Increasing Concurrency in Databases using Program Analysis

Roman Vitenberg[1], Kristian Kvilekval[1], and Ambuj K. Singh[1]

Computer Science Department, UCSB, Santa Barbara, CA 93106, USA
{romanv,kris,ambuj}@cs.ucsb.edu,
WWW home page: http://www.cs.ucsb.edu/{~romanv,~kris,~ambuj}

Abstract. Programmers have come to expect better integration between databases and the programming languages they use. While this trend continues unabated, database concurrency scheduling has remained blind to the programs. We propose that the database client programs provide a large untapped information resource for increasing database throughput.
Given this increase in expressive power between programmers and databases, we investigate how program analysis can increase database concurrency. In this paper, we demonstrate a predictive locking scheduler for object databases. In particular we examine the possibility to predict the client's use of locks. Given accurate predictions, we can increase concurrency through early lock release, perform deadlock detection and prevention, and determine whether locks should be granted before or during a transaction. Furthermore, we demonstrate our techniques on the OO7 and other benchmarks.

1 Introduction

The problem of transaction and lock scheduling is the most fundamental problem in concurrency control in databases. Finding the optimal schedule is known to be an NP-hard even for the offline version of the problem when all events (i.e., transactions) in the system are known in advance [16]. Furthermore, there is no general-case online algorithm that would approximate the optimal offline solution within some small bound. Yet, in many specific cases of the systems that exist in practice, it is possible to design a scheduler that takes advantage of the information about future transactions, producing a more efficient (even though non-optimal) schedule.

Knowledge of the future is the key to efficient scheduling of resources. This knowledge of future events can come from diverse sources, but has been traditionally in the form of programmer annotations. Programmer annotations, though used, are in general difficult for the programmers to construct and are likely to be error prone. Therefore, most systems have adopted an overly conservative view and assume no knowledge of future requests.

However, the knowledge of future access patterns is present in the client programs that use the database system. In recent years, there has been an increasing interest in object database languages with the acceptance of object-oriented and object-relational database systems. These systems already reduce the "impedance-mismatch" between the program code and data storage often experienced in traditional SQL environments.

With the emergence of complex database interface languages such as JDO [13], OQL [15], and the use of complex types in databases that bridge the gap between programming languages and data modelling languages, we explore the benefits that this tighter integration brings. Our work takes a new approach combining program analysis and object databases in order to extract information that will be useful to the database system. In this paper, we present new techniques for scheduling transactions in object-oriented database (OODB) management systems. The technique is also applicable to Object-Relational databases provided that the query language is rich enough to warrant analysis.

This paper's main contributions are: deadlock handling methods based on the types manipulated by the program, a technique that allows early lock release in order to increase concurrency, and a method to determine whether transaction locks should be preclaimed, e.g., before the transaction begins, or taken gradually during the transaction.

In detail, we investigate the use of program analysis to extract interesting properties of programs that can be used by a database system. Our technique is based on shape analysis, a whole-program analysis that has previously used in the compiler research community to determine certain static program properties. The output of shape analysis is a set of graphs representing the way a portion of a program navigates and manipulates its data structures. These structures allow the database system to determine what the client will do as it continues to execute thereby capturing future knowledge of the client's object use.

In order to test our ideas, we constructed a benchmark testbed and ran experiments with the standard OO7 benchmark set and with the prototype of a car reservation system that we have developed. The paper shows the gain obtained from using each proposed scheduling enhancement in terms of the average execution time of a transaction and its standard deviation.

2 Related Work

There has been a tremendous amount of work dealing with transaction concurrency and scheduling [3, 19]. However, schedulers in most database systems that exist in practice do not attempt to predict. The main reason for this is twofold: a) Eliciting and collecting information about future transactions is a non-trivial task, and b) Such predictive schedulers would be highly specialized and tailored to a particular application.

