Automatic Prefetching by Traversal Profiling in Object Persistence Architectures

Ali Ibrahim & William R. Cook

Department of Computer Sciences, University of Texas at Austin
{aibrahim,wcook}@cs.utexas.edu

Abstract. Object persistence architectures support transparent access to persistent objects. For efficiency, many of these architectures support queries that can *prefetch* associated objects as part of the query result. While specifying prefetch manually in a query can significantly improve performance, correct prefetch specifications are difficult to determine and maintain, especially in modular programs. Incorrect prefetching is difficult to detect, because prefetch is only an optimization hint. This paper presents AutoFetch, a technique for automatically generating prefetch specifications using traversal profiling in object persistence architectures. AutoFetch generates prefetch specifications based on previous executions of similar queries. In contrast to previous work, AutoFetch can fetch arbitrary traversal patterns and can execute the optimal number of queries. AutoFetch has been implemented as an extension of Hibernate. We demonstrate that AutoFetch improves performance of traversals in the OO7 benchmark and can automatically predict prefetches that are equivalent to hand-coded queries, while supporting more modular program designs.

1 Introduction

Object persistence architectures allow programs to create, access, and modify *persistent objects*, whose lifetime extends beyond the execution of a single program. Examples of object persistence architectures include object-relational mapping tools [10, 6, 28, 24], object-oriented databases [8, 21], and orthogonally persistent programming languages [25, 2, 19, 22].

For example, the Java program in Figure 1 uses Hibernate to print the names of employees, their managers, and the projects they work on. This code is typical of industrial object-persistence models: a string representing a query is passed to the database for execution, and a set of objects is returned. This query returns a collection of employee objects whose first name is “John”. The fetch keyword indicates that related objects should be loaded along with the main result objects. In this query, both the manager and multiple projects are prefetched for each employee.

[1] This work was supported by the National Science Foundation under Grant No. 0448128.

```
1  String query = "from Employee e
2    left join fetch e.manager left join fetch e.projects
3    where e.firstName = 'John' order by e.lastName";
4  Query q = sess.createQuery(query);
5  for (Employee emp : q.list()) {
6    print(emp.getName() + ": " + emp.getManager().getName());
7    for (Project proj : emp.getProjects()) {
8      printProject(prog);
9    }
10 }
```

Fig. 1. Java code using `fetch` in a Hibernate query

While specifying prefetch manually in a query can significantly improve performance, correct prefetch specifications are difficult to write and maintain manually. The prefetch definitions (line 2) in the query must correspond exactly to the code that uses the results of the query (lines 6 through 8).

It can be difficult to determine exactly what related objects should be prefetched. Doing so requires knowing all the operations that will be performed on the results of a query. Modularity can interfere with this analysis. For example, the code in Figure 1 calls a printProject method which can cause additional navigations from the project object. It may not be possible to statically determine which related objects are needed. This can happen if class factories are used to create operation objects with unknown behavior, or if classes are loaded dynamically.

As a program evolves, the code that uses the results of a query may be changed to include additional navigations, or remove navigations. As a result, the query must be modified to prefetch the objects required by the modified program. This significantly complicates evolution and maintenance of the system. If a common query is reused in multiple contexts, it may need to be copied in order to specify different prefetch behaviors in each case.

Since the prefetch annotations only affect performance, it is difficult to test or validate that they are correct – the program will compute the same results either way, although performance may differ significantly.

In this paper we present and evaluate AutoFetch, which uses traversal profiling to automate prefetch in object persistence architectures. AutoFetch records which associations are traversed when operating on the results of a query. This information is aggregated to create a statistical profile of application behavior. The statistics are used to automatically prefetch objects in future queries.

In contrast, previous work focused on profiling application behavior in the context of a single query. While this allowed systems such as PrefetchGuide [13] to prefetch objects on the initial execution of query, AutoFetch has several advantages. AutoFetch can prefetch arbitrary traversal patterns in addition to recursive and iterative patterns. AutoFetch can also execute fewer queries once patterns across queries are detected. AutoFetch's disadvantage of not optimizing initial query executions can be eliminated by combining AutoFetch with previous work.

When applied to an unoptimized version of the Torpedo benchmark, AutoFetch performs as well as a hand-tuned version. For the OO7 benchmark, AutoFetch eliminates up to 99.8% of queries and improves performance by up to 99.7%. We also examined the software engineering benefits of AutoFetch, by showing that a modular version of a web-based resume application using AutoFetch performs as well as a less-modular, hand-optimized version.

2 Background

The object persistence architectures examined in this paper combine elements of *orthogonal persistence* [1] with the pragmatic approach of relational data access libraries, also known as *call level interfaces* [30].

Orthogonal persistence states that persistence behavior is independent of (orthogonal to) all other aspects of a system. In particular, any object can be persistent, whether an object is persistent does not affect its other behaviors, and an object is persistent if it is reachable from a persistent root. Orthogonal persistence has been implemented, to a degree, in a variety of programming languages [25, 2, 19, 22].

A key characteristic of orthogonal persistence is that objects are loaded when needed. Using a reference in an object is called *traversing* the reference, or *navigating* between objects – such that the target object is loaded if necessary. We use the term *navigational query* to refer to queries that are generated implicitly as a result of navigation.

Relational data access libraries are a pragmatic approach to persistence: they allow execution of arbitrary SQL queries, and the queries can return any combination of data that can be selected, projected, or joined via SQL. Examples include ODBC [15] and JDBC [12]. The client application determines how the results of a query are used – each row of the query result may be used as is, or it may be mapped to objects. Since data is never loaded automatically, the programmer must specify in a query all data required for an operation – the concept of prefetching data that might be loaded automatically does not apply.

The object persistence architectures considered in this paper are hybrids of orthogonal persistence and data access libraries. Examples include EJB [24], JDO [28], Hibernate [6], and Toplink [10]. They support automatic loading of objects as needed. But they also include query languages and the ability to manually prefetch related objects. While query languages can significantly increase performance, they reduce orthogonality because they are special operations that only apply to persistent data.

For example, in EJB 2.1, a query can return objects or a value:

```
select object(p) from Person p where p.firstName="John"
```

The set of objects loaded by a query are called *root* objects. Use of the root objects may result in navigation to related objects, each of which will require an additional query to load.

