CoPO: Collections of Persistent Objects

Stoyan Garbatov  
INESC-ID/IST  
stoyan.garbatov@ist.utl.pt

João Cachopo  
INESC-ID/IST  
joao.cachopo@inesc-id.pt

Jan 2009
The objective of this work is to develop a set of Java APIs that allow an efficient access to persistent information necessary for the processing of tasks by an application and to simplify the implementation of a rich domain model, through the addition of new methods for manipulating collections of objects. The model developed here is inserted in the context of web applications implemented in Java that make use of software transactional memory (STM). An STM is defined as a concurrency control mechanism analogous to database transactions where the access to shared memory, in concurrent computing, is controlled. The solution presented here is applicable to applications that do not make use of an STM without losing too much of its efficiency. Due to the operation in transactional context, only the read operations are optimized since any write operation that may take place during the transaction is persisted when the transaction commits successfully. Nothing is done to increase the efficiency of persistent data write operations. The work is to be developed in the context of the Fenix Framework.

**Keywords:** Software Transactional Memory, Object-Relational Mapping, Pre-Fetching, Persistent Data Manipulation.
CoPO: Collections of Persistent Objects*

Stoyan Garbatov and João Cachopo
INESC-ID/IST

Abstract. The objective of this work is to develop a set of Java APIs that allow an efficient access to persistent information necessary for the processing of tasks by an application and to simplify the implementation of a rich domain model, through the addition of new methods for manipulating collections of objects. The model developed here is inserted in the context of web applications implemented in Java that make use of software transactional memory (STM). An STM is defined as a concurrency control mechanism analogous to database transactions where the access to shared memory, in concurrent computing, is controlled. The solution presented here is applicable to applications that do not make use of an STM without losing too much of its efficiency. Due to the operation in transactional context, only the read operations are optimized since any write operation that may take place during the transaction is persisted when the transaction commits successfully. Nothing is done to increase the efficiency of persistent data write operations. The work is to be developed in the context of the Fenix Framework.

1 Introduction

In the majority of today’s web applications, the domain logic is implemented in an object-oriented programming language such as Java. At the same time, the most common solution for the persistent storage of the information manipulated by the application is to use relational database management systems. Object-oriented languages and relational database systems are associated with different paradigms which makes the development and maintenance of such applications difficult. This paradigm discrepancy is also known as the impedance mismatch problem. In order to eliminate this problem, in the context of the Fenix system (implemented at IST, TU-Lisbon), a new architecture for web applications was developed, with the intent of hiding the existence of a relational database from the application programmer. Even though it achieved its goals completely, this new solution restricted not only the way that some types of information are accessed, but also the efficiency of certain operations.

The work presented here is to be developed in the context of the PASTRAMY project. The objectives of this project are to design and implement an optimized persistence store to Software Transactional Memory (STM) that supports the durability property of software transactions and provides efficient read access to

* This work is being performed in the scope of the Pastramy project (PTDC/EIA/72405/2006).
objects; to design and implement a collection of replication strategies tailored to build a reliable and distributed STM and to deploy and evaluate the solutions in a realistic setting.

A rich domain model is usually composed by many tightly inter-related classes. For instance, in the Fenix system there are more than 1000 types of entities that are related among themselves through over 1300 bidirectional relationships. The experience acquired with the Fenix system shows that most of those relationships are one-to-many, or, in other words, they are relationships where one entity relates itself with several other entities. The most common approach to represent these one-to-many relationships, in Java, is to keep within an object a collection of objects with which it is related. Consequently, in order to implement many of the operations existing in the domain model, it is necessary to access the contents of those collections.

Fig. 1: A simple domain model

```java
for (Employee e : company.getEmployees()) {
    print(e.getName() + "": " + e.getManager().getName());
    print(e.getDepartment().getName());
    for (Project p : e.getProjects()) {
        printProject(p);
    }
}
```

Fig. 2: Sample code traversing entities

In many cases, there is a need to access all the elements of a given collection (for instance, to check if a given element belongs to the collection). These collections may also be subject to additional restrictions, such as having their elements ordered or being indexed by a certain property. On the Java platform there are several different types of collections (such as lists, sets, and dictionaries). These collections do not support (by themselves) persistent data and, as such, this makes them harder to use in the implementation of a persistent domain model. On the other hand, in most situations where it is possible to use them (by making use of mechanisms that take care of the persistence of underlying data), the efficiency of operations over them is not very good (at least when compared with hand written queries for manipulating the same data). An example of a simple domain model can be seen in Fig. 1 whereas Fig. 2 shows an example traversing of that model.
2 Goals

The main objective of this work is the development of a set of Java APIs that allow an efficient access to persistent information necessary for the processing of tasks by an application and to simplify the implementation of a rich domain model, through the addition of new methods for manipulating collections of objects. A comparison of the behaviour of alternative implementations that provide the same functionality is to be performed also.

