The P3L runtime system utilizes the exception handling mechanism extensively and applications are also allowed and encouraged to do the same. The first argument to the *THROW* macro is an exception object, which must be an instance of a subclass derived from the *Exception* class. This object becomes the *current exception object* and is remembered in a data member (*exc*) in the global object(*exc_curr_exception*). Another data member in the object reflects the second argument to the macro. The second argument is a flag which

indicates whether the exception object specified in the first argument must be destroyed when all the exception handlers for the exception are completed. If this flag is true, the *current exception object* is destroyed when the *EXC_LINK* object is destroyed.

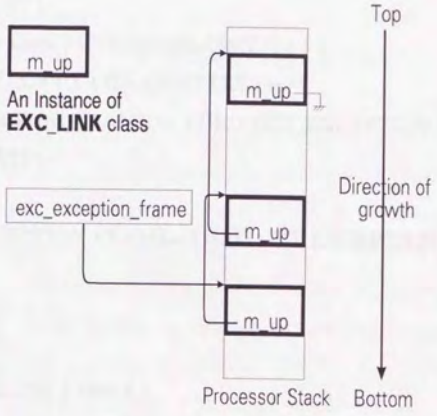


Figure 5.3: The stack of Exception Frames

```
#define THROW(E,D) do { exc_set_exception_context(__FILE__, __LINE__, (E), D);\
                        Throw(); } while(0)

#define THROW_LAST() do { Throw(); } while(0)
```

Figure 5.4: Macros for Exception Handling(1)

```
#define TRY \
{ EXC_LINK lnk; \
    if( setjmp( &lnk.m_jbuf[0] )==0 ) {
#define CATCH(C,V) \
} else if( exc_match_class(RUNTIME_CLASS(C)) ) { \
C * V = (C *)THRD_GET_CURR_CONTEXT.exc; \
THRD_GET_EXCEPTION_FRAME = THRD_GET_EXCEPTION_FRAME→m_up;
#define END_CATCH \
 } else { \
  THRD_GET_EXCEPTION_FRAME=THRD_GET_EXCEPTION_FRAME→m_up;\
  Throw(); \
 } \
}
#define CATCH_ALL(V) } else { \
Exception * V = (Exception *)THRD_GET_CURR_CONTEXT.exc;\
THRD_GET_EXCEPTION_FRAME = THRD_GET_EXCEPTION_FRAME→m_up;
#define END_CATCH_ALL }}
```

Figure 5.5: Macros for Exception Handling(2)

The macros shown in Figure 5.5 are used to construct an exception block and corresponding exception handlers. The *TRY* macro creates a *EXC_LINK* instance on the stack and saves the current processor context in the jmp_buf embedded in the instance. Because the *EXC_LINK* instance is constructed on the stack, the creation is a cheap operation although saving the context via setjmp() is a rather expensive operation especially on RISC processors such as SPARC. The macro is defined in such a way that an exception block is treated by the compiler as the *then*-part of the *if* statement in the

macro. The *CATCH* macro first closes the *then*-part and creates its own *else if* clause. The conditional part of the statement examines the type of the exception raised so that only relevant exceptions are handled by the handler. The class of exceptions to be caught by the exception handler following the *CATCH* macro is specified in the first argument. And the name of the variable to be defined and bound to the *current exception object* is specified in the second argument. The mechanism for determining the class of an exception is described in the next section. When the class matching succeeds, the handler which is compiled as the *then*-part is executed. Before control falls through to the handler, the exception stack is popped so that re-raising the exception in the handler passes control to an enclosing exception handler if there is any. The *END_CATCH* macro closes the blocks which have been opened by the *TRY* macro. The macro re-raises any exceptions which have not been handled by the preceding handlers. *CATCH_ALL* macro is quite similar to the *CATCH* macro except that the corresponding handler is unconditionally executed. The chaining of exception handlers via the *THROW_LAST()* and *CATCH_ALL* macro is useful for guaranteeing the release of resources even when the operation terminates abnormally. Figure 5.6 shows an example of this usage. The exception handler catches any exception raised during the execution of the *do_some_work(p,sz)* function, releases the memory chunk allocated for the work and jump to an enclosing exception handler.

```
void anOperation(size_t sz)
{
    char * volatile p = new char [sz];
    TRY {
        do_some_work( p, sz );
    } CATCH_ALL( ex ) {
        delete [] p;
        THROW_LAST();
    } END_CATCH_ALL
    delete [] p;
}
```

Figure 5.6: Releasing Resource upon Exception.

Determining The Class of an Exception

The exception handling macros rely on the ability to determine the type of the *current exception object*. The RTTI mechanism developed for handling persistent objects could have been utilized. However, a class library based approach was taken so that RTTI would work on statically allocated objects. The implementation was rather easy because the base class *Exception* is assumed for all exception objects.

The class definition of *Exception* includes a virtual member function which returns the type identification. The identification is a pointer to an instance of the class *Class* which is created as a static data member of an exception class. An instance of the class *Class* contains two data members. One is a pointer to another class *Class* instance related to the super class of the exception class. The information on the class derivation must be available to facilitate selection of the exception handler based on the class of the *current*

exception object. The other data member of the class *Class* is the name of the exception class it represents. The availability of printable names aids in building the diagnostic messages given an exception object.

```
/* Exceptions caused by failed lookup of tables */
class LookupException : public Exception {
    DECLARE_DYNAMIC(LookupException)
  public:
    virtual void asString( char * bufp, size_t sz );
    LookupException() : Exception(0,0) {}
    ~LookupException(){};
};
/* The following is found in p3lexc.cc */
IMPLEMENT_DYNAMIC(LookupException, Exception)
```

Figure 5.7: Declaring and Implementing an Exception class

Two macros must be used to declare and define the *Class* class instance in an exception class. These are the *DECLARE_DYNAMIC* and *IMPLEMENT_DYNAMIC* macros. *DECLARE_DYNAMIC* must be used in an exception class definition to declare an instance of class *Class* as a static data member of the exception class. *IMPLEMENT_DYNAMIC* defines the static data member, initializing it appropriately. This macro must be put in an implementation file rather than in a header file. An example of the use of these macros is shown in Figure 5.7.

The logic of matching the class of the *current exception object* and the class specified in a *CATCH* macro is listed in Figure 5.8. The auto variable *c* is initialized to point to the *current exception object*. The virtual member function *GetRuntimeClass()* defined in class *Exception* returns a pointer to the corresponding class *Class* instance. Another

method of accessing class *Class* instances is to directly specify the name of the class whose class object is requested. The *RUNTIME_CLASS* macro, which takes a class name as its sole argument is provided for this purpose, as can be observed in Figure 5.5.

```
int exc_match_class( Class * target_cls ) {
    Exception * c = THRD_GET_CURR_CONTEXT.exc;
    const Class * exc_obj_cls;
    exc_obj_cls = c→GetRuntimeClass();
    int found = 0;
    do {
        if( target_cls == exc_obj_cls )
          { found = 1; break; }
        // reached the root?
        if( exc_obj_cls==exc_obj_cls→m_super ) break;
        exc_obj_cls = exc_obj_cls→m_super;
    } while(1);
    return found;
}
```

Figure 5.8: Matching with the class of an Exception

5.2.3 Limitations

Due to its being based on C macros, the exception handling mechanism has several of limitations:

- Exceptions to be thrown must be derived from the class *Exception*. Exception handling in the ANSI C++ draft standard specifies that an exception of any type can be thrown and caught.

- Class declarations must be furnished by *DECLARE_DYNAMIC* for the type identification mechanism to work. *IMPLEMENT_DYNAMIC* must be added to an implementation file. Mistakenly swapping the two arguments to the latter macro results in a disaster or obscure bugs.

- Auto variables used both in the exception block and the exception handlers must be declared volatile so that the compiler does not assign these variables to a register. This is required because the values of all registers are not restored when the stack is rewound. Certain compilers, Gnu C Compiler for example, detects the use of setjmp() and warns about the use of non-volatile variables.

- Since exception handlers are just disguised *if-else-if* clauses, the sequence of *CATCH* is very important. For example, consider two exception classes, *LookupException* and *InvalidKeyException*. *InvalidKeyException* is a subclass of *LookupException*. Figure 5.9 show a probable erroneous program fragment. The programmer's intent could have been to catch the two exceptions individually and to respond to each of them differently. However, the evaluation of *CATCH* clause is performed from top to bottom at runtime and also *InvalidKeyException* confirms to *LookupException*. Thus an *InvalidKeyException exception* gets handled by the first exception handler.

- When the stack is rewound as a result of throwing an exception, stack frames between where the exception is raised and where the stack is rewound are discarded. However, destructors of objects which have been constructed on the stack frames are not called. A programmer must install an exception handler in order to force the execution of the destructor or devise another cleanup mechanism. Exception handling defined in the ANSI C++ draft standard requires the destructors to be called automatically.

- Because setjmp() is called every time an exception block is entered, and partly because the exception stack must be maintained, defining exception handlers within a

tight loop adversely affects the execution speed of the loop. Certain C++ compilers avoid such problems by constructing a table which associates a program counter value to statically enclosing exception handlers and by stack-walking dynamically at runtime to find dynamically enclosing handlers. This is possible only if exception handling is defined by the language and not by the library.

```
TRY {
    // ... do something
} CATCH( LookupException, e ) {
    // handle    lookup Exception ...
} CATCH( InvalidKeyException, e ) {
    // handle InvalidKeyException ...
} END_CATCH
```

Figure 5.9: A misuse of the CATCH macros

5.3 Address Translation

The P3L runtime library maintains three maps which are utilized in translating various addresses:

- A mapping from a virtual address of lambda to a corresponding persistent identifier.
- A mapping from a persistent identifier to the virtual memory address assigned to the object identified by the identifier.
- A mapping from an arbitrary virtual memory address to a pair consisting of a persistent identifier and an offset within the object.

5.3.1 Dealing with Interior Pointer

The first two mappings are realized with simple hash tables with conflict resolution handled by chaining. The third mapping is maintained as a search tree. In order to prevent the tree from degenerating into a list, which pushes the average search cost to $O(n)$, the tree is implemented as a self balancing tree (AVL-tree). Because the underlining memory allocator aligns dynamically allocated memory chunks at 8 byte boundaries, maintaining a hash map from each block to the top of the containing object may seem to be an alternative. However, this scheme requires one map entry per 8 byte block used. This means an object as large as 1M would requires 1M/8 hash table entries. It is enough of a burden to insert and remove entries as large as 128K, without overhead in the storage requirement. Thus the scheme is not feasible unless objects are extremely fine grained or alignment is coerce enough. As is described in the next chapter, the Texas Persistent Store employs the scheme with page alignment.

```
static char array[MAX_SIZE];
char * p = array;
char * interior1 = &array[4];
char * interior2 = p + 3;
```

Figure 5.10: Creation of an Interior Pointer

The requirement that an arbitrary virtual memory address be resolved to the address of the top of the containing object is unique to implementations based on C/C++ language. Unlike most other programming languages, C/C++ permits the use of a kind of reference so called a *interior pointer*. An interior pointer is created through an arithmetic operation on a pointer or through indexing an array. An interior pointer points to the middle of an object instead of its top. Two examples of interior pointer creation are illustrated in Figure 5.10. Because not all storage managers support the use of interior pointers, for

example Exodus Storage Manager, the runtime system must convert an interior pointer to an object identifier and offset within the object.