Some database systems that attempt to predict based on the history of previous executions, which is collected with profiling. In particular, this technique is commonly used for query optimization in relational databases. However, history-based prediction is different from prediction that relies on program analysis: while the former predicts solely based on past workloads, the latter gives more precise information about the future execution of the currently running transactions.

Many predictors have been demonstrated in practice. Most of these are based on simplifying assumptions about how a program will access data. For example, programs often perform sequential reads from the disk. i.e. while reading a very large array. For this reason, many disk drives automatically perform k-block read-ahead. However, when data is complex and accessed in a scattered way, sequential lookahead not be appropriate behavior as we may read many unused pages.

Object-oriented programs can and usually do have complex object structures. These pointer-based structures make it especially difficult to predict what object(s) the system will be using in the future. Though an important problem, little work has been carried out for predicting access patterns in complex pointer-based structures. Knafla [10] demonstrates prefetching for OODBs using history-based techniques. Cahoon and McKinly [5] have examined a dataflow analysis for prefetching object structures in Java. In contrast, our approach constructs a succinct representation of the program's access pattern and uses it to drive the prediction process. Combining the program representation with real data allows the predictive scheduler to infer the most likely objects to be used by the program in the future.

3 Model

Our model encompasses the design of most existing middleware systems for object management. Clients may send requests to a server which may act over a set of objects that reside solely on the servers. An object server holds multiple objects and the database on a server consists of a set of root objects such that all other objects accessible from these root objects.

Multiple clients are allowed to access the server concurrently invoking server-side transactions. During each transaction, the client may reference multiple objects on the server. The database at the server includes a scheduler module which is responsible for maintaining the consistency of the transactions and the objects. In this work, we consider both pessimistic and optimistic concurrency control models.

4 Predicting Object Accesses and Execution Times based on Shape Graphs

4.1 Background

Shape analysis is a program analysis technique based on an abstract interpretation of a program to determine the "shape" of possible runtime data structures. Shape analysis produces a *shape graph* for a program point, representing the way the program traverses program data structures from that particular point. A shape graph is a directed graph with nodes representing symbolic abstract runtime program values and edges representing program field references from those values. The shape graph is generated by symbolically executing the program code and adding edges for each access.

In order to provide intuition, we present a typical program fragment of integrated databases in Figure 1. The shape graph shown on the right of the figure is derived from the code lines on the left.[1]

[1] The code is taken from the OO7 database [6]; it navigates the database of machine parts in order to weigh the elements. However, the code semantics are not important for this example.

```
class Connector{
  Part partA, PartB; ... }
class Part {
  Connector left, right, up, down;
  Material material;
  Supplier supplier;
  Cost     cost;
  ...
  int volume(); }

weight = 0
while (part != null)
  weight += part.material.density
            * part.volume();
  connector = part.right;
  part = null;
  if (connector != null)
    part = connector.partB;
```

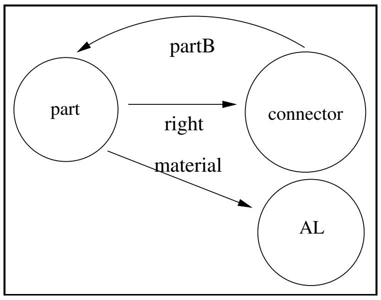

Fig. 1. Code fragment and its shape graph: only items used in the code (`part` and `material`) are in the graph.

The example code uses only the `material` and `right` fields of each `part` in the database ignoring the `cost` and `supplier` among other fields. This is quite typical for database programs: while the database may be large and have a very rich object structure, many programs may use only part of that structure. The access pattern is also revealed in the fact that the code fragment iterates through a list of `part` using the `right` field. In the graph, the node (`part`) is used to represent the values of the variable `part` which access `connector` through the field `right`. The runtime value `part.material` is shown in the shape graph as `AL`. The cycle `part` through the field `right` to `connector` through `partB` and back to `part` contains the needed loop information from the original code. Note that the `volume` function may access additional objects that are not presented in the shape graph.

