DIPS: an efficient pointer swizzling strategy for incremental uncaching environments

Jun-Ki Min *, Chin-Wan Chung

Division of Computer Science, Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, 373-1, Kusong-dong, Yusong-gu, Taejon 305-701, South Korea

Received 11 July 2002; received in revised form 11 January 2003; accepted 9 February 2003

Abstract

Pointer swizzling improves the performance of OODBMSs by reducing the number of table lookups. However, the object replacement incurs the unswizzling overhead. In this paper, we propose a new pointer swizzling strategy, the dynamic indirect pointer swizzling (DIPS). DIPS dynamically applies pointer swizzling techniques in order to reduce the overhead of unswizzling. DIPS uses the temporal locality information which is gathered by the object buffer manager. The information is used to select the object to whose pointers the pointer swizzling techniques are applied and to dynamically bind the pointer swizzling techniques using the virtual function mechanism. We show the efficiency of the proposed strategy through experiments over various object buffer sizes and workloads.

© 2003 Elsevier Inc. All rights reserved.

Keywords: Databases; Persistent objects; Smart pointers; Pointer swizzling

1. Introduction

In the past decades, there has been much research on persistent object systems such as OODBMSs since the object-oriented data model supports many important features such as the class concept, the aggregation relationship, and the inheritance relationship. Recently, due to the emergence of internet and mobile device, the data such as XML, spatial-temporal data, and multimedia data have dramatically increased. These data are commonly represented by the object-oriented data model (eXcelon, 2002; Haahr et al., 2001; Pinheiro et al., 2000). Thus, the proliferation of these kinds of data has awaken the interest of the research community.

The applications of OODBMSs deal with complex structured data. Also, they manage a large volume of data and are computation intensive. Thus, the efficiency of the performance is a key issue for the popularization of OODBMSs.

One of the performance improvement techniques for OODBMSs is pointer swizzling (Brahmath et al., 1998; Kemper and Kossman, 1995; Moss, 1992; McAuliffe and Solomon, 1995). In OODBMSs, each persistent object has a unique object identifier (OID) and the reference to an object is represented by the referenced object’s OID. In general, when an object is referenced, the system searches a table using an OID to find whether the object is in memory or not. This table lookup is a performance bottleneck in OODBMSs. The basic idea of pointer swizzling is to reduce the number of table lookups by replacing an OID with the main memory address of a persistent object.

However, since the applications of OODBMSs handle a massive amount of data, the overhead of unswizzling exists. When an object (called a victim) is replaced by the replacement algorithm (Effelsberg and Härdler, 1984; Johnson and Shasha, 1994; O’Neil et al., 1993) or saved in the storage system by the concurrency control algorithm (Franklin et al., 1997; Voruganti et al., 1999), the swizzled pointers which point to the victim become dangling pointers. In addition, the swizzled pointers in the victim are invalid since the swizzled pointers are not OIDs but main memory addresses. Thus, the system
must replace the dangling pointers and the swizzled
pointers in the victim with proper OIDs. This is un-
swizzling (Kemper and Kossman, 1995; McAuliffe and
Solomon, 1995).

According to the swizzling time and the structure of
objects, the pointer swizzling techniques are very di-
verse. In spite of this variety, some persistent object
systems (Reinwald et al., 1996; White and DeWitt, 1995;
Wilson and Kakkad, 1992) adopt a pointer swizzling
technique without considering their applications’ be-
haviors.

However, there is no one pointer swizzling technique
superior to the others over all situations (Kemper and
Kossman, 1995). Thus, there has been some work about
adaptive pointer swizzling techniques (Brahnmath et al.,
1998; Kemper and Kossman, 1995). However, since
these techniques fix the proper pointer swizzling tech-
niques at compile time, the performance of the appli-
cation can be reduced when the behavior of the
application changes in run time. Furthermore, to apply
these adaptive techniques, an application profile gener-
ator and/or a specific application compiler is needed in
order to select a proper pointer swizzling technique for a
particular application. This induces a difficulty in im-
plementing the components of a DBMS and a difficulty
of implementing application programs.

