Optimising Performance of Object-Oriented and Object-Relational Systems by Dynamic Method Materialisation

Mariusz Masewicz, Robert Wrembel, and Juliusz Jezierski

Poznań University of Technology, Institute of Computing Science
Poznań, Poland
{Mariusz.Masewicz,Robert.Wrembel,Juliusz.Jezierski}
@cs.put.poznan.pl

Abstract. Efficient executions of object methods have a great impact on application response times. Optimising access to data returned by methods is an important issue in object-oriented programs, object-oriented and object-relational systems, as well as in distributed object environments. Since methods are written in high-level programming languages, optimising their executions is difficult. In this paper we present a technique of reducing access time to data returned by methods by means of materialising method results. We have developed a prototype system where the software module, called the *method analyser and optimiser* is responsible for monitoring method access and gathering execution statistics. Based on the statistics, the module selects appropriate methods for materialisation. The experiments that we have conducted show that the overall system's response time decreases while using our optimisation technique.

1 Introduction

Object–oriented (OO) technologies [14] and systems have been developed in order to support storing and processing complex data and ease software development. OO applications usually take advantage of methods as a mean of accessing objects [2]. A method can be a very complex program, that accesses a large number of objects and whose computation may last long. Therefore the efficient execution of a method has a great impact on an application's response time. Optimising method executions is difficult since methods are expressed in an object-oriented language taking advantage of inheritance, overloading and late binding. Moreover, the codes of methods are usually complex using loops, conditional expressions, and calls to other methods.

The systems that profited from OO technologies include among others: Computer Aided Design, Computer Aided Manufacturing, Computer Aided Software Engineering, Geographical Information Systems [13], Computer Supported Co–operative Work, Office Information Systems, various multimedia applications, and distributed computing systems (e.g. CORBA [15]). Since a few years, one can observe a trend towards the integration and analysis of complex data stored in various

systems. To this end, object-relational [5, 6, 7], XML [16], or even multimedia data warehouses [17] are being investigated and developed.

The application of a data warehousing technology to the integration of complex data implies combining object–oriented technology with the technology of data warehousing. In the process of integrating and warehousing complex data, materialised object–oriented views play an important role, e.g. [9, 18]. The materialisation of an OO view includes the materialisation of its structure as well as methods. Moreover, in traditional OO systems, method caching, precomputing, and materialising techniques appeared to be promising, e.g. [1, 8, 10, 20].

A *method materialisation* consists in computing the result of a method once, store it persistently in a database and then use the persistent value when the method is invoked. On the one hand, the materialisation of a method reduces time necessary for accessing a method's result, especially when its execution takes long time. But on the other hand, materialised results of methods have to be kept up to date when data used to compute the results change. Therefore, in order to improve a system's performance, only the right set of methods should be materialised.

The **goal and contribution** of this work is the development of a framework for increasing a system's performance by applying method materialisation. The performance is increased by selectively materialising only the chosen methods, i.e. those whose computation is costly and whose materialised results can be maintained at low costs. The goal is achieved by applying the dynamic method materialisation technique. In this technique, a software module, called *method analyser and optimiser*, is responsible for monitoring access to methods and gathering execution statistics. Based on the statistics, the module automatically selects appropriate methods for materialisation. The statistics are gathered for every method considered for materialisation. When an object used for the materialisation of a method result is updated, the result is invalidated and recomputed. When the number of updates invalidating a materialised result increased beyond a threshold, our system automatically dematerialises a given method. The experimental evaluation of our prototype shows that the overall system's performance increases while using our optimisation technique.

This paper is organised as follows: Section 2 discusses related approaches to method materialisation. Section 3 overviews basic issues on method materialisation. Sections 4 and 5 present the hierarchical method materialisation and the dynamic method materialisation techniques, respectively. Experimental evaluation of our dynamic materialisation technique is discussed in Section 6. Finally, Section 7 summarises the paper.

2 Related Work

Method materialisation (precomputation, caching) was proposed in [7, 1, 10, 11, 4, 19, 20] in the context of indexing techniques and query optimisation. The work of [7] sets up the analytical framework for estimating costs of caching complex objects. Two data representations are considered, i.e. a procedural representation and an object identity based representation. In the analysis, the author has assumed that

procedures are independent on each other, i.e. they do not call each other. Moreover, the maintenance of cached objects was not taken into consideration either.