3 Related Work

Many studies have been done regarding the impedance mismatch problem combined with the need for efficient manipulation of persistent data. In this section, some of the work published and relevant to the current goals is reviewed. There is a great variety of approaches, which can be followed with the intent of achieving the above objectives, and they can be classified in a set of categories sharing common properties.

One of these categories is described as where the object-oriented language itself is extended in such a fashion that there is a straighter relationship between the underlying persistent data and the entities manipulated by the application. This is usually achieved by adding new concepts and/or constructs to the language in consideration, allowing the programmers to benefit from enhanced ability and flexibility to manipulate persistent data. It should be noted that these approaches imply a modification in the programming model that is to be used.

Another category may be identified for solutions that involve some type of analysis (being that static or dynamic) of the behaviour of the application with the intent of being able to predict the data access patterns that are performed. This is subsequently used to load the data that is needed by the application in a pre-emptive manner (that is, pre-fetching it) so that there is no efficiency loss due to idle waiting while the data is being loaded on demand. There is no implied modification in the programming model.

Both of these approaches shall be reviewed in the following sections.

3.1 Language Extensions

Willis et al in [1] present a system denominated Java Query Language (JQL). JQL is an extension to Java that allows the upgrade of queries to first order entities of the language. As such, it is possible to perform queries over all the objects and collections of objects present during the execution of a program. This is done with the intent to create not only a more flexible way for programmers to manipulate data, but also to allow for more efficient persistent data operations.

The system can be decomposed into three main components: a compiler, a query evaluator, and a run time system. The compiler consists in a source-to-source translator (in other words, it takes source code as input and generates
source code as output). It is responsible for the replacement of select declarations by the equivalent Java code.

The source code that is produced generates the query tree (which is the internal representation used for the queries by the system) and passes it to the query evaluator. The objective of the query evaluator is to apply all the possible optimizations that can enhance the efficiency of the query evaluation. Internally, the query evaluator has a staged query pipeline. It is in this pipeline that the processing and optimization of the queries are effectively performed (for each of the conditions present in the query, there is an independent processing phase in the pipeline - it corresponds to a single join operation).

Finally, the run time system (which is implemented in AspectJ) is responsible for keeping a reference to each and every object and collection of objects that might be created during the execution of the program. It is this feature that allows queries to be performed over all objects, including those that are not contained in collections.

One of the concerns that were not considered in the work of Willis et al is the fact that queries may use methods with side effects. This is mainly due to the fact that there are no guarantees as to the order according to which the queries are going to be evaluated in the query pipeline. When a query were to employ methods with side effects, the results that are produced could very well be different with each evaluation of the query. The system does not keep a record of the queries that have already been evaluated and, consequently, every time a given query is found, it is necessary for it to be processed (even if it might have already been evaluated at some previous point in time).

Willis et al present in [2], a follow-up of their work from [1]. The technique described in this newer work covers the caching and incrementalizing of queries in object oriented programming languages with mutable state. This extension to JQL allows the caching of queries that have already been evaluated and reuse their results when they are needed the following time. Additionally, it is possible to update, in an incremental manner, the query cache so that it can reflect correctly all the modifications that may occur to the state of the subjacent objects.

The caching of query results is performed with the intent of optimizing the situations in which certain queries need to be evaluated repeatedly. In these situations, it is of great interest to have the necessary results already available without being necessary to evaluate the considered query once again. There are overheads associated with the caching, and, as such, heuristics are applied so that it is possible to determine the queries for which the caching is most beneficial.

The main parameters used by the heuristics to decide whether a given query should be kept or not in the cache are the number of times that the query under consideration is evaluated and the number of times that the data that supports that query is updated. In other words, the major issue is the relationship existing between these two factors. If, for a given period of time, a query is evaluated relatively few times, while, at the same time, the underlying data is modified frequently, then, the possible gains, in terms of performance that
would be obtained by keeping the query in cache, would be eliminated by the overheads. This can be explained with the incremental updates caused by the constant modification of the data. Taking this into account, only in situations where the modification of the data is much less frequent than the evaluation of the queries is justified to keep the associated results in cache. Statistical data regarding the rates of query evaluation and data modification is kept so that it is possible to decide upon the time intervals during which a query should be kept in cache or not. This book-keeping is performed even for queries that are not currently in cache.