1.1. Our contribution

In this paper, we propose a new pointer swizzling
strategy, dynamic indirect pointer swizzling (DIPS).
Basically, our work can be applied to general persistent
object systems. However, since the representative of
persistent object systems is OODBMS, we present our
work based on OODBMSs. As mentioned above, to
blindly apply a pointer swizzling technique degrades the
performance. Thus, some adaptive techniques have been
proposed. However, these adaptive techniques fix the
pointer swizzling techniques at compile time. In contrast
to these adaptive techniques, DIPS applies the pointer
swizzling technique according to the status of objects at
run time. The contributions of our approach are sum-
marized as follows:

- Dynamic binding of the pointer swizzling technique
  at run time.
- Efficient gathering of the status of objects (i.e., tem-
  poral locality information) for the pointer swizzling
during run time.
- Utilization of a general compiler (e.g., GNU C++)
supporting the object-oriented paradigm.
- Modification of the object buffer manager in order to
  consider the structural relationships among objects.

In addition, we implemented DIPS and conducted an
extensive experimental study over various object buffer
sizes and workloads. Experimental results confirm that
DIPS is superior to the other pointer swizzling tech-
niques.

The rest of this paper is organized as follows: Section
2 presents the basic architecture and the behavior of
persistent object systems and describes various pointer
swizzling techniques. In Section 3, we introduce our
DIPS strategy and the intuitive behavior. Section 4
contains experimental results, which show the efficiency
of our proposed strategy. Finally, conclusions of our
study and some areas for future research are given in
Section 5.

2. Foundation

Above all, we present the procedural steps of the
manipulation of objects on typical OODBMSs (Cattell
and Barry, 2000) for understanding.

As shown in Fig. 1, user defined objects are specified
by the object definition language (ODL) which is pro-
cessed by the ODL preprocessor. The ODL prepro-
cessor generates corresponding metadata and a header file
which is used as a part of the application source code.
Users make application programs using the object ma-
nipulation language (OML) predefined by OODBMSs.
OML supports the mechanism to invoke queries and
procedures using programming languages such as C++,
JAVA, and Smalltalk. The procedures are for the op-
erations (e.g., create, retrieve, update and delete) on
persistent objects and transactions.

Since object-relational databases store and maintain
the objects based on the relational data model, some
features of the object-oriented data model (e.g., multiple
inheritance mechanism) may be restricted. However,
above procedural steps are similarly applied to object-
relational DBMSs (Reinwald et al., 1996) since the ob-
ject-relational DBMSs essentially build an OODBMS
view on the top of relational tables.

As shown in Fig. 2, the relationships between per-
sistent objects are represented by the smart pointers

![Fig. 1. Procedural steps of OODBMS.](image-url)
class Part: public d::Object {
    d_Ref<Connect> conn;
    int price;
    Part(); //default constructor
}

class Connect: public d::Object {
    d_Ref<Part> apart;
    Connect(); //default constructor;
}

d_Ref<Part> part;
...
total_price = part->conn->apart->price
    //part keeps an OID of an Part object
...

Fig. 2. An example of an OML source code.

(e.g., d_Ref(T) in Fig. 2) (Edelson, 1992). Although the class name and operations of the smart pointer are diverse depending on the OML syntax, we use the ODMG C++ OML syntax (Cattell and Barry, 2000) since the ODMG specification is considered as the OODBMS standard. The smart pointer keeps the corresponding object's identifier (OID) and the dereference is done when the dereference operators (e.g., d_Ref(T)::!) of the smart pointer are invoked. Thus, when a persistent object is accessed via a smart pointer, the smart pointer replaces OID with the main memory address of the persistent object, i.e., pointer swizzling occurs.

Now, we describe the various pointer swizzling techniques. Basically, the pointer swizzling techniques are classified into two groups (White and DeWitt, 1992). The first is the software check approach: it uses the table lookup to detect the access to non-memory resident objects. The other is the virtual memory mapping approach: it uses the virtual memory access violation (exception handling) to detect the access to non-memory resident objects (Wilson and Kakkad, 1995).

Generally, the virtual memory mapping approach precludes the object replacement and overcomes the lack of main memory space using the swap space of the client system (Kemper and Kossman, 1995; White and DeWitt, 1992; Wilson and Kakkad, 1992). However, this approach has the following disadvantages (Kemper and Kossman, 1995): (1) The required data size of an application cannot exceed the size of the virtual memory. (2) The number of page faults increases due to the low memory utilization since the unit of the virtual memory management is a page rather than an object and the virtual memory registration table. (3) The cost of a page fault is high since the page fault occurs via the exception handling.