In client-server architectures, the cost of executing a query, which involves a round-trip to a database, typically dominates other performance measures. This is because the latency cost of communicating with the database is significantly greater than the cost of processing the query or producing results [4]. Other factors, like number of joins or subqueries, or the number of columns returned form a query, are insignificant compared to latency. The relative impact of latency on system performance is likely to increase, given that improvements in latency lag improvements in bandwidth and processing power [27]. As a result, number of queries will increasingly dominate all other concerns. In effect, overall response time is directly related to the number of queries executed in a task.

Object persistence architectures have developed a variety of mechanisms for avoiding navigational queries, by allowing programmers to manually specify prefetch of related objects. Prefetch of related objects is especially important in addressing the $n + 1$ select problem in which a related object is accessed for each result of a query. Without prefetch, if there are n results for a query, then there will be $n + 1$ loads. Most JDO vendors extended the standard to allow prefetch to be specified at the class level. Hibernate, and now EJB 3.0, allow prefetch to be specified within each query using the `fetch` keyword. Using `fetch`, a query can specify which related objects to load along with the root objects. These related objects can be either single objects or collections of related objects, depending on whether the association is single- or multi-valued. For example, this EJB 3.0 query returns a collection of persons where the children have been fetched as well:

```
select distinct p from Person p left join fetch p.children
where p.firstName=John
```

Previous versions of Hibernate only allowed one collection prefetch, however, Hibernate 3.1 allows multiple collections prefetches. Hibernate executes a query with a prefetch by augmenting the query with an appropriate join. This strategy causes the data for the container object to be replicated when a collection association is fetched. For a nested collection, the root container is replicated once for each combination of subcollection and sub-subcollection items. Thus replication is multiplied with each level of subcollection. Independent fetch collections are especially expensive because they cause the result set to include the cross-product of independent collection hierarchy elements. If Hibernate used a different query strategy that allowed for multiple SQL queries to be executed, while correlating the results in the client, then this problem could be eliminated.

Safe Query Objects are a type-safe alternative to string-based query interfaces [7]. Safe queries use methods in standard object-oriented languages to specify query criteria and sorting, so that a query is simply a class. Unlike string-based query languages, there is no natural place to specify prefetch in a Safe Query. Thus Safe Queries would benefit significantly from automatic prefetching.

3 Automating Prefetch

In this section we present AutoFetch, a solution to the problem of manual prefetch in object persistence architectures. Instead of the programmer manually specifying prefetches, AutoFetch adds prefetch specifications automatically. By profiling traversals on query results, AutoFetch determines the prefetches that can help reduce the number of navigational queries, i.e. queries executed as a program traverses an association.

To formalize this approach, we define type and object graphs as an abstract representation of persistent data. A type graph represents the class model, or structure of the database. Object graphs represent data. A complete database is represented as an object graph. Queries are functions whose range is a set of subgraphs of the database object graph.

Traversals represent the graph of objects and associations that are actually used in processing each result of a query. These traversals are aggregated into *traversal profiles*, which maintain statistics on the likelihood of traversing specific associations. Queries are classified into *query classes* based on a heuristic that groups queries that are likely to have similar traversals.

For each query executed, AutoFetch computes a prefetch specification based on the traversal profile for the query class. The prefetch specification is incorporated into the query and executed by the underlying database.

3.1 Profiling Traversals

In this section we develop a model for profiling the traversals performed by an object-oriented application. The concept of *profiling* is well known [3, 11]; it involves collecting statistics about the behavior of a program. Profiling is typically used to track control flow in an application – to find hot spots or compute code coverage. In this paper, profiling is used to track data access patterns – to identify what subset of a database is needed to perform a given operation.

We develop a formal model for types, objects, queries, and traversals. The type and object models are derived from work on adaptive programming [18].

Type Graph: Let T be the finite set of type names and F be the finite set of field names. A type graph is a directed graph $G_T = (T, A)$.

- T is a set of types.
- A is a partial function $T \times F \overset{?}{\rightarrow} T \times \{\mathit{single}, \mathit{collection}\}$ representing a set of associations between types. Given types t and t' and field f, if $A(t, f) = (t', m)$ then there is an association from t to t' with name f and cardinality m, where m indicates whether the association is a single- or multi-valued association.

Inheritance is not modeled in our type graph because it is orthogonal to prefetch. Bi-directional associations are supported through two uni-directional associations. Figure 2 shows a sample type graph. There are three types: Employee, Department, and Company. Each company has a set of departments and a CEO, each department has a set of employees, and each employee may have a supervisor. The formal representation is:

T = {Department, Employee, Company}
F = {employees, departments, CEO, supervisor}
A(Department, employees) $\mapsto$ (Employee, collection)
A(Company, departments) $\mapsto$ (Department, collection)
A(Company, CEO) $\mapsto$ (Employee, single)
A(Employee, supervisor) $\mapsto$ (Employee, single)

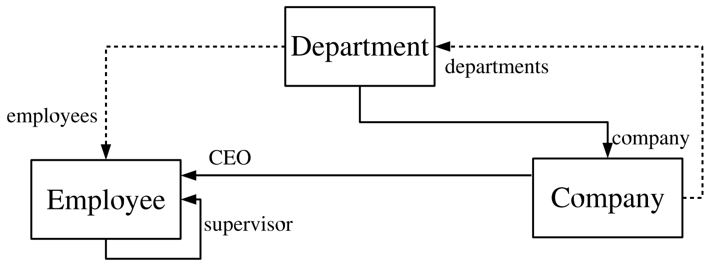

Fig. 2. Simple Type Graph with three types: Employee, Department, and Company. Solid lines represent single associations, while dashed lines represent collection associations.

Object Graph: Let O be the finite set of object names. An object graph is a directed graph $G_O = (O, E, G_T = (T, A), \mathit{Type})$. G_T is a type graph and *Type* is a unary function that maps objects to types. The following constraints must be satisfied in the object graph G_O:

– O represents a set of objects.
– $\mathit{Type} : O \rightarrow T$. The type of each object in the object graph must exist in the type graph.
– $E : O \times F \overset{?}{\rightarrow} \mathit{powerset}(O)$, the edges in the graph are a partial function from an object and field to a set of target objects.
– $\forall o, f$: $E(o, f) = S$
  • $A(\mathit{Type}(o), f) = (T', m)$
  • $\forall o' \in S, \mathit{Type}(o') = T'$.
  • if $m = \mathit{single}$, then $|S| = 1$.

  Each edge in the object graph corresponds to an edge in the type graph, single associations have exactly one target object.

