PRACTIC: A Concurrent Object Data Model for a Parallel Object-Oriented Database System

N. Bassiliades and I. Vlahavas

Dept. of Informatics,

Aristotle University of Thessaloniki,

54006 Thessaloniki,

Greece.

Tel: +3031-998145

Fax: +3031-998419

E-mail: nbassili@athena.auth.gr

vlahavas@olymp.ccf.auth.gr

published at *Information Sciences* **86**, 149-178 (1995)

Abstract

In this paper, a concurrent object data model for a parallel object-oriented database system, named PRACTIC, and its abstract machine are presented. PRACTIC means PaRallel ACTIve Classes and is based on the vertical partitioning and concurrent management of the database schema classes and meta-classes, which are collectively called active objects. Active objects are permanent processes in memory that encapsulate their definitions, methods and management procedures. Semi-active and passive objects exist to realise abstract classes and instances (the actual data), respectively. The object model gives rise to a query/method execution model that provides parallelism on all levels of the instantiation hierarchy. The abstract PRACTIC machine directly maps the model to a MIMD machine. The performance of one of the proposed parallel query/method execution schemes is measured by simulation on the abstract machine.

Keywords: object-oriented databases, active objects, instance hierarchy, inheritance, parallel method execution, vertically partitioned systems.

1. Introduction

Object-oriented databases (OODBs) are an intersection of object-oriented (OO) ideas and conventional databases. They gained attention [4, 16, 27] mainly because they reduce the “semantic gap” between real world concepts and data representation models. This one-to-one mapping helps the development of complex applications, like CAD, simulation, knowledge-based systems, etc. A major drawback of OODBs is the low speed of execution of application programs, due to the dynamic binding of messages to methods at run-time. The advantages gained by dynamic binding [4] are such that one cannot sacrifice them for performance's shake, so performance must be improved by other means.

In this paper an object model for a parallel OODB system and its abstract machine are presented. The object model is named PRACTIC, after PaRallel ACTIve Classes. With the term “active” we mean the representation of classes by permanent concurrent processes that exist in memory and always run a background procedure in addition to servicing messages. Active classes a) describe and aggregate their instances and b) are responsible for the management of their instances. The resulting model is a vertically partitioned system [18] since both abstraction and functionality are encapsulated inside the classes.

A parallel query/method execution model that exhibits parallelism on all levels of the instantiation hierarchy is also described. The latter is executed on the abstract PRACTIC machine, which directly maps active classes onto processing elements, while it provides classes with additional computational resources to manage their instances efficiently.

In the rest of the paper, the various aspects of PRACTIC are described. First, in section 2, the framework of our research is set, related to previous work. In sections 3 and 4, the model is presented, with respect to objects and their response to messages, correspondingly. In section 5 an abstract machine is presented, on which the PRACTIC model is mapped. In section 6, the resulting query/method execution model is described. Some simulation results for the justification of one aspect of the query/method execution model are presented in section 7. Finally, section 8 concludes the paper with a summary of the main points and a discussion of future work.

2. Related Work

The parallelisation of OODBs did not receive much attention, mainly because the main research effort in the early years focused on the improvement of OODB structural capabilities and performance through conventional optimisations [20]. There are very few papers in the literature

[19, 26] and even fewer systems [10] that consider the parallelisation of OODB management in a multi-processor environment. Our main research effort focuses on the development of a parallel OODB system by integrating concepts of concurrent OO programming languages with database management, using a vertically partitioned OO system.

Unlike previous work on parallel OO query processing techniques [19, 26], we discuss a complete parallel OODB system (PRACTIC) that provides a parallel query execution model based on class-hierarchy parallelism [19], while in addition it offers:

A full (concurrent) OO data model,

Intra-class parallelism for query execution,

Stream-parallelism of method execution, and

Inheritance of methods.

Compared to [26] we propose a hierarchical abstract machine architecture for the multi-processor system that will support the parallel OO data model, which significantly increases the system's scalability and performance [8]. Finally, in contrast to AGM [10] we propose a coarser-grained parallel data model which best suits the volumes of data involved in a database system.

Concurrency and parallelism of OO programming languages (OOPLs), on the other hand, has received much attention since the beginning of the development of the field, providing many systems, such as ACTORS [1], POOL [2], Mentat [17], etc. In this work, some of the ideas of concurrent OOPLs are delivered to OODBs. However, PRACTIC differs from concurrent OOPLs in the following key points:

An object is not necessarily active, as in POOL [2],

Class (instead of instance) is the basic concurrent unit, contrasted to Mentat [17].

The above differences spring from the following assumptions for databases:

Databases have many more objects (data) than general-type applications,

There is a greater interest in efficient query processing than efficient execution of general purpose computations.

The advantages of PRACTIC compared to previous OOPL models are the following:

efficient management of large volumes of distributed objects,

clear and efficient, non-centralised mailing system, and

easily exploitable query speed-up, through encapsulated query parallelism.

For the implementation of efficient OO systems, the vertically partitioned approach [18] has been proposed, where data, behaviour and functionality of the system is partitioned according to the type of the objects. PRACTIC adopts the vertically partitioned approach and extends it with a) Inheritance, and b) Schema evolution.

3. The PRACTIC model

In this section we briefly introduce OODB terminology and concurrency in OO systems, before presenting PRACTIC data model. Section ends with the method inheritance and schema evolution strategies of PRACTIC.

3.1. Background to Object-Oriented Databases

Objects are independent entities that encapsulate both their internal state (data) and their behaviour (methods that act upon data). They have a well-defined interface through which all interactions with their environment take place. Object independence leads to message passing as the major computation model.

There are three types of objects, in OODBs: a) instances, which are the actual data, b) classes, which are abstract descriptions of the structure and behaviour of instances, and c) meta-classes, which are abstract descriptions of classes.

```
<object> ::= <instance> | <class> | <meta-class>
```

Except of the “is-instance-of” relationship between classes (or meta-classes) and their instances, there exists the “is-a” relationship, which specialises the structure and behaviour of a class (super-class) in another class (sub-class). A sub-class inherits all attribute and method definitions of its super-class, unless otherwise specified. The “is-a” relationship defines a class inheritance hierarchy:

```
X inherits-from Y ⇐ X is-a Y
X inherits-from Y ⇐ ∃Z, X is-a Z & Z inherits-from Y
```

The set of all the instances of a class is called its “extension”.The extension of a class is a sub-set of all its super-classes. Therefore, we can consider an extended “is-instance-of*” relationship:

```
extension(Y) = { X | X is-instance-of* Y }
X is-instance-of* Y ⇐ X is-instance-of Y
X is-instance-of* Y ⇐ ∃Z, X is-instance-of Z &
                              Z inherits-from Y
```

Abstract [15] or mixin [25] classes are called the ones which cannot have direct instances. However, their extension is not null because of their sub-classes. Abstract classes act as behavioural abstractions for their sub-classes.

`X is-instance-of **abs_class** ⇔ ¬∃Y, Y is-instance-of X`

3.2. Introduction to Concurrency in Object-Oriented Systems

The independent nature of objects implies a simple parallel execution model for OO systems. Each object occupies a concurrent process that encapsulates its computation state, in a multi-processor environment. Concurrent processes cooperate via message communication [1, 2, 17], on a message-passing machine.

In a realistic application, the number of concurrent objects are more than the available processors, therefore some objects may have to share a processor. Since no memory is shared in a message-passing environment, the above approach is efficient only when the number of objects is small, otherwise the system will clutter from two factors:

time overhead caused by the switching of context when an object receives a message,

memory space overflow caused by the copy of methods [11] from a class to its instances.