5.3.2 Organization of the Translation Table

An instance of a parameterized class *OLmapEnt* is utilized for organizing both the 'virtual address to the persistent identifier map' and 'persistent address to virtual address mapping'. These mappings are kept in a two way mapping table which is an instance of the class *OidLidMap*. This class is also a parameterized class. These two classes are parameterized using the template facility of the C++ language so that the difference in the structure of an OID implementation by storage managers can be dealt with and the type of local identifiers can be altered if the need arises. The table is consulted when a pointer is swizzled and unswizzled (Figure 5.11). At fault time, the runtime system first tries to fully swizzle references embedded in the faulted in object. In that attempt, the runtime system consults the instance of *OidLidMap*, (*theOidLidMap*), to see whether the virtual memory address is reserved for the target object. If a corresponding virtual memory address is found, then the reference is fully swizzled with the address. A failed lookup operation means address reservation has not been performed for the object and the runtime system installs a lambda in place of the partially swizzled reference as well as updating the *Addr2Oid table*.

OidLidMap is actually a container of anchor points for the table. The table is populated by adding instances of the class *OLmapEnt*. The table is updated with a new entry when address reservation is performed for an object. The definition of the classes are shown in Figure 5.14 and 5.12. A data member of the class *OidLidMap*, (*lid_tree*), points to the first node of the tree. Another data member *oid_chain* is an array of pointers which are used to maintain hash buckets. (Figure 5.13).

As can be observed in Figure 5.12, an *OLmapEnt* instance does not have pointer fields with which to link the instances to form the tree and the hash chain. Thus the runtime

library creates an instance of the class instantiated from the following parameterized type and copies an instance of *OLmapEnt* to the corresponding part of the object, when populating the table with a new instance. The type of the instance is

```
AvlNode< SListNode< ListNode< ListNode< OLmapEnt <O,L> > > > > >
```

where O is a template parameter which stands for the type of the persistent identifier and L is for the type of a local identifier. The current *P3L* implementation uses a raw pointer as a local identifier.

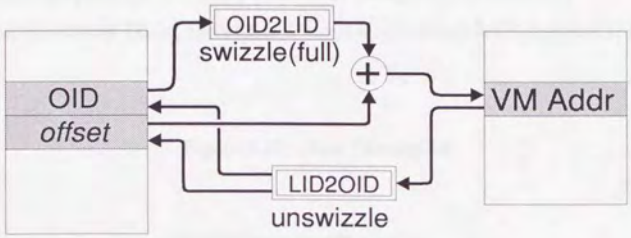


Figure 5.11: Two way mapping between a virtual address and a persistent identifier

```
template <class O, class L> struct OLmapEnt {
    O oid;                                  // OID for a persistent object
    L lid;                                  // LID assigned to the object
    size_t size;                            // the size of this chunk
    C_TYPE type;                            // type of the object
    int tentatively_created;
    OLmapEnt() { }
    OLmapEnt(const O& o) : oid(o) { size = 0; tentatively_created=0; }
    OLmapEnt(const L& l) : lid(l) { size = 0; tentatively_created=0; }
    OLmapEnt(const O& o, const L& l, size_t size_):oid(o),lid(l),size(size_){}
};
```

Figure 5.12: class OLmapEnt

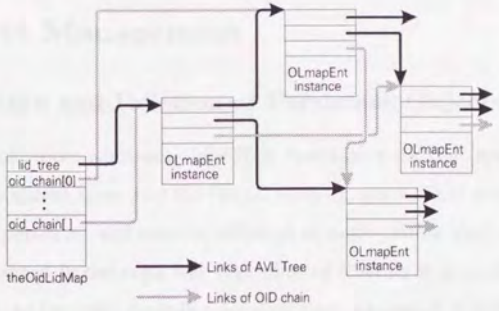


Figure 5.13: Two way mapping between virtual address and persistent identifier

```
template <int size, class O, class L> class OidLidMap {
    AvlTree< SListNode< ListNode< OLmapEnt< O, L> > > > lid_tree;
    SList < ListNode< OLmapEnt< O, L> > > oid_chain[size];
  public:
    List < OLmapEnt< O, L> > creation_order;
    OidLidMap& add( const OLmapEnt< O, L >& e );
    OidLidMap& remove( const OLmapEnt< O, L >& e );
    const OLmapEnt< O,L >& operator [] (L lid);
    const OLmapEnt< O,L >& operator [] (O& oid);
    void Lid2OidAndOffset(L lid, O * o, offset_t *off);
};
```

Figure 5.14: class OidLidMap

5.4 Object Management

5.4.1 Creation and Deletion of Persistent Objects

Implementation of object addressing in *P3L* is based on a multiple space model, which consists of the persistent space and the virtual memory space. That means there are two kinds of objects, persistent and volatile, although accessing either kind of object through a pointer is performed in the same way regardless of whether it is persistent or volatile. Distinction needs to be made regarding creation and deletion of objects. Creation of a volatile object is performed exactly as in an ordinal implementation of C++. Persistent objects are created by promoting volatile objects. The promotion can be requested by calling the *p3l_make_persistent()* function. The function takes two arguments. The first argument is the virtual address of the object to be made persistent and the second argu-

ment is the number of elements in case an array object is being made persistent. Since the second argument as a default value of 1, it can be omitted unless an array object is being dealt with. Objects of any type can be made persistent through this interface. The function is parameterized and is shown in Figure 5.15. The use of the *TexasWrapper* class is explained in section 5.5.

```
template < class T >
void p3l_make_persistent( T * addr, int nArray = 1 )
{
    _p3l_make_persistent(addr,
                         TexasWrapper<T>::texas_unique_class_name,
                        &TexasWrapper<T>::texas_unique_cached_type_id,
                        nArray);
}
```

Figure 5.15: Implementation of the p3l_make_persistent() function.

The Texas Persistent Store employs a similar but different approach. The global operator *new* is overloaded by providing the following declaration in the header file. Because the operator is rather complex, an accompanying macro is provided to ease the creation of persistent objects.

```
/* declaration of overloaded new */
void *operator new (size_t size, ObjStore *store,
    char *typename, unsigned int *cached_type_id,
    unsigned int *cached_hash_val, size_t element_size);

/* macro for persistent object creation */
#define pnew(ps,type) ::new(ps,\
```

```
TexasWrapper< type >::texas_unique_class_name,\
&TexasWrapper< type >::texas_unique_cached_type_id, \
&TexasWrapper< type >::texas_unique_cached_hash_val, \
sizeof(type))\
type
#define pnew_array(ps,type,n) ::new(ps,\
TexasWrapper< type >::texas_unique_class_name, \
&TexasWrapper< type >::texas_unique_cached_type_id, \
&TexasWrapper< type >::texas_unique_cached_hash_val, \
sizeof(type), n) type[n]
```

This operator is called in order create a persistent object. Thus there is no promotion as in *P3L*. This approach is more type-safe because the type of a persistent object is specified when the object is created. The method taken in *P3L* is safe as long as *p3l_make_persistent()* is called with the pointer initialized with a new operator and the apparent type of the pointer exactly matches the return type of the new operator as is illustrated below.

```
Employee * p = new Employee;
p3l_make_persistent( p );
```

This implies that, in the presence of a polymorphic type compatibility rule, the type unsafeness may be revealed through incorrect program behavior. Suppose that the Employee class is a subclass of the Person class. Then the following declaration of the pointer variable 'p' is type safe. The compiler does not complain at all and this portion of the program runs without any problem.

```
Person * p = new Employee;
```

However, calling p3l_make_persistent() with p as an argument will exhibit incorrect runtime behavior because the object pointed to by 'p' will be written to the storage manager

as an instance of the Person class and later retrieved as an instance of the same class. In order to alleviate this problem, the *P3L* runtime system could have implemented checks that the mismatch could be detected at runtime. Unfortunately, implementing such checks requires implementation specific knowledge about the underlining memory allocator. And it could be unable to detect errors all of the time. For example, an array containing 4 instances of class Person cannot be distinguished from an array containing 2 instances of class Employee if the size of the latter is twice as large as the former.

```
template < class T >
void p3l_bind_root(T ** rootPtr, StorageManager *sm, const char *name,
                   int nArray = 1)
{
    if( sm→rootExists(name) ) *rootPtr = (T *)sm→getPersistentRoot(name);
    else {
        * rootPtr = new T;
        tDesc→zerofill( TexasWrapper<T>::texas_unique_class_name,
                        &TexasWrapper<T>::texas_unique_cached_type_id,
                        *rootPtr, nArray );
        // tell the runtime about the type of the Root
        p3l_make_persistent(*rootPtr, nArray);
    }
}
```

Figure 5.16: p3l_bind_root template function

Regardless of the problem outlined above, the current implementation keeps the method of persistent object creation. The reasons are as follows:

- Whether an object should be made persistent or not may not be determined at

the time of creation and creation of a persistent object consumes significantly more memory than a volatile object in the current *P3L* implementation.

- Having functioning *p3L_make_persistent()* means that defining an overloaded new operator similar to the Texas Persistent Store is trivial if strict static type safeness is desired.

- RTTI (Runtime Type Identification) in the draft ANSI C++ standard is becoming common place among C++ compilers. By utilizing this facility, strict runtime type checking can be implemented in a portable way.

When a persistent object is no longer needed, it must be destroyed explicitly by an application unless the storage manager in use supports garbage collection of persistent objects. In order to make an application portable over the use of different storage managers, explicitly destroying objects is recommended. The function *p3L_delete()* must be used to destroy a persistent object. Calling the function removes the persistent storage allocated for the object as well as any copy in the physical storage. The destructor is called if it is defined for the class of the object. Being able to delete every object through the standard *delete* operator would have been a desirable feature. However, achieving that goal requires modification to the implementation of the *delete* operator and the modification may become platform specific with regard to application of destructors on an array object. In order to keep the implementation of *P3L* portable, the two delete operations were not unified.

5.4.2 Root Object Management

One of the benefits of using a persistent object system is the ease of exchanging complex data between applications. Complex data structures can be passed from one application to another easily because the relationship between objects via reference is preserved by the system.

One provision to be made is establishing a standard way to identify objects. Many persistent object systems base object identification on names given to objects. *P3L* is no exception. *P3L* runtime library provides an API with which to assign a name to an object. In the following an object given such a name is called a *root object*.

A root object can be retrieved from the database by specifying its name in a call to the *p3l_bind_root()* API. The API is implemented as a template function defined in a header file provided by the *P3L* implementation. The function utilizes two virtual member functions which form part of the storage manager interface, namely *rootExists()* and *getPersistentRoot()*, as seen in Figure 5.16. Some storage managers implement these routines. The runtime system emulates these functions if they are not provided by the storage manager to be used. The function *p3l_bind_root* is parameterized in order to have the function create a persistent object of the requested type automatically. Without such facility, programmers must determine the existence of a root object first to decide whether it can be retrieved or it must be created. An automatically created object is initialized with zero mimicking the behavior of statically allocated objects. In this process, meta information is utilized to determine the part of an object which needs to be cleared. An example of fields which should not be initialized is a vtbl[3] pointer field. Thus the *p3l_bind_root()* function always binds the pointer whose address is specified in the first argument to a valid heap address. Before an application terminates, it can call the *setPersistentRoot()* virtual member function implemented by a storage manager. The signature of the function is given below.

```
virtual void setPersistentRoot(const char *name, const void *ptr);
```

Calling the function does nothing if the object pointed by 'ptr' is already a root object. Otherwise, the name is assigned to the object. Finally, a name assigned to a root object can be detached by calling another virtual member function *rmPersistentRoot()* abstractly

[3]vtbl is an array of pointers to functions which is built by a C++ compiler for implementing dynamic dispatching to virtual member functions.

defined in the StorageManager class. The argument to be supplied is the name to be detached.

5.5 Type Management

The type of a persistent object is determined from the type of the first argument given to *p3l_make_persistent()*. If the type of the argument is 'Employee *', for example, then the object will be treated as an instance of 'Employee'. When recording the object in a persistent storage, the information about the type must also be recorded so that the type of the object is known when the object is retrieved later by the same or another program.