In the approach of [1], the results of materialised methods are stored in an index structure based on B–tree, called *method–index*. While executing queries that use materialised method *M*, the system searches the method–index for the value of *M* before executing it. If the appropriate entry is found the already precomputed value is used. Otherwise, *M* is executed for an object. The application of method materialisation proposed in [1] is limited to methods that: (1) do not have input arguments, (2) use only atomic type attributes to compute their values, and (3) do not modify values of objects. Otherwise, a method is left as non–materialised.

The concept of [10, 11] uses the so called *Reverse Reference Relation*, which stores an information on: an object used to materialise method *M*, the name of a materialised method, and the set of objects passed to *M* as arguments. Furthermore, this approach maintains also an information about the attributes, called *relevant attributes*, whose values were used for the materialisation of *M*. For the purpose of methods invalidation, every object has appended the set of those method identifiers that used the object. The limitations of this approach are as follows. Firstly, the proposed method materialisation technique does not consider method dependencies. Secondly, a system's designer has to explicitly define in advance data structures for storing the set of method identifiers appended to every object. The defined data structures may never be used when methods are not materialised.

A concept of so called *inverse methods* was discussed in [4]. When an inverse method is used in a query, it is computed once, instead of computing it for each object returned by the query. The result of an inverse method is cached in memory only within a query duration time and it is accessible only by the current query that computed it.

In [20] the authors propose temporarily storing results of method executions in memory in a hash table. In order to increase the usage of cached results, complex functions are decomposed to simple ones, whose results are cached. The approach supports caching methods with constant input values only, i.e. various calls of the same method have to provide the same value of input arguments that strongly limits the application of the approach.

Optimisation of OO queries using methods is discussed in [19]. The work contributes by developing a cost model for method executions. This model includes the number of O/I operations and CPU time, but it does not consider method materialisation.

3 Method Materialisation - Overview

A method whose execution time is long can be materialised, i.e. the result of its execution for a given object and with a given set of input argument values can be stored persistently on a disk. Every subsequent invocation of the same method for the same object and with the same set of input argument values will be handled by reading the already materialised value. A drawback of method materialisation is that values of materialised methods become invalid when objects used for computing

them change. Objects used for computing method values will further be called **base objects**. In a consequence of base object updates, materialised results of methods have to be deleted or set invalid, and then recomputed.

Two important issues arise while working with materialised methods, namely: (1) what technique to use for method materialisation, and (2) which methods to materialise? In our approach, we use the *hierarchical method materialisation technique*. In this technique, not only the method m_i being called is materialised but also other methods called by m_i are materialised.

The second issue concerns selecting right methods for materialisation, as not every method can be materialised and maintained at low costs. First of all, methods having many input arguments with wide domains may not be good ones for materialisation, as materialised results would require huge storage space. Moreover, one would have to implement additional storage structures for efficient searching large sets of materialised results. Second of all, methods whose base objects are being frequently modified should be carefully considered for materialisation. Frequently modified base objects cause that materialised results would have to be invalidated too frequently, deteriorating a system's performance. In our approach we use the *dynamic materialisation technique*. In this technique, a system monitors method usages (method reads and updates of base objects). Based on the gathered statistics, the system decides whether to materialise a given method or not.

4 Hierarchical Method Materialisation

4.1 Concept

In [8, 12] we proposed a novel technique of method materialisation, called the *hierarchical materialisation*. When hierarchical materialisation is applied to method m_i, then not only the result of m_i but also the results of other methods called from m_i are stored persistently. The result of the first invocation of m_i for a given object o_i and with a given set of input argument values is stored persistently. Each subsequent invocation of m_i for the same object o_i and with the same set of input argument values uses the already materialised value. When an object o_i, used to materialise the result of method m_i, is updated or deleted, then m_i has to be recomputed. This recomputation can use unaffected intermediate materialised results, thus reducing the recomputation time overhead.

4.2 Example

In order to illustrate the idea behind the *hierarchical materialisation* let us consider a simplified CAD design of a personal digital assistant (PDA), as shown in Figure 1. A PDA is modelled as the *PDA* class composed of the *MainBoard*, *SoundCard*, and *Display* class. *MainBoard*, in turn, is composed of *CPU*. *SoundCard* is further composed of *Speaker*, whereas *Display* is composed of *HighlightUnit*.