The AGM database system [10], does not consider the above drawbacks and is based on an instance-object level of concurrency. However, in large OODBs certain facts should be taken into account:

the number of instance objects is large, because they represent data,

the number of classes may vary, but is considerably smaller than the number of instances,

there are only few meta-classes in a typical OODB system.

The above remarks imply certain assumptions about a possible mapping of an OODB system onto a multi-processor architecture. Furthermore, objects must also be classified according to their functionality in the OODB; only "useful" objects should be given considerable attention.

3.3. Concurrency in PRACTIC

In PRACTIC model, objects are classified according to their ability to have instances (directly or indirectly) and/or be able to generate new ones. Therefore, there are three types of objects in PRACTIC (fig. 1):

`<object> ::= <active> | <semi-active> | <passive>`

Active objects, are defined as the objects that: a) have direct instances, and b) can create new instances.

`Y ∈ <active> ⇔ ∃X∈<object>, X is-instance-of Y`

They are called active because they are permanent processes in memory. This is vital for being able to respond at any time to messages that request the creation of new instances, the retrieval of old instances, etc. Moreover, active objects do not remain idle when they are not servicing a message, but they execute a background procedure to monitor the efficient management of their extensions.

Semi-active objects, are the objects that: a) have indirect instances, and b) cannot create direct instances.

$$\begin{aligned} \texttt{Y} \in \langle\texttt{semi-active}\rangle \Leftrightarrow {} & (\exists \texttt{X} \in \langle\texttt{object}\rangle,\ \texttt{X is-instance-of}^{*}\ \texttt{Y})\ \& \\ & (\neg\exists \texttt{Z} \in \langle\texttt{object}\rangle,\ \texttt{Z is-instance-of Y}) \end{aligned}$$

Semi-active objects are not permanent processes, but they are processes evoked when a message is sent to them. This is justified by the fact that semi-active objects do not have direct instances to manage. Semi-active object processes are "killed" after they service a message. It is the responsibility of their meta-classes, which are active objects, to monitor messages sent to them and to fork their processes.

Active and semi-active objects are collectively called *non-passive* objects.

$$\langle\texttt{non-passive}\rangle ::= \langle\texttt{active}\rangle \mid \langle\texttt{semi-active}\rangle$$

$$\texttt{Y} \in \langle\texttt{non-passive}\rangle \Leftrightarrow \exists \texttt{X} \in \langle\texttt{object}\rangle,\ \texttt{X is-instance-of}^{*}\ \texttt{Y}$$

Only non-passive objects are connectable through the class hierarchy.

$$\texttt{X inherits-from Y} \Rightarrow \texttt{X} \in \langle\texttt{non-passive}\rangle\ \&\ \texttt{Y} \in \langle\texttt{non-passive}\rangle$$

Passive objects, are the objects that: a) do not have either direct or indirect instances, and b) cannot create instances.

$$\texttt{X} \in \langle\texttt{passive}\rangle \Rightarrow \neg\exists \texttt{Y} \in \langle\texttt{object}\rangle,\ \texttt{X is-instance-of}^{*}\ \texttt{Y}$$

Passive objects are not permanent processes but mere data structures, because they are too many, and they do not have instances to manage. It is the responsibility of their classes, which are active objects, to monitor their state and trigger them to react to messages.

The following sub-sections describe which objects belong to which of the above categories.

3.3.1. Active Objects

Active objects are: a) all the user-defined classes of an OODB schema, except of the abstract classes, and b) some of the meta-classes, described in the following.

$$\langle\texttt{active}\rangle ::= \langle\texttt{class}_1\rangle \mid \langle\texttt{meta-class}_1\rangle$$

$$\texttt{X} \in \langle\texttt{class}_1\rangle \Leftrightarrow \texttt{X is-instance-of}\ \mathbf{class}$$

PRACTIC supports both pre-defined meta-classes and meta-classes supplied by a model designer to tailor the data model ([13, 21, 23]). We have adopted the terminology and instance hierarchy of ADAM [16, 22], which is essentially the same for any type-less, class-based OO system, like SmallTalk [15] or Loops [25]. ADAM is based on Prolog and our choice for it is justified by the intention to build a knowledge base tool on PRACTIC. However, the meta-class hierarchy of PRACTIC is slightly different from ADAM, since the functionality of the three types of objects, according to their “activity”, must be incorporated into the meta-class hierarchy.

According to the PRACTIC meta-class hierarchy (fig. 1) the active meta-classes are all the instances of the special pre-defined meta-class `meta_class`, which is the top of the instance hierarchy, i.e. every other meta-class is created by sending it a `new` message:

$X \in \langle\texttt{meta-class}_1\rangle \Leftrightarrow X$ `is-instance-of` **`meta_class`**

Note that the above definition includes `meta_class`, since it is an instance of itself. The meta-class `meta_class` is essential for the model designer to create new meta-classes that extend the functionality of the OODB [13, 21, 23].

Apart from `meta_class` other active meta-classes include:

Meta-classes which are responsible for defining user-classes. There can be several of these, as classes may have different properties, e.g. support for keys, relationships, versioning, constraints, etc. [5, 23]. Here two generic meta-classes are discussed (fig. 1), `class` and `abs_class` [13, 25], responsible for creating normal and abstract classes, respectively.

The meta-class `mixin`, which is responsible for creating mixin meta-classes (or mixins for short). Mixins hold behaviour shared by several pre-defined or designer-defined meta-classes, through inheritance [13, 23].

3.3.2. Semi-active objects

Semi-active objects are: a) the mixins [13, 23], and b) the abstract classes [25]. The meta-class `object` is also included as a mixin class. We notice here the difference of our approach to ADAM, since we consider `object` as an instance of `mixin` and not of `meta_class` [5, 22]. The `object` mixin is the top of the inheritance hierarchy (fig. 1) for both meta-classes and classes [13, 25]. It defines the general properties of all objects, like deletion and display. Its indirect instances are both instance objects and classes.

$\langle\texttt{semi-active}\rangle ::= \langle\texttt{meta-class}_2\rangle \mid \langle\texttt{class}_2\rangle$

$X \in \langle\texttt{meta-class}_2\rangle \Leftrightarrow X$ `is-instance-of` **`mixin`**

$X \in \langle\texttt{class}_2\rangle \Leftrightarrow X$ `is-instance-of` **`abs_class`**

3.3.3. Passive objects

Passive objects are only the instances of user-defined active classes, and they are the actual data in OODBs. In a large database, passive objects are usually many, so there is a need for large storage devices, which can be either disks, for disk-based database systems, or main memory chips, for modern main memory database systems [3].

`<passive> ::= <instance>`

3.4. Method Inheritance and Schema Evolution

Method inheritance in multi-processor OO systems has been avoided, because it is believed that it compromises encapsulation [2, 18]. The latter is considered as the main OO advantage. Some researchers simply ignore inheritance [19]. We believe that inheritance cannot be neglected, since it is one of the best OO features for structured knowledge representation and PRACTIC is targeted as a core for a knowledge-base management system.

There are two main approaches to realise inheritance, through static or dynamic binding. The static or compiled approach copies all the appropriate methods at compile-time. Binding of method calls to programs is done entirely at compile-time, so run-time overheads are avoided. On the other hand, the dynamic or interpreted approach binds messages to methods at run-time, which delays program execution. Despite that, the interpreted approach is more flexible, since changes to methods and/or database schema (schema evolution) can be easily introduced later, while static inheritance requires re-compilation of the whole system when a change occurs.

In PRACTIC, methods are copied to the appropriate active and semi-active objects at the time the system is initialised, or when an object is created. The system keeps track of which method part origins from which class, in order to be able to replace it upon changes at run-time. When a change of a method occurs at a class, its sub-classes are notified for the change and the corresponding piece of method is replaced by the new one, at run-time. This approach offers the following advantages:

- No inheritance overhead at query-time,
- Interactive, run-time and flexible schema evolution, and
- Fragmented method definition which leads to parallel method execution.