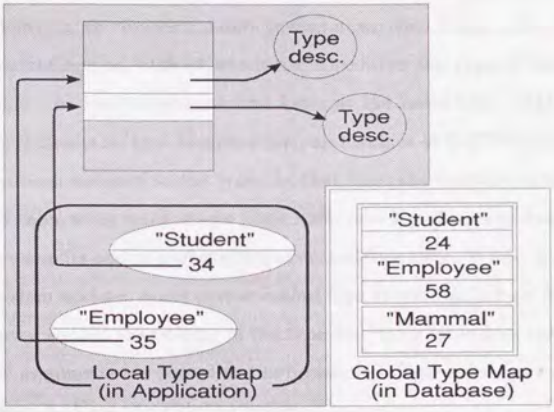


Figure 5.17: Type Maps maintained by P3L Runtime

A naive implementation may store the name of the type along with the contents of the object. However, using a name is disadvantageous in terms of storage requirement and efficiency. That is, a 32 bit integer would be enough to represent every type being

used among a suite of programs and table lookup is faster if the type name is specified as an integer instead of a string.

As described in section 4.3.3, the compiler and related utilities make an arrangement that, for each class T whose instances may be made persistent, a template class TexasWrapper<T> is instantiated and two of its fields (*texas_unique_class_name* and *texas_unique_cached_type_oid*) are initialized properly. The first field is initialized with the name of the class that the wrapper class represents and the second with -1. The pairs of the two fields collectively form a type map, which associates each class name with a unique integer. Each oval on the left of Figure 5.17 represents a TexasWrapper class instantiated from the TexasWrapper<T> templated class.

As explained in section 5.4, *p3l_make_persistent()* function calls *_p3l_make_persistent()* function. In doing so, the function passes *texas_unique_class_name* and a pointer to the *texas_unique_cached_type_id*, both of which are specific to the type of the object being made persistent. The runtime routine first looks at the latter field. If the value of the field is -1, then it means no type identifier (i.e., an instance of C_TYPE, which is defined to be 'int') has been assigned to the type. In that case, the runtime routine probes the type descriptor table using *texas_unique_class_name* as a key. The type descriptor table is created by the compiler and related utilities at compilation time. When an entry is found, the runtime system updates *texas_unique_cached_type_id* with an integer which identifies the relevant entry, so that the probing of the type descriptor table does not happen again when the same arguments are passed to the function. Integer values put in ovals on the left side of Figure 5.17 are these cached indices.

Having obtained an index into the type descriptor table, the runtime system utilizes this value for the purpose of type identification. A new *OLmapEnt* instance is created and its *type* member is filled with the index. An *OlmapEnt* instance associated with a newly promoted object has its *oid* initialized with the default value until the enclosing transaction commits. OIDs are assigned when objects are written to the persistent store instead of when they are promoted.

The type description table is specific to each application. Thus the above index value cannot serve as a global type identifier. Thus a persistent mapping between a type name and a global type identifier is maintained. The mapping is remembered in a special root object named "p3l_typeDb" (Exodus Storage Manager) or as meta data of the store (Simple Storage Manager). Shown on the right of Figure 5.17 is the global type map. The global type map is populated with an entry when an object of a certain type is first written to the persistent store.

When writing an object into persistent store, the runtime system obtains its local type identifier from the corresponding *OLmapEnt* entry in *theOidLidMap*. Then the corresponding type name is retrieved from the type description table and the global type map is consulted to see whether an entry in the global type map exists. If there is, the global identifier is retrieved from the map and prepended at the head of the object contents. Otherwise, an entry is added to the global type map and a global identifier is assigned before writing the identifier and the object contents.

In order to partially or fully swizzle references at fault time, the runtime system must have access to the relevant type description entry. The type description entry is located by the following two steps:

- The global type identifier is available at the header part of the data stream for the object being faulted in. An entry corresponding to the type identifier is available in the global type map, with which the runtime system translates the identifier to a type name. The global type map is hashed both on type name and identifier so that two-way translation is efficiently carried out.

- The type name is used to locate the relevant entry in the type description table, from which the necessary information is available.

5.6 Storage Manager Interface

One of the distinguishing feature of the *P3L* system is that it supports multiple storage managers. The runtime system defines the interface to a storage manager in an abstract base class called *StorageManager*. In this section, those interface functions are grouped and reviewed according to their category.

5.6.1 Transaction Control Interface

```
virtual void StartTransaction(int ac, char **av) = 0;
virtual void CommitTransaction()                 = 0;
virtual void AbortTransaction()                  = 0;
```

StartTransaction() virtual member function takes command line parameters as arguments so that various transaction specification can be controlled. For example, the database volume number to be opened can be specified in the command line and directly handled by the storage manager itself, when the Exodus Storage Manager is used. Variables holding transaction status information must be provided by a concrete subclass of the *StorageManager* class.

5.6.2 OID Manipulation

```
virtual void createOID(C_OID *oid, int sz)                                  = 0;
virtual void createOIDandWrite(C_OID * oid,const void *data,int sz) = 0;
virtual void destroyObject(const C_LID lid)                                 = 0;
```

There are two interfaces defined for object creation. These functions are not called from the runtime system directly. Instead, they are called from the CommitTransaction() function of the class *ESM* for interfacing with the Exodus Storage Manager and the class *SSM* for interfacing with the Simple Storage Manager. When all the references in a newly

create object are unswizzled, the *createOIDandWrite()* function can be utilized to create an OID and write the contents to the persistent store. However, in some cases an OID must be created in order for the contents of another object to be completely swizzled. For example, when two newly created objects refer to each other, forming a cycle in reference, an OID for either one of the two must be created. *createOID()* is used in this case. The function *destroyObject* is called from the *p3l_delete()* function.

5.6.3 Data Transfer Control

```
virtual void * map(const C_OID * oidp, offset_t start,
                   size_t * range,     C_TYPE * type,
                   void * p_advice=0)                        = 0;
virtual void unmap(void * oidp)                              = 0;
virtual void flush( const C_OID *oidp, const void * p, size_t sz) = 0;
```

These functions govern the transfer of objects between the persistent storage and the physical storage. The *map()* function assigns virtual memory space to a part of the object indicated by OID which is passed in the first argument. The part to be mapped is specified by the *start* and *range* arguments. Usually the value of **range* is initialized with -1 by a caller so that the *map()* function calculates the size of the object from its global type identifier. The type identifier is returned in the memory location specified by the fourth argument. In the current implementation of *P3L*, the contents of the object are transferred from the persistent store at this time. It would be beneficial to delay the transfer until a part of the object is accessed, especially when dealing with large objects. The *unmap()* function simply detaches the space and physical storage assigned to an object. The *flush()* function transfers the contents of the storage to the persistent store before detaching the space and physical storage.

5.6.4 Address Translation Management

```
virtual void  rmAddr2OidMapping( C_LID * paddr )                = 0;
virtual const C_OID&  xlatAddr2Oid( C_LID * paddr )             = 0;

virtual void  addOidLidMapping(C_OID& oid, C_LID lid, int n_array,
                               C_TYPE& t, int modified_state)   = 0;
virtual void  rmOidLidMapping( void *p )                        = 0;
virtual const C_OID& lookupOid( C_LID  l )                      = 0;
virtual       C_LID  lookupLid( C_OID& o )                      = 0;
virtual const C_LID  resolve_oid( C_OID * oid_p )               = 0;
```

The *rmAddr2OidMapping()* is called from the *p3l_sw_ptr()* function which is called when a lambda is discovered and swizzled. The function removes an entry from *Addr2Oid* table associated with an object. The argument specifies the virtual address of the object whose entry must be removed. A function for adding an entry to the table is not exposed since it is performed within the *map()* function. The *xlatAddr2Oid()* function is also called from the *_p3l_sw_ptr()* function in order to obtain an OID being represented by the lambda. The rest of the functions are for managing the *OidLidMap* table. One instance of the *OidLidMap* is maintained for each storage manager. These functions delegate the requests to the instance. The difference between the *lookupLid()* and *resolve_oid()* is that the latter allocates virtual memory space for the object designated by the OID if required.

5.6.5 Root Object Management

```
virtual void   setPersistentRoot(const char *name,
                                 const void *ptr)    = 0;
virtual int    rootExists(const char *name)          = 0;
virtual void * getPersistentRoot(const char *name)   = 0;
virtual void   rmPersistentRoot(const char *name)    = 0;
```

These four functions are for managing root objects. The *setPersistentRoot()* function attaches a name to a persistent object which is identified by the second argument. The *rootExists()* function tells whether a root object with the name exists. The *getPersistentRoot()* function maps the specified root object onto a virtual memory space and returns the address assigned to the object. The last function detaches a name from a root object.

5.6.6 Update Tracking

```
virtual void  mark_dirty( C_LID l ) = 0;
```

The current *P3L* implementation requires a programmer to call the *p3l_mark_dirty()* function in order to help the runtime system keep track of updates on objects. The implementation of the function calls the *mark_dirty* member function of the current storage manager.

5.7 Storage Managers

Currently two storage managers are available for use with the *P3L* runtime library. One is the Exodus Storage Manager developed at the University of Wisconsin as part of the Exodus extensible database project. The storage manager will be referred to as *ESM* in the rest of this chapter. ESM is a complete storage manager with support for concurrency control, recovery, large object support, distributed database, client-server architecture, etc.

Since the Texas Persistent Store embeds a storage manager within an application, a lighter weight storage manager than ESM was needed to perform closer apple-to-apple comparison between the Texas Persistent Store and *P3L*. The other storage manager, which is named SSM (Simple Storage Manager) was developed. The next chapter reports on the comparison.

Support for another storage manager is being planned. The *Kala* persistent server is a commercial product developed and sold by Penobscot Development Center. The server was used in developing a previous *P3L* implementation. The reason for planning the support for the server in the current version are (1) to refine the storage manager interface so that adopting a new server becomes easier and (2) to compare the performance of *Kala* to the other storage managers. Kala has interesting features such as the usage of pointer swizzling within the server and reorganization of persistent storage based on the use pattern. In the following, the implementation of SSM is described and the other two servers are overviewed.

5.7.1 Simple Storage Manager

The features set of SSM is very limited due to time constraints in development and its nature as a component for running the OO1 benchmark program in chapter 6. Unlike ESM which supports the client-server architecture, SSM is realized as a library which is to be linked with an application. Other features of SSM are enumerated below.

OID

An OID is 96 bit long. It consists of a volume number (32 bit) and a position within the volume. Each reference consists of an OID plus an offset into the target object in order to handle interior pointers.

Immutability

Objects managed by SSM are immutable. That is, an object receives a new identity when it is updated. That is, an updated image of an object and an image of a newly created object are appended to the tail of the store. Atomic update is guaranteed because SSM forces the updates to the disk surface before updating the MasterRecord at commit time.

Storage Reclamation

Since garbage collection of the persistent store is included in the prospective research topics, the storage manager does not support explicit deletion of objects. An image of updated object is appended to the tail of the store as well as an image of the newly created object. A separate scavenger is run periodically or when the free space is running low to recycle storage allocated to dead objects. Because getting at a root object is the only way for an application to make inroads into the database, dead objects can be identified by tracing references starting at root objects. Objects not reachable through any path for any of the root objects cannot be accessed, i.e. dead.