Shape graphs have previously been used to determine static properties of programs and for many compile time optimizations including removing synchronization primitives [4, 17], parallelization of codes [8], and type safety. Other uses include null analysis, pointer aliasing, cyclicity detection, reachability, and typing [9, 14, 20]. Use of shape graphs for prefetching has been explored in our previous work [11]. This work adopts the shape graph structure and its construction algorithm that are similar to the existing implementations. Yet, this is the first time to the best of our knowledge that shape graphs are exploited for improving lock schedulers in integrated database systems.

4.2 Overview of the Approach

Our lock scheduling techniques that we introduce later in Section 5 rely on the estimation of a) the set of objects to be accessed by a transaction and the order of those accesses, b) whether the access is read or write, and c) the execution time of a transaction during which a given object is accessed. In this section, we describe how this information is obtained by using shape analysis.

Shape graphs that capture the way in which the program's code accesses data, allow us to follow the same datapaths that the original program would take in order to access the data effectively predicting its future access pattern. We cannot follow the exact path, as the code surely has data-dependent branches. However, we capture a unified view of all program paths in the program's shape graph. While this results in a necessarily conservative estimation (a superset which may be several times larger than the actual set of accessed objects), it is incomparably smaller than the entire database, which can be exploited to devise efficient lock schedulers.

Our implementation of shape analysis consists of two components: compile-time construction of a shape graph and runtime prediction, which uses the program's access pattern represented by the shape graph and the actual object graph contained in the OODB to generate the estimated set of future accesses and other required information. It is important to emphasize that deployment of these components does not require rewriting the existing database programs: the construction process can be coupled with the standard compilation process whereas the runtime predictor can be integrated with the scheduler of existing OODB systems in a way that is transparent for the application. Since shape graphs are small even for large programs, their storage does not require a lot of resources and their runtime traversal is computationally effective. Sections 4.3 and 4.4 provide further information about the shape analysis implementation.

In this paper, we also define the type–shape graph which is the reduction of the shape graph to track the types and the access order of the types that are manipulated by the program. The type–shape graph is used to determine static type properties of a transaction. The type–shape graph reduction is one-way in that given a shape-graph, we can construct a type–shape graph.

4.3 Compile-time Construction

The variant of shape analysis we are using is a whole-program, flow-insensitive, context sensitive data flow analysis and is similar in design to those presented in [4, 17, 11]. Previously shape analysis has been used to determine static properties of programs that manipulate heap data structures. In this paper, we take a novel approach examining how the results of program analysis can be combined with an active runtime to increase runtime efficiency.

Shape graphs are created and extended by simulating the actions of the program through abstract interpretation, which creates and connects abstract heap tuples. Simple program actions, such as a field access instruction, create heap tuples. When two variables are determined to point to the same abstract location, we unify their heap representations. Unification is a recursive operation that begins with unifying the abstract locations and continues by unifying the compatible heap tuples that stem from the originally unified location. Heap tuples are compatible when two abstract locations have similarly labelled incoming shape edges. Given two abstract locations, that are to be unified, we first unify their abstract locations and then recursively unify their compatible tuples in the heap.

The construction of the shape graph in Figure 1 begins as follows. The reference to the `part` in line 2 creates an abstract variable in the symbolic interpretation of the program. Line 3 creates the link from the `part` abstract location to some unspecified location (`AL`) when the field `material` is accessed. Similarly, line 5 creates both the abstract location `connector` and the edge `right` between them when interpreted. Finally, in line 7 the edge `partB` is created due to the field access and the resulting abstract location is unified to the original `part` because of the assignment. Line 4 contains a call to a sub-method. The analysis of the sub-method would be similar to the one described above. After both methods have been analyzed, the local parameters passed to the method are unified with the formal parameters by the process described below. Program actions causing unification are summarized in Table 1.