Changes to methods can occur at run-time, without the need to re-compile or re-boot the whole system. Only re-defined methods are re-compiled and broadcasted to all the classes affected by the change. This is viable thanks to the meta-programming facilities and dynamic nature of Prolog.

Broadcasting of changes is done transitively. When a class receives a changed method part, it checks if this change really affects it (the method may be completely cancelled). If it does, then it changes the local method definition and it re-distributes the change to its own sub-classes, and so on, so forth, until classes with no sub-classes receive the change. When a class has its local method definition changed and all its sub-classes have done the same, then it sends an acknowledgment message to its super-class. Meanwhile, the method is locked, therefore messages requesting service from this method are suspended until changes are propagated everywhere and acknowledgment messages have been received.

4. Operational Semantics of Messages

PRACTIC objects can respond to various kinds of messages. The operational semantics of each message varies according to the category of the message and the object. In the following, the operational semantics of generic and general purpose messages are described.

4.1. Messages for Creating New Instances

Only active objects can create new instances, and thus respond to a `new` message. In the following, the re-action of an active object to a `new` message is described, according to the category of its instances.

4.1.1. Creation of Active Instances

To create an active object, a `new` message must be sent to its active meta-object. As a consequence, a new memory-resident process is created in the system. This new process is independent from its parent, therefore it responds to run-time messages without seeking code from the creator process. This is achieved by copying the appropriate method definitions from its meta-object to its private memory at creation-time.

The meta-classes `meta_class` and `class` are typical examples of active objects, that their instances are active as well. A model designer needs to define active instances of `meta_class`, therefore a message `new` sent to `meta_class` is tailored to create a new active meta-class, instance of `meta_class`. On the other hand, a database schema designer needs to define new active classes, instances of `class`, and a message `new` sent to `class` creates active database classes.

4.1.2. Creation of Semi-active Instances

The creation of a semi-active object does not require the creation of a new process, because semi-active objects are only activated when a message is sent to them. Therefore, the creation of a semi-active object causes a creation of a new data structure bound to its parent process, which is an active object.

Examples of active objects with semi-active instances are meta-classes: a) `mixin` whose instances are mixins, and b) `abs_class` whose instances are user-defined abstract classes.

Similarly to the previous section, a model designer may need to define new semi-active instances of `mixin`, while a database schema designer may need to define new semi-active classes, instances of `abs_class`, to abstract behaviour for its (possibly active) sub-classes.

4.1.3. Creation of Passive Instances

Instances of user-defined active classes are passive, therefore their creation does not require a new process. Instead, instance objects are stored in either memory or disk as data structures.

In response to a `new` message, the recipient class checks for applicable constraints about the creation of the new instance and decides if it can be created. Then, an instance with the required characteristics is created and a unique object identifier (OID) for the object is generated. The object is allocated to an appropriate secondary storage or memory device. The `new` message resembles tuple insertion in relational databases.

The OID serves as a unique name of an object and its purpose is to uniquely identify it. Furthermore, the OID serves as a mailing address for the object, and therefore it should be such as to be able to locate the object easily. In PRACTIC, OIDs are represented as *`number%class`*, where *`number`* is a locally unique number of the object created in *`class`*. This representation allows the direct location of the class of the recipient object. In PRACTIC, *`number`* is not globally unique, as in other systems [22], since this requires exchange of information and synchronisation among the class-units. Instead, the combination of *`number`* and *`class`* provides a unique global identification for an object. The latter is a consequence of database distribution to non-shared memories.

4.2. Messages for Retrieving Instances

There are two types of such messages, sent either to active or to semi-active objects: a) unconditional request of the OIDs of all instances, and b) conditional request of some instances, according to the value of a certain attribute.

There is an implicit form of parallelism in both the above types of messages. The message is evaluated in parallel among the recipient class and its sub-classes. Both classes and meta-classes exhibit this kind of parallelism, however, class-hierarchy parallelism is more important, since it can significantly speed-up query processing in parallel OODBs [19], so it will be dealt more thoroughly later.

The general framework of execution of such messages is the following:

Active objects react in two ways when they receive such messages: a) they answer the request by executing the corresponding method, and b) they re-distribute the message to their sub-classes. The execution of the method inside the class can be performed either in parallel or sequentially. The choice of the algorithm lies entirely on the active object and it depends on the volume of its object warehouse and the availability of computational resources of its multi-processor sub-environment for the management of its extension.

Semi-active objects just re-distribute messages to their sub-classes, since they do not have instances of their own. Furthermore, they are responsible for notifying the sender object when message service ends.

4.3. Attribute-accessing and General Purpose Messages

These messages: a) retrieve, update or delete the value of an attribute, b) evoke a general purpose computation through a non-generic method, or c) delete an object. All three object categories respond to such messages, but in considerably different ways.

Active objects are distinct, permanent processes and their actual address is known to the object directories. The system is responsible to deliver directly the message to the active object. However, it may not immediately answer back, since the background procedure is executed and only on certain periods of time the object can receive and answer messages. Messages received when the active object executes its background procedure are queued and serviced on the first occasion in a FIFO manner.

The deletion of an active object causes an extensive process of garbage collection, since all its instances must be deleted as well. The process of the active object is "killed" afterwards.

Semi-active objects are not distinct processes, but their address is known to the object directories, through the "instance-of" tables. When they receive a message their active parent process evokes a new process to service the message. The semi-active process depends on its parent, because methods are not copied to its private memory. Instead it requests the corresponding method body from its parental active object. When the message is serviced, the semi-active process answers back immediately to the sender object and afterwards is "killed". The deletion of a semi-active object furthermore causes the deletion of the data structure from its parent process private memory.

Passive objects are not processes and their exact address is not known to the object directories. The appropriate representation of the OID, allows the easy location of its active class. As a consequence, the system directs the message towards the active class of the passive object. The active class, receives the message, retrieves the specified object and executes the corresponding method in its context. These messages resemble tuple updates-deletions in relational databases. The deletion of a passive object is straightforward since its data structure is just removed.

5. The Abstract PRACTIC Machine

The PRACTIC model is reflected in the abstract PRACTIC machine (APRAM). APRAM [8] consists of shared-nothing computational nodes, that are capable of communicating directly with each other via messages, through a fully connected global communication network (fig. 2). Each node has its own memory and/or disk.

Description. APRAM consists of two kinds of processing elements (PEs):

The Active Object Processors, and

The Active Extension Managers.

The *Active Object Processors (AOP)* are the processors where the processes that correspond to the active objects run. They consist of a CPU, memory and links to other AOPs. AOPs are responsible for a) running the permanent procedure of the active objects, b) accepting messages destined at either the active object they host or at a non-active instance of the active object, c) evoking the semi-active processes, and d) coordinating query execution for the passive instances on the AEMs.

Semi-active instances of an active object are hosted in the same AOP, as their active meta-object, to avoid method copy overhead at run-time. Semi-active processes are only evoked on demand, and thus a) do not occupy much private memory and b) do need code from their meta-objects to respond to messages. When a message arrives for a semi-active object, its active meta-object process is suspended and the local scheduler is responsible for evoking semi-active object processes.

The volume of passive instances dictates the need for multiple PEs to manage the extension of active classes efficiently. The *Active Extension Managers (AEM)* are responsible for processing the passive extension of an active class. AEMs are exclusively devoted to one active class and consequently they are attached to the corresponding AOP. They can directly communicate only to the AOP and to each other, through a local communication network and not to the rest AOPs and AEMs.