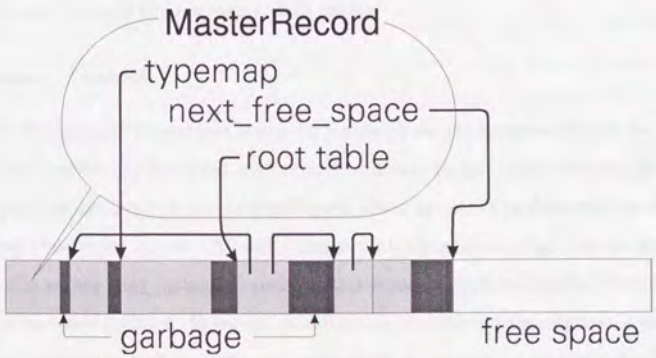


Figure 5.18: Organization of the persistent store in SSM

Meta Data Structure and Atomic Update

A persistent record named *MasterRecord* is kept at a fixed location in the store. This record includes the location of the global type map within the store. Other information included are the location of the root object table, the current size of the store, the next

global type identifier available for use, location of the tail of the store as shown in Figure 5.18. The *MasterRecord* is small enough to be contained within a single disk page and appends to the store are forced before an update is performed to the *MasterRecord*. Thus updates made by a transaction are atomically reflected in the persistent store.

Structure of the Type Map and Root Object Table

Both translation tables are stored as a chunk of storage with no structure at all. In the case of the type map, the chunk contains a *typename*, *global type identifier* pair. In the case of the root object table, the chunk contains a *a name* and *OID* pair to which the name is assigned. Both tables are read entirely, the database is opened and entries in each table are put into the respective hash tables.

Concurrency Control

SSM does not support concurrent access to the database yet because there is no synchronization mechanism implemented with regard to access to the *MasterRecord*. Aside from the synchronization issue on the *MasterRecord*, there are other problems to be solved for supporting concurrent access. Allowing concurrent access from single writer and multiple reader is rather easy once the synchronization mechanism is installed, thanks to the absence of in-place updates. However, coordinating multiple writes requires a substantial amount of code since lock management code must be written. Another potential problem is the fact that the OID of an object changes when the object is updated. Hence references in all the objects on the path from a root object to the updated object must be also updated. This would introduce more lock conflicts than an implementation with fixed OID would.

5.7.2 Exodus Storage Manager

ESM supports two kinds of physical structures, one is for small object that fit within a single page and the other is for large objects which do not. A large object is represented by a B+tree like index on byte offset within the object plus collection of leaf nodes which holds the object contents. The difference of the physical structure is transparent to the application. Unfortunately, advantage of the large object structure that insertion and deletion of bytes into the middle of an object is not taken because the runtime system does not provide interfaces through which an application can perform such operations. Each small object is represented by a (#page,#slot) pair. For large objects, the (#page,#slot) pair points to the object header.

To facilitate sharing, ESM provides mechanisms for concurrency control and recovery. Two phase locking of byte ranges within a storage object is used for concurrency control with the "lock entire object" option being provided for cases where object level locking will suffice. To ensure the integrity of internal pages of large storage objects, a non-two phase locking protocol is employed. For recovery, small objects are handled using before/after-image logging and in-place updating. Recovery for large object is handled using a combination of shadowing and logging. Updated leaf blocks and internal pages are shadowed up to the root level, with updates being installed atomically by overwriting the object header with the new header. In this regard, *MasterRecord* in *SSM* serves a similar purpose.

An OID in ESM is a 96 bit entity which consists of a 32 bit page address, a 16 bit slot number within a page, a 16 bit volume id and a 32 bit unique number. A value is assigned to a newly created object and the value is never recycled. That means, a dangling reference to an already deleted object can be detected even if a new object has overwritten the storage by comparing the unique field in the reference with the unique number assigned to an object. This is an important safety feature because ESM requires explicit destruction for reclaiming storage assigned to dead objects.

As for root object management, ESM supports such a capability. Thus the relevant member functions of the glue class *ESM* merely calls associated APIs provided by ESM.

5.7.3 Kala

Kala is an untyped persistent store for practical object-based systems, such as OODBMS and object-oriented languages with persistence. **Baskets** are dynamic groupings of immutable data elements managed by Kala. Baskets synthesize transaction, configuration management, and access control semantics, and thus offer a platform for implementing arbitrary such models[34].

A *monad* is a unit of storage. A monad consists of bits and references and is immutable. The monad m1 in Figure 5.19 has two references in it. The other part is bits. Two kinds of mechanisms for grouping monads are provided. One is a static and abstract grouping based on *kin*, which is assigned to a monad when it is created. A mid (monad id) has one to one correspondence to a *kid* (kin id) and this correspondence will never be altered. The other mechanism is provided by Baskets. Two ovals in Figure 5.19 are Baskets which contain three and one handles to monad, respectively. Thus this grouping is dynamic and non-partitioning.

The kin mechanism can be used to represent versions of the same database item, e.g. in multi-version concurrency control, as well as to maintain historical versions of the same entity in a version control system. However, Kala does not enforce any semantic relation on the use of kin. It just maintains the grouping.

On the other hand, Baskets can be used to represent a state of a transaction, i.e. correspondence from data item (kin) to mid, and configuration. Also baskets can model access control. The basket A in Figure 5.19(a) holds three handles to the monads, m1, m2 and m3. In order to access a monad, a user needs to have an handle to the monad in an accessible basket. Hence the user A can access all the monads in the figure. Contrarily the user B does not have handles to m2 and m3. Thus the user B cannot follow the

references in m1 to access m2 and m3. In this way, access rights are controlled by the granting of handles.

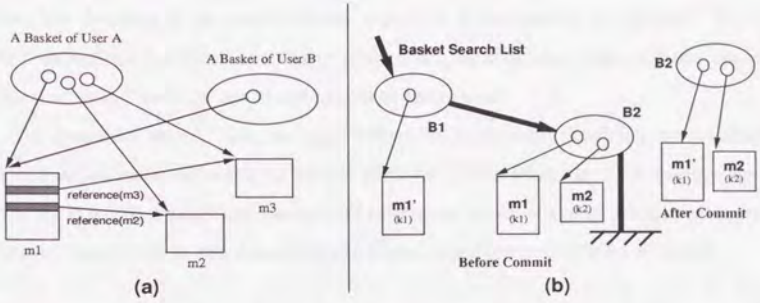


Figure 5.19: An Access Control(left) Transaction and BSL(right)

Using Kala, a simple transaction can be implemented in the following way. Kala addresses a monad with a (kid,bid) or (mid,bid) pair as stated in the above paragraph. And a user may specify multiple Baskets to search in for a handle. In this case, Kala searches all the Baskets to resolve the address. Kala provides each client with Basket Search List (BSL), which is a list of Baskets for specifying the search order.

At the start of a transaction, a client creates an empty Basket(B1) and inserts it at the head of the BSL. B1 is empty at this moment. When the client wants to retrieve the data item represented by kid=k1, it uses an address (k1, allFound). Then Kala searches a handle to a monad whose kin is k1 starting at B1. Since B1 is empty, Kala locates the monad m1 in the Basket B2 and loads it for the client. After updating the dataitem, the client requests Kala to write back the content. When so requested, Kala creates a new monad m1', since a monad is immutable, and puts a new handle to it in B1. If the client loads the dataitem again in the same transaction, Kala resolve the address (k1, allFound) with the handle in B1. Then m1' will be loaded. And it is what should happen.

To commit a transaction, the contents of B1 are dumped into B2. The handle for m1 is supplanted with the one to m1' since they are for the same kin. See the content of Basket B2 on the right of Figure 5.19(b). This supplanting is *the central dogma* in Kala. Since the dumping is an atomic action, atomicity of transaction is achieved. To abort the transaction, the client just has to discard B1. In that case Kala reclaims resources allocated to m1' as there are no references to the monad.

As described above, *Kala* manages references in monads providing an interface to swizzle a particular reference in virtual memory. Thus what the *P3L* runtime system must do is to tell *Kala* about locations of references within a monad when creating it via *CreateDatum()* and to call *LoadMonad()* when a reference needs to be swizzled.

Chapter 6

Performance Evaluation

In this chapter, the overhead introduced by adding reservation checking code is assessed by running a set of small programs and a real program with and without reservation checking code inserted. The set of small programs are pathological ones which induce very dense reservation checking. The results serve as estimation of the worst case overhead introduced by the addition of reservation checking. The real program used is a ray tracing program called POVRAY (Persistence of Vision Ray Tracer) which is written in C. This gives us hints about the overhead introduced in a practical program. Finally, the overall performance of *P3L* is examined by a set of OO1 Benchmarks[35]. Texas Persistent Store was chosen as the reference point for the comparison in order to clarify the subtle trade off between the hardware based implementation and the purely software based implementation, both of which are designed to achieve efficient manipulation of objects in core.

6.1 Micro Benchmarks

For this benchmark we have created a variation of the *P3L* compiler which employs a different strategy for generating reservation checking instructions. In the standard way, reservation checking instructions are added when a parse tree is converted into RTX.

This results in the generation of a compare and a branch instructions for each discovered pointer. As the common subexpression elimination phase of the compiler does not try to optimize beyond a basic block, redundant reservation checking codes are not eliminated. In the special compiler, RTL instructions for fetching a pointer, which is potentially a *lambda*, is just marked as 'need-to-reservation-check' during the conversion phase. Then after finishing the CSE and jump optimizations, an extra RTL sequence for reservation checking is inserted in front of the marked RTLs. In the discussion below, the former is referred to as P3L(tree) and the latter as P3L(RTL). The reason for preparing the variation of the compiler is to assess the effect of folding multiple reservation checks in a visit to a single reservation check at the beginning of the visit.

We have ran three micro tests and one small program by compiling them with the Gnu C++ 2.3.3, two variations of P3L compiler and EG++ (E compiler in Exodus). Both P3L and EG++ are derived from version 2.3.3 of the Gnu C++ compiler. EG++ is the compiler for the E programming language. We made the structure of the nodes to be manipulated in the programs written in E, a *dbtype*. All compilations are done using -O flag. These micro tests examined the overhead of the reservation checking code. The times reported are user CPU time, which was gathered through two *getrusage* system calls placed before and after the measured region of code. The programs were executed on a Sun4/330 with 32MB of memory. For each benchmark, measurements were performed 10 times and the two fastest and slowest results were eliminated and the remaining 6 were averaged.

6.1.1 Traversal over Fully Cached Objects

The first micro test is the most pathological one. The program does pointer chasing only. Both structures being traversed are supposed to reside entirely within the processor cache. Thus the cost of executing reservation checking instructions is not much diluted by the cost of memory access.

```
struct foo {          | int main(int ac, char **av) {
  struct foo * next; |   int i, sum=0;
  int             v; |   struct foo * p = &a;
};                   |   a.next = &b; a.v = 1;
struct foo a, b;     |   b.next = &a; b.v = 2;
                     |   for(i=0;i<1000*1000*10;++i)
                     |       { /* sum += p->v; */ p = p->next; }
                     |   /* return sum; */
                     |   return (int)p;
                     | }
```

compiler	G++	P3L(RTL)	P3L(tree)	EG++
time(msec)	2040	3276	4493	17380
ratio	1.00	1.61	2.20	8.52

In the case of P3L(RTL), the compiler succeeds in placing the instruction for fetching the 'next' field into the delay-slot of the branch for the loop. This reduces the pipeline stall. And the instruction for loading 'p' and the compare instruction for reservation checking are separated by several instructions. This reduces or eliminates the stall time due to data dependencies. These seem to be the reasons why P3L(RTL) does much better than P3L(tree).

6.1.2 Traversal with a Small Work

The next program adds a small work to the navigation loop. The program is the same except that the commented out statements are now executed.

compiler	G++	P3L(RTL)	P3L(tree)	EG++
time(msec)	4088	4506	6135	23131
ratio	1.00	1.10	1.50	5.65

The overhead of the three reservation checks has decreased considerably. This agrees with the results presented in [22] that effectively says that the cost of residency checks becomes rather insignificant when a small amount of work is added. However, the absolute value of the overhead is still high.