From the moment at which the data is saved in cache, it is necessary to guarantee that it reflects correctly the current state of the program. Updates on the objects may lead to inconsistencies in the data contained in the cache. As such, it is necessary to update the cache in an incremental manner, adding and removing objects as is deemed necessary. All types of access that allow the modification of the state of the underlying objects, being that through the insertion/removal of elements from collections or the modification of the values of object fields, are instrumentalized (with AspectJ, in the concrete solution that is presented). This instrumentalization is performed to detect any relevant modification so that it is possible to apply the necessary updates to the cached queries in order to keep them consistent.

The main restrictions present in this solution are associated with the high overheads associated with the incremental updates of the cache. Since it is necessary to keep statistics for all the existing queries, and the number of queries can be potentially very high, the cache manager has to perform its own garbage collection over the statistical data regarding queries that have become inactive. Additionally, it is necessary that the fields which are to be monitored for modifications by the system are annotated.

Additionally, it is necessary, for the fields to be monitored for modifications by the system, to be annotated. There is no other way (at least while employing AspectJ) to know if a certain field is involved or not in some query.

In [3] Meijer et al. present the .NET Language-Integrated Query (LINQ) framework. The goal is to facilitate the manipulation of data originating from different data models. The approach that is followed in the framework is the definition of a design pattern of general purpose standard query operators for traversal, filter, and projection. Apart from the common query operators, this framework provides two domain-specific APIs which work over XML and relational data. They are named XLinq and DLinq respectively.

### 3.2 Pre-Fetching

The types of fetching policies are categorized into on-demand-fetching and pre-fetching. In on-demand fetching, the objects are fetched from the server/database on request. The main advantage of this policy resides in the fact that only objects that are eventually accessed are effectively loaded. Its disadvantage is that it causes many round-trips since, for every object that needs to be accessed, a round-trip will have to be made in order to load it.
With pre-fetching, the objects that are expected to be accessed in the future are loaded in advance, effectively reducing the number of round-trips that need to be made. Consequently, the system performance is increased, assuming that the pre-fetched objects are indeed accessed. However, when the pre-fetched objects are not accessed, the system performance will degrade because of the extra overhead caused by the unnecessary loading of objects. Taking this into account, it is important to correctly predict the future access patterns of the application in order to effectively pre-fetch objects.

Existing pre-fetching techniques can be classified into the following four categories, based on the method of selecting the candidate objects to be pre-fetched: paged-based pre-fetching; object-level/page-level access pattern based pre-fetching; user-hint based pre-fetching, and, lastly, context-based pre-fetching.

The page-based pre-fetching techniques fetch all the objects that are contained in the same page as the object that needs to be loaded. This method works well only when the objects in the same page are accessed consecutively. Otherwise, the benefit obtained from pre-fetching is lost. Since the effectiveness of this class of techniques depends entirely on the clustering of the objects contained in a page, they fail to work as expected when the applications do not access the data according to the order of clustering.

Pre-fetching techniques based on object-level/page level access patterns try to predict future object/page access patterns from recent object/page access references.

The user-hint pre-fetching techniques pre-fetch objects based on hints provided by the user. Their drawbacks reside mainly in the increased burden on the programmers and the fact that they do not allow for any auto-tuning of the system.

The context-based pre-fetching techniques fetch all the objects (or parts of the objects) contained in the context of the object that is requested. The context of an object describes the structure (or some other similar feature) in which the object was loaded. An example of such a context would be a query result or object collections. This method is effective when an application traverses the object hierarchy (graph) in a breadth-first-search (BFS) fashion. The main problem associated with this class of techniques is when applications traverse the object graph in a depth-first-search (DFS) fashion. When this takes place, objects that have been pre-fetched may be replaced from the cache before they are actually accessed. This is aggravated when the context is a large structure.

**Page-Based Pre-Fetching**

Knafla presented in [4] a methodology where the main goal is to facilitate the pre-fetching of persistent objects through the prediction of data page access. The attention is concentrated upon client/server systems with data persistence, where the data transfer unit is the page. In other words, if a client asks for a certain object from the server, then the server will reply with a page of data.
containing the object that was asked for as well as all the adjacent objects that are stored on that page.

