Data Access Pattern Analysis based on Bayesian Updating

Stoyan Garbatov, João Cachopo, João Pereira

Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento,
R. Alves Redol 9, 1000, Lisboa, Portugal
stoyangarbatov@gmail.com, joao.cachopo@inesc-id.pt, joao@inesc-id.pt

Abstract. A new system is presented for analyzing and predicting the most common data access patterns performed during the execution of applications. The correct and precise functioning of the system is demonstrated though the execution of the OO7 benchmark modeled with Monte Carlo simulations. The results obtained show that the precision presented by the system is high. The overheads introduced by the systems are in the order of 4%-5% slower, with respect to the non-instrumented application. The system is flexible enough to be applied to any application where there is a need to identify the access patterns performed.

Keywords: data access pattern, Monte Carlo, Bayesian updating.

1. Introduction

To perform their functions, most applications need to fetch data from external resources (e.g., from files on disk, from a database, or from other applications in a distributed system), which is a costly operation. Because of this high cost in fetching data, applications that access large volumes of data must be carefully engineered to have an acceptable performance.

The cost of fetching data from an external resource depends on a series of factors, but, often, a good design rule is to minimize the number of round-trips made by the application to load the data. On the other hand, we want to minimize the amount of data fetched. Thus, ideally, to perform a given operation, we want to do a single round-trip that fetches exactly the data that we need for the operation. A well-known example of such optimized data-fetching approaches is the use of complex SQL queries to retrieve data from multiple tables at once from a relational database.

Unfortunately, this ideal goal is seldom possible to achieve and is becoming harder to come close to. 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 fetch it. However, as applications become more complex and are developed with higher-level languages (such as object-oriented programming 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.

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

Therefore, we argue that the data-fetching strategy for an application should not be a responsibility of the programmers. Rather, it should be determined automatically, based on the data-access patterns that are dynamically observed for the application.

In this paper, we concentrate on the problem of determining what are the data access patterns of an application. We describe the design and implementation of a system that predicts the data-access patterns for a Java application. To do that, the system captures all of the data-accesses made by the application during its execution, and uses statistical analysis over the captured data to predict future data-accesses.

Our system was designed to integrate seamlessly with the development of a Java application, and requires minimal effort from the programmer. Moreover, our system is efficient both in the amount of memory that it requires and in the performance overheads that it introduces, so that it may be used on production environments.

In the following Section, we describe in greater detail the kind of applications that our system was designed to work with. Then, in Section 3, we describe the design and implementation of our system, both in what regards the capturing of data-accesses and their analysis. In Section 4, we evaluate how our system performs on a well-known benchmark. In Section 5, we discuss related work. Finally, in Section 6, we present the major contributions and conclusions of this work.

2. Characterizing applications and data-access patterns

Applications may access data and be developed in many different ways. Thus, even though the techniques that we describe in this paper may have a broader applicability, for the purposes of this presentation, we concentrate only on a certain class of applications: object-oriented applications that access data through a domain model. More specifically, we assume that applications compile to the Java Virtual Machine, and that they represent the externally accessible data via a set of classes and their corresponding fields. We chose to concentrate on this set of applications because it includes a significant portion of the modern enterprise applications.

So, given such an application, what we need to understand now, is what its data-access patterns are. Informally, the data-access patterns of an application tell us which data is more often used by the application for each of its operations. Yet, this informal definition may range from very fine-grained information (such as knowing that when executing an operation O with arguments A, the application needs to read the slot S of instance I of a class C), to more coarse-grained information (such as knowing that when executing the operation O, regardless of its arguments, the application needs to read some data from class C with a given probability and some data from class C2 with another probability), with a lot of variations in between.

Choosing one solution in this space represents a tradeoff. Using fine-grained

information has the potential of being more accurate for a given situation that was seen before, but on the other hand it may be more difficult to predict what to do in situations never seen before (e.g., a call to operation O with arguments B). Moreover, storing and using such fine-grained information will necessarily introduce higher overheads into the application.