An example object graph is shown in Figure 3 which is based on the type graph in Figure 2. Edges that contain a dark oval represent collection associations. Null-valued single associations are not represented by edges in the object graph, however, empty collection associations are represented as edges whose target is an empty set. We chose this representation because most object persistence architectures represent associations as a reference to a single target object or collection. A null-valued association is usually represented as a special reference in the source object. This means that the persistence architecture can tell if a single-valued association is null without querying the database. On the other hand, the persistence architecture will query the database if a collection association reference is empty, because the collection reference does not have any information on the cardinality of the collection. The ability to represent traversals to empty collections is important when we discuss traversals in Section 3.1, because it allows AutoFetch to represent navigational queries that load empty collections.

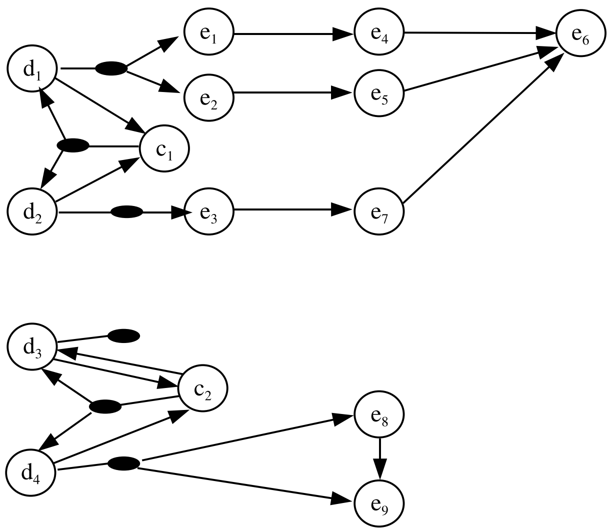

Fig. 3. An example of an object graph based on the type graph in Figure 2. Collection associations contain an oval in the middle of the edge.

Queries A query is a function that returns a subgraph of the database object graph. The subgraph consists of a set of connected object graphs each of which has a distinguished *root* object. The definition of every query includes an *extent type* and *criteria*. The extent type is the type of all the root objects. The criteria are the conditions that an object satisfies to be returned as a root object.

Our approach to prefetching is independent of a particular query language, however, the query language must support an object-oriented view of persistent data, and the underlying persistence data store must allow prefetching associations of the extent type.

Queries are executed by the program to return their results. However, queries are first-class values, because they can be dynamically constructed or passed or returned from procedures. A single program point could execute different queries, depending on the program flow.

Traversals A traversal captures how the program navigates the object graphs that the query returns. A program may traverse all the objects and associations in the result of the query, or it may traverse more or less. Only program navigations that would result in a database load for the query without prefetch are included in the traversal.

A traversal is represented as a forest where each tree's root is a root object in the result of a query and each tree is a subgraph of the entire object graph. Let R denote a single tree from the traversal on the object graph $G_O = (O, E)$.

$$R = O \times (F \rightarrow \{R\}) \text{ where } (o, (f, r)) \in R \text{ implies } |E(o, f)| = |r|$$

If the program navigates to the same object multiple times in a traversal, only the shortest path from the root of the traversal is included in R. Figure 4 shows a sample traversal on the object graph in Figure 3 for a query which returned 3 departments: d_1, d_2, d_3. Edges with dark ovals represent collection associations.

If a program navigates an association, it may not be included in the traversal if it would not result in database load. An association navigation does not result in a database load in three cases:

– The association is a null-valued single association.
– The association is a single valued association whose target had already been navigated to from the root object with a shorter path.
– The association's target was cached by the program.

If a program navigates an empty collection association, there will be a database query and the navigation will be included in the traversal. The last item illustrates an interesting link between caching and query execution; AutoFetch is able to adapt to the caching mechanism of the application by adjusting query prefetch to ignore associations that are likely to be cached.

An important point is that a single query may be used in different contexts that generate different traversals. This will commonly happen if a library function runs a query to load a set of objects, but this library function is called from multiple transactions. Each transaction will have a different purpose and therefore may traverse different associations.

Traversal Profiles A traversal profile represents the aggregation of the traversals for a set of queries. Each traversal profile is a tree representation of all the previous traversals mapped to the type graph. Let P represent a traversal profile for a type graph $G_T = (T, A)$:

$$P = T \times N \times N \times (F \rightarrow P)$$

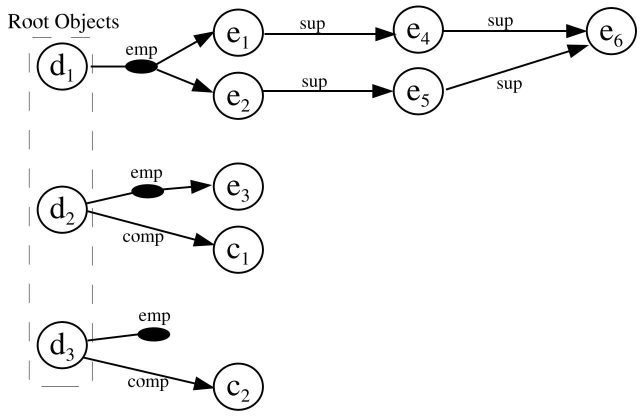

Fig. 4. An example of a traversal on the object graph in Figure 3. Collection associations contain an oval in the middle of the edge.

such that for all $(t, used, potential, (f, p)) \in P$

1. $A(t, f)$ is defined
2. $used \leq potential$.

Each node in the tree contains statistics on the traversals to this node: the number of times this node needed to be loaded from the database (*used*), and the number of opportunities the program had to load this node from the database (*potential*), i.e. the number of times the program had a direct reference to an object representing this node.