The last micro test examines the performance of navigation when pointers are not in the processor cache. A linked list consisting of 500K nodes is created then traversed 10 times calculating the sum. The time for creating the list is not included. The overhead falls to less than 10% for P3L(RTL).

compiler	G++	P3L(RTL)	P3L(tree)	EG++
time(msec)	5721	6228	6948	22676
ratio	1.00	1.09	1.21	3.96

6.1.3 Insertion into an AVL tree

The last test is with a small program which inserts 4096 nodes into an AVL-tree. The type of the nodes are defined as follows.

```
struct node {
    struct node * left, *right;
    int                 balance;
    char                key[32];
};
```

The code to handle the insertion into an AVL tree is taken from [36]. The values of the key are generated at random. The time for generating nodes before insertion into the tree is not included in the results.

compiler	G++	P3L(RTL)	P3L(tree)	EG++
time(msec)	14511	15463	15821	150066
ratio	1.00	1.07	1.09	10.34

Because P3L(RTL) makes it possible for the compiler to take better advantage of the CSE optimization, two successive dereferences of a pointer result in only one pair of load and check instruction. The benefit is not small if time saved is compared to the total overhead. However, the relative value of the difference is so small that it is not tangible. Eliminating redundant translation instructions and check instructions is surely more crucial for faulting methods without swizzling or those with *swizzling upon dereference*, though. The reason why E does so poorly seems to be due to the fact that this test contains many function calls that take pointers in arguments and many accesses to characters through db pointers.

6.2 POVRAY Benchmark

For an application level benchmark, we picked up a freely distributed ray-tracing program, POVRAY (Persistence Of Vision RAYtracer), which is written in C. In this benchmark we compared the execution speed of two executables derived from the same set of sources. One is compiled using plain Gnu C 2.4.8 and the other is compiled by P3L compiler, which is based on Gnu C 2.4.8. Five models of scene were picked up because we expected that different models would result in different traversal density, thus different reservation checking density. The produced images are saved in files instead of being sent to an output device. The results are shown below. Measurements were performed 10 times for each scene and the resulting values were averaged.

Scene	polywood	skyvase	stonewal	magglass	crystal
Plain GCC	37.3	94.2	36.0	32.8	69.3
P3L	40.9	98.4	39.0	36.4	70.5
ratio	1.09	1.04	1.08	1.11	1.02

Time for rendering POV images (in seconds).

6.3 OO1 Benchmark

The OO1 Benchmark was proposed by Cattel[35] to model the work load imposed by design applications and has been used by many researchers for evaluating their research work.

6.3.1 Specifications

The benchmark uses a homogeneous database which consists of two kinds of entity, **Part** and **Connection** as illustrated in Figure 6.1. The benchmark specification does not enforce any physical structure and defines these entities as logical records.

```
Part:       RECORD[id:INT; type:STRING[10]; x,y:INT; build:DATE]
Connection: RECORD[from: part-id; to:part-id; type:STRING[10];
                   length:INT]
```

INT is a 32 bit entity. STRING[N] is a variable length string of maximum size of N. DATE is a some representation of date and time. Id of parts are defined with unique number 1 through 20000. A connection defines a link from a part to another. Each part has three outgoing links to selected parts. Thus the database contains 60000 connections in total. The arrangement of parts are illustrated in Figure 6.1 The 20000 'Part' parts is the specification for the small database benchmarks. The specification also defines the

large database which consists of 200000 'Part' part objects and 600000 connection objects.

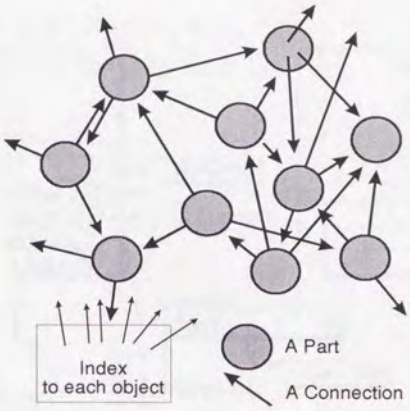


Figure 6.1: OO1 Database Structure

The benchmark specifies a rule that a connection must be made to another part within 1% distance 9 out of 10 times. The distance is determined in terms of the part number assigned to each part and the benchmark makes no specifications about the physical location of parts. Thus two databases which follow the rule were included in the test. In one of them, the part number is assigned following the creation order thus it likely corresponds with physical layout on disk surface. In the other, there are no correlation between the part number and the creation order. Thus there is no correlation between the part number and the position on the disk surface. In addition to these two conforming databases, another database where the locality rule is ignored was tested. The three database are qualified by the term, *local*, *shuffle* and *random*, respectively. Another

deviation is that our large database contains only 100K objects while the benchmark specifies 200K objects for a large database. This is due to the limitation in the available swap space on the test machine. The data structures used to implement the database are shown in Figure 6.2.

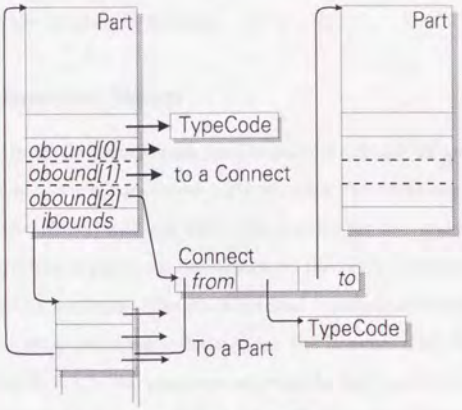


Figure 6.2: OO1 Data Structure

An iteration of a traversal consists of selecting a part randomly and visiting all connected parts reachable from the selected one within 7 hops. For every node visited, a null procedure is called with three arguments. This operation is repeated for specified number of times keeping all data fetched in the previous iterations on the physical storage. A traversal benchmark was performed at most three times under different caching condition. A *cold* traversal is run with disk cache completely clean. A *warm* traversal is run with database in the cache but outside of the virtual memory space of the benchmark process. A *hot* traversal is run just after a warm traversal within the process invocation with all the accessed data completely kept in the physical storage. All the three traversals

use the same random number sequence. Disk cache is flushed before running each hot traversal by mapping a huge file into the virtual memory space and accessing the contents at random repeatedly. To amplify the effect of running benchmark programs on large database, the available physical memory was reduced for benchmark runs on the large database. A program which grabs physical memory was written and run under the root privilege to reduce the amount of memory.

6.3.2 Experimental Setup

Another deviation from the benchmark specification is the use of local disk to store the database. We are not much concerned with whether the database is stored locally or remotely because the choice will not affect the results for hot traversals. However, we do not want to introduce a potential bottleneck to the setup while technologies for LAN connections are rapidly evolving. The machine used is a SparcStation 10 with 96M bytes of memory and the swap partition is located on the internal disk drive. The operating system in use is SunOS 4.1.3. All programs relevant to the benchmark are compiled with -O option using Gnu C++ 2.5.8 or P3L-2.5.8, including the library for Texas PS. The database is placed on a dedicated partition on the externally connected Fujitsu 2694ESA SCSI-2 disk drive. Before creating a new database, the old database is removed from the partition. Thus all databases used for the benchmarks start at the same position on the disk. In this way, the effect of file system fragmentation and zone bit recording of the disk is avoided.

6.3.3 Traversal on Small Databases

The result of cold traversal on small database is shown in Figure 6.3.

P3L (non-clustered) is consistently slower than Texas. Admittedly swizzle per pointer makes longer the code path for swizzling a single pointer. However, it is unreasonable to assume that the difference in the instruction path length makes such a big difference. We

hypothesized that objects are scattered in more disk pages thus the amount of I/O was greater in *P3L* case. To add circumstantial evidence, clustered allocation in the benchmark program for *P3L* was tried. In this version of the program, objects are allocated as an array whose size is quite near to the page size and all slots but the first one shall be used to satisfy subsequent allocation requests. The next allocation grabs the next available slot. This situation is illustrated in Figure 6.4.

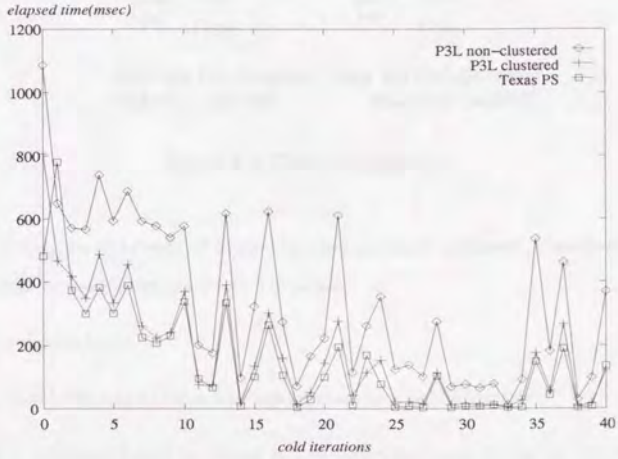


Figure 6.3: Small Database - Local - Cold

The obtained result, *P3L* (clustered), is strikingly similar to the result of Texas. We assume the small difference is due to the difference in the size of reference on persistent storage. Texas uses 32bit reference and *P3L* uses 96bit reference. The size of resulting database are 6.9M (*P3L* non-clustered), 6.1M (*P3L* clustered) and 4.4M (Texas). Similar results were obtained for shuffle and random database. The difference is that the curves go down more steeply and that the ratio of elapsed time for *P3L* (clustered) and *P3L*

(non-clustered) becomes larger.

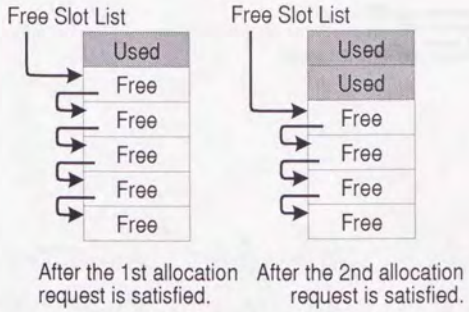


Figure 6.4: Clustered Allocation

Figure 6.5 shows the result of warm traversal on small database. Conceivable reasons why *P3L* does so poorly compared to Texas are:

- *P3L* swizzled twice.
- *P3L* maintains more tables and one of them is a balancing tree.
- Malloc implementation in Texas is more efficient than malloc in libc which *P3L* uses.
- Implementation of Texas is more efficient.

An interesting thing to note in the figure is that most of the elapsed time for Texas is spent in the fault handler. It does not include the time during which the operating system is responding to the fault before passing the control to the user defined fault handler. Thus it can be concluded that the overhead for fielding page access faults is negligible with the system configuration.

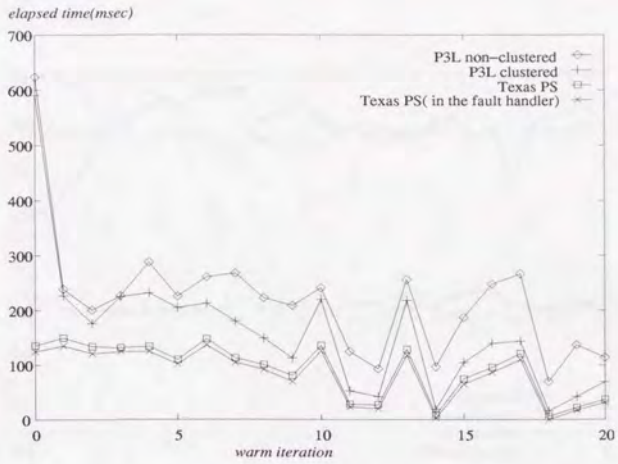


Figure 6.5: Small Database - Local - Warm

Figure 6.6 and Figure 6.7 shows the results of hot traversals on small database. Please note that each iteration is run 10 times to improve the resolution of the results. The result from the local database indicates that although Texas PS consistently do better than *P3L*, *P3L* competes quite well with Texas PS in the hot traversal on the database with the locality of connection.