Moss (1992) first classified the software check approach and analyzed each technique in no uncaching (i.e., non-replacement) environments. Kemper and Kossman (1995) analyzed each technique based on Moss's work in incremental uncaching environments. The software check approach is classified by (1) the time when the pointer swizzling occurs and (2) the structure of objects.

In the first classification, the software check approach is divided into eager and lazy swizzling. Eager swizzling swizzles all pointers before accessing them. Although this guarantees that all the pointers are swizzled, this induces unnecessary object fetches. Lazy swizzling swizzles a pointer when the pointer is dereferenced. Lazy swizzling must check whether the pointer is swizzled or not at each dereferencing time. However, lazy swizzling does not induce the unnecessary pointer swizzling of unused pointers and, therefore, the unnecessary disk I/O.

Additionally, following the structure of objects (see Fig. 3), the software check approach is divided into direct and indirect swizzlings. In direct swizzling, the swizzled pointer points to an object directly. In this case, when the victim is replaced by the object replacement algorithm, the swizzled pointers which point to the victim become dangling pointers. To avoid the dangling pointers, each object keeps backward pointers that point to swizzled pointers using a linked list, called RRL (Kemper and Kossman, 1995), or a hash table (McAliffe and Solomon, 1995) structure. In indirect swizzling, the swizzled pointer points to an object descriptor (OD). The object descriptor keeps the number (num_ref) of swizzled pointers which point to this object descriptor. When the object is replaced, the object descriptor is not removed from memory if this number is not 0. Thus, the dangling pointer problem could be avoided easily in the indirect swizzling technique.

As mentioned earlier, the emergence of internet and mobile device induces the proliferation of persistent object systems. To enhance the performance of systems,
3. Dynamic indirect pointer swizzling

In Section 2, we presented various pointer swizzling techniques. The blind swizzling of pointers incurs the unswizzling overhead in incremental uncaching environments. Therefore, we develop a new pointer swizzling strategy, called DIPS.

3.1. Using the temporal locality information

The performance of pointer swizzling is influenced by the temporal locality of the application and the system configuration. Thus, the basic idea of DIPS is that cold objects do not use the pointer swizzling technique.

Among the components of the object manager, the object buffer manager (i.e., the object replacement algorithm) deals with the temporal locality information to increase the hit ratio of the object in the buffer.

LRU (Effelsberg and Härder, 1984), a well known replacement algorithm, uses a simple heuristic that an object accessed at this time will be used again in the near future. Although it is very efficient, it is too simple to be used for the pointer swizzling information.

To the best of our knowledge, 2Q\(^2\) (Johnson and Shasha, 1994) is as efficient as LRU. Also, since 2Q quickly removes cold objects from the object buffer, the hit ratio is as high as LRU/2 (O’Neil et al., 1993). The basic behavior of 2Q is shown in Fig. 4. 2Q handles the buffer using two separated queues: A1IN and AM. A1IN acts as first-in-first-out queue and AM acts as LRU queue. A1IN and AM contain both objects and their descriptors. When an object which was not used in the near past is loaded, the object buffer manager inserts this object into A1IN. When an object which was used in the near past is loaded, the object buffer manager inserts this object into AM.

In 2Q, the history of the object replacement is maintained by A1OUT queue. A1OUT does not contain the object itself but only the object descriptor. When an object in A1IN is replaced, its descriptor is inserted into A1OUT. If the replaced object is a hot object, the access request for this object will be occurred while the descriptor of this object is in A1OUT. In other words, A1OUT acts as a filter for distinguishing hot objects from cold objects.

The sizes of A1IN, AM, and A1OUT are controlled by parameters, Kin, Km and Kout which are the fractions of the object buffer size such that Kin + Km = 1 and Kout > 0.

---

\(^2\)2Q was originally devised for the page buffer replacement algorithm. However, 2Q can be applied for the object buffer as well.
2Q, like other replacement algorithms, does not consider the structural relationships among objects. Thus, we modified 2Q for DIPS in order to consider the structural relationships among objects. The basic behavior of DIPS is shown in Fig. 5. The dotted lines in Fig. 5 represent the differences from 2Q.