AEMs assist active objects, by sharing the load of method execution for passive instances. The latter are distributed among the AEMs. This introduces the possibility for intra-class parallelism, since the number of passive objects that can be concurrently processed depends on the number of AEMs attached to an AOP. Since I/O has been identified as a major bottle-neck in parallel database systems [12], passive objects are partitioned to multiple disks. Hence, some of the AEMs have a disk attached to store passive objects, while some others are used for processing only and they just have local memory and no disk. However, assuming that AEMs have large enough main memory and the appropriate architectural support [14], the same abstract architecture can be used for a main-memory database system [8].

The existence of multiple AEMs provides a natural declustering for the objects of a class [8]. Declustering criteria can be any one of the popular data partitioning strategies of parallel relational database systems [12], although a hashed partitioning on the OID seems best suited for a general-purpose OODB system. A novel non-uniform declustering strategy has also been proposed to increase the scalability and the performance at the class level [9]. Concurrency is controlled by granting locks at the AEM level, a rather medium-grained (and also adjustable) approach.

6. The PRACTIC Query/Method Execution Model

6.1. Introduction

Classes, which are equivalent to relations in relational databases, are the most interesting active objects, because they usually have lots of passive instances that are oftenly queried. Meta-classes, on the other hand, neither have many instances nor are frequently queried. In the following, the terms class and active object will be used inter-changeably.

Queries in DBMSs are expressed via ad-hoc query languages, which are declarative and expressive, but they are not computationally complete. OOPLs, on the other hand, can easily support queries expressed using primitive language constructs, but they are less declarative. However, query languages are computationally simpler than OOPLs and they can be easily optimised, so that the user does not have to provide optimum access plans.

Ad-hoc queries can be expressed inside the framework of an OOPL by formulating the query in a query language and then translating it into the base OO data language. For example, the following query is formulated in Daplex, a functional query language [24] that has been successfully used in OODBs [16], since its underlying functional data model is portable to a wide variety of object models:

```
foreach p in person
    such that age of p > 50
```

```
print name(p)
```

The above query is translated into the following fragment of the OO data language of ADAM [16], which is an extended Prolog:

```
get(P) => person,
get_age(A) => P, A > 50,
get_name(N) => P, write(N), nl
```

The symbol "=>" denotes message sending. On the left of "=>" is the method selector and on the right is the message recipient.

In the following we describe the various types of parallelism during query evaluation, i.e. inter- and intra-class parallelism. The latter can be further sub-divided to inter- and intra-object parallelism.

6.2. Inter-Class Parallelism

The classification hierarchy of classes implies a form of OR-parallelism during query execution. According to the translated typical query of the previous section and the schema of fig. 3, the class `person` has both direct and indirect sub-classes, therefore the message sent to class `person`, applies to all its sub-classes, as well, because a sub-class "recognises" all the methods of its super-class.

When a class receives a message, it re-distributes it to all its direct sub-classes. At the same time it services the message internally, by finding all the applicable instances and executing the corresponding method on them. The sub-classes that receive the re-distributed message behave similarly, i.e. they re-distribute the message to their sub-classes, and so on so forth, until a class with no sub-classes is reached.

The results are directly returned to the "asker" object and details about which class produced them are hidden. The complete result is the merge of the partial results from each class. Message service proceeds in parallel at each active class involved. Therefore, class-hierarchy implies OR-parallel operational semantics for the inter-class parallel scheme.

When a class finishes with its local message servicing, two things can be done:

If the class has N sub-classes, it just waits for all of them to finish, as well.

If the class has no sub-classes and it is not the target class, it sends an "end-of-message-servicing" (`eom`) message to its super-class (-es).

If the class has no sub-classes and it is the target class, it sends an `eom` message to the sender of the original message.

When a class receives N `eom` messages from its N sub-classes, two things can happen:

If the class is the target of the original message, it sends an `eom` message to the sender of the original message.

If the class is a sub-class (direct or indirect) of the target class, it sends an `eom` message to its super-class (-es).

6.3. Intra-Class Parallelism

The details of query and method execution (ME) inside a class are hidden from the rest of the system, as a result of object encapsulation. The system can only "see" the response time of a class to a message. Each class may employ different techniques for parallelising local query processing to minimise query evaluation time. In the following, we identify two such parallelisation techniques, that are orthogonal to each other, namely inter- and intra-object parallelism.

6.3.1. Inter-Object Parallelism

Intra-class parallelism can be achieved, if the activation of passive instances of an active class is done in parallel. In this way, the processing of objects becomes concurrent. This kind of parallelism is equivalent to the data-parallelism of parallel relational database systems and is named inter-object parallelism.

In order to achieve such an internal parallelisation, an active class needs:

Multiple active processes devoted to the processing of parts of its extension, and

Scheduling and coordination among these processes.

Both the above pre-requisites are satisfied within the abstract PRACTIC machine. More specifically, the multiple cooperating processes for the parallel execution of queries on passive instances are mapped onto the AOPs, while the coordinator process runs on the AEM.

6.3.2. Intra-Object Parallelism

The method inheritance scheme implies parallel execution among the inherited pieces of method code. In order to execute a method on the selected objects the corresponding method body is required. There are three alternative scenarios for the retrieval of method bodies:

Methods can be totally defined at the current class descriptor, so no inheritance occurs.

Methods can be totally absent locally, so they must be inherited from a super-class.

Methods can be specialised versions of definitions in super-classes, so some parts can be found locally and some others must be inherited.

After method bodies are inherited, ME proceeds in parallel at each class. Although the operational semantics of methods are sequential [2], ME can become stream-parallel by parallelising the tasks of object selection and inherited body ME. Methods are still executed sequentially for each object, but each process, responsible for the execution of a different inherited part of the method, executes on a different object.

The notion of stream-parallelism between the inherited method bodies can be easily seen using an example (fig. 3). An attribute *age* is defined at the class *person*, along with generic methods for accessing and updating the value of that attribute. If the method concerning the update of *age* is specialised at a sub-class of *person*, e.g. *student*, then the following outline is used [25]:

```
execute code before the default method
delegate message to super-class (default behaviour)
execute code after the default method
```

The three stages of ME are independent but serialisable, i.e. the order of execution is always *before-default-after*. This is the most general case. Special cases, emerging when any of the three stages is missing, are treated in a different manner. For example, if *default* part is missing, then *before-after* parts are concatenated and override the default method. If *before-after* parts are missing then the default method is just inherited explicitly. If all three parts are missing, then the default method is implicitly inherited.

The three-stage ME holds for every sub-class of *person*, therefore a nested graph of independent pieces of code exists. This graph can be represented as an unbalanced binary tree (fig. 4a). The root of the tree is the *before* part of method definition at the target class. The left sub-tree represents the default method invocation and can be a tree itself, while the right sub-tree consists always of a single terminal node, the *after* part. Pre-order traversal of the tree provides the correct linearised ME order (fig. 4b).

The linearised order implies a stream-parallelism between the processes of the different nodes. Each node executes a part of the method in the context of a different object at the same time. When a process is complete with an object, it forwards it to the next process and receives another object from the previous one. Communication between processes is asynchronous, because the processing time of each may differ. This is achieved through the use of queues of objects to be processed.

The pipeline between different method parts can only be achieved if there are enough free processing elements. In the abstract PRACTIC machine, disk-less AOPs exist to realise stream-parallelism of ME. A pipeline, however, is not required to execute the checking of passive constraints upon the receipt of update messages, since passive constraints are independent from each other. A master-slave technique for independent parallelism of constraint execution can be found in [7].

7. Performance Issues

Inter-class parallelism, has proven to significantly speed-up query evaluation [19]. Furthermore, the hierarchical architecture of APRAM is best suited for inter-class parallelism, since it offers additional speed-up, as it has been analytically proven in [8]. This paper concentrates on the justification of intra-class parallelism, which allows each class-unit to handle internal data independently, encapsulating the parallelisation strategy.