Algorithm 1 $combine((o, AO), (used, potential, t, AP))$

```
for all $(f, (used, potential, t, A)) \in AO$ do
  $w(f) = (used, potential + 1, t, A)$
end for
for all $f, P \in AP$ do
  for all $r \in AO(f)$ do
    $w(f) = combine(r, w(f));$
  end for
end for
return $(used + 1, potential, t, w)$
```

The traversal, a forest of object trees R, is combined with a traversal profile by combining each object tree R in the traversal with the profile using a function *combine* $(R \times P \rightarrow P)$. The combination algorithm is straightforward. Given a traversal and traversal profile, *combine* increments the *used* statistic for the root of the traversal profile and the potential statistic for all the children of the root. The combine method is then recursively called for each child traversal profile and

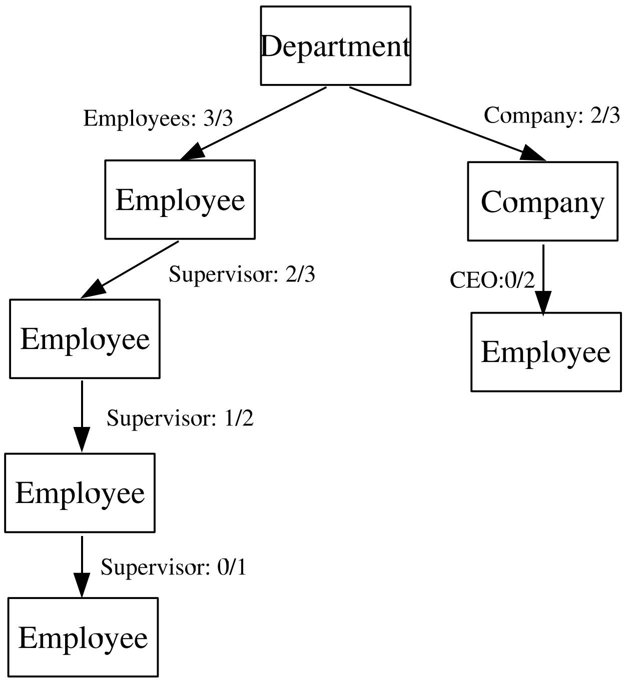

Fig. 5. Traversal profile for query class after traversal in Figure 4. Statistics are represented as (used/potential).

its matching (same association) children of the root node in R. The statistics for the root node of the traversal profile are ignored since they represent the statistics for the root objects returned by a query and AutoFetch assumes those objects should always be fetched. Figure 5 shows a traversal profile updated from an empty traversal profile and the traversal in Figure 4. The traversal profile statistics are given above each type as (used/potential).

3.2 Query Classification

Query classification determines a set of queries that share a traversal profile. The aim of query classification is to group queries which are likely to have similar traversals. A simple classification of queries is to group all queries that have the same query string. There are several reasons why this is not effective.

First, a given query may be used to load data for several different operations. Since the operations are different, the traversals for these operations may be different as well. This situation typically arises when query execution is centralized in library functions that are called from many parts of a program. Classifying based only on the criteria will not distinguish between these different uses of a query, so that very different traversals may be classified as belonging to the same class. This may lead to poor prediction of prefetch. The classification in this case is too coarse.

A second problem is that query criteria are often constructed dynamically. If each set of criteria is classified as a separate query, then commonality between operations may not be identified. At the limit, every query may be different, leading to a failure to gather sufficient data to predict prefetch.

Queries may also be classified by the program state when the query is executed. This is motivated by the observation that traversals are determined by the control flow of the program after query execution. Program state includes the current code line, variable values, library bindings, etc. Classifying queries based on the entire program state is infeasible as the program state may be very large and will likely be different for every query. However, a set of salient features of the program state can be reasonable both in memory and computation. Computation refers to cost of computing the program state features when a query is invoked.

The line number where a query is executed is a simple feature of the program state to calculate and has a small constant memory size, however, it does not capture enough of the program state to accurately determine the traversal of the query results. Specifically the problem is that line number where the query is executed does not provide enough information on how the results of the query will be used outside of the invoking method.

The natural extension to the using the line number where the query is executed is using the entire call stack when the query is executed. Our hypothesis is that the call stack gives more information about the future control flow, because it is highly likely that the control flow will return through the methods in the call stack. The call stack as the salient program state feature is easy to compute and bounded in size. In the programs we have considered, we have found that the call stack classifies queries at an appropriate granularity for prefetch.

Unfortunately, a call stack with line numbers will classify 2 queries with different extent types together if the 2 queries occur on the same line. To address this, AutoFetch uses the pair of the query string and the call stack when the query is executed to classify queries. This limits AutoFetch's ability to prefetch for dynamic queries. Optimally, the call stack would contain information on the exact program statement being executed at each frame.

3.3 Predicting Traversals

Given that an operation typically traverses a similar collection of objects, it is possible to predict future traversals based on the profiling of past traversals. The predicted traversal provides a basis to compute the prefetch specification. The goal of the prefetch specification is to minimize the time it will take to perform the traversal. A program will be most efficient if each traversal is equal to the query result object graph, because in this case only one round-trip to the database will be required and the program will not load any more information from the database than is needed. The heuristic used in AutoFetch is to prefetch any node in the traversal profile for which the probability of traversal is above a certain threshold.

Before each query execution, AutoFetch finds the traversal profile associated with the query class. If no traversal profile is found, a new traversal profile is created and no prefetches are added to the query. Otherwise, the existing traversal profile is used to compute the prefetch specification.

First, the traversal profile is trimmed such that the remaining tree only contains the associations that will be loaded with high probability (above a set threshold) given that the root node of the traversal profile has been loaded. For each node n and its parent node $p(n)$ in the traversal profile, the probability that the association between n and $p(n)$ will be traversed given that p(n) has been loaded can be estimated as $\mathit{used}(n)/\mathit{potential}(n)$. Using the rules of conditional probability, the probability that the association is navigated given that the root node is loaded is:

$$f(n) = (\mathit{used}(n)/\mathit{potential}(n)) * f(p(n))$$

The base case is that the $f(\mathit{root})$ in the traversal profile is 1. A depth first traversal can calculate this probability for each node without recomputing any values. This calculation ensures that traversal profile nodes are prefetched only if their parent node is prefetched, because $f(n) \leq f(p(n))$.

Second, if there is more than one collection path in the remaining tree, an arbitrary collection path is chosen and other collection paths are removed. Collection paths are paths from the root node to a leaf node in the tree that contain at least 1 collection association. This is to avoid creating a query which joins multiple many-valued associations.

The prefetch specification is a set of prefetch directives. Each prefetch directive corresponds to a unique path in the remaining tree. For example, given the traversal profile in Figure 5 and the prefetch threshold of 0.5, the prefetch specification would be: (employees, employees.supervisor, company). The query is augmented with the calculated prefetch specification. Regardless of the prefetch specification, profiling the query results remains the same.

4 Implementation

The implementation of AutoFetch is divided into a traversal profile module and an extension to Hibernate 3.1, an open source Java ORM tool.

