Multigranularity Locking with the Use of Semantic Knowledge in a Layered Object Server[1]

G. Amato[‡], S. Biscari[±], G. Mainetto[±], F. Rabitti[±]
[±]CNUCE-CNR
Via S. Maria, 36, Pisa, Italy
[‡]IEI-CNR
Via S. Maria, 46, Pisa, Italy

Abstract:

Object-oriented database programming languages use a data model that, by its nature, leads to a hierarchical organisation of persistent data. The *Multigranularity Locking* (*MGL*) protocol is the concurrency control framework that allows to better analyse concurrent accesses to such hierarchy of data items.

Furthermore, modern Object–Oriented Database Management Systems are organised accordingly to the client–server architecture, where the server component is often an object server. The application of software engineering criteria to the design of an object server usually leads to a system structured in interpretation layers. In a layered object server, the semantic knowledge necessary to decide which is the "best" granule to lock in the *MGL* data item hierarchy is distributed among all the system layers, therefore a suitable technique to co-ordinate such decisions is necessary.

This paper presents some guidelines on the design of the hierarchical organisation of data items that should be used from an Object–Oriented Database Management System supporting the *MGL* protocol, and an original concurrency control technique called *Expandable MGL* that provides all the system layers with the ability of locking those granules that each layer considers more appropriate on the basis of its partial knowledge of a transaction's behaviour.

Keywords:
Concurrency Control, Multigranularity Locking Protocol, Object–Server, Persistent Object Store, Object–Oriented Databases

1. Introduction

Research is currently trying to overcome the limits of the first generation *Object–Oriented Database Management Systems* (*ODMSs*). One research challenge is the exploitation of new techniques for a better engineering of *ODMS* specific mechanisms such as, for example, those used in concurrency control. For the sake of efficiency, these new concurrency control techniques can take advantage of several sources of semantic knowledge and, particularly, those coming from a richer object–oriented database programming language and from structuring an object server as a layered system.

Object-oriented database programming languages use a data model that, by its nature, leads to a hierarchical organisation of persistent data. Every object belongs

[1] This work has been partly supported by grants from the EU under ESPRIT BRA Project No 6309 FIDE2 (Fully Integrated Data Environment) and from the CNR under bilateral Italy–USA SIENOSP Project.

to a set of objects, i.e. to a class, every class can be a subset of several superclasses, and so on. Such hierarchy of data items can be exploited in the framework provided by *Multigranularity Locking* protocol (*MGL*). In *ODMSs*, ORION and O_2 have investigated the use of this protocol.

The architecture of several modern *ODMSs* is the client–server architecture, where the server component can be an object server [Joseph 91]. An object–server is a complex software that should be structured according to software engineering criteria, among which an important role is played from the information hiding principle. This usually leads to systems structured in interpretation layers. Thus, besides the hierarchical organisation of persistent data, an object server exhibits a hierarchical organisation of interpretation layers.

Since in a layered object server the semantic knowledge necessary to decide which is the “best” granule to lock in the data item hierarchy is distributed among all the system layers, the adoption of the *MGL* protocol involves the use of a suitable technique to co-ordinate such decisions. Furthermore, the layer corresponding to the database programming language should own highest level semantic knowledge, such as for example that about set–oriented accesses.

This paper presents some guidelines to better model the hierarchical organisation of data items for the *MGL* protocol and a technique, called *Expandable MGL* (*EMGL*), that allows to distribute among all object server layers the capability of requesting locks on granules belonging to different portions of a shared global data item hierarchy. In this approach, every system level can choose the granule to lock on the basis of its partial knowledge of the overall transaction’s behaviour. We show the application of these techniques to the *Physically Independent Object Server* (*PIOS*), which is a research prototype that aims to explore new development directions in *ODMSs*. However, the techniques herein described should be applicable to other *DMSs* [2] that support a rich data model and that are structured as layered systems.

The paper is organised as follows. Section 2 outlines the layered architecture of *PIOS* and the type of semantic knowledge owned from each layer that could be exploited in the *MGL* protocol. Section 3 describes in details the new techniques that we propose and their motivations. Section 4 provides the results of some simple performance tests that we have performed in *PIOS*. Section 5 briefly concludes.

2. Architectural Overview of *PIOS*, a Layered Object Server

The *Physically Independent Object Server* (*PIOS*) is a research prototype aimed to explore new development directions in *ODMSs*. The main research direction pursued in the development of *PIOS* is the verification of the effectiveness of *physical independence* in *ODMSs*. To support physical independence *PIOS* makes use of a multi–level architecture founded on three levels of abstraction that is on three different software layers [Aloia 93] [Amato 95].

[2] We use the acronym DMS instead of the more traditional DBMS.

The two upmost levels correspond to two object data models, the *logical data model* and the *physical* one. *PIOS* logical data model is a classical object–oriented data model in which a single logical object can simultaneously belong to several logical classes that are in an inheritance relationship [Atkinson 89]. Vice versa, every physical object of *PIOS* physical data model belongs to exactly one physical class because the logical inheritance relationship has been flattened out.

