The Object-Slicing Technique: A Flexible Object Representation and Its Evaluation *

Harumi A. Kuno, Young-Gook Ra, and Elke A. Rundensteiner
Dept. of Elect. Engineering and Computer Science
Software Systems Research Laboratory
The University of Michigan, 1301 Beal Avenue
Ann Arbor, MI 48109-2122
e-mail: kuno@umich.edu, ygra@umich.edu, rundenst@umich.edu
fax: (313) 763-1503, phone: (313) 936-2971

Abstract

Recently much work has been done towards extending object-oriented database systems (OODBs) with advanced tools such as view technology, advanced schema evolution support, and role modeling systems. These extensions all require that the underlying database system supports more powerful and flexible modeling constructs than are currently supported by existing OODB systems. In this paper, we identify these features as *multiple classification*, *dynamic reclassification*, and *dynamic restructuring*. We then describe a methodology known as *object-slicing* that is capable of extending data models to support these required features. We have successfully implemented an object-slicing software layer using the GemStone system, which while still providing full access to all GemStone DBMS functions, now also offers all required modeling features. In this paper, we describe our experimental results evaluating the relative costs and benefits of adopting the object-slicing technique. This includes an analytical assessment of the storage overhead of the object-slicing representation, and its comparison against the conventional representational models. As clustering is critical to optimizing queries on such models, we present the results of using OO7 benchmark test suites evaluating various clustering strategies for the object-slicing model. We find that for certain types of queries (e.g., those that benefit from the superior blocking factor at the local attribute level resulting from clustering object-slices together by class), the object-slicing model outperforms the conventional approach in spite of its storage overhead, while queries involving inherited attributes with low selectivity are better serviced using a conventional object-clustering approach.

Keywords: Multiple classification, object-oriented database models, performance evaluation, object clustering, benchmarking.

*This work was supported in part by the NSF RIA grant #IRI-9309076, the NSF NYI grant #IRI-9457609, and the University of Michigan Faculty Award Program. Harumi Kuno is also grateful for support from the NASA Graduate Student Researchers Program.

1 Introduction

Recently much work has been done towards extending object-oriented database systems (OODBs) with advanced tools such as view technology, advanced schema evolution tools, and role modeling systems [4, 7, 10, 13, 19, 21, 22, 25]. These extensions all require that the underlying database system supports flexible and powerful modeling constructs that are currently not supported by most existing OODB systems [3, 5, 15]. As we will describe in Section 2, such features include *multiple classification* (allowing an object to be an instance of multiple classes), *dynamic reclassification* (allowing an object to gain and lose class memberships throughout the object's lifetime), and *dynamic restructuring* (allowing an object's structure to change dynamically throughout the object's lifetime).

Unfortunately, this common need of view management, transparent schema evolution, role modeling systems as well as of many knowledge-based systems for flexible and fundamental data model characteristics is not met by current OODB technology. In fact, the object representation assumptions underlying most commercial OODB systems, namely, one most-specific type per object, object type determined at object-creation time, fixed typing, and upwards inheritance, conflict with the requirements of such systems. The *object-slicing* technique [16] is one particularly promising approach of extending an existing OODB system to support the identified required features.

In object-slicing, a real-world object corresponds to a hierarchy of **implementation objects** (one for each class whose type the object possesses) linked to a **conceptual object** (used to represent the object-itself) rather than associating a single implementation with each real-world object, as is commonly assumed in conventional OODB systems [16]. This technique of using implementation objects to represent an object's membership in multiple classes is extremely flexible, and provides a solution that extends an OODB system to support *capacity-augmenting virtual classes*, *multiple classification*, *dynamic reclassification*, and *dynamic restructuring* of objects and classes. For example, an object-slicing system can dynamically reclassify an object from being the instance of one class ($C1$) to becoming that of another class ($C2$), by linking the object instance to an implementation object of $C2$ and discarding that of $C1$.

In addition, object-slicing facilitates the maintenance of materialized views in that (1) it elegantly avoids the need to duplicate data for materialized classes and (2) any update to an object will take place at a unique location determined by the property involved regardless of the source of the update request [14]. Similarly, the flexibility offered by the object-slicing approach naturally lends itself to implementing role systems: object-slicing's implementation objects can easily be adapted to represent the various roles of objects in a role system [11].

We examine the object-slicing representation in the context of the University of Michigan's *MultiView* project, an on-going NSF-funded view management system capable of supporting updatable materialized views and transparent schema evolution. In order to support the features required by views and schema evolution, we have successfully implemented an object-slicing layer [20] on top of the GemStone OODB system [1]. Based on this flexible foundation, we have been able to rapidly prototype the *MultiView* system providing capacity-augmenting virtual classes, updatable materialized virtual classes, and view schemata. Because the object-slicing implementation could be cleanly built on top of an existing DBMS system, the question of how the addition of an object-slicing mechanism impacts performance now arises.

It is to be expected that extending an existing system with object-slicing techniques involves the potential overhead of additional data structures, maintenance costs, and processing time. Although object-slicing is a known technique that is being utilized for view systems [14], schema evolution [21], and role systems [11], to the best of our knowledge *no work has been done evaluating the costs of object-slicing*. The purpose of this paper is to provide such an evaluation.

In this paper, we present experimental results of utilizing our *MultiView* implementation to perform benchmark tests for evaluating the costs and benefits of the object-slicing paradigm. In the remainder of this paper, we will:

- Discuss data modeling requirements that can be addressed by an object-slicing solution (Section 2).
- Compare object-slicing with the conventional intersection class alternative (Section 3).
- Describe the object-slicing data model and its implementation (Section 4).
- Analyze the storage costs of our object-slicing representation (Section 5).
- Present the findings of experiments evaluating the object-slicing model compared to the conventional architecture (Section 6).

[1] GemStone is a registered trademark of Servio Corporation

- Present the experimental results of optimizing query types from the OO7 benchmark on the object-slicing model using clustering (Section 7).

This is followed by a discussion of related work as well as some concluding remarks.

2 Data Model Requirements of Advanced OODB Tools

As we will detail below, view management, transparent schema evolution, role modeling systems as well as many knowledge-based systems share a common need for the powerful data model characteristics of multiple classification, dynamic reclassification, and dynamic restructuring—which typically are not provided in current OODB systems.

View System Needs. In recent years, object-oriented view technology has been touted as an important technique for integrating heterogeneous and distributed systems, for achieving interoperability by hiding idiosyncrasies of component systems to be integrated into one unified, yet federated system, and for security [23, 22, 4, 7, 13]. View mechanisms typically provide the functionality to restructure a base schema by hiding classes, by adding classes, by customizing the behavior or extent of classes, and by rearranging the generalization hierarchy.

The introduction of virtual classes and schemata requires the underlying data model to support certain features. For example, if an object instance qualifies for membership in two virtual classes, then it should belong to both even if no subsumption relationship exists between the virtual classes. For this reason, the data model must provide *multiple classification*, which means that an object can belong to the local extents of multiple classes and can thus be associated with the types of multiple classes. Furthermore, an object instance should dynamically gain (or lose) the type of a *select virtual class* if its data values change so that it fulfills (or ceases to fulfill) the class's selection predicate. Thus, a view system must also support the *dynamic reclassification* and *dynamic restructuring* of object instances, allowing objects to flexibly gain and lose types during their lifetimes (including both the data stored in their state as well as the set of methods to which they can respond).

Extending Views for Schema Evolution. The use of view mechanisms to achieve schema evolution has been advocated by a number of researchers [25, 7, 4, 21]. The basic principle is that given a schema change request on a view schema, the system—rather than modifying the view schema in place—computes a new view that reflects the semantics of the schema change. This approach provides several advantages over direct modification. First, it lends itself naturally to a schema versioning system in which a new schema can be generated using views without destroying the old schema. Without schema versioning, updates to the shared database schema are almost always prohibited because of the risk of incompatibility between existing application programs and the modified schema. Second, the underlying instances are directly shared by different schema versions, which facilitates interoperability and ensures that old versions will always be kept up-to-date. In spite of these advantages, to the best of our knowledge an implementation of view mechanisms to achieve *transparent schema evolution* has not yet been realized. We suspect that one reason for this is that because views correspond to derived data, they by definition do not support the addition of new stored information to the database. Views therefore cannot simulate *capacity-augmenting* schema changes, such as the add-attribute operator.