In this section is examined, through simulation, which form of intra-class parallelism is better and in which context. This study will help the class optimiser to decide which strategy to employ to service a message, based possibly on statistical information. In the following the algorithms and the results of the simulation are described.

7.1. Simulation Algorithms

The simulation model, consists of processors, that play the role of Active Extension Managers (AEM). One of them has access to a disk (AEM_d). The simulation does not consider multiple disks, however, the results can be easily extended so. Method execution (ME) is handled in an abstract way, i.e. when a method is to be executed on an object, an adjustable time period is spent.

In the following three algorithms used in the simulations are described:

Sequential,

Data-parallel or inter-object parallel, and

Pipelined or intra-object parallel.

The *sequential* message service algorithm is very simple; it consists of a loop that a) reads all objects from the disk one by one, b) selects some of them according to the selectivity and c) executes all the method parts on them. The algorithm is called SEQ.

The *inter-object parallel* message service algorithm is more complicated, since synchronisation between the processors must exist. The AEM_d behaves almost as in SEQ, apart from that it waits for the “end-of-processing” (`eop`) messages from all the AEMs to stop. The AEMs receive objects and execute sequentially all method parts on them. When all objects finish, each AEM sends an `eop` message to the AEM_d. The algorithm is called DAT, for “DATa parallelism”.

In the *intra-object parallel* algorithm, ME is pipelined. Each AEM is devoted to a specific part of the method, therefore it executes only a part of the method on each object it receives. When it finishes, it passes the object on to the next AEM. The AEM_d behaves almost as in DAT. The last in chain AEM informs the AEM_d for the end of processing. The algorithm is called PIP, for “PIPelined parallelism”.

For all three algorithms the real time is counted for the AEM_d process to finish. In SEQ the whole processing takes place on AEM_d, while in the other two, the major processing task is distributed among the AEMs. However, synchronisation issues impose the timing of the AEM_d. For the DAT and PIP, the time spent for initialising the AEMs is also counted. All algorithms are included in the Appendix.

7.2. Simulation Results

Simulations are performed using a wide range of simulation parameters, in order to test performance for various cases. The main parameters (table 1) are:

The *object size*. The size in bytes of each object (or at least the average size) determines the I/O time and inter-processor communication time per object.

The *number of objects* determines the total message service time.

The disk *selectivity* determines the number of objects that are selected by the query for method execution (ME) from those that are retrieved. No index is assumed.

The *execution time* of one method part per object determines the total ME time per object.

The *number of parts a method* can be split into, determines the degree of parallelism for the pipelined ME.

Several simulations were conducted for many different combinations of the parameter values, but here the results of the five most representative ones are presented. Each one demonstrates how different algorithms behave as one of the parameters changes, while the rest remain fixed. In the following the term *effective time* is used to describe the rate that an event takes place, instead of the absolute period of time spent.

For the first case (fig. 5), the effect of *object size* is examined. The increase of object size causes an increase in the I/O time and processor communication time per object. Thus the increase on the overall message service time was expected. The increase is slightly over-linear, because I/O and communication increases over-linearly with object size. The SEQ algorithm suffers the least, because only I/O is affected (there is no inter-processor communication). The PIP algorithm suffers the most, because it has both I/O and communication. Communication overhead burdens more PIP than DAT.

The second case (fig. 6) shows the effect of the *number of objects* on the overall execution time. The SEQ algorithm behaves linearly, as expected. On the other hand, PIP and DAT behave slightly under-linear. This happens because the more objects exist, the more the processors are utilised. Also the percentage of initialisation overhead degrades with the number of objects.

The third case (fig. 7) shows the effect of *selectivity*. SEQ is slightly under-linear, because the effective I/O time per object decreases, as the selectivity increases. On the other hand, the other two algorithms are well over-linear, because when selectivity increases, the effective communication time decreases. The ME time remains the same though, therefore a congestion at the input queues of processors occurs. The management of this congestion (by the operating system) causes an increase in the execution time.

The fourth case (fig. 8) shows the effect of the increase on the average *method execution time*. SEQ is linear, as expected, while the other two algorithms are well over-linear, because of the same congestion problems already discussed. Congestion occurs because the more time a single object is processed, the more time the rest are waiting in the queue.

Finally, the last case (fig. 9) shows the effect of the *number of parts* that a method is split (specialisation inheritance). SEQ behaves linearly. DAT is slightly over-linear due to congestion problems. Congestion occurs because more method parts, mean more processing per object, i.e. the rest of objects are waiting in the queue.

The PIP algorithm exhibits two different kinds of behaviour. At the beginning it is over-linear, because the more parts in a method, the more processors are needed. Therefore, the processor initialisation and communication overheads increase. However, PIP is suited to many method parts and many pipelined processors, therefore after some point in the increase of method parts, the overhead by initialisation and communication is insignificant compared to the speed-up caused by the parallel processing of objects in the pipeline. Thus, after (approximately) 3 parts, the behaviour of PIP becomes under-linear.

As a conclusion, it can be said that the object-parallel execution almost always performs slightly better than the pipelined execution and both always perform much better than the sequential one. However, the object-parallel algorithm can perform not so good under non-uniform selectivity at each node (skewness), therefore there is no definite answer which algorithm is better, under the current criteria. Extending the simulation model will shed light to various aspects of these algorithms and will help to develop hybrid ones, that avoid the problems of both. The development of analytic models will help the local class optimiser to choose the best intra-class parallelism strategy for each case.

8. Summary, Conclusions and Future Work

In this paper an object model for a parallel object-oriented database system and its abstract machine have been described. The object model (PRACTIC, after PaRallel ACTIve Classes) is based on the concurrent execution of active objects. Active objects are permanent in-memory processes that have direct instances and can generate new ones. The most interesting active objects are classes, since their passive instances are the actual data.

The PRACTIC query/method execution model, supports two major types of parallelism, the inter- and intra-class parallelism. The former is implied by the class-hierarchy graph and the concurrency of active classes, while the latter can take two forms, which are orthogonal to each other: a) the data-parallel nature of objects (inter-object parallelism), and b) the stream-parallel execution of methods due to specialisation of methods across the inheritance path (intra-object parallelism).

The abstract PRACTIC machine (APRAM) directly maps active objects to distinct processing elements (Active Object Processors), while it provides active objects with additional resources (i.e. disks, memory and processors), to manage their extensions efficiently. Semi-active objects are hosted at the processor of their meta-object, while passive objects are stored and managed by the Active Extension Managers.

Simulations have proven that intra-class parallelism on the APRAM exhibits a considerable potential for query processing speed-up. The comparison between the two algorithms for parallel local query processing showed that the inter-object parallelism performs slightly better than intra-object parallelism. Both, however, can perform badly under different conditions, therefore hybrid algorithms that avoid the pitfalls of both algorithms and a local optimiser are needed.

In order to get optimum performance out of a real implementation of PRACTIC various aspects for global optimisation must be considered, too. These include global parallel query processing models and finer-grained method execution based on the underlying AND-parallelism of Prolog. Prolog has been chosen as the implementation language, because PRACTIC is aimed to be a fast knowledge/data base tool for an intelligent database [6].

Current work includes the development of a prototype of the PRACTIC model on a transputer[1] network, using CS-Prolog[2]. The same platform was also used for developing the simulations.

Acknowledgments

The first author is supported by a scholarship from the Greek State Scholarships Foundation (I.K.Y.).

References

[1] G. Agha, Concurrent object-oriented programming, *Communications of the ACM* 33(9) (Sept. 1990) 125-141.

[2] P. America, POOL-T: A parallel object-oriented language, in: *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro (eds.) (MIT Press, 1987).