PIOS database designer can choose among several different physical organisations of a given logical schema and he/she can define *value and navigation indices* on physical classes [Zezula 93] for the sake of enhancing overall system performance.

The third level of the architecture is the *storage* level. It corresponds to the functionalities for storing and retrieving persistent objects supplied from a concurrent Persistent Object Store [Munro 94]. The storage level abstracts from the operating system details, it provides the upper layers with a heap of persistent objects and with primitives for accessing them expressed in a transactional context.

2.1 *PIOS* process structure

The previously summarised features of *PIOS* lead to the architecture sketched in **Figure 1**. The right part of the figure shows the static information that is the initial input to *PIOS* processes (ovals).

At run–time *PIOS* as a whole represents the server component of a client-server *ODMS*. Internally, *PIOS* is in turn structured according to a client-server architecture in which several *Logical/Physical Translators* are the clients of a single *PHysical Object Server* (*PHOS*). Every *Translator* is a process that acts on behalf of an external *Client* process. Peer *Client–Translator* processes can run on different machines connected through an interconnection structure that usually is a local area network. Every *Translator* interacts with *PHOS* by means of the interprocess communication facilities provided from the operating system.

2.2 *External Clients*

The *External Clients* of *PIOS* are processes such as the compiler of an external language and its run–time support, the front–end of *PIOS* query language processor, a browser, etc.

These clients access the persistent objects and classes stored in the *PIOS* database only by means of the logical data model operators, both those that allow navigational accesses and those that ensure set–oriented accesses. The navigational operators of *PIOS* logical data model can be integrated with the run–time support of object-oriented database programming languages, as the extension of the Common LISP used in ORION [Kim 89], the extension of Smalltalk used in GemStone [Butterwoth 91] or Galileo [Albano 85]. Logical set–oriented operators are combined to form sentences of *PIOS* query language [Rabitti 94].

An *External Client* could be viewed as an additional layer of the system that provides *PIOS* with a logical–level semantic knowledge about a transaction behaviour that can be useful for reducing locking overhead. For example, if it is necessary to perform a simple query that just projects some attributes of all the objects belonging to a logical class, then the current *PIOS* query language processor requires just one lock for the logical class instead of a set of locks, one for every logical object. Another example is the use of static semantic knowledge aimed to support a Conservative Two Phase Locking protocol for a strongly and statically typed object–oriented database programming language [Amato 93]. Such paper illustrates a technique of static analysis of a transaction that automatically infers a safe approximation of the readset and writeset of the transaction. This permits to realise the conservative policy, that is to lock in advance all data the transaction is going to access.

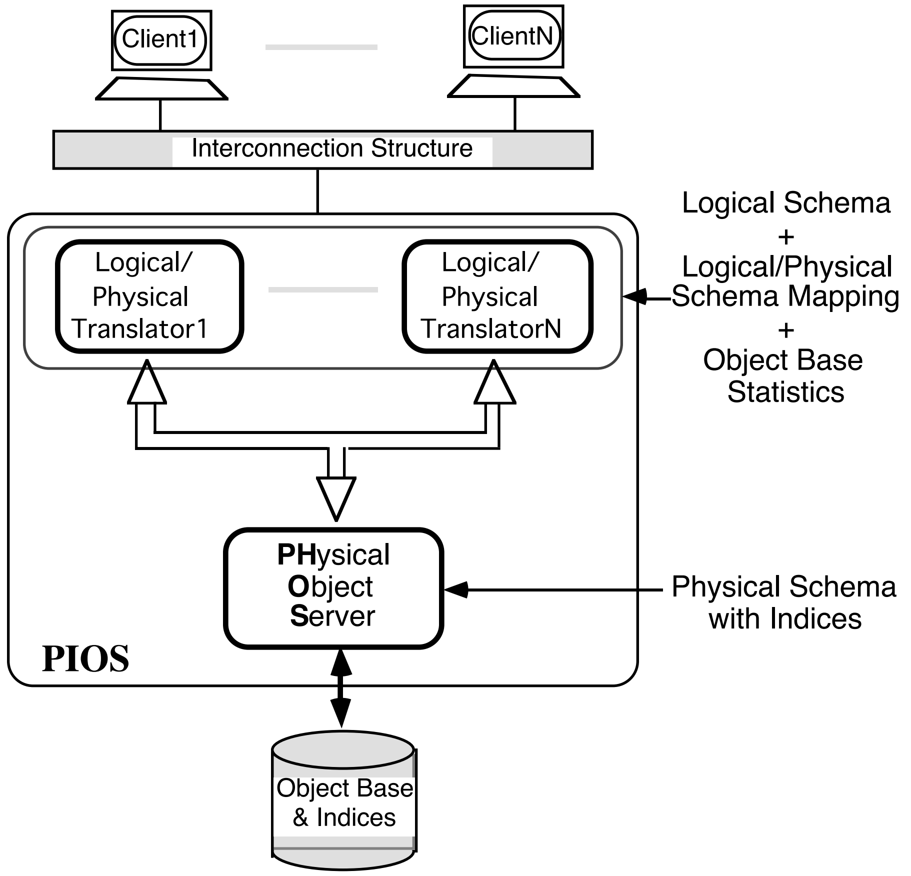
Client1
ClientN
Interconnection Structure
Logical Schema + Logical/Physical Schema Mapping + Object Base Statistics
Logical/ Physical Translator1
Logical/ Physical TranslatorN
PHysical Object Server
Physical Schema with Indices
PIOS
Object Base & Indices

Figure 1. *PIOS* process structure

2.3 *Logical/Physical Translators*

The purpose of each *Translator* is the dynamic mapping of logical access operations (to logical “virtual” objects) into physical access operations (to physical “database” objects). The translation concerns both navigational and set–oriented accesses, but the translation and optimisation of logical queries are of paramount importance for the overall *PIOS* performance.