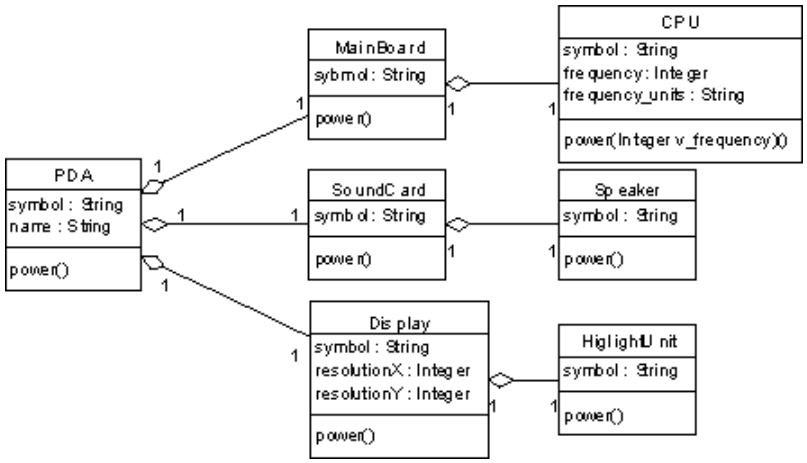

Fig. 1. An example CAD design of a PDA

Each of these classes has method *power()* that computes and returns power consumption of a certain object. A collaboration diagram between the instances of the above classes is shown in Figure 2. The value of *power()* for object *m515* (the instance of *PDA*) is computed as follows: `mb100->power() + sc100->power() + dsp100->power()`. `mb100->power()` is computed as follows: `self->power_cons + cpu33->power(int v_frequency)`, where `power_cons` is an attribute that stores power consumption of a main board itself. `sc100->power()` and `dsp100->power()` are computed similarly as for a main board.

Let us further assume that the *power()* method was invoked for object *m515* and materialised hierarchically. In our example, the hierarchical materialisation results in materialising also *mb100.power()*, *sc100.power()*, and *dsp100.power()*.

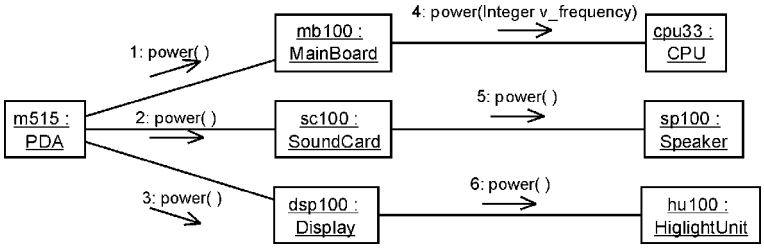

Fig. 2. An example collaboration diagram between class instances of a PDA design

Having materialised the methods, let us assume that the component object *cpu33* was replaced with another central processing unit using 133MHz clock, instead of 33MHz. This change results in higher power consumption of main board *mb100* and of the whole PDA *m515*. In a consequence, the materialised values of *m515.power()* and *mb100.power()* have to be invalidated and recomputed during next invocation. However, during the recomputation of *m515.power()*, the unaffected materialised results of *sc100.power()* and *dsp100.power()* can be used.

4.3 Storage Structures

In order to materialise methods, maintain the materialised results, and use materialised values, we developed five data structures. These structures, which are described below, are called *Materialised Methods Dictionary*, *Materialised Method Results Structure*, *Graph of Method Calls*, *Inverse References Index*, and *Method Value Index*.

Materialised Methods Dictionary (***MMD***) makes available a data dictionary information about all methods, that among others include: a method name and class, the array of input arguments, a method return type, a method implementation, and a flag indicating if a method was materialised.

Materialised Method Results Structure (***MMRS***) stores the following information about every materialised method: (1) the identifier of a method, (2) an object identifier the method was invoked for, (3) the array of input argument values a method was invoked with, (4) the value returned by a method while executed for a given object and for a given array of input argument values.