The system that we propose uses a more coarse-grained approach, aligned with the other end of the space. A data-access pattern in our system describes which classes and which fields within those classes are accessed, and with which probability, for each possible context of the application. We describe these concepts in greater detail in the following Section.

3. Description of the system

Our system is composed of two modules: the data-acquisition module and the data-analysis module. The data-acquisition module is responsible for collecting the information about the data-access patterns during the execution of an application. The data-analysis module is responsible for applying the Bayesian-updating model to the collected data and, subsequently, for generating predictions about the future access patterns. Each of these modules shall be discussed with greater detail in the following sections.

3.1 Data Acquisition

As we saw, the data-access patterns for an application describe which kind of domain objects and which of their fields are accessed during the execution of the application. So, to capture these patterns, we need to identify all the points where data is accessed within the application and instrument them to register those accesses during the execution of the application.

Our system performs this instrumentation at compile-time via bytecode rewriting. Thus, 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*[1] 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 our system. It corresponds to the sequence of method invocations that precede a given point in the execution of the application and define the surrounding scope for each data-access. The actual sequence of methods to take into account when defining a context depends on the *context depth*. This can be better explained through the following example code:

[1] *Javassist* is a class library for editing bytecodes in Java.

```
public void methodA() {
   //method body
   methodB();
   //method body
}
public void methodB() {
   //method body
   methodC();
   //method body
}
public void methodC(){
   //method body
}
```

Assume that methods *methodB* and *methodC* are invoked only as shown above. The *context* of any field access made in the body of *methodC* will be defined by the sequence <*methodA*, *methodB*, *methodC*>, if the context depth is 3, or by the sequence <*methodB*, *methodC*>, if the context depth is 2. The context depth is a parameter that may be adjusted when is necessary to alter the granularity of the input data that is accumulated while the application is in execution (this is the data that will be used for the probabilistic analysis). The context depth can take values from 1 to any deemed practical.

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

The second instrumentation-phase replaces every field access that exists within the application with the invocation of a previously injected statically typed 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 are the data structures 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 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.

3.2 Data Analysis

Once the necessary information has been gathered, we may make predictions about the application's data-access patterns. These predictions result from a probabilistic analysis based on Bayesian techniques. An initial study of this work has been presented in [1] and due to space limitations, the details regarding the formulae used to obtain the final results are omitted, but they may be found in an extended version of the paper in [2].

Bayesian analysis techniques are used for parameter estimation. They give an estimate of the statistical uncertainty of the estimated parameters (corresponding, in this case, to the likelihood of reading/writing a given field of an application class) and can update them when new information becomes available. This is done using two sets of data. The first set is designed by *prior* and corresponds to the data accumulated up to a certain point in the past. The second set is called *current* and is defined by the data acquired from that point in the past, up to the current moment. Based on these two sets, a *posterior* set is calculated. Once all these three sets are available, the prediction for the future access patterns can be performed.

The first phase of the analysis is to generate the *prior* and *current* histograms for the data, which can be seen in Fig. 1. The data on which they are based corresponds to the number of times every field was accessed in each of the identified contexts, during the execution of the application. The x axis of the histogram corresponds to the classes existing in the relative weight of each field access, on a global level. There are 10 distinct classes. The value used to identify each class corresponds to the middle of the class. The y axis corresponds to the number (frequency) of fields whose observed probability of being accessed belongs to a given class.

It is important to point out that it is assumed that the distribution functions that originate from the *prior* and *posterior* histograms share the same type (e.g. normal distribution), a so-called *conjugated prior*. The statistical descriptors[2] for the *prior*, *current*, and *posterior* are determined. In the literature, a number of prior, posterior, and predictive distribution functions can be found, see [3-5].

The continuous normal probability density function is presented by a histogram having the same statistical descriptors. The predicted histogram is adjusted to the statistical descriptors of the predicted normal probability density function by applying a special numerical procedure to make the mean and the standard deviation of the histogram equal to the continuous distribution.

[2] The statistical descriptors correspond to the mean value and variance.

Based on the *prior*, *current*, and *posterior* statistical descriptors, the *predicted* histogram is generated. There is an optimization process involved to calculate the precise *predicted* distribution that adapts with the minimum error with respect to the expected statistical descriptors that were calculated by the Bayesian approach.