The *Translator* is the layer of the system that owns the semantic knowledge about logical–physical object and class correspondence. Furthermore, as shown in

Figure 1, this is the layer that makes use of the statistical information on physical object states for query optimisation purposes. Statistical information is also quite important for deciding the granularity of items to lock during a query evaluation: for example, if a query performs a navigation from the set of physical objects of physical class A to the set of physical objects of physical class B, then it is possible to foresee the percentage of physical objects belonging to B extension that will be accessed during the query evaluation. The locking overheads of setting one lock for a whole physical class and that of setting locks for the physical objects of a physical class have been inserted in the cost model developed for *PIOS* query language [Rabitti 94].

The *Logical/Physical Translator* is the *PIOS* layer that has the intermediate knowledge necessary to connect the high level logical knowledge about a transaction behaviour to the lower level physical one. Thus it plays a central and important role in *PIOS*: it is the layer responsible for generating some lock primitives of the *MGL* protocol. In particular, it generates those lock primitives that concern the granules of the *LTG* in which the physical data items are organised.

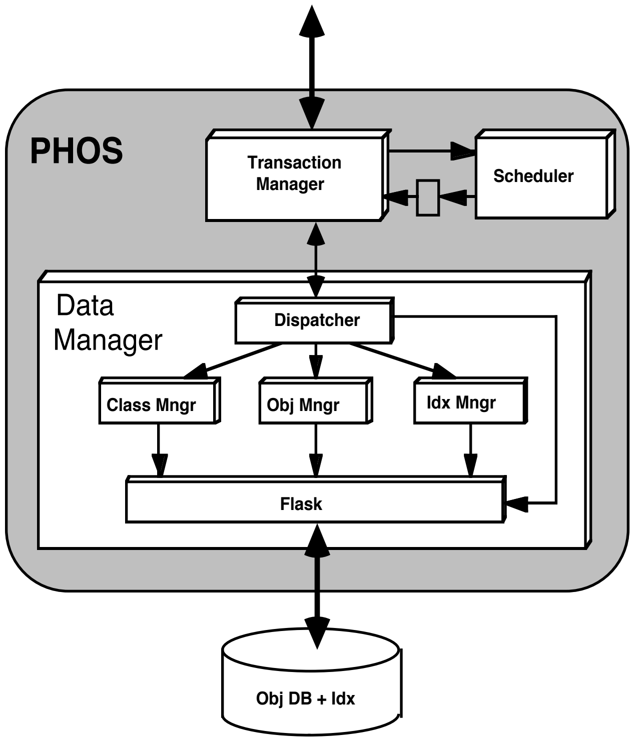

Figure 2. *PHOS* architecture

2.4 *PHysical Object Server*

PHOS provides the upper layers with both the operations of *PIOS* physical data model and the operations for interacting with navigation and value indices. *PHOS* is a process that integrates the functionalities of a Transaction Manager, a Scheduler and a Data Manager (**Figure 2**). *PHOS* is also responsible for generating statistical information about the state of the database.

PHOS is the layer of the system that owns the semantic knowledge about physical–storage object and class correspondence. Such knowledge is concentrated in the components of the *Data Manager* that directly deal with the storage level to map physical level operations (*Class*, *Object* and *Index Managers* in **Figure 2**).

The Persistent Object Store library called FLASK represents the storage level [Munro 94]. FLASK offers some basic facilities to manipulate persistent objects, it provides a great aid concerning recovery and it helps in implementing concurrency control. The key feature of FLASK is the *Concurrent Shadow Paging* mechanism that ensures the database consistency after system and transaction failures. FLASK programming interface contains operations, indexed by a transaction identifier, to create, delete, update, and access persistent objects of a generic format. The interface provides primitives for beginning, committing and aborting transactions.

Clearly, the lowest level components of the *PIOS* architecture have a specific semantic knowledge close to implementation details of the storage level. This knowledge is as important as that owned from higher layers and it should be integrated with the others in a unique framework. A classical example of *PIOS* is the index locking: a *Translator* has the knowledge to decide if it is convenient to lock an index in its entirety, but if this is not the case, then it should delegate the *Index Manager* of *PHOS* the management of the locking policy.

3. *MGL* protocol and the *Expandable MGL* technique

3.1 Multigranularity Locking in *ODMSs*

The first proposal of use of the *MGL* protocol appeared in the context of relational *DMS*s [Gray 76]. In this context, the use of *MGL* aims at minimising the number of locks that a transaction needs to acquire when it accesses the set of tuples of a relation. To this purpose, a transaction can lock a granule in five different modes (S, X, IS, IX, SIX). In this paper we assume the reader's knowledge of the details of *MGL* protocol as described in [Gray 78].