When materialised method m_i is invoked for a given object o_i and with a given array of input argument values, then ***MMRS*** is searched in order to get the result of m_i. If it is not found then, the value of m_i is computed and stored in ***MMRS***. Otherwise, the materialised result of m_i is fetched from ***MMRS***. When an object used to compute the materialised value of m_i is updated or deleted, then the materialised value becomes invalid and is removed from ***MMRS***.

Dependencies between methods, where one calls another one, is called *Graph of Method Calls* (***GMC***). ***GMC*** is used by the procedure that maintains the materialised results of methods. When materialised method m_j becomes invalid all the materialised methods that use the value of m_j also become invalid. In order to invalidate those methods the content of ***GMC*** is used. ***GMC*** stores pairs of values: the identifier of a calling method and the identifier of a method being called.

In order to invalidate dependent methods the system must be able to find also inverse references in object composition hierarchy. In order to ease the traversal of a composition hierarchy in an inverse direction we use so called *inverse references* for each object. An inverse reference for object o_j is the reference from o_j to other objects that reference o_j. The references are maintained in a data structure called *Inverse References Index* (***IRI***). For example, an inverse reference for object *cpu33* (cf. Figure 2) contains one reference to object *mb100*. At the implementation level, a separated ***IRI*** is created for every, but a root class in a composition hierarchy.

Method Value Index (***MVI***) is an index defined on results of methods. Every method of a class has its own ***MVI***. The index stores the following: (1) the value of a method input argument, (2) a method result, and (3) an object identifier a method was invoked for. By using this index, the system is able to quickly find answers to queries that use methods, e.g. show all PDAs with power consumption lower than 1W. The content of ***MVI*** is filled in with data when methods are materialised.

5 Dynamic Method Materialisation

Finding the right set of methods for materialisation is difficult as one have to take into account for each method m_i its execution/response time as well as the number of reads of a materialised method and the number of invalidations of this method, i.e. the number of updates of its base objects.

The best performance is achieved when there are only reads of materialised method values in a system. The worst performance is achieved when there are in the system only updates of base objects, but methods are rematerialised after every update. In this case, however, there is no sense in materialising methods. Scenarios between the best and the worst cases are the subjects of detailed analysis. In real systems one may expect a mixture of reads and updates addressing a given materialised method. In a consequence, the system will profit from materialisation for read transactions. Whereas, for update transactions a system will spend some time on rematerialising previously invalidated method results.

The goal of this work is to increase a system's performance for transactions reading method values. Materialising only those methods whose computation is costly and whose base objects are not frequently updated increases the performance. The goal achieved by applying the *dynamic method materialisation technique*. In our approach methods are materialised hierarchically, (cf. Section 4) since it was proved to be a promising and efficient technique for some types of methods [8]. However, the *dynamic materialisation* can be applied to any other materialisation technique.

5.1 System Tuning with Dynamic Method Materialisation

The *dynamic method materialisation technique* consists in: (1) gathering method usage statistics and based on the statistics (2) finding methods whose materialisation increases system's performance and methods whose materialisation deteriorates system's performance. A software module, called the *method analyser and optimiser* does the final selection of methods for materialisation. It also monitors method access patterns and gathers execution statistics.

For a given method m_i the *execution statistics* include:

- method execution times and the number of disk accesses for every object and every set of input argument values,
- the number of base object updates,
- the number of reads of m_i materialised values,
- method invalidation times and the number of disk accesses for every object and every set of input argument values,
- method recomputation times and the number of disk accesses for every object and every set of input argument values,
- time and the number of disk accesses required for finding an already materialised value.

Tuning of a system is performed in two following steps.

Step 1

A system administrator selects the set S^M of methods for materialisation. After that, first calls of these methods materialise their results. Having materialised the methods in S^M, the *method analyser and optimiser* is constantly monitoring the usage of the methods and is gathering execution statistics. It is so called *sampling cycle*.

The set of transactions using m_i and its materialised values will further be called the *batch transaction set*. The size of the *batch transaction set* is parameterised by a system administrator. For every method m_i in S^M a sampling period lasts until a given number of transactions in the batch transaction set is reached. After that, the system enters the second step.

Step 2

In this step, the *method analyser and optimiser* identifies methods whose materialisation increases system's performance and methods whose materialisation deteriorates the performance. To this end, the execution statistics are used. Methods from the second category are automatically dematerialised. After that, the module enters the *sampling cycle* (cf. Step 1) for the remained materialised methods.