4.1 Traversal Profile Module

The traversal profile module maintains a 1-1 mapping from query class to traversal profile. When the hibernate extension asks for the prefetch specification for a query, the module computes the query class which is used to lookup the traversal profile which is used to compute the prefetch specification. The module computes the query class as the pair of the query string and the current program stack trace and uses this as the key to lookup the traversal profile. To decrease the memory requirements for maintaining the set of query classes, each stack trace contains a maximum number of frames. If a stack trace is larger than this limit, AutoFetch removes top-level frames until the stack trace is under the limit. Each frame is a string containing the name of a method and a line number. If a traversal profile does not exist for a query class, the module adds a mapping from that query class to an empty traversal profile. Finally, the module computes a prefetch specification for the query using the traversal prediction algorithm in Section 3.3 applied to the traversal profile.

4.2 Hibernate

Hibernate was modified to incorporate prefetch specifications and to profile traversals of its query results. The initial AutoFetch implementation used Hibernate 3.0 which did not support multiple collection prefetches. Fortunately, Hibernate 3.1 contains support for multiple collection prefetches and AutoFetch was migrated to this version. Support for multiple collection prefetches turns out to be critical for improving performance in some of the evaluation benchmarks.

Hibernate obtains the prefetch specification for a query from the traversal profile module. The code in Figure 6 illustrates how a HQL query is modified to include prefetches and the SQL generated by Hibernate. Queries which already contain a prefetch specification are not modified or profiled allowing the programmer to manually specify prefetch. The hibernate extensions profile traversals by instrumenting each persistent object with a dynamically generated proxy. The proxy intercepts all method calls to the object and if any object state is accessed that will require a database load, the proxy increments the appropriate node in the traversal profile for the query class. Hibernate represents single association references with a key. Therefore, accessing the key is not considered as an object access because it never requires a database query. Collections are instrumented by modifying the existing Hibernate collection classes. Although there is a performance penalty for this type of instrumentation, we found that this penalty was not noticeable in executing queries in our benchmarks. This performance penalty may be ameliorated through sampling, i.e. only instrumenting a certain percentage of queries. The AutoFetch prototype does not support all of Hibernate's features. For example, AutoFetch does not support prefetching or profiling for data models which contain weak entities or composite identifiers. Support for these features was omitted for simplicity.

5 Evaluation

We evaluated AutoFetch using the Torpedo and OO7 benchmarks. The Torpedo benchmark measures on the number of queries that an ORM tool executes in a simple auction application, while the OO7 benchmark examines the performance of object-oriented persistence mechanisms for an idealized CAD (computer assisted design) application. We also examined the software engineering benefits of avoiding manual specification of prefetches in a resume application.

Original query

HQL:

```
from Department d where d.name = 'foo'
```

SQL:

```
select * from Department as d where d.name = 'foo'
```

Query with a single prefetch

HQL:

```
from Department d
 left outer join fetch d.employees where x.name = 'foo'
```

SQL:

```
select * from Department as d
 left outer join Employee as e on e.deptId = d.id
where d.name = 'foo'
```

Fig. 6. Augmenting queries with prefetch specifications.

Both benchmarks were executed on an Intel®Pentium®4 2.8 GHz machine with 1 Gb of RAM. The OO7 benchmark connects to a database on a separate machine, an Intel®Pentium®4 2.4 Ghz machine with 885 Mb of RAM on the same University of Texas Computer Science department local area network. The AutoFetch parameters maximum extent level and stack frame limit were set to 12 and 20 respectively unless otherwise noted. The benchmarks did not use any caching across transactions.

5.1 Torpedo Benchmark

The Torpedo benchmark [23] measures the number of SQL statements executed by an ORM tool over a set of test cases. The benchmark consists of a Java client and a J2EE auction server. The client issues requests to the auction server, such as placing a bid or retrieving information for a particular auction. There are seven client test cases which were designed to test various aspects of the mapping tool such as caching or prefetching. The number of SQL statements executed is used as the measure of the performance of the mapping tool. The benchmark can be configured to use different object-relational mapping tools (EJB, JDO, Hibernate) as the persistence backend.

We created two versions of the Hibernate persistence backend, the original tuned backend included with the benchmark and that same backend minus the prefetch directives. The latter backend can be configured to have AutoFetch enabled or disabled. We ran the Torpedo benchmark for each version and possible options three times in succession. The results of the first and third iterations are shown in Figure 7. The second run was omitted in the graph since the first and second iterations produce the same results. A single set of iterations is sufficient, because the benchmark is deterministic with respect to the number of queries.

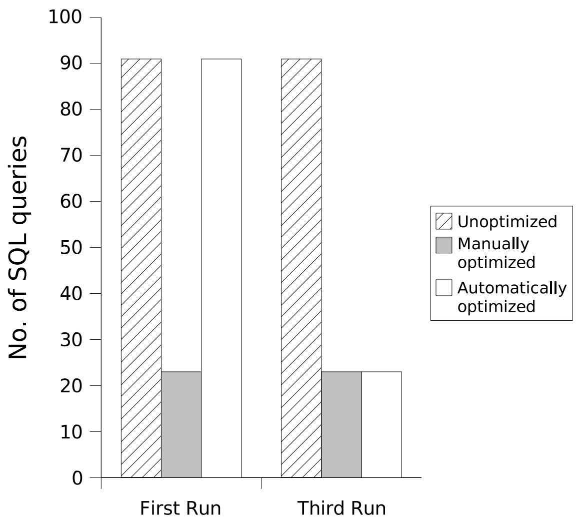

Fig. 7. Torpedo benchmark results. The y-axis represents the number of queries executed. Maximum extent level is 12.

As Figure 7 shows, the prefetch directives reduce the number of queries executed. Without either the prefetch directives nor AutoFetch enabled the benchmark executed three times as many queries. Without prefetch directives but with AutoFetch enabled, the benchmark executes many queries on the first and second iterations; however, from the third iteration (and onward) it executes as many queries as the version with programmer-specified prefetches.

A simple query classification method using the code line where the query was executed as the query class would not have been sufficient to match the performance of manually specified prefetches for this benchmark. For example, the findAuction method is used to load both detailed and summary information about an auction. The detailed auction information includes traversing several associations for an auction such as the auction bids. The summary auction information only includes fields of the auction object such as the auction id or date. These different access patterns require different prefetches even though they use the same backend function to load the auction.

5.2 OO7 Benchmark

The OO7 benchmark [5] was designed to measure the performance of OODB management systems. It consists of a series of traversals, queries, and structural

Table 1. Comparison with prefetch disabled and with AutoFetch. Maximum extent level is 12. Small OO7 benchmark. Metrics for each query/traversal are average number SQL queries and average time in milliseconds. Percentages are for percent improvement of AutoFetch over baseline.