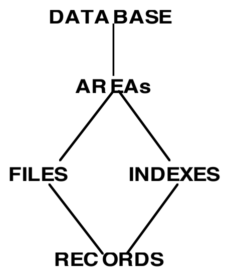

Figure 3. Relational Lock Type Graph

We recall that a *Lock Type Graph* (*LTG*) represents the hierarchy among the types of the data items. **Figure 3** shows the traditional *LTG* described in [Gray 78] and reported in [Bernstein 87]. The *Lock Instance Graph* (*LIG*) represents the same hierarchy instantiated on the actual data items stored in the database.

In the *ODMS* context, *MGL* protocol has been realised in ORION, an *ODMS* that supports multiple inheritance [Garza 88], and proposed in O_2 [Cart 90]. The motivation for using the *MGL* protocol in an *ODMS* is that the object–oriented data model provides a natural hierarchical organisation of data items in granules of different size: every object is a member of the set representing the extension of a class; a class can be a subclass of one or several superclasses, which implies that there is a subset relationship among the extension of a subclass and the extensions of its superclasses. In this context, when most of the objects belonging to the extension of a class are to be accessed, it makes sense to set one lock for the whole class extension, rather than one lock for each individual object of the extension.

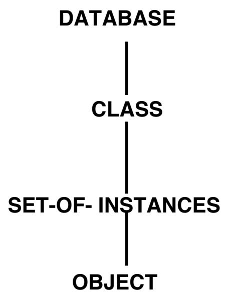

Figure 4. *ODMS* Lock Type Graph

In [Garza 88] the hierarchies representing the inheritance relationship and the *instance_of* relationship are inserted into the traditional *LTG*. The **Figure 4** (taken from [Kim 90]) shows the resulting *LTG*. Given an ORION database schema, a *LIG* will be a *rooted direct acyclic graph* (*RDAG*) that directly connects the database root node to base class nodes (roots in the inheritance hierarchy). Subclasses of base classes are represented as subnodes of base class nodes, and so on representing all the inheritance hierarchy. Class nodes are connected to nodes representing their extensions and extensions to objects thus characterising the *instance_of* relationship. The *LIG* is not a tree: it is a *RDAG* because in ORION there is multiple inheritance.

We notice that the *LTG* in **Figure 4** distinguishes common properties of classes, such as class variables and the class definitions, from class extensions (set of instances). Unfortunately, these two different aspects of the notion of class are related in the hierarchy. For example, if a class is locked in X mode because a transaction is changing the value of a class variable, then also the set of instances of the class will be implicitly locked. To overcome this difficulty, [Garza 88] proposes to add to the standard lock modes new ones, having their own new semantics, and to modify the semantics of standard lock modes when they regard classes. Vice versa, our approach will be to operate on the organisation of the *LTG*.

3.2 Designing the Lock Type Graph

A trivial choice of the *LTG* can lead to modest performance and it can render of no use the decision of adopting a complex protocol such as the *MGL*. To understand the origins of these potential shortcomings, we will firstly suppose to adapt the *LTG* proposed in [Gray 78] and used in relational *DMS*s to *PIOS* (**Figure 5**). We remind that in *PIOS* physical data model the inheritance relationship among physical classes does not exist, physical class attributes are not present, and it is impossible to modify a physical class definition while *PIOS* system is operational.

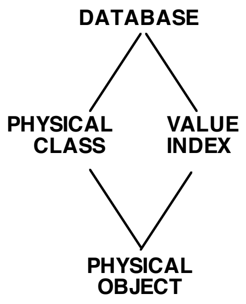

Figure 5. A trivial *PIOS* Lock Type Graph

In this hypothesis, a *PIOS* physical database will be the topmost granule of a containment relationship that has physical classes and indices in intermediate positions, and physical objects as the smallest granules. Physical objects' parents are physical classes and indices, since a physical object is contained in the extension of a physical class and it can also be accessible through an index, if a value index for that physical class has been defined[3].

This *LTG* is simple and easy to understand, but a deeper analysis reveals its inadequacy as illustrated from the following example. Let us suppose that at a certain moment in *PIOS* there are two running transactions T1 and T2. T1 counts the number of objects in a physical class. T2 updates the state of a physical object that belongs to the same physical class accessed from T1. T1 does not need to access any object state: it just has to access the data structure that represents the physical class. On the contrary, T2 does not access any physical class since it accesses directly the physical object state by means of the its identity.

These considerations lead us to conclude that T1 and T2 should be able to execute in parallel because there is no logical conflict between them. Vice versa, the trivial *LTG* generates a conflicting situation. According to the *MGL* protocol, T1 should set one single shared lock S on the physical class it wants to count, and T2 should lock the same physical class in intention-exclusive IX mode, since it has to lock in X mode a physical object that belongs to the same physical class. S and IX lock modes on the same granule are not compatible and so the two transactions cannot execute in parallel. Similar examples could be provided for indices.