[3] P.M.G. Apers et al., PRISMA/DB: A parallel main memory relational DBMS, in M.H. Eich, ed., Special section on main-memory databases, *IEEE Trans. on Knowledge and Data Engineering* 4(6) (Dec. 1992) 541-554.

[4] M.P. Atkinson et al., Object-oriented database system manifesto, *Proc. 1st Internat. Workshop on Deductive and Object-Oriented Databases* (North-Holland, 1990).

[5] N. Bassiliades and P.M.D. Gray, CoLan: A functional constraint language and its implementation, to appear in *Data & Knowledge Engineering* 14 (1995).

[6] N. Bassiliades and I. Vlahavas, A multi-processor machine for the parallel management of an active object-oriented database (in Greek), *Proc. 4th Hellenic Conf. on Informatics*, Patras, Greece (Dec. 1993).

[7] N.Bassiliades and I.Vlahavas, Constraint checking in a parallel object-oriented database system, to appear in *Journal of Parallel Algorithms and Applications* 5(3+4) (1995).

[8] N.Bassiliades and I.Vlahavas, A hierarchical abstract architecture for parallel query execution in an object-oriented database system, submitted for publication (1994)

[9] N.Bassiliades and I.Vlahavas, A non-uniform data fragmentation strategy for parallel main-memory database systems, submitted for publication (1995).

[10] L. Bic and L.R. Hartmann, AGM: A dataflow database machine, *ACM Trans. Database Systems*, 14(1) (1989) 114-146.

[11] W. Bronnenberg, L. Nijman, E. Odijk and R. Twist, DOOM: a decentralized object-oriented machine, *IEEE Micro Machine* (Oct. 1987) 52-69.

[12] D. DeWitt and J. Gray, Parallel database systems: The future of high performance database systems, *Communications of the ACM* 35(6) (June 1992) 85-98.

[13] O. Diaz and N.W. Paton, Extensibility through metaclasses in object-oriented systems, *Proc. 3rd Internat.. Conf. on Information Systems Developers Workbench*, Gdansk, Poland (Sept. 1992).

[14] H. Garcia-Mollina and K. Salem, Main memory database systems: An overview, in M.H. Eich, ed., Special section on main-memory databases, *IEEE Trans. on Knowledge and Data Engineering* 4(6) (Dec. 1992) 509-516.

[15] A. Goldberg, D. Robson, *Smalltalk-80, The Language and its Implementation* (Addison-Wesley, 1983).

[16] P.M.D. Gray, K.G. Kulkarni and N.W. Paton, *Object-Oriented Databases, A Semantic Data Model Approach* (Prentice Hall, 1992).

[17] A.S. Grimshaw, Easy-to-use object-oriented parallel processing with Mentat, *IEEE Computer* 26(5) (May 1993) 39-51.

[18] S.P. Hufnagel and J.C. Browne, Performance properties of vertically partitioned object-oriented systems, *IEEE Trans. on Software Engineering* 15(8) (Aug. 1989) 935-946.

[19] K.C. Kim, Parallelism in object-oriented query processing, *Proc. 6th Internat. IEEE Conf. on Data Engineering* (1990) 209-217.

[20] W. Kim, Research directions in object-oriented database systems, *Proc. 9th Conf. on Principles of Object Database Systems* (1990) 1-15.

[21] W. Klas, K. Aberer, E. Neuhold, Object-oriented modelling for hypermedia systems using the VODAK model language, in *Object-Oriented Database Management Systems*, NATO ASI Series, Springer-Verlag, Berlin (Nov. 1993).

[22] N.W. Paton, ADAM: An object-oriented database system implemented in Prolog, in: *Proc. 7th British National Conf. on Databases*, Williams (ed.) (CUP, 1989) 147-161.

[23] N.W. Paton and O.G. Diaz and M.L. Barja, Combining active rules and meta-classes for enhanced extensibility in object-oriented systems, *Data & Knowledge Engineering* 10(1) (Jan. 1993) 45-63.

[24] D.W. Shipman, The functional data model and the data language DAPLEX, *ACM Trans. on Database Systems* 6(1) (1981) 140-173.

[25] M. Stefik and D. Bobrow, Object-oriented programming: Themes and variations, *AI Magazine* 6(4) (1986) 40-62.

[26] A.K. Thakore and S.Y.W. Su, Performance analysis of parallel object-oriented query processing algorithms, *Distributed and Parallel Databases* 2(1) (1994) 59-100.

[27] S.B. Zdonik and D. Maier (eds.), *Readings in object-oriented database systems* (Morgan Kaufmann, 1990).

Appendix

Here the algorithms for the simulations described in section 7 are presented. In the description of the algorithms, `eof` stands for "end-of-file", `eob` stands for "end-of-objects" and `eop` stands for "end-of-processing".

The following is the SEQ algorithm for the AEM_d:

```
BEGIN
    REPEAT
        READ an object from the disk;
        IF selection criteria is true, THEN EXECUTE all method parts (sequentially)
    UNTIL an eof is READ
END.
```

The following is the DAT algorithm for the AEM_d:

```
BEGIN
    REPEAT
        READ an object from the disk;
        IF selection criteria is true, THEN
                BEGIN
                        CHOOSE an AEM;
                        SEND the object to the AEM
                END
    UNTIL an eof is READ;
    SEND an eob message to all AEMs;
    WAIT-UNTIL an eop message is RECEIVEd from all AEMs
END.
```

The CHOOSE algorithm just picks-up an AEM using the round-robin algorithm. However, alternative algorithms can be used as well [7].

```
BEGIN
    IF the algorithm runs for the first time, THEN SET CurrentAEM to 0;
    SET CurrentAEM to CurrentAEM + 1;
    IF CurrentAEM > MaxAEM THEN SET CurrentAEM to 1;
    RETURN CurrentAEM
END;
```

The DAT algorithm for the AEMs is the following:

```
BEGIN
    REPEAT
```

```
        WAIT-UNTIL a message is RECEIVEd;
        IF the message is NOT eob, THEN EXECUTE all method parts (sequentially)
    UNTIL an eob is RECEIVEd;
    SEND an eop message to the AEMd
END.
```

The following is the PIP algorithm for the AEM_d:

```
BEGIN
    REPEAT
        READ an object from the disk;
        IF selection criteria is true, THEN SEND the object to the first-in-chain AEM
    UNTIL an eof is READ;
    SEND an eob message to the first-in-chain AEM;
    WAIT-UNTIL an eop message is RECEIVEd from the last-in-chain AEM
END.
```

The PIP algorithm for all the AEMs, except for the last-in-chain, is the following:

```
BEGIN
    REPEAT
        WAIT-UNTIL a message is RECEIVEd from the (N-1)th in-chain AEM;
        IF the message is NOT eob, THEN
                BEGIN
                        EXECUTE the Nth part of the method;
                        SEND the object to the (N+1)th in-chain AEM
                END
    UNTIL an eob is RECEIVEd;
    SEND an eob message to the (N+1)th in-chain AEM
END.
```

The last-in-chain AEM follows a slightly different PIP algorithm:

```
BEGIN
    REPEAT
        WAIT-UNTIL a message is RECEIVEd from the (LAST-1)th in-chain AEM;
        IF the message is NOT eob, THEN EXECUTE the LASTth part of the method
    UNTIL an eob is RECEIVEd;
    SEND an eop message to the AEMd
END.
```

Figure Captions

Table 1. Simulation parameters

Figure 1. Instance hierarchy of PRACTIC objects

Figure 2. The abstract PRACTIC machine

Figure 3. University database schema

Figure 4. Pipelined method execution scheme

Figure 5. Simulation results for various object sizes

Figure 6. Simulation results for various numbers of objects

Figure 7. Simulation results for various selectivity ratios

Figure 8. Simulation results for various method execution times

Figure 9. Simulation results for various numbers of method parts

Name	**Symbol**	**Range**
Object size	o	10 - 1000 (bytes)
No. of objects	N	10 - 1000
Method parts	m	1 - 4
Method part execution time	t_m	0.01 - 1 (sec)
Selectivity	s	0.01 - 1