In order to use view technology to support schema evolution, the traditional view management system must be extended to support the creation of *capacity-augmenting virtual classes*. A capacity-augmenting virtual class is a virtual class that includes, for example, instance variables that are not derivable from the source classes of the virtual class. Thus the underlying view system providing such capacity-augmenting views does require the *dynamic restructuring* of objects in addition to the features of multiple classification and dynamic reclassification.

Role Modeling System Needs. Finally, *role modeling* approaches have become increasingly popular [11, 19]. Role systems strive to increase the flexibility of the model by enabling objects to dynamically change types and class membership. In role modeling systems, objects dynamically gain and lose multiple interfaces (a.k.a. *roles*) throughout their lifetimes, hence the need for multiple classification. Such changes are done explicitly by user request, and on an object-by-object basis. Because role systems must reflect the evolution of an object as it dynamically gains and loses roles throughout its lifetime, the properties of *dynamic reclassification* and *dynamic restructuring* must also be supported by the data model underlying a role system.

3 Object-Slicing v.s. Intersection-Class Creation

3.1 An Introduction of Two Multiple-Classification Techniques

In Section 2, we identified the following object model requirements: (1) efficient dynamic restructuring of object representations, (2) multiple classification, and (3) dynamic reclassification in addition to the well-established object-oriented features such as encapsulation, generalization hierarchy, etc.. To the best of our knowledge, no current OODB system supports all of these features. Furthermore, with the exception of the IRIS functional database system [8], which uses a relational database as storage structure and stores data from one object across many relations, most OODBs represent each database object as a chunk of contiguous storage determined at object creation time. Thus they adhere to the invariant that *an object belongs to exactly one class* at a time — as well as indirectly to all of that class's superclasses [1].

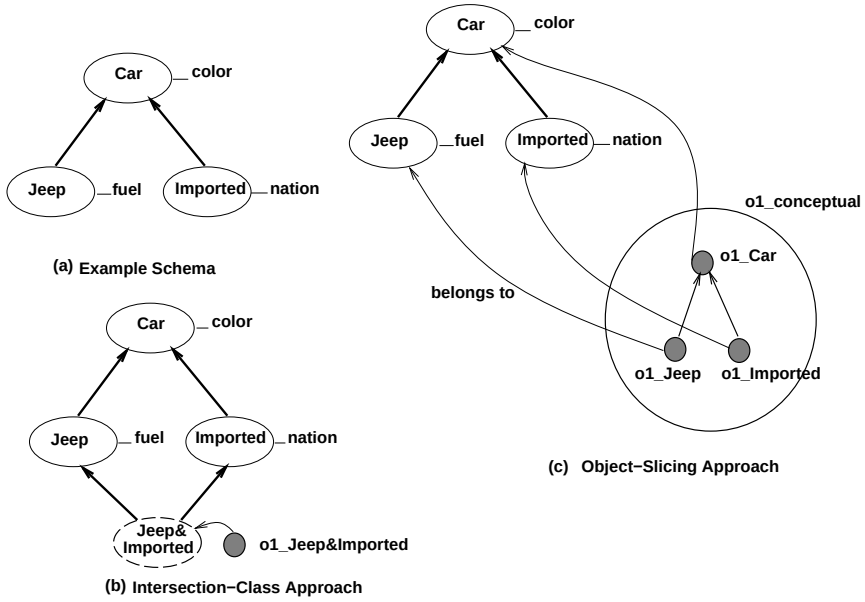

Figure 1: Two Approaches For Implementing Multiple Classification.

Object-slicing, as introduced in Section 2, is one way to extend an existing OODB system to provide the required features. The *intersection-class* approach is another. The intersection-class approach simulates multiple classification and dynamic restructuring in existing systems by creating *intersection classes* to reflect the structure of a multiply-classified object. For example, suppose that given the schema in Figure 1 (a), we want to create a new car object $o1$ that is both of type *Jeep* and of type *Imported*. We cannot find a class in which to store $o1$ without violating the invariant that an object belongs to exactly one class. To resolve this dilemma, the intersection-class approach would create a new intersection class *Jeep&Imported* that is a subclass of both the *Jeep* and *Imported* classes. It then would create $o1$ as member of the new class (Figure 1 (b)). Alternatively, given the same schema suppose that $o1$ were a member of the *Jeep* class and that we wanted to dynamically reclassify it to be a member of the *Imported* class. In this case, the intersection-class approach would require us to create a new object $o2$ as member of the *Imported* class, to copy all attribute values from $o1$ to $o2$, and finally to swap the object identities of these two objects. If $o1$ should not lose its membership in the *Jeep* class, then this dynamic reclassification of the object would again cause the creation of a *Jeep&Imported* intersection class.

The object-slicing approach (Figure 1 (c)) would implement the required multiple classification by manipulating the implementation objects representing the $o1$ object. As shown in the figure, the $o1$ conceptual object corresponds to a hierarchy consisting of the $o1_{Car}$, $o1_{Jeep}$ and $o1_{Imported}$ implementation objects. Each implementation object acts as its class's interface to the object. For example, when the current class of the $o1$ object is Jeep, the $o1_{Jeep}$ object represents the $o1$ object.

The object-slicing approach also enables efficient dynamic restructuring of object representations to account for the addition of new instance variables. Suppose that we were to extend our schema, which originally contains the *Car* and the *Jeep* classes, with a new class named *Imported* that refines the *Car* class by adding the stored attribute *nation*. Each *Car* object (as well as each *Jeep* object) can potentially acquire

	object-slicing	intersection-class
casting	change representative implement. obj.	need additional mechanism
#oids for one object	$1 + N_{impl}$	one
storage for managerial purposes	$(1 + N_{impl}) \cdot sizeOf(oid)$ $+2 \cdot N_{impl} \cdot sizeOf(pointer)$	sizeOf(oid)
storage for data values	no redundancy	no redundancy
#classes	#user-defined classes	#user-defined classes + #intersection classes (exponential)
performance for queries	good access to local attributes	fast access to inherited attributes
dynamic classification	by creating and destroying implementation object	by creating new classes, another object and copying values and removing old one
multiple inheritance resolution	has flexibility to dynamically resolve	must be determined statically

N_{impl} : the number of implementation objects for each object.

Table 1: Comparison of Two Multiple Classification Approaches.

the type of the *Imported* class. This means that the *Car* object representation should be restructured so that it can store the data for the new attribute. This can be accomplished simply by creating implementation objects of the *Imported* class and adding them to each *Car* object. Thus, the restructuring the object representation in the object-slicing paradigm is relatively efficient and simple when compared with the conventional architecture of having each object carry all of its state information in a contiguous fashion and permitting objects to belong to only one most specific class.

3.2 Comparing the Two Approaches

Each approach has its own advantages and disadvantages. A summary of the following comparison is also presented in Table 1.

- Casting an object to a type is readily provided by switching between the representative implementation objects in the object-slicing approach. However, in the intersection-class approach, we need an additional mechanism (possibly implemented by the compiler) to implement the casting facility.

- In the object-slicing approach, a single database object is represented by as many implementation objects as the number of types the object possesses, plus one conceptual object. Thus, the number of object identifiers needed to implement a single object is equal to $1 + N_{impl}$, where N_{impl} denotes the number of implementation objects needed by the object. The intersection-class approach requires only one object identifier per conceptual object.

- OODBs use storage for purposes other than storing data values, e.g., for indices and object identifiers. The object-slicing approach requires additional storage for the internal object identifiers described above ($N_{impl} \cdot sizeOf(oid)$) [2] and pointers to link the implementation and conceptual objects ($2 \cdot N_{impl} \cdot sizeOf(pointer)$) [3]. On the other hand, the intersection-class approach requires no additional storage, other than that required to create the intersection-class itself.

- Neither approach requires duplication of data values, and thus they are non-redundant in terms of the storage for data values. (Note that the designer is still able to duplicate data for performance reasons in both approaches.)