First of all, we adopted the object descriptor as the queue element for all queues. This avoids an additional indirection overhead for the queue management. However, since the object descriptor whose num_ref is not 0 cannot be removed from A1OUT, this incurs the overhead of A1OUT queue handling. Thus, we made another queue, called AMOUT which only contains object descriptors. All object descriptors in AMOUT come from A1OUT or AM. When an object descriptor is removed from A1OUT or AM by the object buffer manager, num_ref of the descriptor is checked. If num_ref is greater than 0, the descriptor is inserted into AMOUT. Furthermore, if num_ref of an object descriptor in AMOUT is set to 0 as a result of unswizzling, the descriptor is removed from AMOUT. Thus, num_ref of any object descriptor in AMOUT is not 0. Since the objects whose descriptors are in AM are hot, the pointers of only those objects can be swizzled. This means that there are structural relationships between some objects whose descriptors are in AM and all object descriptors in AMOUT.

Let the object descriptor of the object A be in AM, the descriptor of the object B be in AMOUT and there be a structural relationship between A and B. When B is accessed via a swizzled pointer (i.e., a structural relationship) in A, the object B and its descriptor are moved to the front of AM since A is a hot object and the structural relationship between A and B had been used in the near past. Thus, our modified object buffer manager for DIPS efficiently uses the structural relationship as the buffer management information based on the lazy indirect pointer swizzling technique.

In our object buffer manager, most of all objects in AM queue are hot objects. In DIPS, the internal pointers of objects in AM can be swizzled only. Thus, when an object in A1IN is replaced, there is no overhead of unswizzling.

3.2. Dynamic binding of pointer swizzling

In the object-oriented programming environment, the reference is represented as the smart pointer (Edelson, 1992). The smart pointer overloads some operations (e.g., →, *) in order to use the normal pointer syntax of the general compiler (Edelson, 1992).

As mentioned earlier, one of our contributions is the utilization of a general compiler that supports the object-oriented paradigm. In DIPS, the smart pointer changes its behavior according to the temporal locality. Thus, in order to change the behavior of the smart pointer during run time, some operators of the smart pointer are implemented as virtual functions.

As shown in Fig. 6, the smart pointer contains a hidden pointer that points to a virtual function table, called vtbl. vtbl contains addresses of virtual functions. The vtbl pointer (called vptr), which is a volatile pointer, points to vtbl. Generally, vptr in an object is initialized by the constructor of an object implicitly.

As addressed in (Biliris et al., 1993), when a persistent object is loaded into main memory from disk, the hidden pointers such as vptr should be reinitialized since these hidden pointers are volatile pointers. Thus, to reinitialize the hidden pointers, ODL preprocessor generates the specific constructor which does not modify any attribute value of the object.

The specific constructor calls the specific constructors of all attributes of the class associated with the object. Thus, for the dynamic binding of the pointer swizzling technique, the object manager applies a particular constructor of the class when an object is fetched from the page buffer and its descriptor moves to AM. This constructor calls the proper smart pointer constructor (e.g.,

---

3 This technique is also applied to lazy indirect and eager indirect in the experiment.
the constructor of d_Ref(T)_S which initializes vptr of the smart pointer to point to vtbl containing the addresses of various virtual functions each of which performs pointer swizzling.

It is very important that vptr of the smart pointer changes according to the object’s temporal locality information. In DIPS, there are two vtbls. The other vtbl contains addresses of the identical virtual functions except that they do not perform pointer swizzling. This is the way to avoid additional software check whenever a pointer is dereferenced. Also, the call to the specific constructor has a little overhead because the constructor call for the hidden pointer initialization is processed whenever an object is fetched by the object manager.

3.3. Variation of DIPS

When the size of the object buffer is extremely large, the object buffer manager does not distinguish cold objects and hot objects since there is no object replacement. It degrades the performance of DIPS. Thus, we device another version of DIPS (called Full-DIPS). In Full-DIPS (see Fig. 5(b)), when an object in A1IN is accessed and the fault ratio is less than a threshold, its descriptor moves to AM and the eager indirect pointer swizzling technique is applied to the internal pointers of it using the reinit() function, for which there are two alternatives as follows:

- The reinit() function simply calls a specific constructor of the object, since the specific constructor does not change the attribute value but can change all the vtbl pointers in the object.
- The reinit() function gets the offsets of the smart pointers in the object from the schema manager, and then only calls the specific constructors of the smart pointers.