The relationships between the objects are modelled by a discrete-time Markov Chain. A discrete-time Markov Chain is a stochastic process (that is, a probabilistic model) through which it is possible to model the behaviour of systems. The model is constructed in such a way that the following state to which the system is going to transit does not depend on any previous states through which the system has already passed (but only on the current state).

The method that was developed is designated as hitting times and has as a final result the probability for a certain page of data to be accessed, as well as the average access time. The probabilistic calculation is performed taking into account the structure of the existing relationships between the objects. The probabilities associated to the (possible) transitions between objects are used for accessing adjacent pages, from the current position in the graph of objects. If the calculated probability exceeds a threshold limit defined while taking into account cost/benefit parameters (that include not only the benefits obtained in case the estimate is correct but also the penalty associated with an unnecessary pre-fetch), then the page is a candidate for pre-fetching.

An additional aspect which is considered in this approach is the best moment to do a given pre-fetch. This is due to the fact that the benefits that can be obtained, in terms of performance, with a given pre-fetch when it is successful, increase (up to a certain maximum limit from which on there is no more improvement) with the distance between the current position in the graph and the data that are going to be pre-fetched. However, when the previously referred distance is bigger, then the less precise is the method and, as such, the probability of making a correct decision decreases. Keeping this in mind, in the solution presented by Knafla, it tries to reach a balance between these two factors in order to maximize the benefits without compromising the accuracy of the method.

This work presents an interesting set of ideas. However, a less positive aspect is the fact that systems, which offer support regarding the localization of persistent application objects within the data pages to which they belong, as well as the relative position among the pages, are not a very common occurrence. Furthermore, no method is presented for calculating the probabilities that are associated with the transitions existing in between the objects, being in a somehow implicit manner considered as input from some external entity to the system.

**Object-level access pattern based pre-fetching**

Han et al presented in [5] a new technique for pre-fetching. One of the fundamental ideas behind their work is the fact that the access patterns employed by applications can be modelled in terms of the attribute references made when manipulating objects instead of the pattern of object references.

They introduce the concepts of type-level access locality and type-level access pattern. Applications traverse the graph of objects, which represent the domain entities, through the references existing among objects. Whenever distinct ob-
jects of the same type are navigated, it is frequent for the same attributes to be referenced repeatedly. This repeated referral to the same attributes in objects of the same type is what defines the type-level access locality. The type-level access pattern corresponds to the pattern of attributes that are referenced in such situations.

Based on these access patterns, the authors identify an iterative and a recursive type-level access patterns for ORDBMSs. These are subsequently used to perform prefetching of persistent data according to the information contained in the patterns.

**Context-Based Pre-Fetching**

Bernstein et al. presented in [6] a technique for data pre-fetching. The main concept behind it is to associate a context to every object at the time it is loaded. This subsequently allows using the context of the object and the concrete portion of an objects state that is loaded to interpolate about the probability of the application needing to manipulate the same portion of state of other objects sharing that context. That enables the loading of data that would be needed by the application in a pre-emptive fashion (pre-fetching it), effectively reducing the time lost while waiting for the data to be loaded on demand.

The pre-fetch technique is called a context controlled pre-fetch. The objective of this approach is to associate a context to each existing object. The context corresponds to the structure within which an object is loaded. It is used to direct the pre-fetch decisions which are taken in later phases. The authors identified several access patterns which can be used as indication for future similar types of accesses and on which are based pre-fetch optimizations. Since the indiscriminate use of those patterns is not guaranteed to produce an always accurate prediction, their use should be constrained. These constrains can be summed up as follows: when accessing a given object, pre-fetch all of its scalar attributes as well as all references it has to other objects, and whenever accessing a scalar or relationship attribute of an object then pre-fetch that attribute for all objects that are in the same context as the considered object. Whereas all these pre-fetch constraints are useful, the context associated with a given object may be potentially large and the loading of all scalar and relational attributes of the objects may not be feasible.

Bernstein et al. make several important contributions in [6]. There are a couple of aspects that have not been considered in it. One of them is regarding the fact that no pre-fetching across the graph of objects is performed. At most, the relational attributes of an object towards another object can be pre-fetched, but no actions are taken about the actual contents of the related object. Another aspect that is not taken in consideration is the possibility of loading the state of sets of objects in a lazy manner so as to improve the performance of situations where there are only insert operations to be performed over the set. In such situations, there is no actual need for any part of the state of the contained objects to be loaded.
Ibrahim and Cook presented in [7] a solution to the problem of the manual specification of pre-fetching in architectures with persistent objects. The system presented by them, called Autofetch, adds automatically specifications for the realization of pre-fetch operations. It is implemented as an extension to Hibernate.