5.2 Selecting Methods for Materialisation - Cost Model

The decision taken by the *method analyser and optimiser* whether to keep a method as materialised or not is based on the below formula. Let r be the number of transactions reading the materialised value v of method m_i. Let u be the number of transactions updating a base object of m_i. The sum of r and u yield the number of transactions in the *batch transaction set*.

Let t^R_{MAT} represent the time of reading a materialised value of m_i. This value is red from *MMRS* (cf. Section 4.3). Let t_{EXEC} be the execution time of non-materialised method m_i. Let t_{REMAT} be the time of rematerialising value v of m_i, after its base object was updated. All the discussed times include I/O as well as CPU times.

The materialisation of method m_i will reduce query response time if the following holds: the overall time spent on reading materialised value v of m_i by the number r of reading transactions plus the overall time spent on rematerialising the value of m_i by the number of u update transactions is lower than the overall time spent on computing the result of non-materialised m_i for every reading transaction (cf. Formula 1). Δ represents a coefficient by which an overall system's response time is to be reduced. It takes its value from the range of (0, 1) and it is considered as a tuning parameter set up by an administrator.

$$r * t^R_{MAT} + u * t_{REMAT} < r * t_{EXEC} * \Delta \qquad \text{(Formula 1)}$$

In the worst case, i.e. when all branches in the *GMC* have to be invalidated, the rematerialisation time (t_{REMAT}) includes: the invalidation time of a materialised result (t_{INV}), computation of a method result from scratch (t_{EXEC}), and writing the materialised result on disk (t^W_{MAT}). Thus can be expressed as follows:

$$t_{REMAT} = t_{INV} + t_{EXEC} + t^W_{MAT} \qquad \text{(Formula 2)}$$

After replacing t_{REMAT} in Formula 1 by t_{REMAT} from Formula 2, we obtained Formula 3 expressing the number of updates to the number of reads. If for a given method m_i and a given batch transaction set the inequality is true, then it is profitable

to materialise m_i as it will increase system's performance. Otherwise, m_i has to be dematerialised.

$$\frac{r}{u} > \frac{t_{INV} + t_{EXEC} + t^{W}_{MAT}}{\Delta * t_{EXEC} - t^{R}_{MAT}} \qquad \text{(Formula 3)}$$

6 Experimental Evaluation of Dynamic Materialisation

The described *dynamic method materialisation* combined with the *hierarchical materialisation* has been implemented in Java, on top of the *FastObjects t7* (ver. 9.0) object–oriented database system, which was used as a storage system for our data structures and test objects. The prototype and the experiments were run on a PC with the Pentium III (1.13GHz) processor and 128 MB of RAM, under Windows2000. The goal of the experiments was to evaluate our *dynamic materialisation technique*, i.e. to compare an overall system's response time for three cases where: (1) methods were not materialised at all, (2) methods were kept materialised all the time, (3) methods were materialised dynamically.

The graph of method calls (cf. Section 4.3) was composed of 4 levels, each of which having 10 sublevels, as shown in Figure 3. Method m_1 was calling 10 methods $m_{1\text{-}1}$ to $m_{1\text{-}10}$ for objects $o_{1\text{-}1}$ to $o_{1\text{-}10}$, respectively. Every method from the lower level was calling 10 other methods. The size of base objects was constant and equalled to 100 kB. The total size of our test database equalled to 1.1GB.

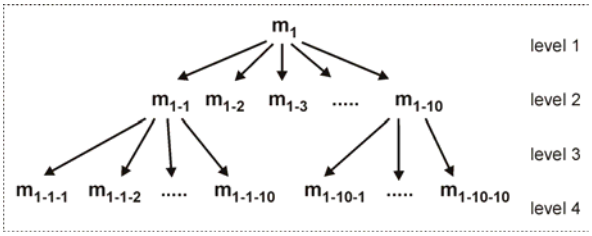

Fig. 3. A test *GMC*

The carried out experiments present overall system's response time for a given batch transaction set of 1000 transactions. The number of read (r) and update (u) transactions in the set was parameterised from 0 to 1000, so that r+u remained constant and equalled to 1000. The execution time of every method was also parameterised and ranged from 10 ms to 100000 ms. However, the results presented in this paper (due to space limitations) are for methods whose execution times equalled to 100 ms. The obtained overall system's response times were measured for root method m_1, cf. Figure 3. Every update transaction modified based objects at the lowest level that caused the invalidation of one branch.