- The object-slicing approach does not require the creation of any hidden classes; all classes in the global schema are user-defined classes. However, the intersection-class approach requires intersection classes in order to accomplish multiple classification. For each object that takes two types, we must create

[2] Object identifiers are necessary for one conceptual object and N_{impl} implementation objects.
[3] Each implementation object keeps the pointer to its conceptual object, and vice versa.

a class to hold the combination of the two types, if it does not yet exist. The number of intersection classes may increase to $2^{N_{class}}$, where N_{class} is the number of user-defined classes of the global schema. In the worst case, the number of intersection classes could grow exponentially with respect to the number of user-defined classes. Also, as demonstrated above, dynamic classification may require the creation and/or removal of intersection-classes on the fly. The intersection-class approach thus requires *dynamic schema evolution support.*

- The state of a given object in the object-slicing approach is separated by class and distributed over a number of implementation objects. Because each implementation object is smaller than a complete object, the blocking factor of the implementation objects of a given class under the object-slicing model should be significantly better than the blocking factor of complete objects of the same class under a conventional architecture such as that of the intersection-class model. Sequential access of clustered implementation objects should therefore be faster than sequential access to the same number of complete objects. However, access to an inherited attributes in the object-slicing approach can involve the random access retrieval of conceptual object and multiple implementation objects. The intersection-class approach may therefore be faster in accessing an inherited attribute because the values of all attributes of an object reside in the same location. We present detailed comparison studies by benchmark experiments and simulations that confirm this hypothesis in Section 6.

- Changing an object from being an instance of one class ($C1$) to being an instance of another class ($C2$) is called *dynamic classification* [16]. In the object-slicing approach, when an object is dynamically reclassified to be an instance of the class $C2$ rather than one of class $C1$, the object instance takes an implementation object of the class $C2$ and discards that of the class $C1$. By combining the operators for adding and removing class membership, we can easily achieve this functionality. In the intersection-class approach, we first must identify the proper class for the new classification and, if it does not exist, create the class. This requires schema evolution support with classification capability. Second, we need to create an object of this new class and copy the values of the object to be reclassified into this newly created object. To preserve object identity, we must copy the object identity of the old object to the new object by utilizing a swap mechanism.

In conclusion, on most fronts the object-slicing approach promises comparable performance to the more conventional intersection-class method, and in some cases even improves performance. Furthermore, because object-slicing avoids the creation of cumbersome intersection classes and offers flexible restructuring of both state and behavior, it appears to be a cleaner method for extending existing OODBs. However, the additional overhead required for the object-slicing structures and the random accesses needed to retrieve the inherited state from implementation objects are disadvantages associated with the object-slicing methodology. In the remainder of this paper, we thus describe our experimental results in evaluating these two approaches. In Sections 5, 6 and 7, we perform a closer examination of the benefit provided by object-slicing's larger blocking factor at the class, the cost of the additional random accesses, and the potential of clustering to alleviate that cost.

4 The Object-Slicing Model

We formalize the object-slicing paradigm used in our implementation of *MultiView* below. Let $O_i \in O$ be a user-defined object. In the object-slicing model, O_i is represented using two kinds of objects: a single conceptual object, $O_{i_{conc}}$ and one implementation object for each class $C_j \in C$ to which the object belongs, $O_{i_{impl_{C_j}}}$. A **conceptual object** consists of a tuple, <***implObjects, OID***>, where OID is the unique, system-generated object-identifier of the conceptual object, and *implObjects* is the set of **implementation objects** that are linked to the conceptual object. An **implementation object** is a tuple <***OID,oid,class,state***> where OID is the object-identifier of the linked conceptual object, *oid* is the object-identifier of the implementation object itself [4], *class* is the class of which the implementation object is a direct instance, and *state* corresponds to the values of the local instance variables stored for the given

[4] Each implementation object by default possesses its own object identifier. However, because the implementation object serves as an interface for a specific conceptual object, the object-identifier of the conceptual object supersedes that of the implementation object for most practical purposes, such as determining object-equality.

object [5]. Each implementation object $O_{i_{impl_{C_j}}}$ is an object instance of the database class $C_j \in C$ it represents. Conceptual objects are object instances of a special system class, *ConceptualObject*, rather than of a user-defined class.

Because a single real-world object is now represented using multiple database objects, we define a number of functions to operate on objects in the model, including object creation, equality comparison, etc. For example, we say that two objects are *equal* if and only if they are linked to the same conceptual object. Similarly, given an object $O_i \in O$ and a class $C_j \in C$, we define functions **MakeImpl**($O_{i_{conc}}$,C_j) and **DeleteImpl**($O_{i_{conc}}$,C_j) to create and destroy implementation objects of class C_j for object $O_{i_{conc}}$.

If an object O_i possesses an implementation object $O_{i_{impl_{C_j}}}$ for some $C_j \in C$, then O_i must also possess implementation objects for all classes C_k s.t. C_j *is-a* C_k. Thus the set of implementation objects associated with a given conceptual object mirrors the structure of the class hierarchy. Object-slicing intrinsically includes its own inheritance mechanism. Let there be an implementation object $O_{i_{impl_{C_j}}}$, $O_i \in O, C_j \in C$. If the method $m_k \in M$ were to be invoked upon object $O_{i_{impl_{C_j}}}$, and m_k is not defined locally in **type**(C_j), then $O_{i_{impl_{C_j}}}$ will delegate the method m_k to $O_{i_{conc}}$. It will in turn conduct a search "upwards" through C_j's superclasses. If method m_k is not found in the type of a superclass C_l of C_j, then an error is returned; otherwise, the method is invoked upon the object $O_{i_{impl_{C_l}}}$ [6]. Figure 2 illustrates this idea using an example.

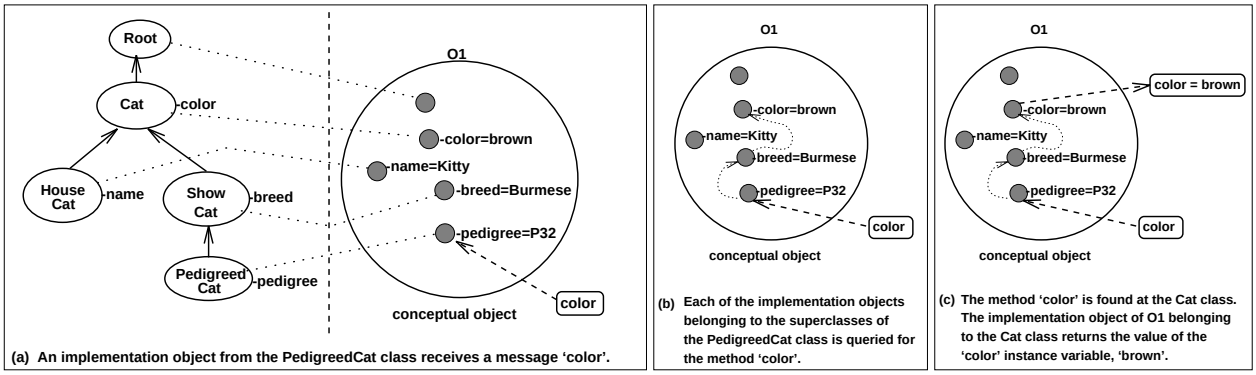

Figure 2: Object-Slicing Example

Because objects in the object-slicing model can dynamically add and lose implementation objects, and because the object-slicing model includes its own inheritance mechanism, the object-slicing technique can be used to extend object-oriented data models to provide the features of *multiple classification*, *dynamic reclassification*, and *dynamic restructuring*. It is relatively straightforward to see how one could implement the object-slicing model on top of an existing OODB. In our technical report [20], we describe our solution of realizing the model using a Smalltalk-based OODB, GemStone system. Due to space limitations, a discussion of these implementation issues is not repeated here.

5 Object-Slicing Storage Costs

In evaluating the performance of databases, I/O operation time typically dominates CPU operation time. Consequently, an evaluation of object-slicing must consider the effect of object-slicing on I/O time.

[5] The methods used to set or retrieve an instance variable's value are called **accessing methods**. Accessing methods are always located at the same class as the instance variables for which they are defined, and thus when an instance variable migrates from one class to another, that instance variable's accessing methods must make the same migration.

[6] If a method with selector m_i is found in more than one class in the superclass hierarchy of C_j, then the user is prompted to **cast** O_i into a non-ambiguous implementation object.

One major variable for calculating I/O time is the number of objects that can be stored in a disk block, known as the *blocking factor* (bf), namely: ⌈ disk block size / object size ⌉.