One of the concepts introduced in their work is that of a traversal of the graph of objects that represent the entities of the application. A traversal is defined as the sub-graph of objects and associations that are exercised as a consequence of processing the results of a given query. The traversals capture the way that applications navigate through the graph of objects returned by the queries. Taking into account the fact that the navigation may involve more or less objects than the ones contained in the results, only the navigations that result in additional accesses to the database are included in the traversals.

While the system is in operation, the traversals are aggregated into traversal profiles. The traversal profiles keep information regarding the way that the associated object graphs were navigated through. In each of the nodes of the trees that represent a profile is kept information about the number of times a given object was loaded from the database as well as its potential. The potential of an object is defined as the number of times that an object that has a direct reference to the considered object was accessed.

A classification is applied to each query to allow several distinct queries share the same traversal profile. The criteria that is used to effectively classify the queries is the stack of invocations at the moment of execution of the query. In this way, queries that share an execution context are grouped into the same profile, because they will have similar traversals associated.

Each time a query is to be executed, the associated traversal profile is consulted, and a pre-fetch specification is defined. The concrete objects that will make part of the pre-fetch are determined based on the accumulated statistics for their nodes in the profile. When the quotient between the number of times that the object was loaded and its potential exceeds a pre-established threshold, the object in consideration will be a part of the concrete pre-fetch specification.

The work of Ibrahim and Cook presents some interesting contributions, yet, there are a few less positive aspects about it. One of those is related to the fact that, in order for the application to actually enjoy the benefits offered, it is necessary for the application to work for a certain period of time so as to allow for the accumulation of enough statistical data so that it may be used to generate an optimized pre-fetch specification. Additionally, the authors themselves point out towards the fact that it is not being taken into consideration situations in which the traversals may depend on certain conditions and, as such, unexpectedly influence the results.

Another work in the area of context-based pre-fetch can be found in [8] by Wiedermann and Cook. Their objective is to solve the problem of the impedance mismatch. What is sought is to combine the benefits from automatic object-relational mapping techniques and the manual optimization of queries. They carry out a static analysis of the source code which extracts the queries underly-
ing in the conditions and traversals of the graph of objects executed in programs with transparent access to persistent data and the subsequent replacement of the code with equivalent optimized versions of itself.

The process is split into three phases: identification of the traversals executed in the program, identification of the conditions under which the data is used and the creation of the queries, and the consequent modification of the program so that it may use the results of the query execution. The traversal analysis is defined as being an abstract semantic interpretation, where the values of the database are replaced by paths. A path associated to a given value is defined as the sequence of field names traversed in order to load the considered value. The accuracy of the results produced in the first phase can be improved significantly if the conditions under which the program traverses the data path are taken into account. It is relevant to point out that only conditions which can be expressed in the query language employed by the system (which is named Object Query Language - OQL) are taken into consideration. In the third and last phase, the results produced in the previous phases are used to split the initial program into queries and their client. The queries are responsible for obtaining the data over which the client works. It is possible to apply simplifications over the source code resulting in the removal of conditions that become redundant after being integrated in the queries.

One of the restrictions present in the work of Wiedermann and Cook is the fact that the imperative language over which the work was performed does not support procedures nor recursive flows of execution. Another aspect to be mentioned is that only read operations are considered, being left aside all the update and write operations.

In continuation from this work, Wiedermann, Ibrahim and Cook presented a better and expanded version of their system in [9]. The main features that were enhanced, when compared with what was previously achieved, are the fact that the system operates over Java (in contrast with the imperative kernel language used in the previous work), virtual method invocation is supported through the insertion of additional queries (where deemed necessary) and the situations where there is recursion are treated with a finite expansion of the associated code. The queries resulting from the analysis make use of Hibernate. In terms of the architecture of the solution, there are no significant modifications.

3.3 STM

As was referred in previous sections, an STM is a concurrency control mechanism for working with concurrent computing. Since the work is to be developed in the context of applications using STM, it is necessary to provide a better description of what an STM is and how it works.

The main idea behind the use of an STM is for the programmers to specify the operations which should be executed atomically instead of protecting the data accesses with locks. As such, these operations should execute as if they were the only ones being executed at the time, independently of the data accesses they may perform. Consequently, the task of the STM is to allow the greatest
parallelism in an application while preserving the semantics of atomic operations. A recently developed approach in the context of STM has been reported by Cachopo in [10].