The origin of this shortcoming comes from the meaning of the physical class granule. A physical class in the *LTG* has a double meaning since it is used for representing a set of individuals (the extension of the physical class intended as a set of identities of physical objects) and the set of values that the individuals take (the extension of the physical class intended as a set of *states* of physical objects). A *LTG* with two types of granules for the two concepts should enhance parallelism. If

[3] For simplicity reason we will limit our attention to value indices.

a system is able to separate the two concepts, then it could allow to execute in parallel: two operations that change the values of single different individuals without modifying the set of individuals, an operation that increments or decrements the set of individuals with another operation that just accesses the value of a different individual, an operation that queries about the cardinality of the set of individuals from an operation that modifies the value associated with an individual, and so on. In terms of *LTG* this separation means that both the data structures representing set of individuals and the data structure representing set of values of individuals should be leaf granules.

From a pragmatic point of view, we notice that a *DMS* can arrange for a greater parallelism if it organises granules according to criteria that strictly depends on the operations it provides. Here are the general criteria we believe should be followed:

- all the data structures representing persistent values that can individually be accessed from a single operation should be leaf granules;
- non-leaf granules should be introduced in the hierarchy when the system provides at least one operation that manipulate a set of leaf granules as a whole.

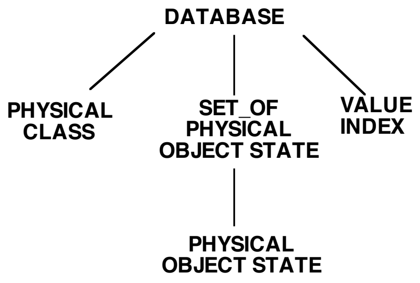

Figure 6. *PIOS* Lock Type Graph

Following these two general criteria, *PIOS LTG* becomes that shown in **Figure 6**. The previous granule has been split into two granules. The first one, still called Physical Class, represents the physical class data structure that realises the extension of a physical class and it is not a parent of the Physical Object State granule. The second granule is called Set_of Physical Object State and it is the unique parent node of all physical object states of the physical class. This last granule allows to reduce locking overhead since *PHOS* provides a simple query primitive based on the values stored in the states of physical objects (belonging to one physical class). There are also more complex situations in which the *Translator* of *PIOS* can decide if a great percentage of the physical objects of a physical class will be accessed during the execution of a query, and thus it can order a single shared lock on the physical class.

The index granule has been positioned in a different place with respect to the trivial *LTG*. A *PIOS* value index plays the same role of a physical class in the *LTG*. The reason is that a value index is defined on some descriptive attributes of a physical class and thus it can be considered as an "associative" physical class that maps a value to a set of identities. Similarly to physical classes, a value index will contain all the identities of the physical objects of the physical class on which it has been defined.

With this solution, the transactions T1 and T2 can run in parallel without any interference. T1, the transaction that counts, only asks for a shared lock on a Physical Class granule and an intention–share lock on the Database. T2, the transaction that updates, asks for an exclusive lock on a Physical Object State granule, an intention–exclusive lock on Set_of Physical Object State and Database granules. In this case, there is only one shared entity, the Database, between the two transactions, and IS and IX lock modes are not conflicting.

3.3 The *Expandable MGL* technique

A well-designed *LTG* is the basis on which all the semantic knowledge on a transaction can be fully exploited. Two kinds of semantic knowledge are useful to decide how to strike a balance between parallelism and locking overhead.

The first one is related to a semantic analysis of a transaction in its entirety. It would be useful if *PIOS* could infer properties of the submitted transactions that allow to choose the most appropriate granule to lock. An example of this is the previously mentioned analysis performed from *PIOS* query processor when it foresees, using statistical information, the percentage of the physical objects accessed during the navigation of physical classes. For obtaining this knowledge is usually necessary a complex analysis performed from components of the system that are close to the end user. These components of a layered system are placed in the highest layers of abstraction and they can infer information for reducing locking overhead in *MGL* protocol, i.e. they can reason about properties that regard coarse granules of *LTG*.

The second kind of semantic knowledge concerns the organisation of the data structures used in the implementation of a complex layered system. In a complex system organised in layers of different abstractions, only the system component that realises an abstract data type knows the details about the data structures used in the implementation. For example, in *PIOS* only the index manager holds the semantic information to decide the strategy that guarantees the greater parallelism when a transaction uses an index because it knows the organisation of the data structures used in the implementation of the index. Similar examples in *PIOS* could be provided considering the implementations of physical classes or physical objects. The components of a layered system that hold this knowledge are those placed in the lowest layers of abstraction. They can provide information for increasing parallelism of *MGL* protocol, i.e. they can use properties that regard fine granules of *LTG*.

The previous considerations suggest that the following properties should characterise the layers of a complex system (see **Figure 7**):

- The higher layers of abstraction know how to choose coarse granules to reduce overhead
- The lower layers of abstraction know how to choose fine granules to increase parallelism

It is worth noting that if a system only shows the first property, then there can be a low overhead in the management of concurrency control and a small parallelism among transactions since coarse granules can potentially raise several conflicts. Vice versa, if a system only presents the second property, then it could achieve a greater parallelism at the price of a greater locking overhead.