4. Experimental results and Evaluation of the System

We used the OO7 benchmark [6] to evaluate our system. We chose this benchmark because it presents itself as a data intensive application that performs complex manipulations. The OO7 benchmark was built to evaluate the performance of object-oriented persistence mechanisms for an idealized CAD (computer aided design) application. The benchmark performs various types of traversals and read/write operations (grouped into 14 main methods) over its domain objects, which are organized into an elaborate hierarchy.

The benchmark is completely deterministic and to effectively validate the new probabilistic method that we propose in this work, a stochastic[3] application should be used. Therefore, we modeled the number of invocations of the main methods of the benchmark through Monte Carlo simulations. By doing that, the benchmark is converted to a stochastic model.

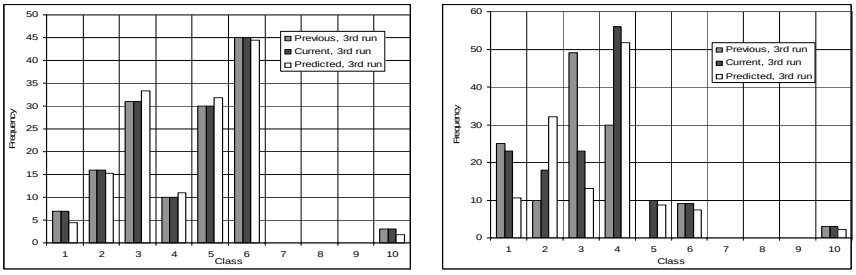

Fig. 1. Result histograms, left and right mode location input

We used uniform random numbers to generate the Monte Carlo simulations and the obtained output is a triangular distribution that represents the uncertainty in the calls of different methods.

Monte Carlo simulation is categorized as a sampling method because the inputs are randomly generated from probability distributions to simulate the process of sampling from an actual population. So, a distribution is chosen for the inputs, that most closely matches the current state of knowledge. The data generated from the simulation can be represented as probability distributions (or histograms).

The triangular distribution is typically used as a subjective description of a population for which there is only limited sample data. It is based on knowledge of the minimum and maximum and an inspired guess as to what the modal value might

[3] A stochastic process is one whose behavior is non-deterministic in that a system's subsequent state is determined both by the process's predictable actions and by a random element.

be. Despite being a simplistic description of a population, it is a very useful distribution for modeling processes where the relationship between variables is known, but data is scarce, because of the high cost of collection.

Three different triangular distributions for the call of the fourteen main methods of the benchmark were generated, based on Monte Carlo simulations. They are left, right, and middle mode location.

Random number generation for a triangular distribution is performed by transforming a continual uniform variable into the range 0 to 1 with the distribution's inverse probability function.

As described in the previous section, the Bayesian updating technique uses two sources of data to make a prediction. One of them is the *previous*, and the second one is the *current*. The *current* data is used to "update" the *previous* collected information, by making it reflect the patterns that have been most recently observed. By doing this, the *predicted* histogram is obtained, as can be seen in Fig. 1. The x axes of the histograms represent the index of the classes within which a field belongs. The index *i* is defined by $i = floor\left[(p + 0.1)/0.1\right]$ where $p \in [0,1]$. It should be noticed that, whereas the bars in the *previous* and *current* histograms indicate the number of fields whose observed probability of being read/written belongs to a given class, the *predicted* histogram reflects the fluctuation in the relative importance of each of the classes, which is expected to be seen in the following executions of the application, see textbook on Bayesian statistics, e.g. Box & Tiao [7] and Lindley [8].