We choose the reinit() function using the schema information for Full-DIPS since the reinit() function using the schema information is a little bit more efficient than the reinit() function using the constructor which calls constructors of all attributes. However, the performance of Full-DIPS is lower than that of lazy indirect swizzling and DIPS due to the reinit()'s overhead and the threshold check overhead. In Section 4.2, we describe the analysis of the performance of Full-DIPS with experimental results in detail.

4. Experiment

4.1. Experimental environment

To show the efficiency of our strategy, we compare it with no swizzling, lazy indirect swizzling, and eager indirect swizzling. But, we preclude the virtual memory mapping approach in our experiments since our work is mainly focused on limited sized buffers which require the effective object replacement. The gain ratio is

\[
\text{Gain ratio} = \frac{(R_{time} - I{time_{NS}}) - (R_{time_{NS}} - I{time_{NS}})}{R_{time_{NS}} - I{time_{NS}}}
\]

As in the most related OODBMS literature, our work is mainly focused on reducing the access time to in-memory persistent objects, not on reducing the disk I/O time. Thus, to show the efficiency of our strategy, we extract the I/O time (I{time_{NS}}) of the application with no swizzling from the response time (R_{time}) of the application with each pointer swizzling since I{time_{NS}} is the default I/O time of the application and some pointer swizzling techniques (e.g., eager indirect) induce the additional disk I/O.

Also, we perform experiments over various object buffer sizes since the behavior of the object manager is dynamically changed by the ratio of the size of the object buffer to the amount of data required by an application. Thus, we change the size of the object buffer from 1% to 100% of the required data size of each traverse.

The experiments are performed on a Sun Ultra II 168 MHz platform with solaris 2.5.1 and 384 MBytes of

4 Since the performance of pure eager indirect swizzling is low on small sized object buffer, we implemented eager indirect swizzling such that pointer swizzling occurs at the call time of the smart pointer constructor.
main memory. We use GNU C++ compiler 2.8.1 as the application compiler.

The conceptual structure of the experimental system is shown in Fig. 7. In our experiments, we assume that each application keeps its own object buffer such as the object-server architecture (DeWitt et al., 1990). The experimental system consists of four major components: the application API, the object manager, the schema manager and the storage manager.

We use OMEGA 5.0 1.0 C++ OML class library, which is compatible with ODMG C++ OML (Cattell and Barry, 2000), as the application API.

The object manager consists of the object table (OT), the object buffer manager (see Section 3.1) and the transaction manager (TM). We implemented the OT as a hash table. We run the object buffer manager with parameters set to \( K_{in} (25\%) \), \( K_{out} (75\%) \), \( K_{out} (50\%) \) which are the values of the parameters recommended in 2Q (Johnson and Shasha, 1994). If super objects \(^6\) are replaced, the heap pointer (i.e., return pointer) in a traverse is invalid. Thus, we adopt the object pinning mechanism in order to avoid the replacement of the super objects in traverse. The TM keeps the information of dirty objects and saves dirty objects at transaction commit time to ensure the inter-transaction cache consistency (Franklin et al., 1997).

We use Shore 1.1.1 Storage Manager (Shore Project Group, 1995) for a storage manager.

The evaluation of the proposed pointer swizzling strategy is based on a variety of experimental tests on OO7 benchmark data (Carey et al., 1993) which is the famous benchmark data for OODBMSs. The structure of OO7 is illustrated in Fig. 8. We stored objects of OO7 data in disk following the first-come-first-serve policy. In our experiments, since we use the same data over various pointer swizzling techniques and preclude the disk I/O time, our performance analysis is not affected by the location of disk-resident objects (e.g., clustering effect). The configuration of OO7 for our experiment is summarized in Table 1. The DB size is 15 MBytes.

For showing the effect of the temporal locality and the cache consistency algorithm, we implemented following four traversal operations as the application program.

- T1: traverses the assembly hierarchy. As each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, perform a depth-first-search (DFS) on its graph of atomic parts (Carey et al., 1993). (The same as the definition in OO7.)
- T2a: traverses T1 and updates one atomic part per composite part (Carey et al., 1993). (The same as the definition in OO7.)
- T4: traverses the assembly hierarchy. As each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, visit all atomic parts of it in a breath-first-search (BFS) fashion but does not access any connect of atomic parts.

---

5 Object management systEm for geo-spatial applications (OMEGA) 1.0 is a spatial OODBMS which has been developed at KAIST.