In traditional (non-object-slicing) architectures, the size of an object is calculated to be the total amount of storage needed to store the state of the object (data size), oid size, and pointer size (to reference the object's class), and some fields for the system use. Because the object-slicing model represents any given object using a conceptual object and some number of implementation objects, objects in the object-slicing architecture inherently require more storage space than their counterparts in traditional architectures. Like a traditional object, an object under the object-slicing architecture contains data, an oid, and a pointer to its class. In addition, an object-slicing object that belongs to l classes also requires l implementation objects (each with a reference to its class and to the object's conceptual object), the conceptual object (which has a dictionary of references to its implementation objects, respectively indexed by the class the implementation object belongs to), and the system fields [7]. That is, while in a *conventional architecture* we would have:

$$obj\ size = data\ size + oid\ size + pointer\ size + system\ fields;$$

in the *object-slicing architecture* we now have:

$$obj\ size = data\ size + (l+1) \cdot oid\ size + (4l+1) \cdot pointer\ size + system\ fields.$$

To simplify, we assume that oid size is equal to pointer size. The ratio of the sizes is now:

$$SizeRatio(SR) = \frac{datasize + (5l+2) \cdot pointersize + systemfields}{datasize + 2 \cdot pointersize + systemfields}.$$

Ignoring the size of the system fields for simplicity and assuming that oid size = pointer size, the ratio becomes:

$$SR = \frac{DS + (5l+2) \cdot pointersize}{DS + 2 \cdot pointersize},$$

where SR is the size ratio and DS is data size. When the size of the pointer is 4 bytes, which is a reasonable assumption for current systems, this becomes

$$SR = \frac{DS + (5l+2) \cdot 4}{DS + 8}.$$

Given these assumptions, Figure 3 shows the blocking factor ratio for the fixed data size values of 10, 20, 30, 50, 100, and 1000 bytes while increasing l, the number of implementation objects required, from 1 to 20. On the other hand, Figure 4 shows the blocking factor ratio by fixing the number of implementation objects and varying data size. In general, the ratio increases as the value of l increases and decreases as the data size increases. This means that the disadvantage of the object-slicing architecture's storage overhead is ameliorated by an increased object size/decreased schema size (depth of schema) ratio.

6 Evaluation of Object-Slicing Approach Using the OO7 Benchmark

In order to determine the base cost of implementing the object-slicing representation paradigm, we have run several test queries from the University of Wisconsin's OO7 benchmark suite [6] with the intention of comparing GemStone's native implementation versus our object-slicing extension to GemStone. GemStone is a Smalltalk-based system, while the four systems compared in the OO7 benchmark paper [6] are all C++ based. Because GemStone thus supports dynamic method resolution, run-time augmentation of the schema with new methods, etc., we did not compare GemStone against other systems, and instead limited our study to comparing "pure" GemStone with *MultiView*. For this study, we used GemStone, version 3.2 Opal; and created a randomly populated database of the parts-assembly benchmark example with 10,000 Atomic Parts.

First, we compare results for navigation-type queries, e.g., for the "Traversal 1" query. The "Traversal 1" query tests raw pointer traversal speed with a high degree of locality [6]. The query requires a traversal of

[7] While this storage of references linking conceptual to implementation objects and back could be reduced, we've chosen this representation for reason of efficiency.

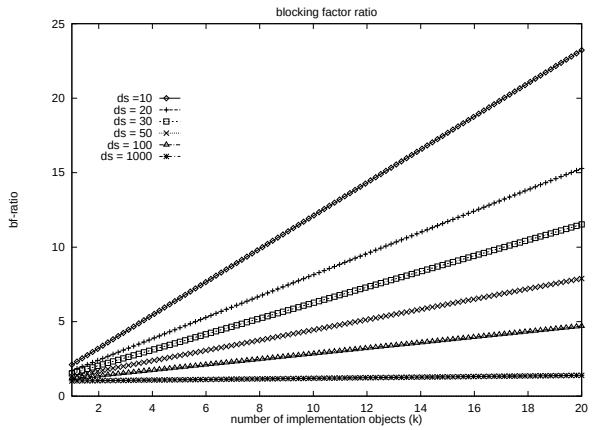

Figure 3: Blocking Factor Ratio with Fixed Data Size

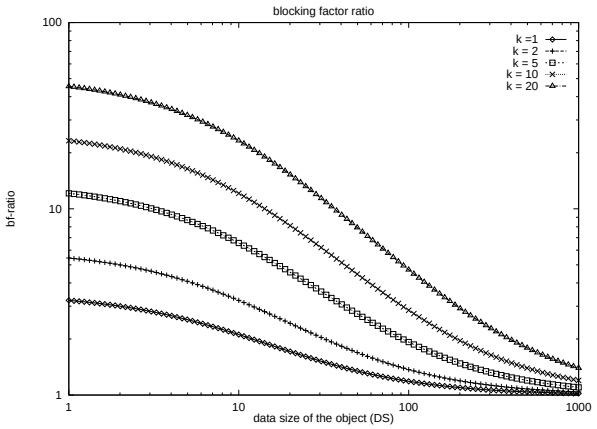

Figure 4: Blocking Factor Ratio with Fixed Number of Implementation Object

the assembly hierarchy shown in Figure 5 and performs a depth-first search on each part's graph of atomic parts. *MultiView*'s performance slightly improved upon GemStone's time (by $\approx 4\%$) despite the fact that *MultiView* is built on top of GemStone rather than directly into the GemStone kernel—thus adding an extra layer of indirection [8]. The improved performance can be explained as follows. First, the navigation was limited to access of local instance variables (rather than inherited ones). Thus there is no overhead of finding appropriate implementation objects for *MultiView*. Consequently, only one implementation object has to be retrieved per queried object. The query avoids having to perform random accesses to retrieve additional implementation objects. Not only was retrieval of the single implementation objects sequential, but also these navigated implementation objects are much smaller in size (containing only local instance variables) compared to GemStone's native objects (containing both local and inherited instance variables in one contiguous allocation) and thus offered a higher blocking factor (See Section 5 for a more formal analysis of these factors).

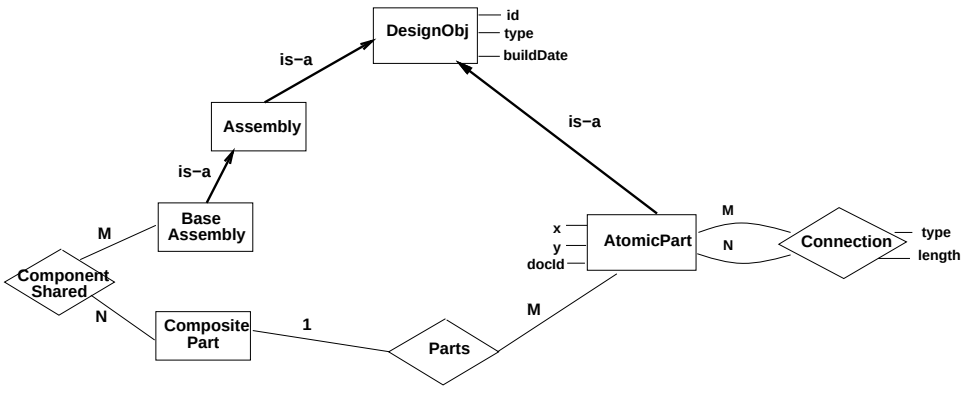

Figure 5: A Portion of OO7 Benchmark Database.

Next, we tested the "Query 1 Search" benchmark, which performs a random lookup of an atomic part based on the inherited property "id" as shown in Figure 5. For this lookup, *MultiView* performed 3-1/2 times slower than GemStone. The major overhead stems from the fact that for each of the 10,000 lookups, we must traverse from each Atomic Part's implementation objects to the corresponding conceptual object, and then to its corresponding DesignObj's implementation object in order to retrieve the "id" data value. Accessing an inherited attribute in *MultiView* requires two additional traversals: one to traverse the

[8] We do not have access to the GemStone code to override the kernel with *MultiView*.

`conceptualLink` to get from the implementation object to its conceptual object, and the other to traverse the `implementationLink` to get from the conceptual object to the correct implementation object holding the desired data value. As demonstrated in this paper, these two links improve the flexibility of our object model greatly but it also causes the observed performance degradation.