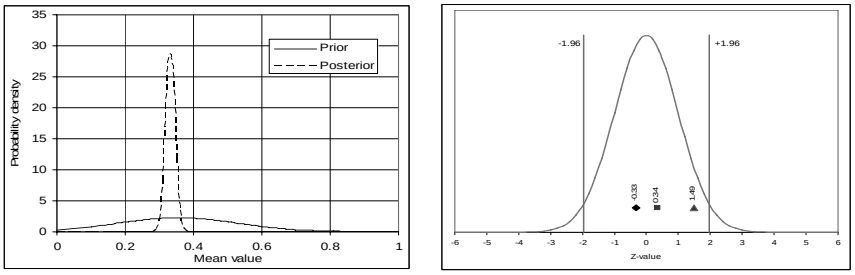

Fig. 2. *Prior* and *posterior* probability function (mid mode location) and respective predicted probability density function for the expected mean value (left), Z values test (right)

Based on the Bayesian method, both the physical uncertainty related to the considered variable as well as the statistical uncertainty related to the model parameters can be quantified. An important property of the Bayesian analysis is the fact that the uncertainty of the prediction is reduced. This can be seen in Fig. 2, left.

To evaluate the precision of the results generated by the system, we need to compare the predictions generated during the second runs of the benchmark, with the actual patterns observed during its third runs. To check whether the mean of the prediction is significantly different from the one observed in the next execution, we resorted to a null hypothesis test. The Z test statistic for this type of check can be found in [7] as:

$$z = \mu_1 - \mu_2 \Big/ \sqrt{\sigma_1^2 / n_1 + \sigma_2^2 / n_2} \qquad \textbf{(1)}$$

where μ_1 and μ_2 are the mean values, σ_1^2 and σ_2^2 are the variances and n_1 and n_2 are the sample sizes, for the predicted and the observed, respectively.

The critical value for z is defined as ± 1.96, that corresponds to 0.05 level of significance to reject the null hypothesis.

For the three different input modes tested here, the calculated z values are presented in Fig. 2, right. As can be seen, all three z values belong to the area of 95% confidence that the mean values of the predicted and subsequently observed are not different. Consequently, we may conclude that the predictions created by the system are precise enough. They are able to indicate correctly the tendencies that are actually observed in the next executions of the application.

As has been previously referred to, there is a significant amount of code injected in the application. The need to execute this code causes overheads and penalizes the performance of the application, in comparison with its non instrumented version. So, it is important to measure those overheads. The execution times of the main methods of the OO7 benchmark used for the results are summarized in Fig. 3.

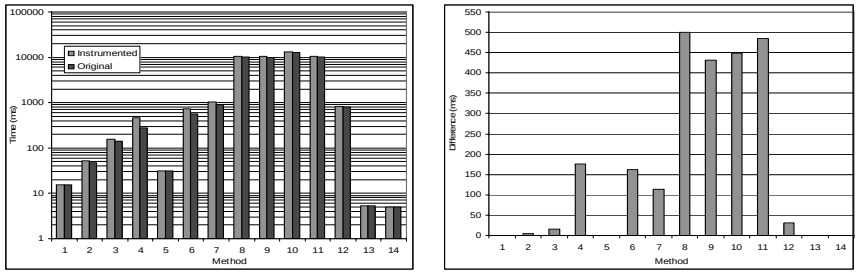

Fig. 3. Execution times (logarithmic scale), left, difference between execution times, right

In the x-axis of the histogram presented in Fig. 3 we have each of the 14 main methods that define the benchmark, and the values shown are in milliseconds. The weighted average of the overhead equals 5.15%. As a result of that, the instrumented version is, on average, about 5% slower, in its execution. It is deemed that the performance penalty is acceptable.

5. Related Work

Recently, many related works have been reported in the literature. Knafla presented in [9] a methodology where the main goal is to facilitate the pre-fetching of persistent objects through the prediction of data access.

The relationships between the objects are modeled by a discrete-time Markov Chain. A discrete-time Markov Chain is a stochastic process through which it is possible to model the behavior 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).

The work above offers an interesting view regarding the use of stochastic models for predicting the behavior of applications. However, the method is stateless and, consequently, cannot make use of any previous information that may be available.

Han et al presented in [10], a new technique for pre-fetching. One of the fundamental ideas behind the work is the fact that the access patterns employed by applications can be modeled in terms of the attribute references made when manipulating objects instead of the pattern of object references.

Bernstein et al. presented in [11] 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 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 (pre-fetching it), effectively reducing the time lost while waiting for the data to be loaded on demand. Other interesting works, with respect to efficient persistent data manipulations, can be found in [12-16].