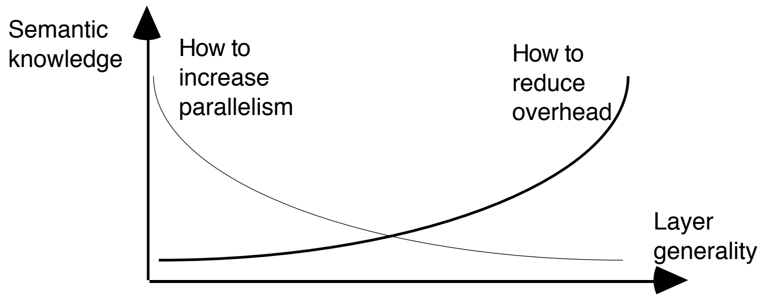

Figure 7. Different kinds of semantic knowledge in a layered system

PIOS approach aims at providing a unifying framework that takes advantage of both types of semantic knowledge. The idea is to allow each *PIOS* layer to require locks on granules corresponding to its level of semantic knowledge and to use the *LTG* as a mean for exchanging lock information among layers. If a higher layer infers from its own semantic knowledge that it cannot take a final decision on a coarse granule, then it can delegate lower layers of the system the task of exploiting the semantic knowledge on finer granules. Vice versa, if a higher layer holds all the knowledge necessary to take a definitive decision on a coarse granule, then it can inhibit lower layer requests on some finer granules.

PIOS obtains this by modifying the standard way in which a system layer manages the *LTG*. As far as *PIOS LTG* is concerned, we introduce the idea of *leaves' expandability*: a granule, that is viewed as a leaf from a higher layer, can be expanded from a lower layer with a subtree of finer granules (**Figure 8**). The *MGL* protocol allows an intermediate layer to lock what it sees as a leaf granule also in IS or IX mode. *PIOS* uses these apparently unnatural lock modes to communicate among layers. When a higher layer wants to permit a lower layer to expand a granule, it will set an intention lock on the interested granule. We call this technique *Expandable MGL.*

Let us consider a *PIOS* example that illustrates this technique. There are two concurrent transactions T3 and T4. T3 is going to insert a new physical object in an indexed physical class. T4 is willing to update the state of a physical object belonging to the same indexed physical class.

Let us firstly analyse the case in which *EMGL* technique was not used. T3 during its execution has to lock in X mode a Physical Class granule representing the extension of the physical class and a Value Index granule representing the index defined on the physical class. T4 must lock in X mode a Physical Object State granule and the Value Index granule. T3 and T4 cannot run in parallel since they ask for an incompatible lock on the same Value Index granule. This happens because some higher layer of *PIOS* had not a detailed semantic knowledge of the index data structure and it decided to lock the whole index.

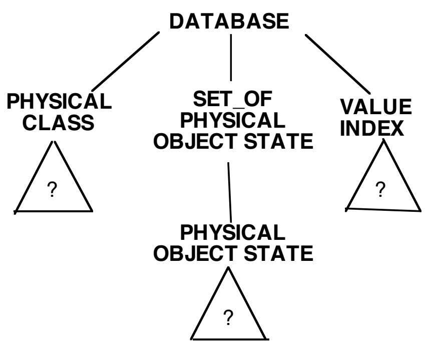

Figure 8. Leaves' expandability

The use of *EMGL* technique and leaves' expandability can ensure parallel executions. Both T3 and T4 ask for a IX lock on the same Value Index granule, and the index manager adopts its own ad-hoc policy for expanding this granule. If for instance T3 and T4 work on different pages of the B-Tree data structure, then they can run in parallel without interference.

4. Performance Samples

This section presents some performance figures regarding a *PIOS* version that fully exploits the *EMGL* technique. To appreciate the performance improvements achieved, there is a comparison between results obtained running the same tests on two versions of *PIOS*. The former version, called *Old PIOS*, is a version in which the lock requests are generated from a unique centralised component, i.e. the *Logical/Physical Translator*, which is just aware of the indices' and physical classes' existence and consequently it orders the lock of the entire data structures when they have to be accessed. The latter version, called *New PIOS*, is the version in which the lock management policy is distributed among the various layers of the architecture. In particular, the *Index* and *Physical Class Managers* have the tasks of deciding the lock request generation for the data structures that they manage.

The tests always run on the same fixed database populated with 2M objects. The logical object–oriented database schema is depicted in **Figure 9**. The two inheritance relationships in the figure have been resolved with the fragmentation of logical objects belonging to Library–UnivLibrary and Book–TechBook pairs of classes into two physical objects, one for every logical class of each pair. In the physical database schema, there are four value indices defined on the physical classes corresponding to Topic, Book, Library and TechBook logical classes.

Indices are loaded with 64K keys. Each node of the index contains at most 200 keys and the B-Trees have three levels. In the B-Trees there are around 600 leaf nodes and 7 non leaf nodes.