6 Super objects are in the reference sequence from the root object to the lastly accessed object at the certain time of a traversal operation.

---

Table 1

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumAtomicPerComp</td>
<td>50</td>
</tr>
<tr>
<td>NumConnPerAtomic</td>
<td>3</td>
</tr>
<tr>
<td>DocumentSize</td>
<td>2000</td>
</tr>
<tr>
<td>ManualSize</td>
<td>100k</td>
</tr>
<tr>
<td>NumCompPerModule</td>
<td>500</td>
</tr>
<tr>
<td>NumAssmPerAssm</td>
<td>3</td>
</tr>
<tr>
<td>NumLevel</td>
<td>7</td>
</tr>
<tr>
<td>NumCompPerAssm</td>
<td>3</td>
</tr>
<tr>
<td>NumModule</td>
<td>1</td>
</tr>
</tbody>
</table>
• T5a: traverses T4 and updates one atomic part per composite part.

A specification of the above operations is the hierarchical navigation of complex assemblies and DFS/BFS traversal of atomic parts. Thus, the utilization of extents (i.e., the set of OIDs of a certain class) is not available. Although it depends on the OML specification, if the API of extents is supported as persistent objects, the behavior of accessing objects using an extent changes according to the temporal locality in DIPS. In our experiments, the behavior is partially reflected on T4 and T5a since these operations retrieve atomic parts of a composite part iteratively.

Each read-only traverse is run in “cold” and “warm” ways. In a cold run, the traverse begins with the empty database buffer. The warm run consists of the first cold run and then running of the same traverse three more times and reporting the average of the last three runs. The required data sizes of T1, T2a, T4, and T5a are 8.6, 8.6, 3.9, and 3.9 MBytes, respectively.

4.2. Experimental result

Table 2 summarizes the symbols and the names of the techniques that we use in the experimental results.

When the size of the object buffer is extremely small (1%) or extremely large (100%), it is observed that most of all objects are in A1IN. However, the meanings of two cases are different. When the size of the object buffer is extremely small, most of all objects are cold objects because the fault ratio is very high. When the size of the object buffer is extremely large, the object buffer manager cannot distinguish hot objects from cold objects since there is no object replacement. Thus, the performance of DIPS in these cases is slightly lower than that of no swizzling.

As shown in Figs. 9 and 10, DIPS is the most efficient when the size of the object buffer is from 10% to 60% of the required data sizes of T1 and T4. Moreover, lazy indirect swizzling is less efficient than no swizzling when the size of the object buffer is 1–30% of the T1’s required data size due to the unswizzling overhead. Eager indirect swizzling shows the worst performance over 1–70% sized object buffer due to the overhead of swizzling/unswizzling of unused pointers and, therefore, the unnecessary disk I/O overhead. Furthermore, eager indirect is less efficient than lazy indirect when the size of the object buffer is extremely large (100%). Generally, eager swizzling is more efficient than lazy swizzling for an extremely large sized buffer because there is no software check overhead in eager swizzling. However, in our experiment, eager swizzling loads more objects than the required objects of a traverse to ensure all pointers swizzled. Thus, there are the object replacement overhead and the unswizzling overhead. In addition, Figs. 9 and 10 show the gain ratios of Full-DIPS with threshold 0.2.

In the cold run of T1 (see Fig. 9(a)), the gain ratio of DIPS increases while the size of the object buffer is 1–70% and then decreases. As mentioned earlier, the object buffer manager for DIPS distinguishes hot objects from cold objects based on the object replacement history. When the size of the object buffer is in 1–70% of the required data size of T1, the object buffer manager distinguishes hot objects from cold objects in the cold run of T1. However, the more the object buffer size increases, the more the duration of hot objects in A1IN increases. Thus, more smart pointers behave as the pointers of no swizzling in the cold run.