The other results are even more interesting. At the first several initial iterations, *P3L* outperforms Texas PS despite of added reservation checking instructions. We speculate that better utilization of cache and/or TLB due to the tight clustering of accessed objects is the cause of this phenomenon. We tried clustered allocation for this traversal as well. However, the result was flat lines similar to the results for Texas PS. The synergy of objectwise address assignment and lazy address assignment seems important.

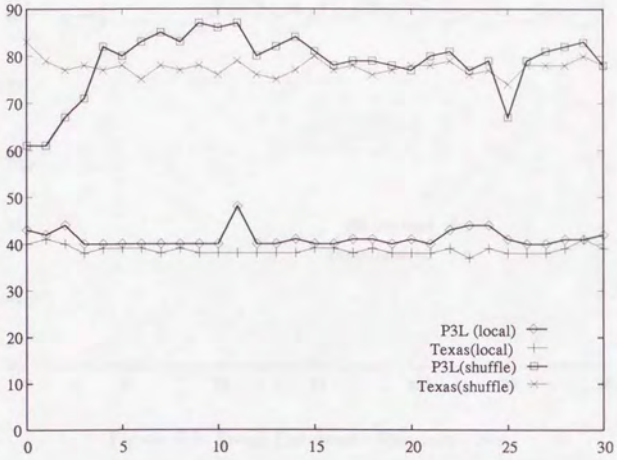


Figure 6.6: Small Database - Local/Shuffle - Hot

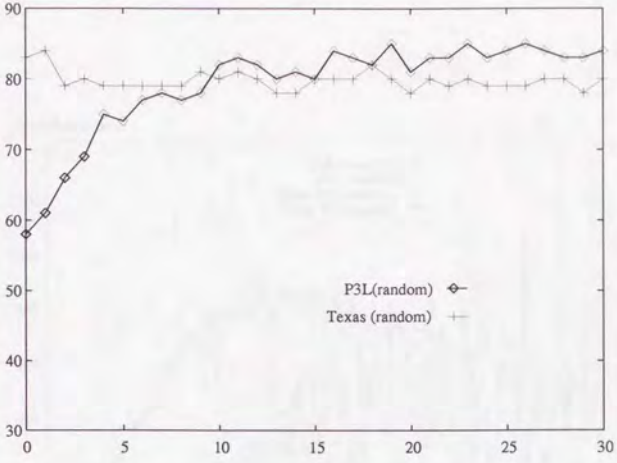


Figure 6.7: Small Database - Random - Hot

6.3.4 Traversal on Large Databases

The available memory was reduced to 15M bytes for these benchmarks. Figure 6.8 shows how badly *P3L* performs in the cold traversal if enough memory is not available. The larger database size, overhead in free space management and the consumption of storage for resident object table entries are responsible for this. This figure gives an indication of what price must be paid to get an advantage in the hot traversal described in the next paragraph.

Figure 6.9 shows the result of running the hot traversal on large database with locality in connection. *P3L* does better in these initial iterations because accessed objects are packed more tightly. A good clustering for initial iterations means a bad clustering for later iterations. After the 100th iteration, the elapsed time occasionally goes beyond

2000msec.

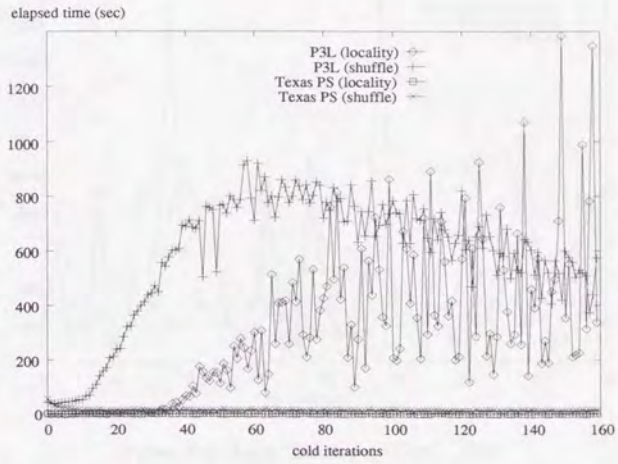


Figure 6.8: Large Database - Locality/Shuffle - Cold

Figure 6.10 shows the result of running the hot traversal on large shuffled database. The result for the random database is omitted because it is almost identical. Texas PS shows the effect of clustering of accessed object for the first few iterations. However, the effect is more significant in *P3L* (non clustered) although it becomes worse than Texas PS after 35th iteration and, at about 150th iteration, it reaches 600 seconds. At the first 20 iterations, *P3L* (non clustered) does better, as much as twice as good as *P3L* (clustered), though it cannot be seen from the figure. These facts lead us to believe that the objectwise and lazy address allocation in *P3L* system is very effective in converting access locality to spatial locality.

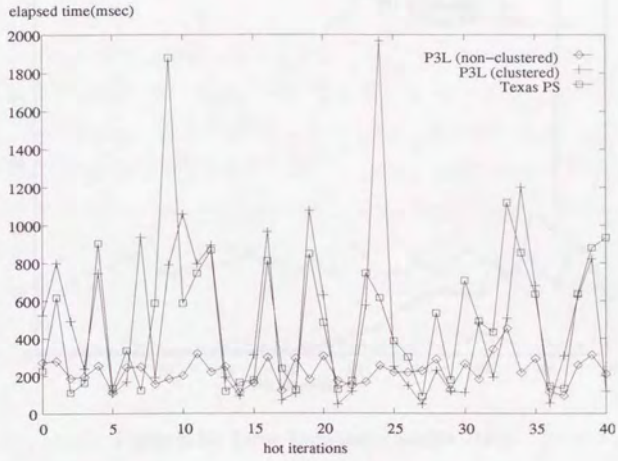


Figure 6.9: Large Database - Local - Hot

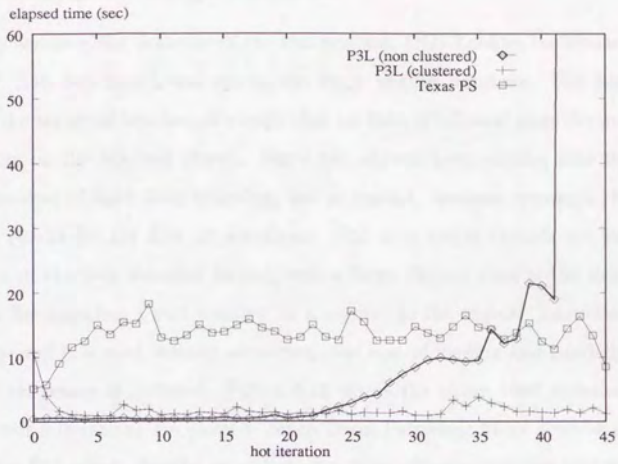


Figure 6.10: Large Database - Shuffle - Hot

6.3.5 Lookup on Large Database

In order to compare the behavior of the two systems, OO1 Lookup benchmark was also performed. The benchmark was run on the large 'shuffle' database. The benchmark is similar to the traversal benchmark except that no links is followed after the null function call is made on the selected object. Since the objects keep coming into the physical storage, the cost of fault time swizzling, full or partial, becomes apparent. Figure 6.11 shows the results for the first 50 iterations. *P3L* does better though not significantly. The reason of the first iteration having such a large elapsed time is the existence of a hash table for mapping a part number to a pointer to the object. Since this object is rather huge and it is read entirely at startup, the cost of loading and partially swizzling embedded references is incurred. Figure 6.12 shows the entire 1000 iterations. Values averaged over 5 iterations are plotted. Since Texas Persistent Store absorbs a page at a time, unlike *P3L* which absorbs an object at a time, objects migrates into the physical storage of the benchmark program more quickly. This is reason why the reduction of the elapsed time is much steep in Texas.

Since the above cold test includes the disk I/O in the result, a warm lookup was also performed so that the actual cost of swizzling references at fault time may be examined. The result is shown in Figure 6.13. The reason why Texas does not do so well initially seems to be related to the cost of manipulating protection status of virtual memory pages.

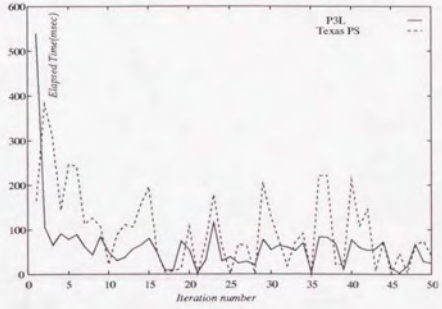


Figure 6.11: Lookup Large Database - Shuffle - Cold

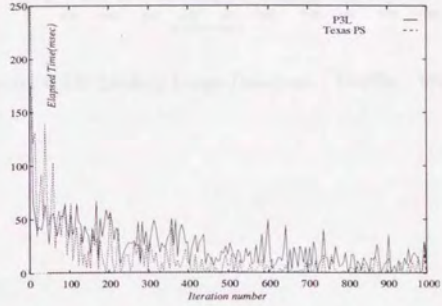


Figure 6.12: Lookup Large Database - Shuffle - Cold

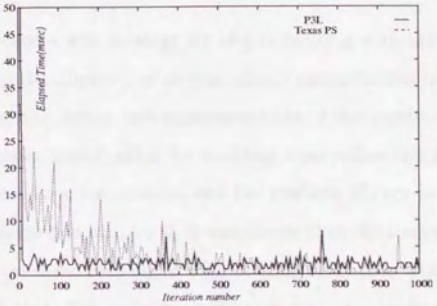


Figure 6.13: Lookup Large Database - Shuffle - Warm

Chapter 7

Conclusions

This thesis has proposed a new strategy for object faulting with pointer swizzling, which is aimed at improving the efficiency of on-core object manipulation typical in design applications and presented the design and implementation of the persistent C++ system *P3L* in terms of the compiler modification for emitting reservation checking instructions and for generating runtime type information and the runtime library organization. Through the various tests presented in chapter 6, it was shown that the reservation checking strategy avoids the large overhead associated with existing strategies which does not rely upon MMU hardware and that *P3L* compete quite well with the hardware based strategy as implemented in the Texas Persistent Store.

To ascertain the contributions of the proposal made and the results obtained through this thesis work, the next section compares the work presented in this thesis with related works in the field. The final section presents future prospects.

7.1 Summary

One of the early reports on the performance of pointer swizzling is [28]. In the paper, the cost and merit are studied along two orthogonal axis, (1) in-place v.s. copy swizzling and (2) eager v.s. lazy swizzling. The paper focuses mostly on the cost of swizzling pointers

rather than the cost of reservation checking. Our concern has been about reducing the overhead of adding checks, which needs to be performed repeatedly even after all relevant pointers have been swizzled. We proposed to recognize that there are different degrees of laziness in swizzling than the simple categorization and that the choice of laziness affects the characteristic of a resulting system in subtle ways.

Pointer swizzling at page fault time was proposed in [26] for the Texas Persistent Store. The Texas Persistent Store provides persistence support to C++ programs by supplying C++ a class library. It does not require a compiler to issue extra instructions for reservation or residency checks. To a running program, all pointers appear as a raw virtual memory pointer. This scheme defines the lower bound of the number of instructions needed to deal with persistent objects, that is, it adds 0 instructions for persistence support. This strategy is believed to be the fastest in a tight traversal, and in fact it is in many cases. There have been no reports of comparison of the Texas Persistent Store with a competitive software implementation. This study provided a result of such comparison.