There are two important concepts in the execution of an STM. They are the \textit{read set} and the \textit{write set}. From the perspective of an STM, a transaction is a sequence of read/write operations over shared locations. Consequently, the read set of a transaction is the set of locations read during the transaction. In a similar fashion, the write set of a transaction consists in the set of locations to which was written during the transaction.

Since transactions can execute in parallel, it is possible for their read/write sets to have elements in common. There is no problem if the common elements are among the read sets. However, if there is an intersection between a read and write set or the write sets of distinct parallel transactions, then the atomicity property will most likely be broken. As such, it is up to the STM to manage the transactions’ read and write sets and prevent such situations from occurring.

It should be noted that any transaction can terminate with only one out of two possible operations. These are the commit and the abort operation. As such, it is the commit operation that indicates that all the values that were written during the transaction should become accessible to others. If a transaction fails, then it is with the abort operation that is guaranteed that none of the values written during the transaction can be accessed by other transactions.

The property that must be verified for a transaction to be successfully committed is linearizability. For transactions to be linearizable, the informal definition states that: each transaction should appear to occur instantaneously at some point in time, between its beginning and end; and the execution of non-concurrent transactions preserve their invocation order. The last condition says that a transaction that starts after another ends should not appear to have executed before the previous one.

The general approach in STMs is an optimistic one. Consequently, transactions proceed with their execution without acquiring any synchronization locks and at some point of their execution verify if there are any conflicts with other concurrent transactions. Usually, a conflict takes place whenever a transaction needs to read (or write) a piece of data that has been modified by some other transaction. Once a conflict is detected, the procedure to be followed is to restart at least one of the transactions involved, in order to solve the conflict. Based on the assumption that conflicts are rare, it is acceptable to redo the execution of a transaction once a conflict has been detected. If the probability of a conflict is high, however, this optimistic approach can lead to the starvation of some transactions.

4 Solution’s Architecture

There are certainly many alternatives to approach the problem of impedance mismatch combined with the poor efficiency of automatic object-relational mapping techniques, and not all of them are viable to be applied in combination with
one another. This can be explained with the fact that either the properties they present are not compatible or because interoperating with one another in order to achieve the benefits of all the parts involved is simply not possible. This shall be taken into consideration when defining the techniques that will be employed in order to achieve the objectives presented in previous sections.

Since what is sought after is the development of a methodology that is as transparent, in its use, as possible (with regards to the operation optimizations), for the application programmer, approaches that introduce new concepts to the language (like upgrading queries to first order language entities) shall not be made use of.

The attempt to achieve the greatest possible transparency entails many benefits (like the simplicity of use, lack of a learning curve, and the minimal amount of modification necessary to be applied over already existing code in order for it to make use of the developed functionality) but it also carries with itself several restrictions. These restrictions can make the solution less flexible or with a smaller range of application than solutions that sacrifice more of their transparency. However, as is this case, a compromise solution should be defined.

Before the approaches that are intended to be followed in this work are presented, let us contextualize ourselves so that the decisions presented later on can be understood in a better light. The work that is to be developed is inserted in the context of web applications implemented in Java that make use of software transactional memory (STM). An STM is defined as a concurrency control mechanism that is analogous to database transactions with the objective of controlling access to shared memory in concurrent computing. Whereas the solution is designed with that end in mind, it should be applicable to similar applications without losing too much of its efficiency. It should be noted that due to the operation in transactional context, only the read operations can be actually optimized, since any write operation that may take place during the transaction is only persisted when the transaction commits successfully.

As was referred in previous sections, the main objective of this work is the development of a set of APIs with the use of which it is possible to continue working in a flexible manner with rich domain models while at the same time benefiting from better efficiency while manipulating persistent objects. Speaking in more concrete terms, all the work is going to revolve around the implementation of collections of (persistent) objects that present the same interface as any other collection of similar type (being that a map, set, list or any of their derivates) and, at the same time, allow for the execution of the standard operations they support, over the underlying persistent objects, in a more efficient manner.

In order for the collections to be able to apply the most appropriate optimizations, it is necessary that information is gathered regarding the way that the underlying persistent objects are manipulated while the application is in execution. For this information to be actually useful, it needs to be associated to the context in which a particular situation takes place. Consequently, the next time the application passes through that situation, there is already information about
the most probable course of action it will take, and, as such, the collections are able to pre-fetch and/or load partially the objects to be manipulated.