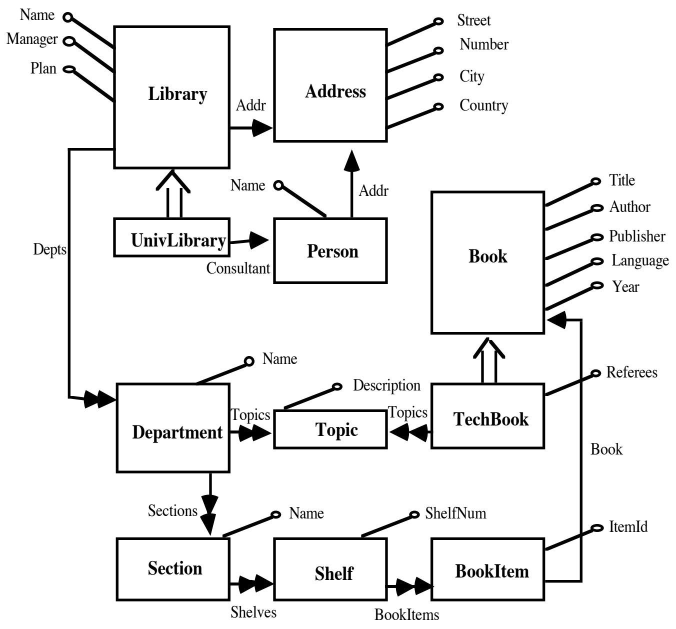

Figure 9. The sample database schema

The tests run on a Sun Spark 4[4]. It is a single processor machine with 24 MByte of RAM and 1 GByte of disk. During the experiments, the machine was completely dedicated to the tests.

The test transactions are generated in a way that allows to have both navigational and set–oriented accesses to the database. Every *PIOS* transaction in the tests executes just one logical *PIOS* operation. The operations are randomly chosen among the insertion of a new logical object in a randomly chosen logical class, the update of an old logical object of a randomly chosen logical class and the query on a randomly chosen logical class. We notice that the choice of just one *PIOS* operation for every *PIOS* transaction is not a heavy restriction: each logical *PIOS* operation can be even translated into hundred of physical operations and several transaction conflicts are possible.

The tests use the multi-programming level (number of active transactions) as input parameter to control data and resource contention. The experiments range on the multi-programming level. The algorithm that drives tests keeps constant the multi–programming level, i.e. it starts a new transaction as soon as a previously executing transaction reaches its end.

Two kinds of tests are executed. The first test provides as a result the average of the quotient between the number of running transactions and the number of active transactions (see **Figure 10**). The running transactions are those active transactions that are not waiting for some lock. The *PIOS* scheduler obtains the result computing the quotient at short fixed time intervals. Then it computes the average of all the quotients.

[4] The improvements in software components are put in evidence from ancient hardware!

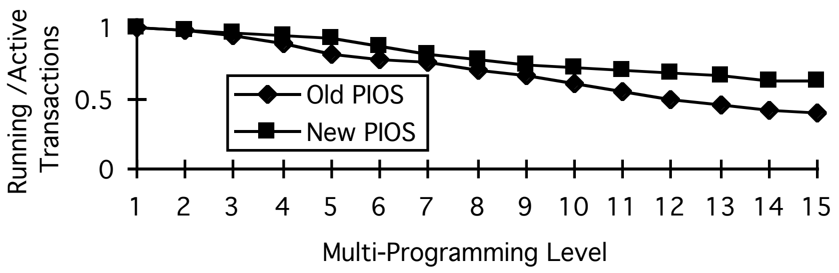

Figure 10. Running transaction ratio

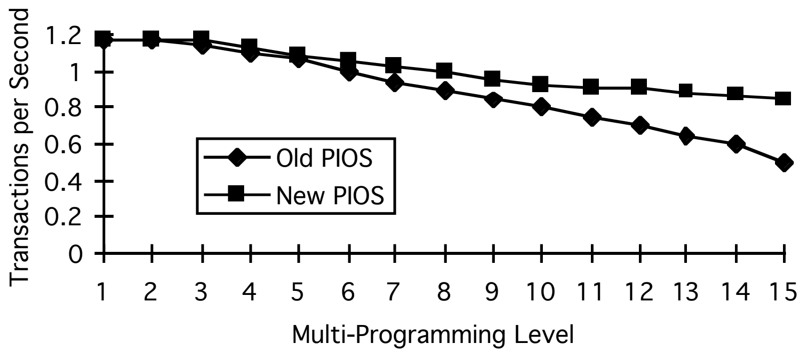

Figure 11. Throughput

The second test gives the throughput in transactions per second. This result is computed by measuring the time needed to execute a fixed number of transaction ranging on different multi-programming levels. **Figure 11** shows the outcome of this test.

It is clear from the previous figures that *EMGL* performs better than standard *MGL* under the same circumstances. The reason is that highly concurrent algorithms are absolutely necessary to deal with contention in heavily used data structures. In the tests, the data structures adopted to implement the indices and the physical classes represent bottleneck for concurrent transactions. *Old PIOS* uses the trivial policy of locking the entire index when a physical object belonging to an indexed physical class is updated and the entire physical class when a new physical object is inserted into a physical class. The risk of reducing parallelism is clear because the transaction locks all indexed physical objects and all the objects of a physical class. *EMGL* allows to adopt ad-hoc policies in each structure representing a leaf in the lock type graph, as physical classes or indices, that result in finer granularity locking and hence in better performance.