[22] compares the performance of *ObjectStore*(V 1.2) and various versions of *E* in terms of traversal operations modeled from the OO1 benchmark. A result obtained shows ObjectStore runs 5 times faster than *E* in a tight traversal. In a traversal with sparse updates, E variants are shown to run faster because then the log writes becomes the dominant factor. One reason for the 5 times slower execution is the overhead of software implemented checks. Other conceivable reasons are the long persistent pointer (96 bits) and the memory indirection introduced to the dereference operation on a swizzled pointer. In a similar vein, [7] provides a comparison between a hardware based residency checking scheme as implemented in *QuickStore* and a software based reservation checking scheme in E (EPVM 3.0), using the OO7 benchmark. The traversal test T6 (the sparse traversal) produces a result that shows the version of E is 3.6 times slower than QuickStore. We perceive that a performance hit as high as 5 or 3.6 times is prohibitive to application domains we target *P3L* to, although comparison should not be made directly because we

have not yet experimented using the OO7 benchmark. Our study proved that a software implementation can do better in a OO1 traversal than their paper suggests. And the result obtained so far indicates that it can do better also in the T6 traversal in OO7 benchmark. Of course a definite answer must wait for the OO7 implementation using *P3L*.

[38] evaluates various schemes for adding reservation[1] checks in the context of Persistent Smalltalk. This paper examines the overhead introduced by reservation checks in a hot tight traversal as well as the cost of swizzling and fielding faults. They conclude that the software implementations can be as efficient as the hardware implementations even in terms of tight traversals. The results of our study supports this conclusion. However, the conclusion in the paper states: *However, some languages (e.g., Modula-3, C++) do not enforce the pure object-oriented style of execution that enables residency checks to be piggybacked with method invocation. Operations on an object can be performed without necessarily invoking a method on it.* A contribution of our study is the finding that the same conclusion can be drawn in the context of the conventional language, C/C++.

[31] proposes a unique hardware based object faulting scheme with pointer swizzling. It forces allocation of virtual memory address as late as dereference time. This scheme is interesting in that it assigns virtual memory addresses very lazily, while the other hardware based implementation do so as early as at page fault time. The lazy allocation of addresses is expected to provide different tradeoffs than other hardware schemes. The proposed implementation pays much effort to swizzling pointers as early as possible so that the number of address translations and page faults can be reduced, while avoiding forcible reservation. Unfortunately, we are aware of no implementations of this scheme thus we could not include performance evaluations of the scheme in the study.

In [39] and [40], the runtime cost of introducing provision for object eviction[2] is studied. Two mechanisms are considered. One uses indirection block (descriptor) via which

[1] Called *residency checks* in the paper.

[2] Object eviction is replacement of an object from virtual memory space.

objects are referred to. Having the indirect block allows one to mark the evicted object as *not present* by updating only the corresponding indirection block instead of updating references pointing to the object. The other mechanism maintains a RRL (reverse reference list) for each object. A RRL associated with each object records the set of the locations of pointers which point to the object. All pointers are located and deswizzled by consulting the relevant RRL when an object is evicted. Our performance study does not include consideration on the cost because we assume the backing store is large enough so that no eviction is needed. However, a mechanism for object eviction needs to be implemented in order to guarantee proper functioning of the system in unexpected situations. Our plan is to cope with the problem of locating pointers, which need deswizzling, by tracing pointers like garbage collectors do. Pointers on the stack will be conservatively scanned [19][41][42]. Research in garbage collection concluded reference counting and RRL are generally inferior to garbage collectors in terms of introduced overhead.

7.2 Future Prospects

7.2.1 Improvements to *P3L* Implementation

As has been pointed out in section 3.5, the current implementation of *P3L* performs the contents transfer when a virtual memory address is assigned to it. This would likely results in unnecessary I/Os being performed when larger objects are manipulated. Because of the object at a time reservation and transfer assumed in the current implementation of *P3L*, it is not trivial to perform residency checking at dereference time. However, it is not difficult either since many of the necessary piece of software are already in the runtime library. For example, being organized as an AVL tree, the *Lid2OidMap* provides enough information needed to enumerate all objects which are fully or partially mapped within a virtual memory page. So what needs to be done is to write a page fault handler and modify the *map()* member function of a storage manager. The handler fetches objects residing

in the faulted on page from the persistent store. The modified *map()* function places a read barrier on the pages surrounding the specified object so that the page fault hanlder will be activated when the object is accessed. One thing to be noted about installing the page fault handler is that debugging an application program becomes tricky, because the application is stopped every time a non-resident persistent object is accessed as the debugger catches an access fault exception first.

There are many improvements which can be made about optimizations in the compiler. Since the current compiler inserts reservation checking instructions when RTXs are generated, multiple access to an object within a visit are not folded. For example, two reservation check are performed to evaluate the following expression, given the type definitions in Figure 4.5.

```
p->m_father->age == 35 && p->m_father->salary < 400000
```

Another optimization is to delay reservation checking to dereference time if possible so that MMU can be utilized for reservation checking. Consider the following code fragment:

```
0:  Person * tmp_p; // p is also 'Person *'
1:  tmp_p = p->m_father
2:  // [r0 <= &p->m_father]
3:  // [r1 <= *r0]
4:  age = tmp_p->age;
5:  // [r2 <= r1 + offset(age)]
6:  // [r3 <= *r2] # r2 << *r0 + offset(age)
```

If the principle of *pointer swizzling upon discovery* is strictly followed, the reservation checking instruction must be issued for the statement in line 1. If the check is omitted, the execution of instruction from the statement in line 4 will generate a segmentation fault because the location pointed to by *lambda* is usually not accessible. This runtime error can be utilized in order to elide the reservation check for the expression in line 1. What the fault handler must know are:

- The memory location from which the lambda was fetched.
- Registers which contain the *lambda* fetched in line 1.
- Any offsets added to each of the registers containing the lambda.

The compiler should emit a record which contains the information listed above and associate the record with the location of the instruction which causes the segmentation fault so that the fault handler can extract the information from the value in the instruction counter at the fault time. This would require extensive modifications to a code generator part of a compiler. However, the benefit is not small considering that many of reservation checking instructions are eliminated while avoiding the problems associated with *pointer swizzling upon dereference.*

7.2.2 Evaluation by means of OO7 Benchmark

The OO1 benchmark used to be one of the most popular benchmark. Recently, however, the OO7 benchmark[37] draws more attention since the benchmark examines wider range of OODB features and performance issues. We have not used the OO7 benchmark in this thesis work because the application model that the OO1 benchmark assumes (design applications) coincide nicely with the model for which *P3L* is optimized. The range covered by the OO1 benchmark was sufficient to verify that our design goals were met. However, now that we have clarified the tradeoffs, under the assumed application scenario, in comparison with the Texas Persistent Store, it is the time to examine how well or bad the *P3L* implementation fares under application scenario which were not assumed.

7.2.3 Garbage Collection

As this thesis work progresses, it has dawned that the issues in building persistent object system overlap significantly with those in garbage collecting language implementation. Persistent object system must govern the migration of object between persistent store

and virtual memory storage while copying collectors manage migration of objects between semi-spaces. The issues to be tackled by both systems are:

- The mechanism with which references within an object is located.
- The mechanism with which a reference to one space is distinguished from a reference to the other space(s).
- The mechanism with which the top of an object is located, given an interior pointer.

The first one concerns with the management of runtime type information and runtime type identity. The second is for handling the migration of objects. A small difference is that the structure of a pointer does not change in copying collectors while the structure can vary from the one on the persistent store to the one in virtual memory storage in a persistent object system. As for the last one, an interior pointer must be associated with the object it points to so that the pointer can be normalized before being written to the persistent store in case of *P3L* and that live objects can be marked or copied in case of garbage collectors. This is a requirement specific to the C/C++ language which allow interior pointers.

Since they share so much, the work expended on the implementation of *P3L* can be utilized for building a garbage collecting C++ system. In fact *P3L* need garbage collection support in order to handle applications with large working set which may grow bigger than available virtual memory space. Currently, objects are evicted back to the persistent store at the end of transaction only. In order to handle such applications, objects must be sent back to the persistent store while a transaction is running so that exhaustion of virtual memory space is avoided. Safely evicting object requires to select the copies of persistent objects to which there are no references or to deswizzle all references to an evicted object. Adding object eviction and garbage collection will improve the usability of the *P3L* system significantly.

Not only the implementation techniques but also the strategies coincide. The idea of *pointer swizzling at fault time* was inspired by Appel, Ellis and Li's concurrent incremental collector [43][3]. The collector access-protects a region in *tospace* where some of objects in *fromspace* have migrated into. When an access to the protected region triggers a page fault, the fault handler scans pointer in the protected pages to update pointers to the *fromspace* so that they points to the *tospace*. The referents of these pointers are migrated to the *tospace* if necessary and pages containing the newly migrated objects are access protected. On the other hand, Baker's incremental copying collector [44] examines each pointer fetched from memory in order to trigger update of a pointer with associated object migration. This is exactly the strategy taken in the *P3L* implementation. It seems to be the belief in GC research community that Baker's scheme incurs significant overhead in excess of 20% even with special compiler techniques. The results obtained in this thesis work indicates that this number may be lower than 20% for typical C/C++ applications. If that is the case, the scheme must be revisited considering its benefits such as no reliance on MMU and copying of no dead objects. Application of the optimization technique described in section 7.2.1 to the Baker's collector may also be interesting.

7.2.4 Distributed Computing

Distributed computing also deals with the problem of object migration between multiple spaces thus the techniques presented in this thesis can be applied. A standard for inter operability in distributed computing, OSF/DCE RPC (Remote Procedure Call), solves some of the problems through the use of IDL (Interface Definition Language). Interface functions are described in IDL and stored in a file, which is compiled by an IDL compiler. The result of the compilation is two C source files for server stub functions and client stub functions. On making a remote procedure call, a client stub marshalls function parameters into a packet then send it over the wire. Then the server accepts the packet

[3]From personal communication with Paul Wilson.

and unmarshall the packet to extract arguments, with which a function on the server is called. For returning the result of the function call, the data flows back to the client in the same way. The client and the server must agree on the packet structure for marshalling and unmarshalling. OSF/DCE RPC achieves the goal by generating marshalling and unmarshalling functions from interface specification in an IDL file using an IDL compiler. The current spec assumes the marshalling be performed for an entire set of objects to be passed at one time. When object must be accessed on demand, it is the programmer's responsibility to request a piecemeal transfer. That places a burden on the programmer since calls to RPC functions must be embedded in source codes. The process is error prone, too. The technique for automatically emitting reservation checking code can be applied to solve this problem and techniques for runtime type information may eliminate the need to prepare an IDL definition file and to precompile it.

Bibliography

[1] Karen E.Smith and Stanley B.Zdonik, 'Intermedia:A Case Study of the Differences Between Relational and Object-Oriented Database systems',Brown University Techinical Report No. CS-87-18, Junly 1987

[2] Thomas M. Atwood 'The case for object-oriented databases', IEEE SPECTRUM FEBRUARY 1991

[3] M. Atkinson, F. Banchilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik, 'The Object-Oriented Database System Manifesto', 1st Int'l Conference on Deductive and Object-Oriented Databases. Kyoto, 1989

[4] Shinji Suzuki, M. Kitsuregawa, M. Takagi "An Efficient Pointer Swizzling Method for Navigation Intensive Applications", Proc. of the Sixth Int'l Workshop on Persistent Object Systems, Tarascon, France Sept 1994, Springer Verlag Workshops in computing series, Malcolm Atkinson, David Maier, Veronique Benzaken ed., ISBN 3-540-19912-8

[5] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson, 'Texas: An Efficient, Portable Persistent Store', Proc. Fith Int'l Workshop on Persistent Object Systems, San Miniato, Italy, Sept 1992.