Table 1

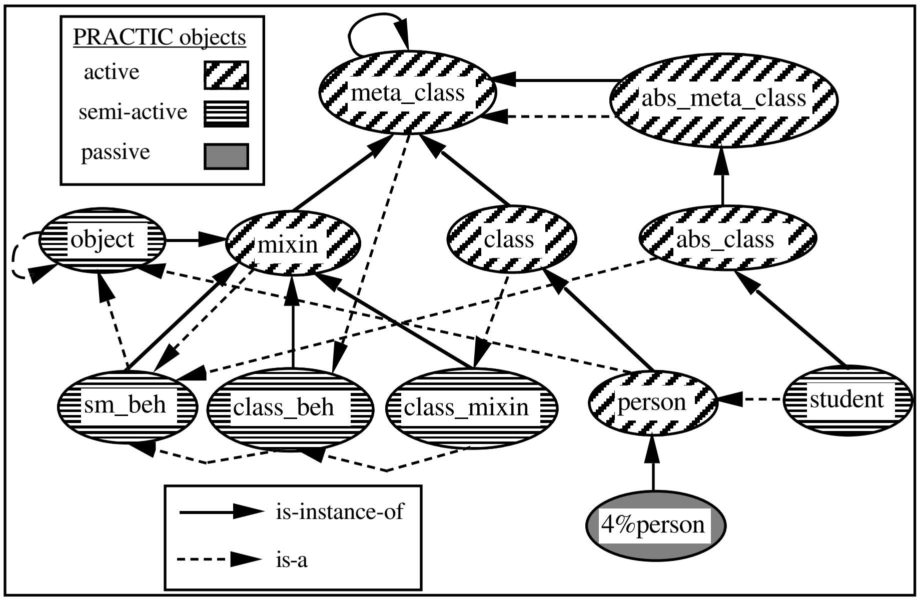
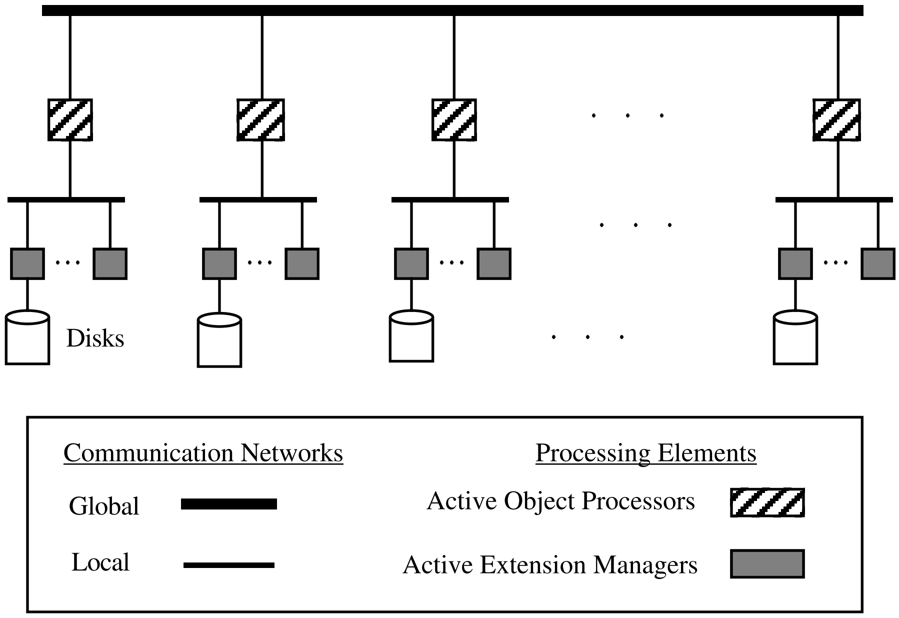

PRACTIC objects
active
semi-active
passive
meta_class
abs_meta_class
object
mixin
class
abs_class
sm_beh
class_beh
class_mixin
person
student
4%person
is-instance-of
is-a

Figure 1



Disks
. . .
Communication Networks
Global
Local
Processing Elements
Active Object Processors
Active Extension Managers

Figure 2

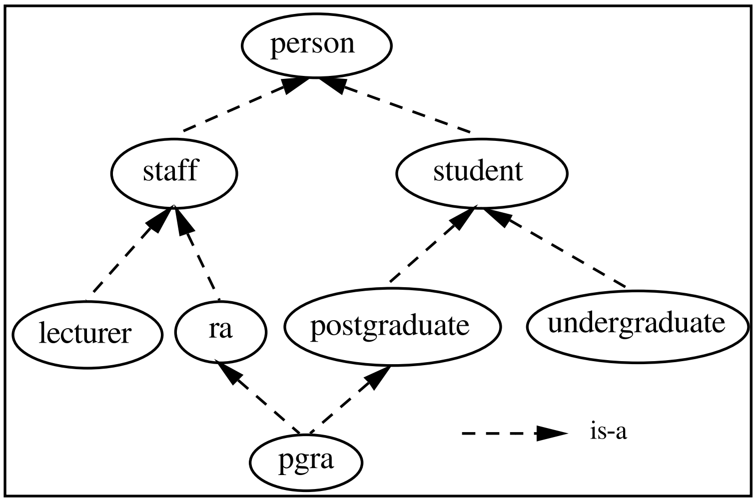

person
staff
student
lecturer
ra
postgraduate
undergraduate
pgra
is-a

Figure 3

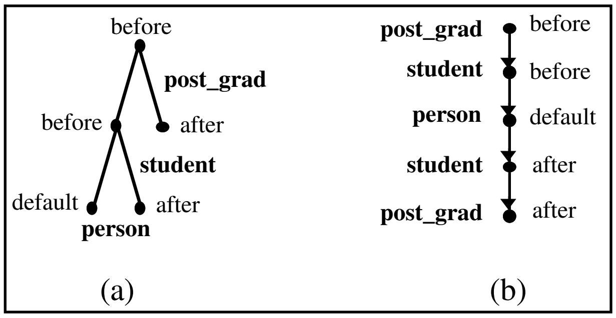
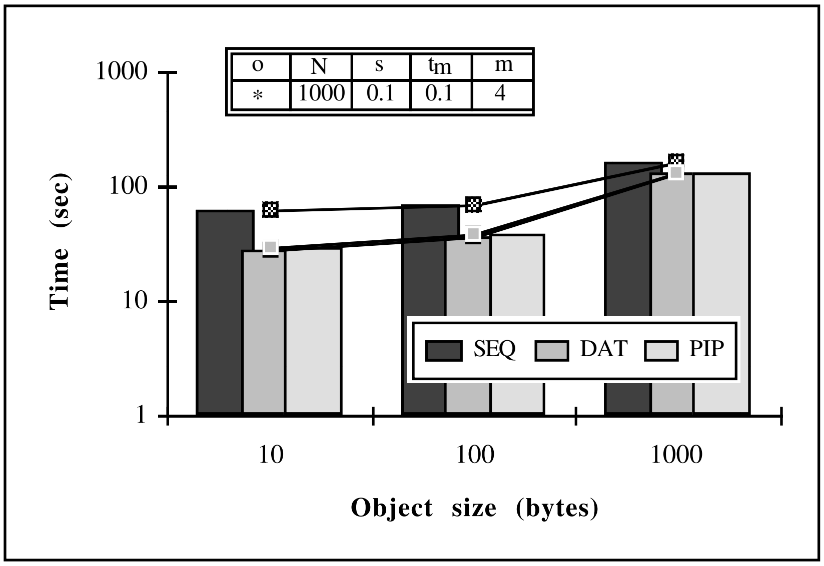

before
post_grad
before
after
student
default
after
person
(a)
post_grad
before
student
before
person
default
student
after
post_grad
after
(b)