Query	Iteration	No Prefetch		AutoFetch			
		queries	ms	queries	%	ms	%
Q1	1	11	45	11	–	43	(4%)
	2	11	44	11	–	43	(2%)
	3	11	43	11	–	43	–
Q2	1	2	10	2	–	9	(10%)
	2	2	10	2	–	10	–
	3	2	11	2	–	10	(9%)
Q3	1	2	59	2	–	58	(2%)
	2	2	89	2	–	59	(34%)
	3	2	58	2	–	60	-(3%)
Q6	1	2	70	2	–	69	(1%)
	2	2	66	2	–	65	(2%)
	3	2	67	2	–	81	-(21%)
Q7	1	2	532	2	–	504	(5%)
	2	2	472	2	–	508	-(8%)
	3	2	498	2	–	471	(5%)
Q8	1	2	43	2	–	48	-(12%)
	2	2	46	2	–	46	–
	3	2	48	2	–	44	(8%)
T1	1	3096	21750	2909	(6%)	20875	(4%)
	2	3096	22160	2907	(6%)	20694	(7%)
	3	3096	21009	38	(98.8%)	248	(98.8%)
T6	1	1146	8080	1099	(4%)	8266	-(2%)
	2	1146	7900	1096	(4%)	8115	-(3%)
	3	1146	7831	2	(99.8%)	21	(99.7%)
T8	1	2	36	2	–	38	-(6%)
	2	2	46	2	–	36	(22%)
	3	2	36	2	–	40	-(11%)
T9	1	2	40	2	–	35	(13%)
	2	2	44	2	–	38	(14%)
	3	2	40	2	–	36	(10%)
RT	1	10	63	4	(60%)	43	(32%)
	2	10	63	3	(70%)	39	(38%)
	3	10	61	3	(70%)	39	(36%)

modifications performed on databases of varying sizes and statistical properties. We implemented a Java version of the OO7 benchmark based on code publicly available from the benchmark's authors. Following the lead in Han [13], we omitted all structural modification tests as well as any traversals that included updates, because updates have no effect on AutoFetch behavior and otherwise these traversals were not qualitatively different from the included traversals. Q4 was omitted because it requires using the medium or large OO7 databases. Traversal CU was omitted because caching and AutoFetch are orthogonal, and the traversal's performance is very sensitive to the exact caching policy.

Only a few of the OO7 operations involve object navigation, which can be optimized by AutoFetch. Traversal T1 is a complete traversal of the OO7 object graph, both the assembly and part hierarchies. Traversal T6 traverses the entire assembly hierarchy, but only accesses the composite and root atomic parts in the part hierarchy. Traversal T1 has a depth of about 29 while Traversal T6 has a depth of about 10. Neither the queries nor traversals T8 or T9 perform navigation; however, they are included to detect any performance penalties for traversal profiling.

We added a reverse traversal, RT, which chooses atomic parts and finds their root assembly, associated module, and associated manual. Such traversals were omitted from the OO7 benchmark because they were considered not to add anything to the results. They are significant in the context of prefetch, since single-valued associations can be prefetched more easily than multi-valued associations.

Table 1 summarizes the results of the OO7 benchmark. Neither the queries nor traversals T8 or T9 show any improvement with prefetch enabled. This is to be expected since they do not perform any navigational queries. These queries are included for completeness, and to show that AutoFetch does not have high overhead when not needed.

Both traversals T1 and T6 show a large improvement in the number of queries and time to execute the traversal. T6 shows a larger improvement than T1 even though T1 is a deeper traversal, because some of the time executing traversal T1 is spent traversing the object graph in memory; repeatedly traversing the part hierarchies. The number of queries and the time to execute a traversal are tightly correlated as expected. Both T1 and T6 are top-down hierarchical traversals which require multiple collection prefetches to execute few queries. Table 2 shows a comparison of the number of queries executed by AutoFetch with Hibernate 3.1 and AutoFetch with Hibernate 3.0 which was unable to

Table 2. The number of queries executed by AutoFetch with Hibernate 3.0 and AutoFetch with Hibernate 3.0 for traversals T1, T6, and RT. Only 3rd iteration shown. Maximum extent level is 12. Small OO7 benchmark.

AutoFetch Version	T1	T6	RT
AutoFetch with Hibernate 3.0	2171	415	3
AutoFetch with Hibernate 3.1	38	2	3

prefetch multiple collection associations. The ability to fetch multiple collection associations had a greater effect on deep traversals such as T1 and T6 than on shallow traversals such as RT.

Figure 8 shows that the maximum depth of the traversal profile is important to the performance of prefetch system in the presence for deep traversals. The tradeoff for increasing the maximum depth of the traversal profile is an increase in the memory requirements to store traversal profiles. It should be noted that deep traversals such as T1 and T6 in OO7 are relatively rare in enterprise business applications.

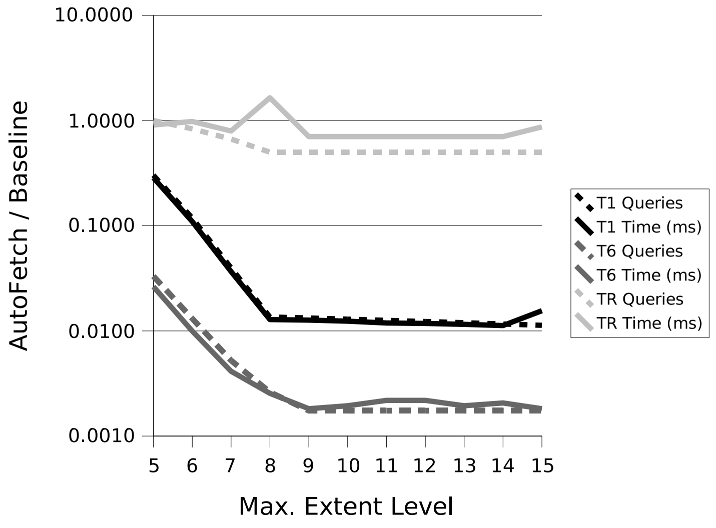

Fig. 8. Varying maximum extent level from 5 to 15. Only 3rd iteration shown. Small OO7 database.

5.3 Resume Application

In addition to the synthetic benchmarks, we applied AutoFetch to a resume application that uses the AppFuse framework [29]. AppFuse is a template for a model-view-controller (MVC) architecture that integrates many popular Java libraries and tools. AppFuse includes a flexible data layer which can be configured to use one of several persistence providers. Users of the framework define interfaces for data access objects (DAO) that are implemented using the persistence provider.