In short, there are two possible causes for the degradation in performance associated with object-slicing. First, the storage overhead to store the conceptual object, the oids' of the implementation objects, and those two extra links may require a larger number of blocks and result in more page faults. Second, because the implementation objects belonging to the same class are by default clustered together, each traversal to an implementation object for getting an inherited value requires a partial random block access, which can result in a page fault.

In the next section, we investigate the impact of the object-slicing mechanism upon an implementation's storage overhead in terms of the *blocking factor* of various objects and also in terms of the page faults incurred by the retrieval of object-slicing implementation objects. We also evaluate the ability of clustering strategies to alleviate these object-slicing penalties.

7 Evaluating and Optimizing Object-Slicing Using the OO7 Benchmark

7.1 Clustering Strategies for Object-Slicing

By specifying that certain categories of objects be placed in contiguous storage on the disk (thereby *clustering* them), database designers try to match the traversal patterns of objects in a database to storage sequences of objects to minimize disk I/O costs. In traditional (non-object-slicing) environments, the granularity of this optimization is often restricted to the clustering of objects. A *MultiView* object's state is distributed among multiple object-slicing implementation objects, however, which lends itself to clustering strategies that resemble the vertical partitioning of the relational model. In order to determine under which circumstances it is preferable to cluster the implementation objects by class (which we call *class clustering*), and under which circumstances it is better to cluster all the components of a *MultiView* object together (which we term *object clustering*), we designed and carried out an extensive experimental study evaluating both clustering techniques.

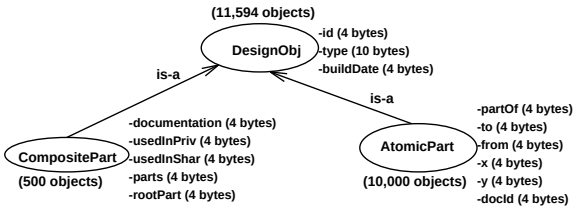

Figure 6: Subschema of OO7 Database.

For our evaluation, we again use a subsection of the OO7 Benchmark's example database (Figure 5). Figure 6 shows the three class subschema that is central to our test queries. Figure 7 illustrates clustering options of object versus class clustering using an example. In this example, there are three objects that are instances of the AtomicPart class from the schema shown in Figure 6. Figure 7(a) depicts the initial object relationships, in which there are three objects—Object1, Object2, and Object3—each of which has two implementation objects (one for the DesignObj class and one for AtomicPart). If we were to cluster the database objects by class, as shown in Figure 7(b), then we would cluster the implementation objects of each class and the conceptual objects separately. If, however, we were to cluster by objects, then all the data associated with each individual object would be clustered together, as shown in Figure 7(c). The second option resembles clustering at the granularity of complete objects.

Tables 2 and 3 compare the relative object sizes and blocking factors involved with the object clustering and class clustering strategies. The classes in the OO7 schema are listed as the rows of each table. The columns of Table 2 indicate, respectively, the number of objects that initially exist in each class's extent (#impls), the number of objects that have the given class as a most-specific type (#objs), the average size of an object in the class's extent using the class clustering methodology (CC size), the blocking factor of the class using the CC strategy (CC BF), the number of blocks that would be needed to store the extent of

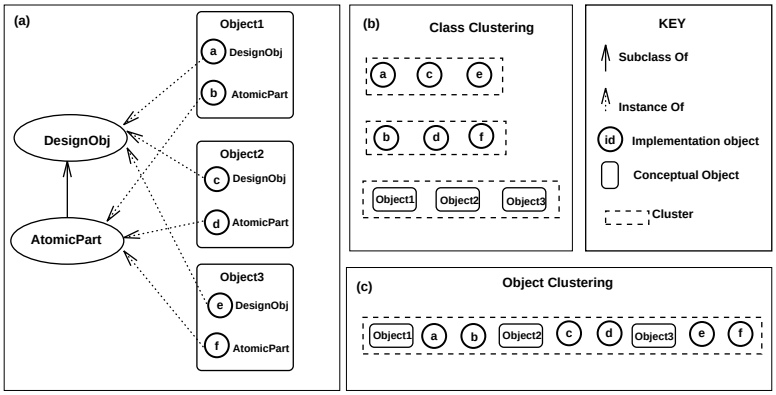

Figure 7: Example of Object Clustering v.s. Class Clustering.

	#impl	#objs	CC size	CC BF	#CC	OC size	OC BF	#OC
DesignObj	11594	0	30	17	680	54	9	0
Assembly	1093	0	20	25	43	74	6	0
BaseAssembly	729	729	20	25	29	86	5	146
ComplexAssembly	364	364	16	32	12	90	5	73
AtomicPart	10000	10000	36	14	704	82	6	1667
CompositePart	500	500	32	16	32	78	6	84
Connection	30000	30000	34	15	1993	50	10	3000
Module	1	1	24	21	1	78	6	1

Table 2: **Data sizes with 512 byte blocks.**

	# Objects	Conc. Obj. Size	Conc. Obj. BF	Conc. Obj. Blocks
DesignObj	0	12	42	0
Assembly	0	16	32	0
BaseAssembly	729	20	25	30
ComplexAssembly	364	20	25	15
AtomicPart	10000	16	32	313
CompositePart	500	16	32	16
Connection	30000	12	42	715
Module	1	16	32	1

Table 3: **Conceptual Object sizes for ClassClusterDictionary with a blocksize of 512 bytes.**

the class using the CC strategy (#CC), the average size of an object in the class's extent using the object clustering (OC)methodology (OC size), the blocking factor of the class using OC (OC BF), and the number of blocks that would be needed to store the extent of the class using OC (#OC).

The columns of Table 3 indicate, respectively, the number of objects that have that class as most specific type, the size of a conceptual object that has that class as most specific type, the blocking factor of such a conceptual object, and the number of blocks needed to store the conceptual objects of that class's extent. For example, we can see from Table 2 that there are 10,000 instances of the AtomicPart class. If we were to cluster implementation objects according to their classes, then the blocking factor of the AtomicPart class's extent would be sixteen objects per 512 byte block. If, on the other hand, we were to cluster all of an object's conceptual object plus all of its implementation objects together, then the AtomicPart class's extent would have a blocking factor of only six objects per 512 byte block.

We ran several benchmark queries adopted from the OO7 benchmark test suite on *MultiView* to compare the performance of the class clustering and object clustering strategies. We modified the accessing methods of all objects in *MultiView* so as to log all object accesses and method invocations. We used the resulting logs as input to trace-driven simulations of a least-recently-used (LRU) buffering policy. The simulator takes as input the trace data, a clustering policy, and the amount of memory in the simulated machine, and performs the following tasks:

1. Uses the specified clustering policy to map each data item in the trace data to a simulated block.
2. Simulates the effect of the clustering policy given the specified amount of memory and the access patterns contained in the trace data.
3. Returns the total number of block misses incurred by the simulation. (A block miss occurs when an object that does not reside in main memory is accessed.)

We use this simulator to compare the number of block misses that are incurred under the class clustering and object clustering strategies to the number of block misses that would occur if we did not use an object-slicing mechanism. Thus each of the following graphs plots three lines:

1. The block misses incurred by an object-clustering policy.
2. The block misses incurred by an class-clustering policy.
3. The block misses incurred by a non-object-slicing implementation.

From these graphs we should be able to identify the degree to which either the object-clustering or class-clustering strategy approaches the memory size / blocks missed ratio of a non-object-slicing implementation.

7.1.1 Retrieving Local Attributes

Class clustering and object clustering are each associated with inherent strengths and weaknesses. The chief advantage of class clustering is that it greatly increases the blocking factor of the objects in each class's extent, thereby facilitating the retrieval of local attributes. A comparison of the blocking factors of the two strategies for the various classes is presented in Tables 2 and 3. From these tables, we can see that the number of blocks needed to store the extent of the classes in our example schema using the class clustering strategy is between one half and one third the number of blocks that would be needed if an object-clustering strategy were used. This ratio clearly depends on other factors, such as the datasize and the number of implementation objects, as indicated by the blocking factor ratios depicted in Tables 2 and 3.