<table>
<thead>
<tr>
<th>Sizzling technique</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>No swizzling</td>
<td>NS</td>
</tr>
<tr>
<td>Lazy indirect pointer swizzling</td>
<td>LIPS</td>
</tr>
<tr>
<td>Eager indirect pointer swizzling</td>
<td>EIPS</td>
</tr>
<tr>
<td>Dynamic indirect pointer swizzling</td>
<td>DIPS</td>
</tr>
<tr>
<td>Full-dynamic indirect pointer swizzling</td>
<td>Full-DIPS</td>
</tr>
</tbody>
</table>
The gain ratio pattern of Full-DIPS is similar to that of DIPS when the object buffer size is 1–70% in the cold run of T1. Since there is the threshold check overhead at each object access time in Full-DIPS, Full-DIPS is less efficient than DIPS. When the object buffer size is 80–100%, Full-DIPS shows the worst performance. There are two reasons. First, as mentioned earlier, more smart pointers behave as no swizzling pointer until the object fault ratio drops the threshold of Full-DIPS. Finally, as mentioned in Section 3.3, the reinit() function overhead occurs in the cold run.

In the warm run of T1 (see Fig. 9(b)), the gain ratio of DIPS still increases when the size of the object buffer is greater than or equal to 70% because the object buffer manager can distinguish hot objects from cold objects more exactly in the warm run. The gain ratio of DIPS approaches 30% when the size of the object buffer is 90% of the required data size.

Also, when the object buffer size is 70–80%, the gain ratio of Full-DIPS decreases since the reinit() function overhead exists in the warm run of T1. When the object buffer size is 90–100%, the reinit() function overhead does not exist in the warm run of T1.

As following the increment of the object buffer size, the object fault ratio decreases. Thus, the unswizzling overhead of the victim’s internal pointers injures less the effect of pointer swizzling. Therefore, the lazy indirect technique is the most efficient in the cold run and the warm run when the size of the object buffer is 70–100% of the T1’s required data size.

Compared T4 with T1 (see Fig. 10), the gain ratio of T4 is less than that of T1. In traverse T1, all atomic parts of a composite part are, at least, visited twice in order to check whether the atomic part is already visited or not. In contrast to T1, the atomic part of a composite part is visited only once in T4. Thus, the effect of pointer swizzling in T4 is less than that in T1. However, the pattern of T4’s gain ratio is similar to that of T1’s gain ratio. Also, the object fault ratio of T4 is higher than that of T1. Thus, in the cold run of T4, Full-DIPS with threshold 0.2 acts as DIPS. But, when the object buffer size is 90–100% in the warm run of T4, the performance of Full-DIPS is significantly reduced due to the reinit() function overhead.

Figs. 11 and 12 show the gain ratio of each pointer swizzling in traverses containing some update operations. As mentioned early, dirty objects are saved at commit time to ensure the inter-transaction cache consistency (Franklin et al., 1997). Thus, the benefit using pointer swizzling is injured by the unswizzling overhead at commit time. However, as shown in Fig. 11, DIPS is...
still most efficient than others while the size of the object buffer is 20–60% of T2a’s required data size.

Since T5a performs a BFS fashioned traverse, the pointer swizzling effect is less shown than T2a. Moreover, the unswizzling at transaction commit time injures the swizzling effect. However, DIPS is injured less than other pointer swizzlings. As a result, the performance of DIPS is not lower than that of no swizzling.

Consequently, DIPS is shown to provide good performance, particularly, when the size of the object buffer is smaller than the required data size of an application.

5. Conclusion

In this paper, we present a new pointer swizzling strategy, called DIPS. Our strategy is based on two ideas: (1) gathering the temporal locality information from the object buffer manager and (2) dynamic binding of the pointer swizzling using virtual function mechanism. Furthermore, our proposed strategy can be easily implemented using a general compiler.

The results of experiments indicate that the effect of pointer swizzling is injured by the unswizzling overhead in incremental uncaching environments. But, DIPS shows good performance for navigational and materialized accesses in incremental uncaching environments. The relative improvement obtained by DIPS over no swizzling is from 6% to 31% except when the size of the object buffer is extremely small or extremely large. The experiments with updates show that DIPS has a good property for incremental uncaching since the effect of pointer swizzling in DIPS is injured less by the unswizzling overhead at transaction commit time.

In this study, we only consider the temporal locality for pointer swizzling. Thus, for the future work, we would like to extend our pointer swizzling strategy in a way that the temporal locality information and the spatial locality information are considered together. Using both types of information together may improve the performance of DIPS when the size of the object buffer is extremely small or extremely large.

References


eXcelon Corporation, 2002. eXtensible Information Server (XIS).


Moss, J.E.B., 1992. Working with persistent objects— to swizzle or not to swizzle. IEEE TSE 18 (8), 657–673.