Figure 4 presents overall system's response times of method m_1 in a function of the number of reading transactions in the test batch transaction set.

- *t: NM* - represents the execution time of a non materialised method;

- *t: 1 branch NM* - represents an execution time when a frequently invalidated branch was never materialised;
- *t: MAT select* - represents an execution time when previously invalidated branch was rematerialised just before reading the value of m_1;
- *t: dynamic* - represents an execution time when the dynamic materialisation was applied;
- *t: error cost* - represents the estimated cost of wrong decision made by our *method analyser and optimiser*. This cost is expressed by additional time spent on computing the value of m_1 if its branch was not materialised although it was supposed to be materialised.

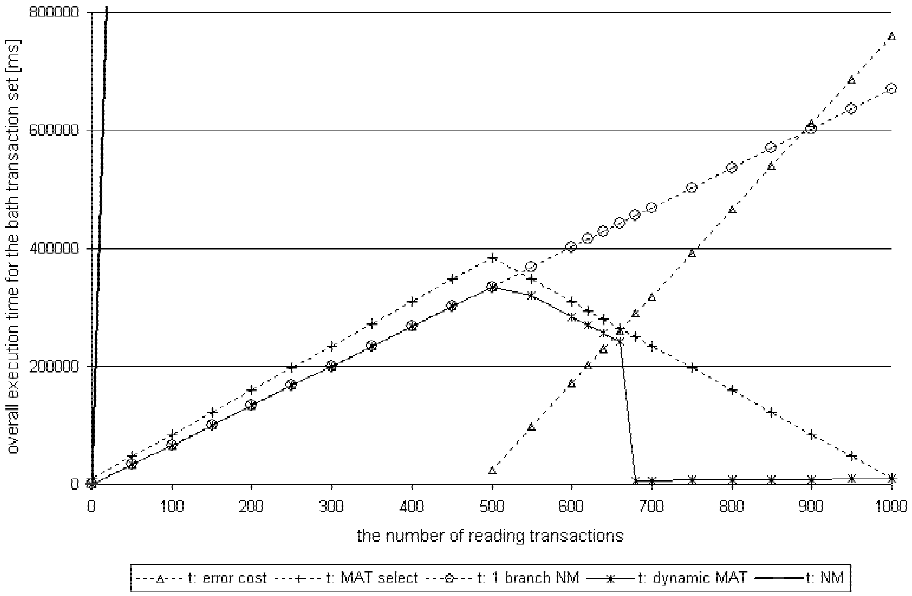

Fig. 4. Overall system's response time for method execution times equalled to 100 ms

As we can observe from the chart, the system with methods being hierarchically materialised, cf. the "*t: MAT select*" time, is much more efficient than without materialised methods, cf. the "*t: NM*" time. Additional efficiency improvement is achieved by the *dynamic materialisation*, i.e. an overall response time of a system with dynamically materialised methods is lower (cf. "*t: dynamic*") than an overall system's response time with materialisation (cf. "*t: MAT select*").

Within the range of reads between 500 and 680 (i.e. 320 to 500 updates) the system may leave a given method as materialised or dematerialise it. The cost of taking a wrong decision is represented by "*t: error cost*". When the number of reading transactions exceed 680 the *method analyser and optimiser* decides to keep method m_i materialised and rematerialises it just after its base object update. Thus, from a user's point of view, the value of m_i is available immediately when required.

In our prototype system we have experimented also with methods having 10 ms, 100 ms, and 100000 ms execution times. Similar performance improvement was achieved as for the cases presented in Figure 4.

Although in the presented experiments one branch of *GMC* was invalidated, our hierarchical materialisation improves system's performance also when more branches have to be invalidated, cf. [8].

7 Conclusions

In this paper we discussed our approach to optimising access to data returned by methods. To this end, we proposed applying the hierarchical materialisation technique as it appeared to be promising for accessing data returned by methods [8]. The process of selecting the right methods for materialisation is based on the *dynamic materialisation*. To the best of our knowledge, it is the first approach to automatic optimisation of method executions by using materialised methods. The *dynamic materialisation* can be applicable to every system that uses methods and to other materialisation and caching techniques, as discussed in Section 2.