Method calls are combined in a bottom-to-top fashion. The static call-graph is used to drive the entire interprocedural analysis. The call-graph is partitioned into strongly connected components (SCC), then topologically sorted. The method contexts (locals, globals, return value, and exceptions) for each method are propagated bottom-up through all possible call sites. The shape graphs are propagated from callee to caller during this phase through the unification of shape graphs. This allows the analysis to be context-sensitive as the caller's shape information is not mixed into callee. We lose this sensitivity for methods belonging to the same SCC (mutually recursive methods) as all methods will share a single shape context [17]. In many cases the actual method receiver cannot be determined at compile time and this is a cause of uncertainty in the graph. Rapid Type Analysis [2] is applied to each call site in order to reduce the number of possible targets for each call site. For each target, the actual parameters are unified with a copy of callee method context in the caller's method context. As an example, the method call `part.volume()` in line 4 of Figure 1 generates a sub-shape graph based on the type of `part` at runtime. This sub-shape graph is merged into the caller's context at the call point. Since we cannot determine at compile time which runtime type `part` will have, we must unify all shape graphs from the target set.

Table 1. Statements causing unification of shape graphs and their effect. The fields *array*, *formal* and *return* are special fields for array reference, method local and return values respectively.

Statement	Abstract Location	Description
`x = y`	unify$(AL(x), AL(y))$	Assignment
`x.`*field*`=y`, `y=x.`*field*	unify$(AL(x)$.*field*, $AL(y))$	Field assignment
`x = a[i], a[i] = x`	unify$(AL(x), AL(a)$.array$)$	Array assignment
`return x`	unify$(AL(x), AL(m)$.return$)$	Function return
`v = f(`$a_1, ..., a_n$`)`	$\forall_{t\in\text{target}(f)}$unify$(AL(a_i), AL(t)$.formal$(i))$ $\forall_{t\in\text{target}(f)}$unify$(AL(v), AL(t)$.return$)$	Invocation
`x = new T`	$AL(x) = \emptyset$	Allocation

During shape analysis, we decorate the shape graph with attributes depending upon how the shape graph will be used. For example, a simple extension to the basic shape analysis described above labels each edge with `r/w` value depending if the resulting abstract location is the result a field read or field write operation respectively. During unification, a read operation unified with a write results in a write otherwise the attribute remains the same. This information can be used to statically determine whether the element could ever be the target of a write operation.

We also label each shape edge with the count of the first and last access instruction. This value is calculated by examining the basic blocks of the transaction and finding the minimum/maximum number of the instructions over all paths needed to access some abstract location. This value will be used to determine the object access order and assist in determining the expected execution time as explained in the following section.

Type-shape graphs are constructed by merging edges of a shape graph. Edges on which the end points have compatible types may be merged. During the analysis, abstract location (shape edge endpoints) are labelled with the set of types that they refer to in the actual program source. Compatible abstract locations are those that have a common super type in the class hierarchy.

The analysis must create shape graphs for the entire program as described above. However, we need to store only those shapes that will be useful at runtime. At a minimum, we must store an entire graph for each top-level variable in the transaction, in which case the predictor will run once before the beginning of the transaction. Further graphs may be stored depending on how often the predictive process will be used during the transaction. Each additional run will refine the prediction results but impose certain runtime overhead.

The shape graphs themselves may be stored either on the client or on the database server. The shape graphs are quite small (usually no more than several hundred nodes per transaction) and need to be communicated at most once during the entire client session.

4.4 Runtime Use

The runtime system can be triggered in a variety of ways to perform the actual prediction: either through programmer annotations or through automatic identification and instrumentation of transaction routines.

Upon entrance, the runtime interprets the shape graph over the actual program data generating the set of objects used by program. The runtime algorithm produces the future accessed objects based on the shapes extracted from the program. Along with each object to be accessed it also produces whether the object can be the target of a write operation, the expected order in which the objects will be accessed and finally the time the algorithm needs to compute while accessing the objects.

Before a transaction starts, we can follow the associated shape graph to generate an unordered set of the possibly accessed objects in the database. Given a transaction root object and the program point's associated shape graph, we generate all actual objects that *might* be accessed during the transaction. Each shape graph represents how the transaction will manipulate structures referred to in the future by its visible references (the object, arguments, globals) in the transaction body.