4.1 Context

The context can be defined in several ways. As such, the method for obtaining it should be chosen so as to provide some desirable properties. One of these properties is to contain enough information so that it may identify a given situation in an as unique as possible manner. It should be able to distinguish between two calls to the same operation of a given collection made in different parts of the application. An example of this can be seen in Fig. 3.

```java
/*code in AClass.java*/
...
List<Employee> emps = Company.getEmployeeList();
Iterator<Employee> iter = emps.iterator(); //context A
while(iter.hasNext()) {
    print(iter.next().getName());
}
...
/*code in BClass.java*/
...
List<Employee> emps = Company.getEmployeeList();
Iterator<Employee> iter = emps.iterator(); //context B
while(iter.hasNext()) {
    print(iter.next().getDepartment().getName());
}
...
```

Fig. 3: Invocations of the same method belonging to different contexts

Another property is the ability to detect similar or identical situations and recognize them as such, grouping them under the same context. This is especially important because if each and every execution of a given operation were to be associated with a separate context, then the usefulness of the context would not be too great. It would make it impossible to actually extract any information regarding the probable behaviour of the application for a given situation. An example of this can be seen in Fig. 4.

```java
...
Employee mng;
Iterator<Employee> iter = Company.getManagerList().iterator();
Map<Employee,Project> proj = Company.getProjects();
while (iter.hasNext()) {
    mng = iter.next();
    if (proj.containsKey(mng)) {//all invocations belong to same context
        print(mng.getName() + " is leading " proj.get(mng));
    }
}
...
```

Fig. 4: Independent method invocations belonging to the same context
The third property is the context to have a compact spatial representation. This can be explained with the fact that there is a potentially unlimited number of operations and manipulations over collections of objects in an application. Consequently, were the context to be demanding in terms of memory requirements, it would influence the general performance of the system in a negative way. The last property is for the method to be computationally light so that it does not degrade the system operation.

Two alternative methods for defining the context are to be reviewed and evaluated against the properties presented in this section. The first alternative is to use part of the invocation stack of the application as the context. The last few method invocations preceding the current position in the execution of the application are considered to define the context. This method both identifies each manipulation/operation invocation over a collection of objects in a unique way and allows the agglomeration of similar situations under the same context. While this technique presents a very high affinity with regards to the first two properties, it should be used with care because it can easily violate the limited spatial requirements property and as such the section of the stack to be taken into account should be the minimum possible that has an acceptable precision.

However, it should be noticed that while this approach is not computationally intensive, its implementation would imply byte-code manipulation of the stack in order to extract the necessary information and this is not a completely trivial matter. From a more practical point of view, this method can be developed with the use of the ASM Java byte-code analysis and manipulation framework.

```java
/*original code*/
List<Employee> emps = Company.getEmployeeList();
Iterator<Employee> iter = emps.iterator();
...  
/*code after applying AOP*/
List<Employee> emps = Company.getEmployeeList();
Iterator<Employee> iter = emps.iterator(contextId);
...

Fig. 5: Result of applying AOP to add context information
```

The second approach is to make use of an aspect oriented programming framework such as AspectJ. With its help, all the method invocations and other manipulations executed over collections with underlying persistent objects would be replaced with overloaded versions of those methods. These overloaded methods would accept as an additional input argument a system-wide unique identifier to which would be associated the context (see an example of this in Fig. 5).

This approach adheres in a weaker way to the first property than the previously presented method. This is due to the fact that while it still identifies in a unique way every distinct invocation of a method over a collection, it can go no further than that. It is unable to capture inter-procedural dependencies, such as one method being called from various different points in an application and, consequently, for each of those cases being inserted in a different context. With
regards to the other properties, in terms of the volume of information that is necessary to be kept, it is minimal, since the context is defined by the identifier that is passed as argument. Regarding its computational weight, there is none, since, once again, the context is explicitly indicated in the method to be invoked.

To sum up, the second approach discussed here may be qualified as being more lightweight than the first one, at the cost of some loss of accuracy. The two approaches are to be implemented and compared in practice with the benchmarks used to evaluate the overall performance of the system.

4.2 Statistical Information

All the objects and relationships existing among them in a program can be modelled through a graph. When the program is in execution, it is traversing its graph of objects. The statistical information that is gathered is to give information about the access patterns that are most common or likely to be executed by the program in any context. If it is considered that the nodes of the graph represent individual objects, then there are two types of accesses that are of interest. One can be classified as a special case of the other. Both of them fall under the category of accessing an objects attributes, but, when an attribute contains a reference to another object, then, in practice, we traverse from one node of the graph to another.