Figure 4



o	N	s	tm	m
*	1000	0.1	0.1	4

1000
100
10
1
Time (sec)
SEQ
DAT
PIP
10
100
1000
Object size (bytes)

Figure 5

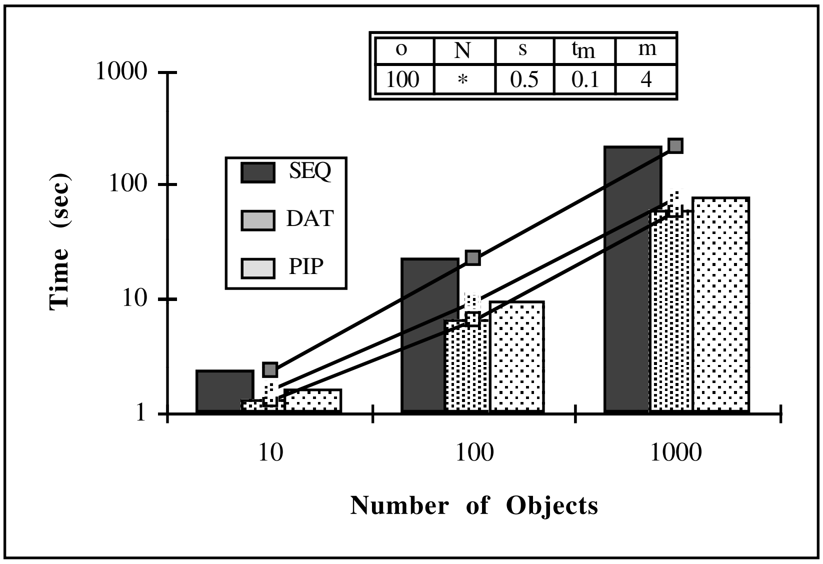

o
N
s
tm
m
100
*
0.5
0.1
4
1000
100
10
1
Time (sec)
SEQ
DAT
PIP
10
100
1000
Number of Objects

Figure 6

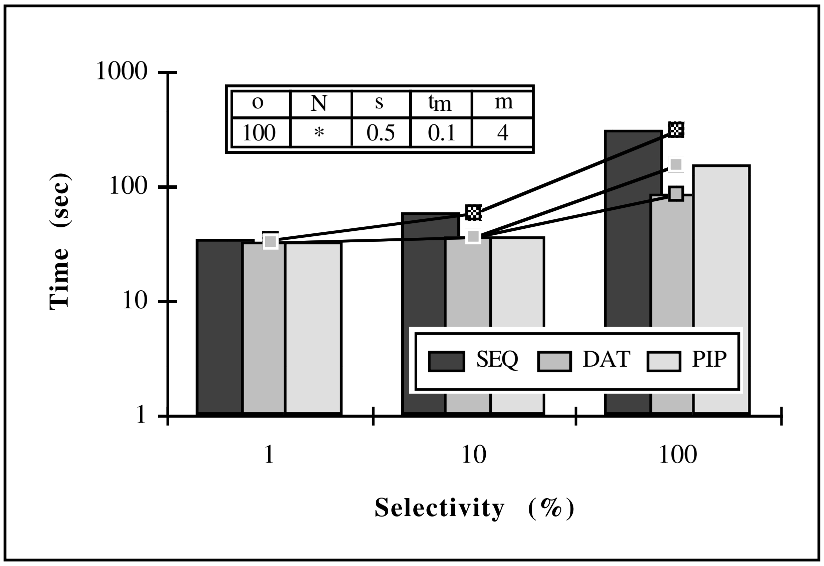
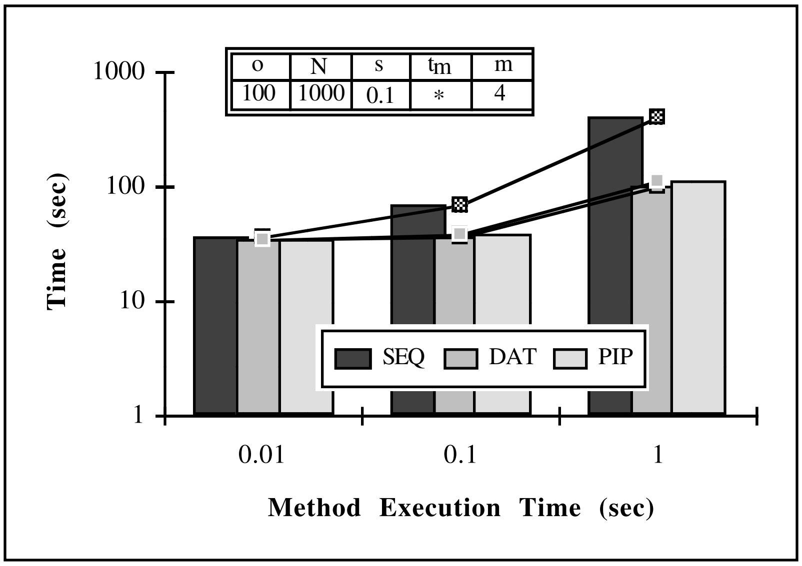

1000
100
10
1
Time (sec)
o
N
s
tm
m
100
*
0.5
0.1
4
SEQ
DAT
PIP
1
10
100
Selectivity (%)

Figure 7



o
N
s
tm
m
100
1000
0.1
*
4
1000
100
10
1
Time (sec)
SEQ
DAT
PIP
0.01
0.1
1
Method Execution Time (sec)

Figure 8

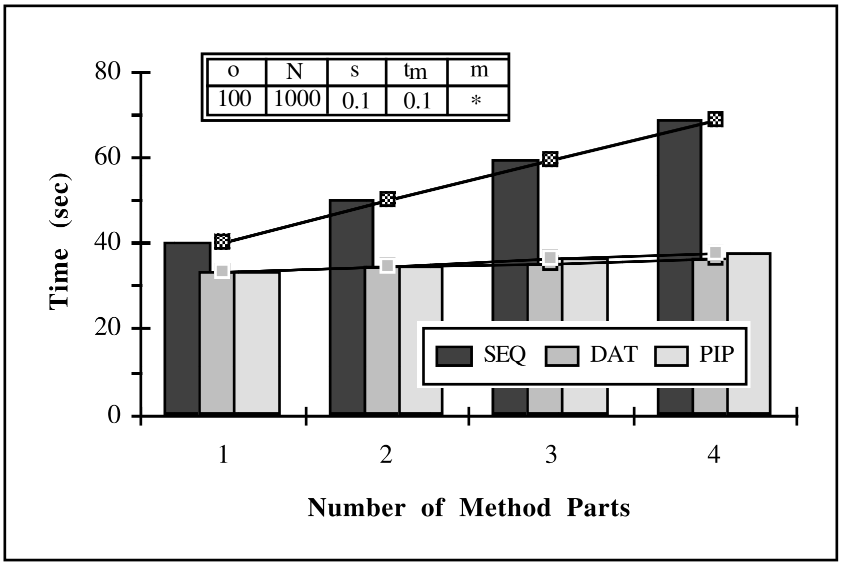

o
N
s
tm
m
100
1000
0.1
0.1
*
80
60
40
20
0
Time (sec)
SEQ
DAT
PIP
1
2
3
4
Number of Method Parts

Figure 9

Footnotes

[1]Transputer® is a trademark of Inmos Ltd.

[2]CS-Prolog® is a trademark of Multilogic Computing Ltd.