In Test 1, we compare the performance of object clustering and class clustering regarding the retrieval of objects' local attributes. In this test, we generate 10 random numbers, n, then search the extent of the AtomicPart class looking for a part that has a Document Id equal to the random number n. Note that docIdArray is an attribute defined by the AtomicPart class, and is thus an attribute local to the AtomicPart implementation objects.

Query Q1:

```
index := (Rand RandomNumberLessThan: (docIdArray size)) + 1.
document  := docIdArray at: index.
(AtomicPart extent) select: [ :elem | elem docId = document].
```

This test is equivalent to performing ten sequential searches of the 10,000 objects in the extent of the AtomicPart class while retrieving a locally defined attribute.

The class clustering strategy produces a blocking factor of sixteen objects per block for the AtomicPart class, versus only six objects per block using the object clustering strategy. We thus would expect the class clustering strategy to produce about $\frac{1}{3}$ of the block misses of the object clustering strategy. Our experiments confirm these expectations. Namely Figure 8 shows a 37% decrease in block misses between the object clustering and class clustering strategies.

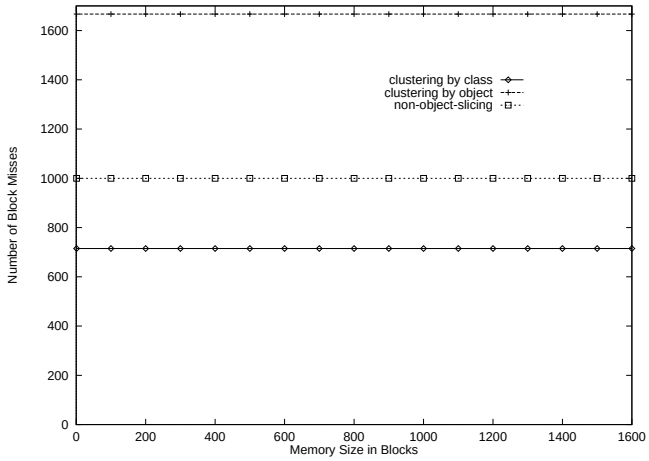

Figure 8: Block Misses for Retrieving Local Attribute

The vertical axis of Figure 8 indicates the number of block misses, while the horizontal axis indicates the memory size in blocks. Because the AtomicPart class's extent is stored on 715 contiguous blocks under the class clustering strategy and 1667 contiguous blocks under the object clustering strategy, even with an infinite amount of memory, there will be a minimum of 1667 block misses in the object-clustering case, and 715 block misses in the class-clustering scenario. The third line shows the blocks misses that would be incurred if we did not use object-slicing. In that case, the AtomicPart class's extent would be stored on 1,000 contiguous blocks, and the first query would result in a minimum 1,000 block misses. As the number of blocks in memory decreases, the number of block misses should naturally increase. Because accesses are sequential, both class clustering and object clustering object layouts are optimal for this query, in that every object will be accessed exactly once, and no object will be accessed after any of the objects that succeeds it in storage has been accessed. Therefore, independent from the number of blocks in memory, the class clustering strategy will result in 715 block misses and the object clustering strategy will result in 1667 block misses. The class clustering strategy is the clear winner for these types of queries. It improves upon a non-object-slicing implementation and is superior to the object-clustering strategy for the retrieval of local attributes.

7.1.2 Sequential Retrieval of Inherited Attributes

In the case of queries that retrieve inherited attributes, we might expect object clustering to outperform class clustering. While the object clustering strategy can access methods in a sequential manner, in the class clustering scenario each retrieval of an inherited attribute requires a semi-random access to retrieve the object's conceptual object and a semi-random access to retrieve the implementation object associated with the inherited property [9]. Our second test, Test 2, iterates through the extent of the AtomicPart class, comparing each object's inherited buildDate instance variable to a randomly generated date:

Query Q2:

[9] Semi-random because the sets of implementation objects and conceptual objects are themselves clustered sequentially.

```
aDate := RandomNumber.
(AtomicPart extent) select: [ :elem | elem buildDate < aDate].
```

Test 2 measures the impact of semi-random accesses caused by retrieving an inherited attribute from the 10,000 objects in the AtomicPart extent. In this test, the buildDate of each object in the AtomicPart extent was compared to a randomly generated number. Test 2 is equivalent to performing a sequential search of the 10,000 objects in the extent of AtomicPart class and retrieving a property inherited from the DesignObj class from each of them.

Recall from Section 4 that whenever an inherited instance variable of an implementation object is accessed, three objects must be accessed in order to retrieve the inherited instance variable–the recipient object (the object that is initially accessed), the recipient's conceptual object, and the implementation object where the inherited instance variable is defined. For example, if we were to apply this query to the example scenario shown in Figure 7, the sequence of objects we retrieve might be as follows:

1. `Obj b` (O1's AtomicPart object, accessed when it receives buildDate)
2. `Obj 1` (O1's conceptual object, accessed when `Obj b` does not understand buildDate)
3. `Obj a` (O1's DesignObj impl. object, accessed because buildDate is defined at DesignObj)
4. `Obj d` (O2's AtomicPart impl. object, accessed when it receives buildDate)
5. `Obj 2` (O2's conceptual object, accessed when `Obj d` does not understand buildDate)
6. `Obj c` (O2's DesignObj impl. object, accessed because buildDate is defined at DesignObj)
7. `Obj f` (O3's AtomicPart impl. object, accessed when it receives buildDate)
8. `Obj 3` (O3's conceptual object, accessed when `Obj f` does not understand buildDate)
9. `Obj e` (O3's DesignObj impl. object, accessed because buildDate is defined at DesignObj)

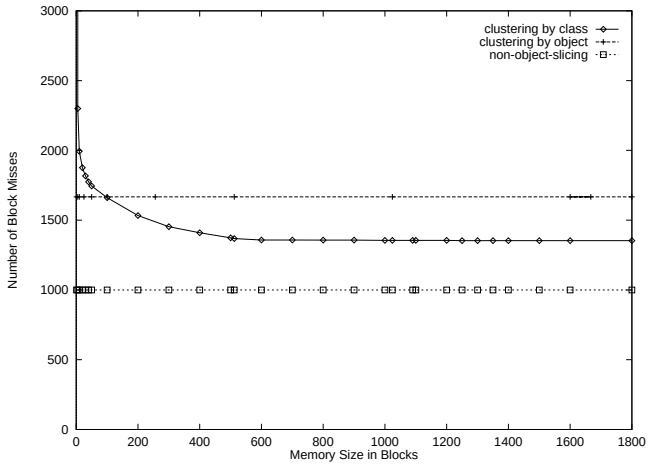

Figure 9: Block Misses Involved in the Sequential Retrieval of an Inherited Attribute

The results of this test (Figure 9) show that the cost of semi-random access to inherited attributes needed by the class clustering strategy is counterbalanced by the superior blocking factors at both the DesignObj and AtomicPart classes. Although for small memory sizes object-clustering causes fewer faults than class clustering, for larger memory sizes class clustering out-performs object clustering and more closely approaches the baseline of a non-object-slicing implementation. Note that the number of object-clustering misses remains constant regardless of the number of blocks in memory. This is because from the object-clustering perspective, we are simply performing a sequential search of the AtomicPart class. On the other hand, because the class clustering approach automatically references three different blocks for every object

processed, when the number of blocks in memory dips below three, class clustering results in 30,000 block misses (one for each implementation or conceptual object accessed).

In order to refine the results from Test 2, we next set out to vary two parameters that affect the performance of object clustering and class clustering. First, Test 2 involved only a single inherited attribute. Since the block miss gap between class and object clustering was reduced by the presence of a single inherited attribute, we designed Test 3 to measure the effect of an increased number of inherited attributes (each inherited from a different class). Next, note that the 10,000 implementation objects associated with AtomicPart objects make up a significant percentage of the 11,594 implementation objects in the extent of the DesignObj class. In Test 4 we compare class and object clustering performance in a scenario where the inherited attribute is inherited from a class where the queried objects make up only a small portion of the class's extent—that is, where a condition of low selectivity exists between the original and inheritance class.