[6] Charles Lamb, Gordon Landis, Jack Orenstein and Dan Weinreb:"THE OBJECT-STORE DATABASE SYSTEM" Vol.34 No.10 CACM91

[7] Seth J. White, David J. DeWitt, 'QuickStore: A High Performance Mapped Object Store', Proc. int'l Conf. on the Management of Data, Minneapolis, 1994

[8] J.Eliot B. Moss,'Addressing Large Distributed Collections of Persistent Objects:The Mneme Project's Approach', Database Programming Languages,*Secondnd intl. workshop*, Morgan Kaufmann 89.

[9] Bjarne Stroustrup, 'The Design and Evolution of C++', Addison-Wesley 94, ISBN 0201543303

[10] James Coplien, 'Advanced C++ Programming Styles and Idioms', Addison-Wesley 91, ISBN 0201548550

[11] Daniel R. Edelson, 'Smart Pointers: They're Smart, but They're Not Pointers', Technical Report UCSC-CRL-92-27 1992

[12] Bjarne Stroustrup, 'The C++ Programming Language', Addison-Wesley 2nd edition, 1991

[13] W.P. Cockshot, M.P. Atkinson and K.J. Chisholm, 'Persistent Object Management System', Software Practice and Experience Vol.14 1984

[14] M.Atkinson et,al."Algorithm for a Persistent Heap", Software Practice and Experience Vol 13, 1983

[15] Edited by R.G.G. Cattell, 'The Object Database Standard', Morgan Kaufmann Publishers, ISBN 1-55860-302-6

[16] Malcolm P. Atkinson and O. Peter Buneman,'Types and Persistence in Database Programming Languages', ACM Computing Surveys 19,2 1987

[17] Thomas Atwood, 'Two Approaches to Adding Persistence to C++', Proc of the Fourth Intl. conf. of Persistent Object Systems 1992

[18] Eric N. Hanson, Tina M.Harvey and Mark A.Roth, 'Experience in DBMS Implementation Using an Object-Oriented Persistent Programming Language and a Database Toolkit', Proc. of the Conference on Object-Oriented Programming Systems, Languages and Application, Phoenix, AZ, October 1991.

[19] Boehm Hans-J. and Mark Weiser,'Garbage collection in an uncooperative environment', Software Practice & Experience 18,9 1988

[20] Carey, M., et. al.,'The EXODUS Extensible DBMS Project: An Overview' in Readings in Object Oriented Databases., S. Zdonik and D.Maier,eds.,Morgan-Kaufman,1989

[21] Joel E.Richardson and Michael J.Carey,"Persistence in the E Language: Issues and Implementation" ,University of Wisconsin-Madison Computer Sciences Technical Report #791,1988

[22] Seth J. White and David J.Dewitt, 'A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies', Proc. of Conference on Very Large Data Bases 1992

[23] Dan Schuh, Michael Carey, David Dewitt, 'Peristence in E Revisisted — Implementation Experience', Proc of the Fourth Intl. conf. of Persistent Object Systems 1992

[24] R. Agrawal and N.H. Gehani, 'ODE (Object Database Environment): The Language and the Data Model', Proc. int'l Conf. on the Management of Data, Portland, 1989

[25] Ted Kaehler and Glenn Krasner, 'LOOM - Large Object Oriented Memory for Smalltalk-80 Systems', Smalltalk-80 Bis of History, Words of Advice, Addison Wesley, ISBN 0-201-11669-3, pp.251-271

[26] Paul R.Wilson and Sheetal V.Kakkad,'Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Space on Standard Hardware', Proc. Int'l Workshop on Object Orientation in Operating Systems, Paris, Sept 92,pp. 364-377

[27] Paul R.Wilson,'Uniprocessor Garbage Collection Techniques', Proc. Int'l Workshop on Memory Management 92, St. Malo France, Sept 92.

[28] J. Eliot B. Moss, 'Working with Persistent Objects: To Swizzle or Not to Swizzle', Trans. on Software Engineering Vol 18, Aug 1992

[29] W.P.Cockshott, 'Addressing Mechanism and Persistent Programming', in Data Types and Persistence, Springer Verlag, 1988 ISBN 3-540-18785-5

[30] Gordon Russell, Paul Shaw and Paul Cockshott, 'DAIS: An Object-Addressed Processor Cache', Proc. of the Sixth Int'l Workshop on Persistent Object Systems, Tarascon, France Sept 1994, Springer Verlag Workshops in computing series, Malcolm Atkinson, David Maier, Veronique Benzaken ed., ISBN 3-540-19912-8

[31] Francis Vaughan, Alan Dearle, 'Supporting large persistent stores using conventional hardware', proc. of the 5th International Workshop on Persistent Object Systems, San Maniato, Italy

[32] Amer Diwan, Eliot Moss, Richard Hudson,'Compiler Support for Garbage Collection in Statically Typed Language', SIGPLAN '92 conf. on Programming Language and Implementation

[33] Michael J. Franklin, Michael J. Zwilling, C.K. Tan, Michael J. Carey, David J. Dewitt, 'Crash Recovery in Client-Server EXODUS', Proc. int'l Conf. on the Management of Data, San Diego, 1992

[34] Sergiu S.Simmel and Ivan Godard,'The Kala Basket, A semantic primitive Unifying Object Transactions, Access Control, Versions, and Configurations', Proc. of the Conference on Object-Oriented Programming Systems, Languages and Application, Phoenix, AZ, October 1991.

[35] R.G.G. Cattel and J. Skeen, 'Engineering Database Benchmark', Database Engineering Group, Sun Micro Systems Technical Report 1990

[36] Aaron M.Tenenbaum, Yedidyah Langsam and Moshe J. Augenstein, 'DATA STRUCTURES USING C' Prentice Hall ISBN 0-13-199746-7

[37] Michael J. Carey, David DeWitt, Jeffrey F. Naughton, 'The OO7 Benchmark', Proc. int'l Conf. on the Management of Data, Washington D.C., 1993

[38] Antony L. Hosking, J.Eliot B. Moss, 'Object Fault Handling for Persistent Programming Lanugages: A Performance Evaluation', Proceedings ACM Conference on Object Oriented Programming Systems, Languages and Applications, Washington DC, Sep. 1993, pp.288-303

[39] Alfons Kemper and Donald Kossmann,'Adaptable Pointer Swizzling Strategies in Object Bases', Int'l Conf. on Data Engineering, 1993, pp.155-162

[40] Mark L. McAuliffe and Marvin H. Solomon, 'A Trace-Based Simulation of Pointer Swizzling Techniques', proc. Int'l Conf. on Data Engineering, Taipei, 1994

[41] Hans-J. Boehm and David Chase,'A Proposal for Garbage-Collector-Safe C Compilation', The Journal of C Language Translation, Volume 4, Number2, December 1992, pp. 126-141

[42] Joel F.Bartlett,"Compacting Garbage Collection with Ambiguous Roots", Digital Equipment Corp, Western Research Center, 1988

[43] Andrew W. Appel, John R. Ellis, and Kai Li., 'Real-time concurrent garbage collection on stock multiprocessors', proc. of SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, 1988

[44] Henry G. Baker, Jr. 'List Processing in real time on a serial computer', Communication of ACM, 21(4) 1989

論文リスト

国際会議・研究会・学会誌

Comparison of Two Eager Pointer Swizzling Strategies:
A software implementation can be as Fast
Shinji Suzuki, Masaru Kitsuregawa, Mikio Takagi
21th Int'l Conference on Very Large Databases 投稿中

Dimensions and Mechanisms of Persistent Object Faulting
Shinji Suzuki, Masaru Kitsuregawa, Mikio Takagi
International Symposium on Advanced Database Technologies
and Their Integration (ADTI94), Nara Japan 1994.10

An Efficient Pointer Swizzling Method
for Navigation Intensive Applications
Shinji Suzuki, Masaru Kitsuregawa, Mikio Takagi
Proc. of the Sixth Int'l Workshop on Persistent Object Systems,
Tarascon, France 1994.9, Springer Verlag Workshops in computing
series, Malcolm Atkinson, David Maier, Veronique Benzaken ed.
ISBN 3-540-19912-8

Persistent Programming Language P3L and its Application
to Global Database as an Implementation Tool
Shinji Suzuki, Masaru Kitsuregawa, Mikio Takagi
International Workshop on Global GIS, ISPRS Tokyo 1993.8

永続プログラミング言語 P3L の実装方式
鈴木慎司 喜連川優 高木幹雄
情報処理学会データベースシステム研究会 69-77 1992.7

永続的プログラミング言語におけるオブジェクト識別子の主記憶内表現について
鈴木慎司 喜連川優 高木幹雄
情報処理学会データベースシステム研究会、情処研報 83-5 1991.5

Accessing Objects in Virtual Memory
Shinji Suzuki, Masaru Kitsuregawa, Mikio Takagi
電子情報通信学会「若手のためのデータベースワークショップ」 1991.3

VLSI ソートプロセッサ
喜連川優 鈴木慎司 楊維康
情報処理学会誌 VOL.31 No.4 1990.4

全国大会

永続プログラミングシステム P3L の OO1-Lookup ベンチマークを用いた性能評価
鈴木慎司 喜連川優 高木幹雄
情報処理学会第５０回（平成７年前期）全国大会論文集 1G-02

永続プログラミングシステム P3L の OO1 ベンチマークを用いた性能評価
鈴木慎司 喜連川優 高木幹雄
情報処理学会第４９回（平成６年後期）全国大会論文集 2U-8

サロゲート OID を用いたポインタ書き換え方式
鈴木慎司 喜連川優 高木幹雄
情報処理学会第４８回（平成６年前期）全国大会論文集 4G-4

永続的プログラミング言語 P3L 処理系の GCC と Exodus Storage Manager による実装
鈴木慎司 喜連川優 高木幹雄
情報処理学会第４７回（平成５年後期）全国大会論文集 D4-4

永続的プログラミング言語 P3L における分岐特性を考慮したレジデンシィ検査について
鈴木慎司 喜連川優 高木幹雄
情報通信学会 1993 年春期大会講演論文集

永続的プログラミング言語 P3L の実装におけるポインタ書き換え方式のコスト評価
鈴木慎司 喜連川優 高木幹雄
情報処理学会第４５回（平成４年後期）全国大会論文集 (4)

永続プログラミング言語のためのコピー方式ゴミ集めの実行時コストについての考察
鈴木慎司 喜連川優 高木幹雄
情報処理学会第４４回（平成４年前期）全国大会論文集 4G-10

永続性を備えた C 言語における可動オブジェクトの実装について
鈴木慎司 喜連川優 高木幹雄
情報処理学会第４３回（平成３年後期）全国大会論文集 6N-9

プログラミング言語 P3L における永続オブジェクト参照方式とその評価
鈴木慎司 喜連川優 高木幹雄
情報処理学会第４２回（平成３年前期）全国大会論文集 3L-9

オブジェクト指向データベースプログラミング言語 P3L の概要
鈴木慎司 喜連川優 高木幹雄
情報処理学会第４１回（平成２年後期）全国大会論文集 4F-11

永続的オブジェクト空間内のオブジェクトアクセス法について
- 効率的な画像・テキストデータベース処理に向けて -
鈴木慎司 喜連川優 高木幹雄
情報処理学会第４０回（平成２年前期）全国大会論文集 2L-11

ピラミッドアルゴリズムを用いた並列テンプレート
マッチングの実装とその評価
鈴木慎司 喜連川優 高木幹雄
情報処理学会第３９回（平成元年後期）全国大会論文集 3K-9

NOAA 衛星画像処理の並列化 - センサー較正 -
鈴木慎司 喜連川優 高木幹雄
情報処理学会第３８回（昭和６４年前期）全国大会論文集 7C-3


inches
cm
Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
C
Y
M
Kodak Gray Scale
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19