Importance Analysis for Predicting Data Access Behaviour in Object-Oriented Applications

Stoyan Y. Garbatov and João P. Cachopo

Abstract: This work presents an innovative system for analyzing and predicting the behaviour of object-oriented applications, with regards to the data they manipulate. The system is validated by the execution of the oo7 benchmark, which has been modelled as a stochastic process through Monte Carlo simulations. The overheads introduced are in the order of 4%-5%, with regards to the non-instrumented application. The system is sufficiently flexible to be applied to any object-oriented application.

1. Introduction

Most applications need to fetch data from external resources to perform their functions. This is a time consuming operation. Because of this, applications that access large volumes of data must be carefully engineered to have acceptable performance.

The cost of fetching data from an external resource depends on a series of factors. A good design rule is to minimize the number of round-trips made by the application to load the data. On the other hand, it should be also sought out to minimize the amount of data fetched. Consequently, performing a single round-trip that fetches exactly the data that is needed for a given operation would be the optimal solution. A common example of such data-fetching approach is the use of complex SQL queries to retrieve data from multiple tables at once from a relational database. Unfortunately, this idealized goal is seldom possible to achieve and the reasons are manifold.

The current state-of-the-practice is to leave to the programmers the burden of knowing what to fetch and when to do it. However, as applications become more complex and are developed with higher-level languages, knowing beforehand which data is needed for a particular computation, becomes humanly unfeasible.

Moreover, even if the data-fetching patterns for an application are fine-tuned, these adjustments may become useless or even counterproductive when either the requirements for the application or the data structure change, even if slightly.

Additionally, many applications use caches to reduce the need to fetch data from external resources, and, thereby, accelerate the application. Obviously, the use of these caches and their dynamic contents influence the optimal fetching strategy.

The data-fetching strategy for an application should not be a responsibility of the programmers. It should be determined automatically, based on the data-access behaviour observed.

Several related works have been published. Knafla [1] presents a methodology where the main goal is to facilitate the prefetching of persistent objects through the prediction of data access. The relationships between the objects are modelled by a discrete-time Markov Chain. The work offers an interesting view regarding the use of stochastic models for predicting the behaviour of applications. However, only the actually existing relationships between domain objects are taken under consideration with the model. It may be more interesting to consider the effectively observed data accesses, instead of only the static domain object hierarchy.

Bernstein et al. [2] present a technique for data prefetching. 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 object's 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 (prefetching it), effectively reducing the time lost while waiting for the data to be loaded on demand.

The work presented here is a continuation of the one developed in [3],

where the analysis is based on the Bayesian inference model, and is also a part of [4] and [5].

This paper deals with the problem of determining what the behaviour of an application is, with regards to the data accesses that it performs. It describes the design and implementation of a system that predicts the data-access patterns for a Java application. It captures all of the data accesses made by the application during its execution, and uses Importance Analysis model over the collected data to predict future data accesses.

The system was designed to integrate seamlessly with the development of a Java application, and requires minimal effort from the programmer. Additionally, the system is efficient both in the amount of memory that it requires and in the performance overheads that it introduces.

2. System Description

The system is composed of three modules: a code injection module, a data acquisition module, and a data analysis module. The code injection module is responsible for injecting code into the target applications. This code is necessary for the invocation of functionality present in the other two system modules and its injection is performed in a completely automatic fashion by the system to avoid the need for the application programmers themselves to perform any modifications whatsoever to their applications. The second module deals with collecting data about the actual behaviour of the target application, with regards to the data accesses that are performed. Finally, the data analysis module is responsible for applying the Importance Analysis model over the acquired data and, subsequently, for generating the prediction about the most likely behaviour to be observed by the target application, in terms of the data it will access.

2.1. Code Injection

The data-access patterns of an application describe which types of domain objects and which of their fields are accessed during its execution. Consequently, to capture these patterns, it is necessary to identify all the points where data is accessed within the application and instrument them. This allows the subsequent recording of those accesses during the execution of the application.

The system performs this instrumentation at compile-time by means of bytecode rewriting. Consequently, the system performs all the necessary modifications to the target application in an automatic way, without the intervention of the programmer. The modifications are achieved through two instrumentation phases, both of which make use of the Javassist library to inject the code.