7.1.3 Retrieval of Multiple Inherited Attributes

In Test 2, inherited attributes are retrieved with semi-random accesses. Because object clustering permits a given object's state to be retrieved in a sequential manner (whereas class clustering requires at best semi-random accesses for inherited attributes), we designed Test 3 to increase the number of classes involved in the access pattern. Test 3 traverses the assembly hierarchy of the root Module object, visiting all components and their sub-components in a recursive fashion.

Query Q3:

```
method: Assembly
TraverseAH
(self isInstanceOf: BaseAssembly )
  ifTrue: [ ^(((self asClassOf: BaseAssembly) componentsPriv)
                do: [ :elem |
              visitedNode at: nodeNumber put: nodeCount ])
          ]
  ifFalse: [ ((self asClassOf: ComplexAssembly) subAssemblies)
                do: [ :elem | elem TraverseAH].
  ].
%

run
aModule := (Module extent) at: 1.
aModule designRoot TraverseAH.
%
```

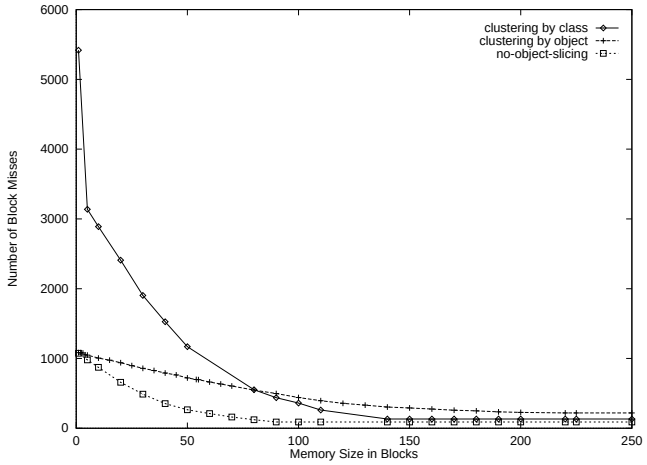

Figure 10: Random Access on Inherited Attribute

Graph 10 compares the block misses that are incurred by the object and class clustering layouts. Because this query does not use all of the state of the accessed objects, we benefit from the superior blocking factor at the implementation object level offered by the class clustering approach. Under the object clustering scenario, the accessed objects reside on 220 unique blocks, whereas under the class clustering scenario, only 107 unique blocks are retrieved during Test 3. The graph shows that object clustering is better for small amounts of memory, but that class clustering's performance more successfully approaches non-object-slicing performance as the number of blocks in main memory increases. Class clustering is better for larger amounts of main memory due to the inherently higher blocking factors of class clustering. Under the object clustering, when an instance is brought into main memory, all the constituent implementation objects residing in the same block must also be brought into the memory—including implementation objects that are not needed by the query being performed. Object clustering thus accesses a number of unnecessary blocks. However, because the implementation objects of an object under the class clustering scenario inhabit separate blocks, class clustering requires cross-block references to access the values of the inherited attribute. Thus, when only an extremely small number of blocks can be stored in memory, some of the necessary blocks will be swapped out from memory before they are re-referenced by the query.

7.1.4 Retrieval of Inherited Attributes with Low Selectivity

In Test 4, we compare class and object clustering performance in a scenario where the inherited attribute is inherited from a class where the queried objects make up only a small portion of the class's extent. Test 4's query iterates over the extent of the CompositePart class, retrieving each object's DesignObj implementation object.

Query Q4:

```
(CompositePart extent) select: [ :elem | elem buildDate < RandomNumber]
```

Since CompositePart objects make up about 5% of the DesignObj class's extent, the retrieval of each CompositePart instance's buildDate requires a random access into the extent of the DesignObj class's extent. Graph 11 contrasts the impact of the two clustering strategies on the block misses incurred by query 4. The graph shows that under conditions of low selectivity, object clustering is far better than class clustering.

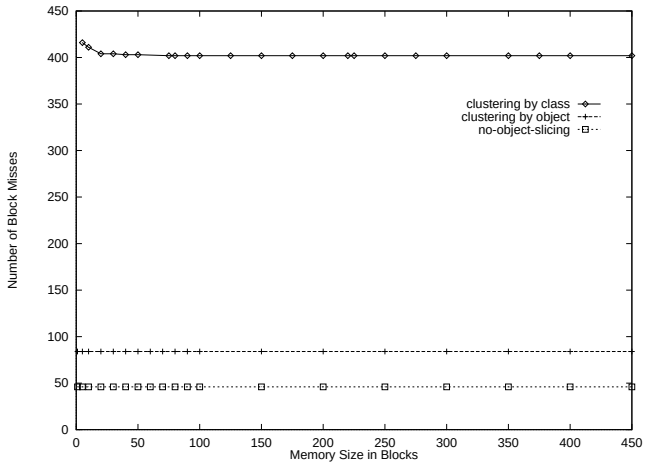

Figure 11: Query on Inherited Attribute with Low Selectivity

When a given implementation object receives a reference to a local attribute, the local attribute is stored in the receiver and thus no additional blocks must be retrieved. The retrieval of an inherited attribute in the object-slicing architecture also involves additional accesses to the receiver's conceptual object and to the implementation object that stores the inherited attribute value. If an object-clustering strategy is in effect, then both implementation objects will be stored in the same block as the conceptual object, and thus all three objects can be accessed with the retrieval of a single block. If, however, a class-clustering strategy is being used, then the reference to the implementation object will require a random access to retrieve the

receiver's conceptual object from the store of conceptual objects, as well as a random access to retrieve the appropriate implementation object from the extent of whichever of the receiver's superclasses defines the referenced property.

Thus, under object clustering the number of blocks that must be randomly accessed is not affected by the size of the extent of the *DesignObject* superclass, because the *DesignObject* implementation objects of objects belonging to both the *DesignObject* and *CompositePart* classes will be clustered together with the extent of the *CompositePart* class. However, the class clustering strategy does not differentiate between the implementation objects belonging to different classes when storing the implementation objects that make up a class's extent. Thus under class clustering, the *DesignObject* implementation objects of objects that belong to both the *DesignObject* and *CompositePart* classes will be intermingled with all other *DesignObject* implementation objects in the extent of the *DesignObject* class.

As shown in Figure 6, the number of *AutomicPart* instances, *CompositePart* instances, and *DesignObject* instances are 10,000, 1000, and 1000, respectively. Thus, under class clustering the *DesignObject* class indiscriminately stores the 12,000 implementation objects that belong to *DesignObject*, *AtomicPart*, or *CompositePart* instances. Because these implementation objects are intermixed, most of the blocks to store implementation objects of *DesignObject* could be brought into memory in order to retrieve the *buildDate* values of the *CompositePart* instances. This means that under the class clustering strategy, the execution of query 4 could require the retrieval of all blocks of the *DesignObject* implementation objects, whereas for object clustering, the query only needs to bring the portion of blocks that contain the implementation objects belonging to the *CompositePart* instances. This difference causes the performance gap between class clustering and object clustering as shown in Graph 11. This gap is quite dramatic, and indicates that if an application heavily uses this kind of queries (queries on inherited attribute with low selectively), object-clustering is superior to class-clustering for the sake of average performance.

8 Related Work

In this paper, we identified data modeling requirements that should be met by an OODB model in order to support advanced tools, such as view mechanisms, schema evolution, and role systems. Identified features include multiple classification, dynamic reclassification, and dynamic restructuring. We are not the first who have identified the utility of such flexible modeling constructs.

One example of this is the work by Scholl et. al. on object-oriented views [23]. Other examples are recently emerging *role modeling* approaches [11, 19]. In *role modeling* systems, objects dynamically gain and lose multiple interfaces (aka *roles*) throughout their lifetimes. These roles can be compared to the implementation objects of an object-slicing implementation, in that both permit objects to belong to multiple classes and change types dynamically. In some sense, accessing an object through one of its implementation objects is like accessing an object while it is playing one of its roles.

In [24], Sciore proposed an object specialization approach, in which a real world entity is modeled by multiple objects arranged in an object hierarchy. These object hierarchy objects inherit from each other, enabling each individual entity object to decide its own inheritance hierarchy. Although our implementation objects resemble object hierarchy objects in that they inherit from each other, objects in our implementation always conform to the existing global class hierarchy. The role system proposed by Gottlob et al. [11] was implemented using techniques similar to object-slicing . This system, like ours, is implemented in Smalltalk by overriding the `doesNotUnderstand:` method.