5. Conclusions

In this paper we have presented some guidelines on the design of the hierarchical organisation of data items that should be used from an *ODMS* supporting the *MGL* protocol, and a technique of concurrency control that provides all the system layers with the ability of locking those granules that each layer considers more appropriate on the basis of its partial knowledge of a transaction's behaviour. We have shown the application of these techniques to *PIOS*, an object server that use the *Concurrent Shadow Paging* mechanism provided from FLASK persistent object store, and some simple experimental results.

The techniques we propose to adopt seem to be well–suited for the high levels of a complex system, and they appear to be less adequate for the low levels. We have the impression that the complementary situation holds for multi–level transaction [Weikum 91], and in the next future we will investigate the possibility of combining in a single framework the two approaches. Another future work will be about a closer examination of the measurements provided from the proposed techniques.

6. References

[Albano 85] Albano A., L. Cardelli and R. Orsini, "Galileo: A strongly typed interactive conceptual language", *ACM Transactions on Database Systems, Vol. 10, N. 2,* pp. 230-260.

[Aloia 93] Aloia N., S. Barneva and F. Rabitti, "Supporting physical independence in object databases", *Database Technology*, Vol. 4, No. 4, pp. 265–286.

[Amato 93] Amato G., Giannotti F., Mainetto G.,"Data sharing analysis for a database programming language via Abstract Interpretation", *19th International Conference on Very Large Data Base,* Dublin, Ireland, 24-27 August 1993, pp. 405-415.

[Amato 95] Amato G., Biscari S., Mainetto G. and Rabitti F.,"Overview of *PIOS*: a Physically Independent Object Server", in *Fully Integrated Data Environments,* M.P. Atkinson (ed.), 1995. to be published by Springer Verlag.

[Atkinson 89] Atkinson M.P., Bancilhon F., DeWitt D., Dittrich K., Maier D. and Zdonik S., "The Object–Oriented Database System Manifesto", *Proc. Int. Conf. DOOD* , Kyoto, Japan, pp. 40–57, 1989.

[Bernstein 87] Bernstein P., V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database System*, Addison–Wesley, Reading, MA, 1987.

[Butterwoth 91] Butterworth P., A. Otis, J. Stein, "The Gemstone Object Database Management System", *Communication of ACM, Vol. 34, N. 10*, pp. 64–77.

[Cart 90] Cart M. and Ferriè J., "Integrating Concurrency Control into an Object–Oriented Database System", *Proc. Int. Conf. EDBT* , Venice, Italy, pp. 363–377, 1990.

[Garza 88] Garza J. F. and W. Kim, "Transaction Management in an Object–Oriented Database System", *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Chicago, Illinois, pp. 37–45, 1988.

[Gray 76] Gray J., R. Lorie and G. Putzolu, "Granularity of locks and degrees of consistency in a shared database", *IBM Res. Rep. RJ1654,* IBM Research Laboratory, San Jose, CA, also in *Modeling in Database Management Systems*, Nijssen (ed.), North Holland, 1976.

[Gray 78] Gray J., "Notes on Database Operating Systems", *IBM Res. Rep. RJ2188,* IBM Research Laboratory, San Jose, CA also in *Operating Systems – An Advanced Course*, R. Boyer, R. M. Graham and G. Siegmüller (eds.), Springer Verlag, LNCS 60, 1978.

[Joseph 91] Joseph J. V., Thatte S.M, Thompson C.W. and Wells D.L., "Object-Oriented Databases: Design and Implementation", *Proc. of IEEE*, Vol. 79, No.1, pp. 42-63,1991.

[Kim 89] Kim W., Ballou N., Chou H. T., Garza J. F., Woelk and D. "Features of the ORION Object-Oriented Database." In *Object-Oriented Concepts, Databases, and Applications*, edited by Won Kim and Frederick H. Lochovsky, ACM Press Frontier Series, 1989, pp. 251-282.

[Kim 90] Kim W., *Introduction to Object-Oriented Databases*, The MIT Press, Cambridge, Mass., 1990.

[Munro 94] Munro D.S., Connor R. C. H., Morrison R., Scheuerl S. and Stemple D.W. "Concurrent Shadow Paging in the Flask Archiecture", *Proc. of Sixth Int. Workshop on Persistent Object Systems*, Tarascon, France, September 6-9, 1994, Workshops in Computing, Springer, pp. 16-42.

[Rabitti 94] Rabitti F., Benedetti L. and Demi F., "Query Processing in *PIOS*", *Proc. of Sixth International Workshop on Persistent Object Systems*,Tarascon, France, September 6-9, 1994, Workshops in Computing, Springer, pp. 415-440.

[Weikum 91] Weikum G., “Principles and Realization Strategies of Multilevel Transaction Management”, *ACM Transactions on Database Systems, Vol. 16, No. 1,* pp. 132–180.

[Zezula 93] Zezula P. and Rabitti F., “Object Store with Navigation Accelerator”, *Information Systems, 18(7):*429-460, 1993.