The first instrumentation phase injects the code necessary for the identification of the contexts within which all data accesses take place. The context is a key concept in the system. It defines the scope within which all data manipulations take place. For the work presented here, the context corresponds to the method in which the data accesses take place, but it can be defined in a different way, if the situation so demands (for instance, the context can correspond to the execution of a given service or even a sequence of method invocations that precede the current point of execution).

To identify the contexts at runtime, the first instrumentation phase modifies each method of the application. It injects code that updates the context information associated with the method, upon entering it, and code that cleans the same context information when returning from it.

The second instrumentation phase replaces every field access that exists within the application with the invocation of a previously injected static method. There is a distinct method for each of the fields for every class of the application. These methods determine the surrounding context in which the field is accessed and update the associated statistical information. This is done according to the type of access performed: a distinct method is invoked whether the field is read or written.

Once these two phases are complete,

the application can be put into operation, and it will automatically collect the statistics about each data access, without any further modifications.

An important aspect of the solution presented here is the data structure used to store the statistical data. During the instrumentation phases, the system performs an analysis of the structure of the target application via reflection. As a result of this analysis, it generates a set of *PClass* instances to model the structure of each of the target application classes. Each of these *PClass* instances contains a set of *PField* instances, which are used to represent each of the fields that exist in the application class. The information kept in a *PField* covers not only the name and type of a field, but also the number of times that it has been accessed in a context. The subsequent probabilistic analysis uses this information to make its calculations.

The relationship between *PClass* instances and an application class is many-to-one. This can be explained with the fact that the *PClass* instances store information regarding the way that application classes are accessed in the contexts during execution. Consequently, if the same class is used in different contexts, distinct *PClass* instances will keep the information about how that class was used in each.

Taking this into account, it is possible to estimate the upper bounds on the memory requirements for maintaining these structures. The memory will be proportional to the number of classes, times their average number of fields, times the average number of distinct contexts where each class is accessed during the execution. More importantly, note that the memory requirements will not grow continuously, independently of the duration of the execution of the application. In general, this consumption of memory is negligible when compared to the memory needed by the application.

Moreover, given that the probabilistic analysis iterates through the *PField* and *PClass* instances, the time spent in the probabilistic analysis will also take time that is proportional to the previously stated bound.

2.2. Model Implementation

The way how the Importance Analysis model operates is presented here. The data acquisition process, for the model, is decomposed into several steps. While the application is executing, at each field access, the surrounding context is determined. The context has a set of *PClass* instances, which contain the data about what types of accesses have been seen. Once the surrounding context is available, the system performs a lookup over the collection of *PClasses*, resulting in the *PClass* whose field is currently being accessed. After this has been done, the data of how many times the field has been accessed is updated, and the application may proceed with its normal operation.

Based on the Risk Priority Numbers (RPN) technique [6-8] the Importance Analysis model uses three sets of data as input, namely:

- The local probability of a field to be accessed in a given context;
- The global probability of a field being accessed in the application;
- The impact factor, which corresponds to expert judgment of the application developer, indicates how much a given field, is deemed important for the operation of the application.

$$RPN = (L)(G)(I) \quad (1)$$

These three factors are employed as may be seen in Eqn (1), where L is the local access probability, G is the global access probability and I is the impact factor. Once the resulting RPN is calculated, all application fields may be classified and subsequently ordered according to their RPN values. These values reflect the importance of the data, with regards to the effectively observed application behaviour. Based on them, it is possible to make decisions about what are deemed to be the most likely data types to be accessed in a given moment of the application execution and, subsequently, perform optimization techniques that may lead to improving the overall performance. Examples of such techniques are prefetching, and guided

caching policies.

3. Results and Evaluation of the System

The oo7 benchmark has been employed for evaluating the system presented here. This benchmark, firstly presented by Carey et al. [9], is often used to evaluate the performance of object-oriented persistence mechanisms. It strives to present a broad set of operations, allowing for the building of a comprehensive performance profile of the system under evaluation.

The benchmark has been designed to boast properties common to different CAD/CAM/CASE applications, although in its details it does not model any specific application. The evaluation is performed through the execution of a series of traversals, updates, and query operations over the underlying object model. The performance metric used is the time that these operations take to execute.

It should be noted that even though there are some random elements present in the functionality performed by the benchmark operations, these are deterministic in their behaviour. In other words, the complete sets of actions which are executed by the oo7 benchmark are predictable. This property makes it somehow inappropriate for evaluating the precision of a system that seeks to predict the behaviour of its target applications.