Hibernate is used as the persistence provider in the sample resume application. The resume application data model is centered around a Resume class. A Resume contains basic resume data fields and associations to related objects, including education listings, work experiences, and references. The ResumeDAO class includes methods to load and store resumes. A simple implementation of the ResumeDAO and Resume classes is shown in Fig 9. The ResumeDAO.getResume(Long) method loads a resume without prefetching any of its associated objects. To load the work experience in a resume, a programmer first uses ResumeDAO to load the resume, and then getExperiences() to load the work experience.

```
interface ResumeDAO {
  Resume getResume(Long resumeId);
   ...
}

class Resume {
   List getEducations() { ... }
   List getExperiences() { ... }
   List getReferences() { ... }
   ...
}
```

Fig. 9. Struts resume code without any optimizations

Although this implementation is very natural, it is inefficient because the resume application has several pages that display exactly one kind of associated object; a page for work experience, a page for references, etc. For these pages, the application would execute 2 queries: one to load the resume and another to load the associated objects. There are several alternative implementations:

1. Modify the ResumeDAO.getResume(Long) method to prefetch all associations.
2. Add ResumeDAO methods which load a resume with different prefetch directives.
3. Add ResumeDAO methods which directly load associated objects without loading the resume first.

The actual implementation uses the third approach. The first alternative always loads too much data and would be infeasible if the data model contained cycles. The other two alternatives are fragile and redundant. For example, if a new user interface page was added to the application that displayed two resume associations, then a new method would have to be added to the ResumeDAO class. The code is also redundant because we have to copy either the ResumeDAO.getResume(Long) method in the second alternative or the Resume getter methods in the third alternative. By incorporating AutoFetch, the simple code in Figure 9 should perform as well as the optimized code after some initial iterations.

We tested the code in Figure 9 version with AutoFetch and found that indeed it was able to execute a single query for all the controller layer methods after the initial learning period. Our modified code has the advantage of being smaller, because we eliminated redundant methods in ResumeDAO class. With AutoFetch, DAO methods are more general because the same method may be used with different traversal patterns. AutoFetch also increases the independence of the user interface or view layer from the business logic or controller layer, because changes in the traversal pattern of the user interface on domain objects do not require corresponding changes in the controller interface.

5.4 General Comments

In all of the evaluation benchmarks, the persistent data traversals were the same given the query class. Consequently, AutoFetch never prefetched more data than was needed, i.e. AutoFetch had perfect precision. While our intuition is that persistent data traversals are usually independent of the program branching behavior, it is an open question whether our benchmarks are truly representative in this respect. Similarly, it is difficult to draw general conclusions about the parameters of the AutoFetch such as the maximum extent level or stack frame limit without observing a larger class of persistent programs. The maximum extent level was set to 12, because this produced reasonable memory consumption on our benchmarks. The stack frame limit was set to 20 to preserve enough information from the stack frame about control flow in the presence of the various architectural layers in the Torpedo benchmark and the recursive traversals in the OO7 benchmark.

6 Related Work

Han et al. [13] classify prefetching algorithms into five categories: page-based prefetching, object-level/page-level access pattern prefetching, manually specified prefetches, context-based prefetches, and traversal/path-based prefetches.

Page-based prefetching has been explored in object-oriented databases such as ObjectStore [17]. Page-based prefetching is effective when the access patterns of an application correspond to the clustering of the objects on disk. Since the clustering is usually static, it cannot efficiently support multiple data access patterns. Good clustering of objects is difficult to achieve and can be expensive to maintain when objects are updated frequently. However, when it works it provides very low-cost prefetching. Finally, if the amount of object data that should be prefetched is larger than a page, than this prefetching algorithm will be unable to prefetch all the objects needed.

Object-level or page-level access pattern prefetching relies on monitoring the sequence of object or page requests to the database. Curewitz et al. [9] implemented an access pattern prefetching algorithm using compression algorithms. Palmer and Zdonik [26] implemented a prefetch system, Fido, that stores access patterns and uses a nearest neighbor algorithm to detect similar patterns and issue prefetch requests. Knafla [16] models object relationship accesses as discrete time Markov chains and uses this model in addition to a sophisticated cost model to issue prefetch requests. The main drawback to these approaches is that they detect object-level patterns, i.e. they perform poorly if the same objects are not repeatedly accessed. Repeated access to the same objects is not typical of many enterprise applications with large databases.

Bernstein et al. [4] proposed a context-controlled prefetch system, which was implemented as an extension of Microsoft Repository. Each persistent object in memory is associated with a context. This context represents a set of related objects, either objects that were loaded in the same query or objects that are a member of the same collection association. For each attribute access of an object O, the system prefetches the requested attribute for all objects in O's context. When iterating through the results of a query or collection association, this prefetch strategy will avoid executing $n + 1$ queries where n is the number of query results. A comparison of this strategy and AutoFetch is given below. While AutoFetch only profiles associations, Bernstein et al. use "MA prefetch" to prefetch scalar attributes for classes in which the attributes reside in separate tables. MA prefetch improves the performance of the OO7 benchmark queries, which were not improved by AutoFetch, because OO7 attributes and associations are separated into multiple tables. The implemented system only supported single-level prefetches, although prefetching multiple levels (path prefetch) is mentioned as an extension in the paper. The system also makes extensive use of temporary tables, which are not needed by AutoFetch.

Han et al. [14, 13] extended the ideas of Bernstein et al. to maintain not only the preceding traversal which led to an object, but the entire type-level *path* to reach an object. Each query is associated with an attribute access log set which contains all the type level paths used to access objects from the navigational root set. The prefetch system then monitors the attribute access log and prefetches objects if either an iterative or recursive pattern is detected. The prefetch system, called PrefetchGuide, can prefetch multiple levels of objects in the object graph if it observes multi-level iteration or recursive patterns. However, unlike the Bernstein prefetch implementation, there are no results on prefetching for arbitrary queries, instead only purely navigational queries are supported. PrefetchGuide is implemented in a prototype ORDBMS.