The work presented here is a part of a project which deals with optimization techniques, such as pre-fetching or caching. In order to achieve this, it is necessary to know the access patterns performed. Most of the works mentioned above, do not apply any specialized analysis techniques. The main contribution of this work resides in the development of a new methodology for identifying access patterns based on high level statistical techniques.

6. Conclusions

In this paper, a new system for analyzing the most common data access patterns performed during the extensive operation of an application was presented. The data accesses are analyzed and predicted using advanced probabilistic techniques employing Bayesian Updating. Several scenarios were generated and executed, and the subsequent results were analyzed. The system developed for data access pattern analysis was applied over the OO7 benchmark adapted to make use of Monte Carlo simulation.

The overheads introduced by the system were shown to be in the order of 4-5%, when comparing the instrumented application with the original one. For the three different input modes tested here, the calculated z values belong to the area of 95% confidence that the mean values of the predicted and subsequently observed are not different. Consequently, we may conclude that the predictions created by the system are precise enough. All the initially identified requirements were satisfied and the objectives were achieved.

The system developed here is flexible enough to be used as a basis for a set of optimization techniques. Special benefits could 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 (e.g. pre-fetching), is stored (e.g. multiple data-bases store the application data) or is cached (caching policies).

Acknowledgments

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).

References

1. Garbatov, S. and Cachopo, J., CoPO: Collections of Persistent Objects, INESC-ID Tec. Rep. 9/2009.
2. Garbatov, S., Cachopo, J. and Pereira, J., Data Access Pattern Analysis based on Bayesian Updating, Tec. Rep. 2009.
3. Raiffa, H. and Schlaifer, R.: Applied Statistical Decision Theory. MIT Press, Cambridge, (1968).
4. Aitchison, J. and Dunsmore, I.R.: Statistical Prediction Analysis. Cambridge University Press, Cambridge, (1975).
5. Rackwitz, R. and Schrupp, K.: Quality Control, Proof Testing and Structural Reliability. Structural Safety, Vol. 2, pp. 239-244, (1985).
6. Carey, M.J., Dewitt, D.J. and Naughton, J.F.: The oo7 benchmark, pp. 12–21, (1993).
7. Box, G. E. P. and Tiao, G. C.: Bayesian Inference in Statistical Analysis, Wiley, New York, (1992).
8. Lindley, D.V.: Introduction to Probability and Statistics from a Bayesian Viewpoint, Vol. 1+2, Cambridge Univ. Press, Cambridge, (1976).
9. Knafla, N.: Analysing object relationships to predict page access for prefetching. In Proc. of the Eighth Int. Workshop on Persistent Object Systems: Design, Implementation and Use (POS-8). pp. 160–170, (1998).
10. Han, W. S., Whang, K.Y. and Moon, Y. S.: Prefetching based on the type-level access pattern in object-relational dbmss. In Proc. of the 25th Very Large Data Base Conference (VLDB99), (1999).
11. Bernstein, P. A., Pal, S. and Shutt, D.: Context-based prefetch for implementing objects on relations. In Proc. of the 25th Very Large Data Base Conference (VLDB99), (1999).
12. Willis, D., Pearce, D. J. and Noble, J.: Efficient object querying for java. In: In Proc. of the European Conference on Object-Oriented Programming (ECOOP), (2006).
13. Willis, D., Pearce, D. J. and Noble, J.: Caching and incrementalisation in the java query language. In Proc. of the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA08), (2008).
14. Ibrahim, A. and Cook, W. R.: Automatic prefetching by traversal profiling in object persistence architectures. In Proc. of the European Conference on Object-Oriented Programming (ECOOP), (2006).
15. Wiedermann, B. and Cook, W. R.: Extracting queries by static analysis of transparent persistence. In: In Proc. of the 34th Symposium on Principles of Programming Languages (POPL07), (2007), 199–210.
16. Wiedermann, B., Ibrahim, A. and Cook, W.R.: Interprocedural query extraction for transparent persistence. In: In Proc. of the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA08), (2008).