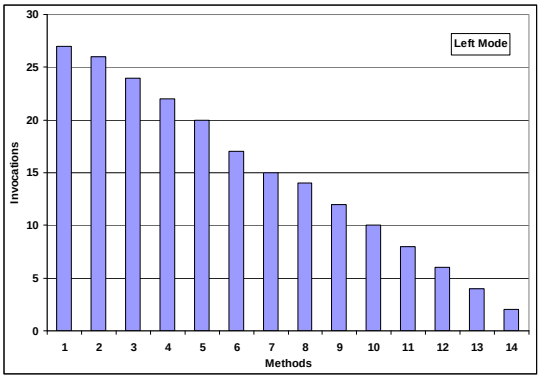

Figure 1 - Method invocations, left mode

With this in consideration, we employed Monte Carlo simulations [10] to make the oo7 benchmark behave like a stochastic process. A stochastic process is one whose behaviour is non-deterministic in that a system's subsequent state is determined both by the process's predictable actions and by a random element. This was done by modelling the number of invocations of the main methods that compose the benchmark. A triangular distribution is used to perform the modelling. This concrete type of distribution is chosen because it is the one most commonly employed when dealing with a system about which there is little or none information about how it behaves. Three distinct triangular distributions are generated to model the benchmark invocations, namely with left (see Figure 1), middle and right mode location.

The results generated by use of the Importance Analysis model can be observed in Figure 2.

The *z*-axis of the histogram corresponds to the number of fields whose RPN values equal the number obtained by multiplying their associated *x* and *y* values. The *x*-axis indicates the local probability access class to which a given field belongs, while the *y*-axis is relative to the global probability access class of the fields.

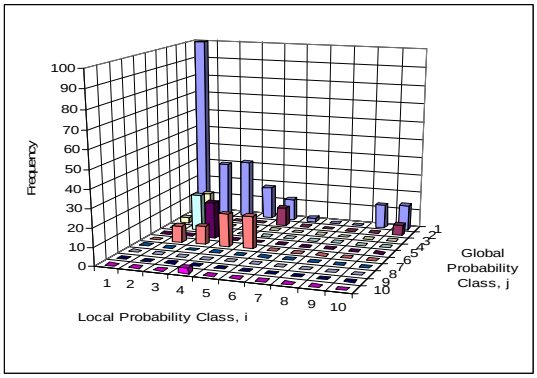

Figure 2 – Importance analysis, left mode

A remark to be made is that most of the application fields belong to the lowest probability classes, while, at the same time, very few of them belong to the higher probability classes. This trend is rather beneficial inasmuch as a smaller data set of likely to be needed data is easier to manage when trying to perform optimizations.

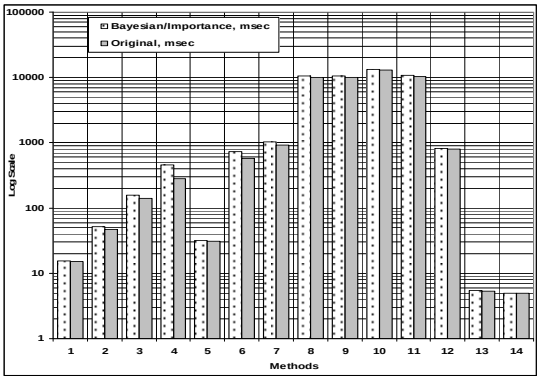

Figure 3 – Method execution times, msec

There is a significant amount of code injected in the application to collect all the information needed to build the models. The need to execute this code causes overheads and penalizes the performance of the application, in comparison with its non-instrumented version. Consequently, it is important to measure those overheads to determine whether they are acceptable in the context of the normal operation of the application. Another aspect to be taken under consideration is the memory space needed for the storage of the statistical data gathered up to a given point in time.

The execution times of the main methods of the OO7 benchmark are summarized in Figure 3 and Figure 4.