Unlike many role systems, which allow object hierarchies to exist independently from class hierarchies [24], objects in our model always conform to the existing global class hierarchy. To recap, if an object possesses an implementation object of a given class's type, it must also possess an implementation object for every class that is a superclass of that given type. This achieves an efficient and uniform inheritance scheme. Also, unlike many role systems, in our implementation conceptual objects can be associated with at most one implementation object of a given type [11]. Our proposed object-slicing approach is thus a compromise between extremely flexible role models on the one side and rigid class-based data models on the other side.

In short, variations of the *object-slicing* technique have been repeatedly recognized as a powerful and flexible method for extending an existing OODB system to support the identified required features. In spite of this work, the object representation assumptions underlying most OODB systems are of a different nature. Current OODB systems allow only one most-specific type per object and the object type is determined and fixed at object-creation time [2, 18, 12].

The Iris functional database system is the most well-known exception to this; i.e., it is a DBMS system that supports multiple classification. However, IRIS, being built on top of a relational engine, distributes its data over several relational tables [8], and hence can support multiple states per object. There are, however, several significant differences between our work and IRIS. For one, our object-slicing representation is built on top of a pure object-oriented kernel rather than a relational system. Consequently, our system supports

inheritance of both instance variable and methods. IRIS, however, does not provide for any encapsulation. Instead, foreign functions written in other programming languages can be imported. In our system, each implementation object is still a fully functioning object in the sense of the object-oriented paradigm, and thus is encapsulated and can respond to methods, such as, typeof(), etc. In short, in this paper we examine the impact of an object-oriented realization of object-slicing rather than a horizontal partitioning type of an approach.

Extending an existing DBMS with object-slicing techniques necessarily involves the overhead of additional data structures, maintenance costs, and processing time. To our knowledge no work has been done evaluating the costs of object-slicing. In this paper, we address this issue. More precisely, we present results from our evaluation of the performance costs incurred by object-slicing, and examine the potential of various clustering techniques to alleviate this cost.

Unlike previous work on the clustering and partitioning of OODBs [26, 27], our goals are not to invent new clustering strategies to dynamically adapt to various access patterns. Instead, we are interested in understanding what the innate differences are between the conventional and the object-slicing representation — and whether the flexibility gained comes with an added cost. Furthermore, previous work in the area of clustering such as [9, 17], is still applicable in our representation model. In fact, we expect that object-slicing is amendable to horizontal and vertical partitioning techniques, since conceptual objects are already naturally partitioned into smaller self-maintained chunks.

9 Conclusions

In this paper, we identify extensions required from an OODB system in order to support advanced tools such as view technology, advanced schema evolution support, and role modeling systems. These features include *multiple classification*, *dynamic reclassification*, and *dynamic restructuring*. While we are not the first to identify the need for such flexible data modeling support, we point out that the majority of currently available OODB systems do not provide such support. Instead, they support comparatively poor data models—requiring an object to belong to exactly one most specific type, and not allowing an object to dynamically migrate to new types over the lifetime of the object.

In this paper, we describe a methodology known as *object-slicing* that is capable of extending existing OODB systems to support these required features. We have successfully implemented an object-slicing representational layer on the GemStone OODBMS system, which while still providing full access to all GemStone DBMS functions, now also offers these flexible modeling features. Our experience with building this object-slicing paradigm should be valuable to other researchers or OODB system builders that are exploring the extension of OODB systems with more flexible constructions.

Extending an existing DBMS with object-slicing techniques necessarily involves the overhead of additional data structures, maintenance costs, and processing time. However, although object-slicing is a known technique that is being utilized for view systems [14], schema evolution [21], and role systems [11], to our knowledge no work has been done evaluating the costs of object-slicing. In this paper, we therefore provide an in depth evaluation of the object-slicing technique.

We describe our experimental results evaluating the relative costs and benefits of adopting the object-slicing techniques. First, we compare object-slicing with the conventional *intersection class* alternative by contrasting their capabilities. We back up our experimental results with an analytical storage cost model of our object-slicing implementation, which confirms the expected improvement of performance for access of local attributes and the additional cost associated with accessing inherited objects with low selectivity. As clustering is critical to optimizing queries on such models, we present results gathered using OO7 benchmarks queries to evaluate the object-slicing model using two of the more standard clustering strategies. In particular, we report our experimental results of both class-clustering and object-clustering strategies. Based on these results, we conclude that clustering can be utilized to effectively reduce the overhead associated with object-slicing, and that as in a conventional architecture various types of access patterns can best be optimized by providing distinct types of clustering techniques.

References

[1] S. Abiteboul and A. Bonner. Objects and views. *SIGMOD*, pages 238–247, 1991.

[2] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonic. The object-oriented database system manifesto. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: The Story of O2*, chapter 1, pages 3 – 20. Morgan Kaufmann Pub., 1992.

[3] J. Banerjee, H. Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, and H. J. Kim. Data model issues for object-oriented applications. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 197–208. Morgan Kaufmann Pub., 1990.

[4] E. Bertino. A view mechanism for object-oriented databases. In *3rd International Conference on Extending Database Technology*, pages 136–151, March 1992.

[5] P. Butterworth, A. Otis, and J. Stein. The gemstone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[6] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. *SIGMOD*, 1993.

[7] C. Souza dos Santos, S. Abiteboul, and C. Delobel. Virtual schemas and bases. International Conference on Extending Database Technology (EDBT), 1994.

[8] D.H. Fishman. Iris: An object oriented database management system. In *ACM Transactions on Office Information Systems*, volume 5, pages 48–69, January 1987.

[9] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Partition-based clustering in object bases: From theory to practice. In *Foundations of Data Organization and Algorithms, 3rd International Conference*, pages 301–316, October 1993.

[10] G. Gottlob, P. Paolini, and R. Zicari. "Properties and Update Semantics of Consistent Views". *ACM Trans. on Database Systems*, vol.13(4):486–524, December 1988.

[11] G. Gottlob, M. Schrefl, and B. Rock. Extending object-oriented systems with roles. manuscript, 1993.

[12] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the orion next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[13] H. A. Kuno and E. A. Rundensteiner. Implementation experience with building an object-oriented view management system. Technical Report CSE-TR-191-93, University of Michigan, Ann Arbor, August 1993.

[14] H. A. Kuno and E. A. Rundensteiner. Materialized object-oriented views in *MultiView*. In *ACM Research Issues in Data Engineering*, March 1995.

[15] C. Lecluse, P. Richard, and F. Velez. o_2, an object-oriented data model. *SIGMOD*, pages 424–433, 1988.

[16] J. Martin and J. Odell. *Object-Oriented Analysis and Design*. Prentice-Hall, Inc., 1992.

[17] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems*, 9(4):680–710, December 1984.

[18] O2 Technology. *O2 Views User Manual*, version 1 edition, December 1993.

[19] M. P. Papazoglou. Roles: A methodology for representing multifaceted objects. In *International Conference on Database and Expert Systems Applications*, pages 7–12. Springer-Verlag, 1991.

[20] Y. G. Ra, H. A. Kuno, and E. A. Rundensteiner. A flexible object-oriented database model and implementation for capacity-augmenting views. Technical Report CSE-TR-215-94, University of Michigan, 1994.

[21] Y. G. Ra and E. A. Rundensteiner. A transparent object-oriented schema change approach using view schema evolution. In *IEEE International Conference on Data Engineering*, March 1995.

[22] E. A. Rundensteiner. *MultiView*: A methodology for supporting multiple views in object-oriented databases. In *18th VLDB Conference*, pages 187–198, 1992.

[23] M. H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *Proceedings of the Second DOOD Conference*, December 1991.

[24] E. Sciore. Object specialization. *ACM Transactions on Information Systems*, pages 103–122, April 1989.

[25] M. Tresch and M. H. Scholl. Schema Transformation without Database Reorganization. In *SIGMOD RECORD*, pages 21–27, 1993.

[26] M. M. Tsangaris and J. F. Naughton. A stochastic approach to clustering in object bases. *SIGMOD*, pages 12–21, 1991.

[27] M. M. Tsangaris and J. F. Naughton. On the performance of object clustering techniques. *SIGMOD*, pages 144–153, 1992.