As our experiments show, by using the *dynamic materialisation*, the overall system's performance can be substantially improved. Moreover, the lower number of update transactions the shortest system's response time for a given dynamically materialised method m_i.

In the current implementation of the prototype system, the body of method m_i being materialised may not contain OQL commands as it would cause difficulties in registering in *MMRS* the used object identifiers and values of the method. Future work will focus on removing this limitation.

References

1. Bertino, E.: Method precomputation in object–oriented databases. SIGOS Bulletin, 12 (2, 3) (1991)
2. Cattell, R., Barry, D., Berler, M., Eastman, J., Jordan, D., Russel, C., Shadow, O., Stanienda, T., Velez, F.: Object Database Standard: ODMG 3.0, Morgan Kaufmann Publishers (2000)
3. Czejdo, B., Eder, J., Morzy, T., Wrembel, R.: Design of a Data Warehouse over Object–Oriented and Dynamically Evolving Data Sources. Proc. of the DEXA'01 Workshop Parallel and Distributed Databases, Munich, Germany (2001)
4. Eder, J., Frank, H., Liebhart, W.: Optimization of Object–Oriented Queries by Inverse Methods. Proc. of East/West Database Workshop, Austria (1994)
5. Gopalkrishnan, V., Li, Q., Karlapalem, K.: Efficient Query Processing with Associated Horizontal Class Partitioning in an Object Relational Data Warehousing Environment. In Proc. of DMDW'2000, Sweden (2000)
6. Huynh, T.N., Mangisengi, O., Tjoa, A.M.: Metadata for Object–Relational Data Warehouse. In Proc. of DMDW'2000, Sweden (2000)
7. Jhingran, A.: Precomputation in a Complex Object Environment. Proc of the IEEE Data Engineering Conference, Japan(1991)
8. Jezierski, J., Masewicz, M., Wrembel, R.: Prototype System for Method Materialisation and Maintenance in Object-Oriented Databases. Proc. of the ACM Symposium on Applied Computing (SAC), Nicosia, Cyprus (2004)

9. Dobrovnik, M., Eder, J.: Logical data independence and modularity through views in OODBMS. Proc. of Engineering Systems Design and Analysis Conference, Vol. 2 (1996)
10. Kemper, A., Kilger, C., Moerkotte, G.: Function Materialization in Object Bases. Proc. of the SIGMOD Conference (1991)
11. Kemper, A., Kilger, C., Moerkotte, G.: Function Materialization in Object Bases: Design, Realization, and Evaluation. IEEE Transactions on Knowledge and Data Engineering, Vol. 6, No. 4 (1994)
12. Morzy, T., Wrembel, R., Koszlajda, T.: Hierarchical materialisation of method results in object–oriented views. Proc. of the ADBIS–DASFAA 2000 Conference, Czech Republic, LNCS 1874 (2000)
13. Loomis, M.E., Chaudhri, A.B.: Object Databases in Practice. Prentice Hall PTR (1998)
14. Loomis, M.E.: Object Database the Essentials. Addison–Wesley Publishing Company (1995)
15. Object Management Group. The Common Object Request Broker: Architecture and Specification (1995)
16. Yin, X., Pedersen, T.: Evaluating XML-Extended OLAP Queries Based on a Physical Algebra. Proc. of the DOLAP 2004 Conference, Washington, USA (2005)
17. Ben Messaoud, R., Boussaid, O., Rabaséda, S.: A New OLAP Aggregation Based on the AHC Technique. Proc. of the DOLAP 2004 Conference, Washington, USA (2005)
18. Motschnig-Pitrik, R.: Requirements And Comparison of View Mechanisms for Object-Oriented Databases. Information Systems, Vol. 21, No. 3 (1996)
19. Gardarin, G., Sha, F., Tang, Z. H.: Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System. Proc. of the 22nd VLDB Conference, India (1996)
20. Pugh, W., Teitelbaum, T.: Incremental Computation via Function Caching. Proc. of the Annual Symposium on Principles of Programming Languages, Texas, USA (1989)