While the systems created by Bernstein and Han prefetch data within the context of a top-level query, AutoFetch uses previous query executions to predict prefetch for future queries. Context-based prefetch always executes at least one query for each distinct association path. AutoFetch, in contrast, can modify the top-level query itself, so that only one query is needed. AutoFetch can also detect traversal patterns across queries, e.g. if certain unrelated associations are always accessed from a given query result, AutoFetch prefetches those objects even though it would not constitute a recursive or iterative pattern within that single query. One disadvantage of AutoFetch is that the initial queries are executed without any prefetch at all. The consequence of this disadvantage, is that the performance on the initial program iteration is equivalent to a program with unoptimized queries. However, it would be possible to combine AutoFetch with a system such as PrefetchGuide. In such a combined system, PrefetchGuide could handle prefetch in the first query, and also catch cases where the statistical properties of past query executions do not allow AutoFetch to predict correct prefetches. We believe that such a combination would provide the best of both worlds for prefetch performance.

Automatic prefetch in object persistence architectures is similar to prefetching memory blocks as a compiler optimization. Luk and Mowry[20] have looked at optimizing recursive data structure access by predicting which parts of the structure will be accessed in the future. One of their approaches, history pointers, is similar in philosophy to our traversal profiles.

7 Future Work

We presented a simple query classification algorithm which only relies on the call stack at the moment the query is executed. Although we found this to work quite well in practice, a more complex classification algorithm could include other features of program state: the exact control path where the query was executed, or the value of program variables. This richer program state representation might classify queries too finely. Unsupervised learning techniques could be applied to richer program state representations to learn a classification that clusters the queries according to the similarity of their traversals. Consider the following program fragment, where findAllFoos executes a query:

```
List results = findAllFoos();
if (x > 5)
  doTraversal1( results );
else
  doTraversal2( results );
```

A learning algorithm could learn a better classification strategy than the one described in this paper. In this case, the value of the variable x should be used to distinguish two query classes.

A cost model for database query execution is necessary for accurate optimization of prefetching. AutoFetch currently uses the simple heuristic that it is always better to execute one query rather than two (or more) queries if the data loaded by the second query is likely to be needed in the future. This heuristic relies on the fact that database round-trips are expensive. However, there are other factors that determine cost of prefetching a set objects: the cost of the modified query, the expected size of the set of prefetched objects, the connection latency, etc. A cost model that takes such factors into account will have better performance and may even outperform manual prefetches since the system would be able to take into account dynamic information about database and program execution.

8 Conclusion

Object prefetching is an important technique for improving performance of applications based on object persistence architectures. Current architectures rely on the programmer to manually specify which objects to prefetch when executing a query. Correct prefetch specifications are difficult to write and maintain as a program evolves, especially in modular programs. In this paper we presented AutoFetch, a novel technique for automatically computing prefetch specifications. AutoFetch predicts which objects should be prefetched for a given query based on previous query executions. AutoFetch classifies queries executions based on the client state when the query is executed, and creates a traversal profile to summarize which associations are traversed on the results of the query. This information is used to predict prefetch for future queries. Before a new query is executed, a prefetch specification is generated based on the classification of the query and its traversal profile. AutoFetch improves on previous approaches by collecting profile information across multiple queries, and using client program state to help classify queries. We evaluated AutoFetch using both sample applications and benchmarks and showed that we were able to improve performance and/or simplify code.

References

1. M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987.
2. M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *SIGMOD Record*, 25(4):68–75, 1996.
3. T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.
4. P. A. Bernstein, S. Pal, and D. Shutt. Context-based prefetch for implementing objects on relations. In *Proceedings of the 25th VLDB Conference*, 1999.
5. M. J. Carey, D. J. DeWitt, and J. F. Naughton. The 007 benchmark. *SIGMOD Rec.*, 22(2):12–21, 1993.
6. D. Cengija. Hibernate your data. *onJava.com*, 2004.
7. W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 97–106. ACM Press, 2005.
8. G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 316–325. ACM Press, 1984.
9. K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, 1993.
10. J.-A. Dub, R. Sapir, and P. Purich. Oracle Application Server TopLink application developers guide, 10g (9.0.4). Oracle Corporation, 2003.
11. J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 85–95. ACM Press, 1992.

12. G. Hamilton and R. Cattell. JDBC^{TM}: A Java SQL API. Sun Microsystems, 1997.
13. W.-S. Han, Y.-S. Moon, and K.-Y. Whang. PrefetchGuide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational DBMSs. *Information Sciences*, 152(1):47–61, 2003.
14. W.-S. Han, Y.-S. Moon, K.-Y. Whang, and I.-Y. Song. Prefetching based on type-level access pattern in object-relational DBMSs. In *Proceedings of the 17th International Conference on Data Engineering*, pages 651–660. IEEE Computer Society, 2001.
15. ISO/IEC. Information technology - database languages - SQL - part 3: Call-level interface (SQL/CLI). Technical Report 9075-3:2003, ISO/IEC, 2003.
16. N. Knafla. Analysing object relationships to predict page access for prefetching. In *Eighth International Workshop on Persistent Object Systems: Design, Implementation and Use, POS-8*, 1998.
17. C. Lamb, G. Landis, J. A. Orenstein, and D. Weinreb. The ObjectStore database system. *Commun. ACM*, 34(10):50–63, 1991.
18. K. J. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.
19. B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 318–329. ACM Press, 1996.
20. C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Architectural Support for Programming Languages and Operating Systems*, pages 222–233, 1996.
21. D. Maier, J. Stein, A. Otis, and A. Purdy. Developments of an object-oriented DBMS. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 472–482, 1986.
22. A. Marquez, S. Blackburn, G. Mercer, and J. N. Zigman. Implementing orthogonally persistent Java. In *Proceedings of the Workshop on Persistent Object Systems (POS)*, 2000.
23. B. E. Martin. Uncovering database access optimizations in the middle tier with TORPEDO. In *Proceedings of the 21st International Conference on Data Engineering*, pages 916–926. IEEE Computer Society, 2005.
24. V. Matena and M. Hapner. Enterprise Java Beans Specification 1.0. Sun Microsystems, 1998.
25. R. Morrison, R. Connor, G. Kirby, D. Munro, M. Atkinson, Q. Cutts, A. Brown, and A. Dearle. The Napier88 persistent programming language and environment. In *Fully Integrated Data Environments*, pages 98–154. Springer, 1999.
26. M. Palmer and S. B. Zdonik. Fido: A cache that learns to fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases*, 1991.
27. D. A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, 2004.
28. C. Russell. Java Data Objects (JDO) Specification JSR-12. Sun Microsystems, 2003.
29. Raible's wiki: StrutsResume. `http://raibledesigns.com/wiki/Wiki.jsp?page=StrutsResume`, March 2006.
30. M. Venkatrao and M. Pizzo. SQL/CLI – a new binding style for SQL. *SIGMOD Record*, 24(4):72–77, 1995.