Since the objective is to be able to predict if a given attribute or reference to another object will be needed for the computation at hand, what the statistical information will contain, in concrete terms, is the number of times that the program held an object and the number of times that it accessed each of its fields (including those containing references to other objects). If all field accesses are made through getter and setter methods (as they should be in every well designed application) then the actual data acquisition for the statistics can be implemented with the use of aspect oriented programming. Additional code would be injected in every access method, with the intent of updating the information regarding the use of the field in consideration.

4.3 Mode of Operation

When the system is in operation, whenever there is an operation that involves persistent objects contained in a collection, then the procedure that is presented next is to be followed. Firstly, determine the context in which the operation is to be executed. If there is no context associated (depending on the implementation, as was referred before, the contexts can be determined in a more static, with the use of aspect oriented programming, or a more dynamic way, making use of the stack of invocations present at the time of invocation), create a new context. Once the context is determined, consult the statistical information it contains, regarding the access patterns that are more likely to be followed. In the case when there is no information gathered yet or that it is not enough, then the loading of the persistent objects shall proceed as it would if there were no additional mechanisms for optimizing the persistent data access. The objects
will be loaded lazily as they are needed by the application. If there is enough information so as to make a decision with a certain degree of certainty of it being correct, then all the fields that are deemed as likely to be needed by the program for its execution in the context under consideration are to be pre-fetched and kept in cache so that they are readily available when the application needs them.

However, it should be noted that the pre-fetch does not necessarily include only fields and/or objects that are directly related with the objects to be manipulated by the program but it will be possible to pre-fetch the portion of the graph of objects whose access probability is evaluated to be above an established threshold. Independently of whether there is a pre-fetch to be performed or not, at every object manipulation, the statistical data that is associated to it, in the underlying context, is updated. This should be always performed so that the statistical information is as up-to-date as possible. Consequently, it would allow that the most appropriate measures can be taken to optimize the loading of persistent objects.

Since the statistical information is to be used every time the application executes, it will be persisted in some fashion (using, for instance the Java object serialization mechanism). This prevents the need for statistical information to be created from scratch at each execution and it allows it to grow more complete and comprehensive as time passes.

The behaviour of the application can change over time. This can be either due to external factors, such as the supplied input, or to internal ones, of which the modification of the application code is a common example. Consequently, it is desirable to give greater importance to statistical information generated more recently, when compared to the one collected from the past. The latter may be out-dated and, as such, will not fit the behaviour of the application as precisely as intended. Accounting for that, measures shall be taken so that an indicator of the freshness of the information associated to each statistical item is kept. This would allow for the application to evolve with the passing of time without there being a need to reset all the statistical information collected up to a given moment. Subsequently, old and out-dated items would simply be disregarded.

5 Methodology for Evaluating the Work

The solution to be developed in the course of this work is not an autonomous system. It is to be used by systems and applications that wish to benefit from the properties provided by the solution. The evaluation of the present work will fall upon the execution of several benchmarks, among which are OO7 [11], DayTrader[12], Rubis[13] and TPC-W [14].

The OO7 benchmark examines the performance of object-oriented persistence mechanisms for an idealized CAD (computer aided design) application. DayTrader is a benchmark application built around the paradigm of an online stock trading system. Rubis is an auction site benchmark used to evaluate application design patterns and application servers performance scalability. TPC-W is a transactional web benchmark, where the workload is performed in a con-
trolled internet commerce environment that simulates the activities of a business oriented transactional web server.

The evaluation of the developed solution shall be derived from the comparison between the execution of a version of each of the benchmarks using the Fenix Framework and a version using the Fenix Framework extended with the solution presented here.

6 Conclusions

This work proposes to develop a set of Java API’s that allow an efficient access to persistent information necessary for the processing of tasks by an application and to simplify the implementation of a rich domain model. The solution is designed so that it may be used for applications that do not make use of an STM without losing too much of its efficiency. The read operations are the only operations to be optimized. Nothing is done to increase the efficiency of persistent data write operations. The work is to be a part of the Fenix Framework.

7 Acknowledgments

This work is being performed in the scope of the Pastramy project (PTDC/EIA/72405/2006), which is funded by the Portuguese FCT (Fundação para a Ciência e a Tecnologia).

References