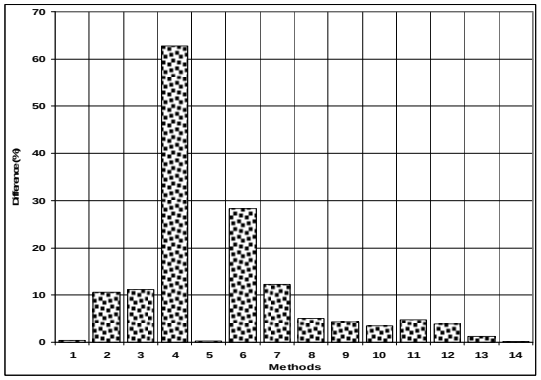

Figure 4 – Difference in execution times, %

In the *x*-axis of the histogram present in Figure 3, we have each of the 14 main methods that define the benchmark. For each of these methods there are two bars: the right one indicates the execution time (in milliseconds) of the method in the original benchmark and the left one is the execution time (in milliseconds) of the instrumented version. The histogram in Figure 4 shows the difference between the bars in percentage. The weighted average, $\overline{overhead}$, is as:

$$\overline{overhead} = \sum_{i=1}^{n} overhead_{method,i} \cdot t_{method,i} \Big/ \sum_{i=1}^{n} t_{method,i} \quad (2)$$

where n is total number of methods, $overhead_{method,i}$ is the overhead, in percentage, associated with method with index i and $t_{method,i}$ is the execution time of method indexed by i. The weighted average of the performance overhead equals 5.15%. As a result of that, the instrumented version is, on average, about 5% slower, in its execution, when compared with the original one. It is deemed that this performance penalty is acceptable.

With regards to the additional memory requirements, due to the storage of the statistical data of observed access patterns, it is not possible to verify any "observable" difference in the memory needed by the original benchmark and the one employing the system developed here. A concrete example would be to say that the statistical data gathered from several executions of the benchmark did not exceed several hundred kilobytes, when saved on the hard disk, while the benchmark process would require a couple of hundred megabytes of memory, when executing. This allows concluding that the memory requirements for the storage of the statistical data necessary for the generation of predictions by the system are negligible.

4. Conclusion

This paper presented a new system for analyzing the behaviour of target applications with regards to the data accesses they perform. The data accesses are analyzed and predicted using advanced probabilistic techniques employing Importance Analysis. Several scenarios were generated and executed, and the subsequent results were analyzed. The system developed for data access behaviour analysis was applied over the oo7 benchmark randomized through the

use of Monte Carlo simulation.

The overheads introduced by the system were shown to be in the order of 5%, when comparing the instrumented application with the original one.

The system developed here is flexible enough to be used as a basis for a set of optimization techniques. Special benefits may be obtained when the data being analyzed is persistent. This would allow for the application of optimizations regarding the way that the information is loaded, stored, or cached.

5. Acknowledgements

This work has been 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).

6. References

1. Knafla, N. Analysing object relationships to predict page access for prefetching. in Eighth Int. Workshop on Persistent Object Systems: Design, Implementation and Use (POS-8), 1998.
2. Bernstein, P., S. Pal, and D. Shutt. Context-based prefetch for implementing objects on relations. in 25th Very Large Data Base Conference (VLDB99), 1999.
3. Garbatov, S., J. Cachopo, and J. Pereira. Data Access Pattern Analysis based on Bayesian Updating. in INForum 2009. Lisbon, 2009.
4. Garbatov, S., J. Cachopo, and J. Pereira, Data Access Pattern Analysis based on Bayesian Updating. 2009, INESC-ID.
5. Garbatov, S., Data Access Patterns Analysis and Prediction for Object-Oriented Applications, in Computer Science. 2009, Technical University of Lisbon, Instituto Superior Técnico: Lisbon.
6. Langford, J.W., Logistics: Principles and Applications: McGraw Hill, 1995.
7. DoD, U.S., Procedures for Performing a Failure Mode Effects and Criticality Analysis. 1984.
8. Kececioglu, D., Reliability Engineering Handbook. Vol. 2. Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1991.
9. Carey, M., D. Dewitt, and J. Naughton. The OO7 benchmark. in ACM SIGMOD International Conference on Management of Data, 1993.
10. Berg, B., Markov Chain Monte Carlo Simulations and Their Statistical Analysis: Hackensack, NJ: World Scientific, 2004.
Knafla, N. Analysing object relationships to predict page access for prefetching. in Eighth Int. Workshop on Persistent Object Systems: Design, Implementation and Use (POS-8), 1998.

Contacts:
Stoyan Garbatov, MSc and João Cachopo, PhD
Instituto de Engenharia de Sistemas e Computadores - Investigação e Desenvolvimento,
R. Alves Redol 9, 1000, Lisboa, Portugal
E-mails: stoyan.garbatov@gmail.com
joao.cachopo@inesc-id.pt