In our system, the database is responsible for interpreting the shape graphs of the client. Upon receiving a transaction request, the server will walk the shape graph with a real object database object. This effectively simulates all possible program paths taken during the transaction over the database.

```
// Input:  An initial object o
//         A shape graph sg)
// Return: A set of accessed objects
List DetermineObjects(Object o, ShapeGraph sg)
 Queue search  // Tuples of form (object, abstract node)
 Set   objects  // Set of objects found
 Set   seen     // Tuples (object, abstract node) already visited
 push (o, root (sg)) on search;
 while not empty search
   (o, rv) = pop search;
   if (o, rv) not in seen
     seen = seen ∪ (o, rv);
     for each edge e in adjacent edges of rv
       next = read field e.toNode of object o;
       push (next, rv.e.toNode) on search;
       objects = objects ∪ next;
 return objects;
```

Fig. 2. Algorithm to determine objects using shape graph and object graph

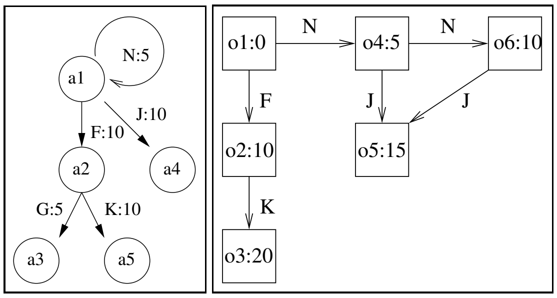

Fig. 3. Shape graph and object graph: Objects are linked through fields and have been labelled with expected access time using shape graph.

Our pseudo-code for walking the object graph is shown in Figure 2. The algorithm traverses the object graph based on program's field accesses represented by the shape graph. We search through the object graph in a breadth-first manner based on earliest expected access through the field. Thus, the runtime cost of prediction equals the cost of interpreting the shape graph over the input object graph.

The prediction walk is usually only done at the beginning of the transaction. However, the runtime system is capable of re-running the walk if further precision is desired and the associated shape graphs are available as discussed in Section 4.3.

The basic algorithm outlined above returns a unordered set of possibly accessed objects. We have extended the above algorithm in several ways to:

Determine R/W attributes As described in the Section 4.3, a read/write attribute was added to each shape edge depending on whether the edge was created by a read/write operation in the source text. The runtime algorithm was modified to track these edges and return whether, during the transaction, an object *might* incur a write.

While visiting a shape node-object pair, the algorithm is allowed to visit all outgoing shape edges. If any edge had a write attribute, then the object was the target of a possible write and was labelled as such.

Determine access order We modified the basic shape graph construction algorithm to label each shape edge with a minimum number of basic instructions passed through while reaching the shape node. During unification, we found the minimum of instructions over all paths before reaching a particular node.

While collecting the accessed objects at runtime, we maintain the number of instructions the code would take to reach the object. Each edge taken increases the total number of instructions needed to reach the object. The runtime algorithm was also modified to use a priority queue in order to maintain a sorted list of objects in expected access order.

For example, Figure 3 has a shape graph with each abstract location node labelled with the count of the earliest access instruction. On the right, we show a database object graph. The database objects (`o1..o6`) have been labelled with their expected first access time. In the example, Object `o5` was reachable both through `NNJ` with an instruction count of 20, or simply `NJ` with a count of 15. Note that it is possible that two objects may have the same first access time. This occurs when a data dependent branch in the program code is merged together in the shape graph.

Determine expected execution time We measure the instructions contained in the basic-blocks creating the longest path between field accesses. By estimating the instruction time for these instruction sequences, we can arrive at an expected time of computation between accesses.

The object finding algorithm is modified again to keep track of the maximum number of instructions to be executed during the navigation of the data structure. After visiting the data structure guided by the shape graph, we estimate the number of instructions to completely execute the transaction. This technique gives a conservative measure of the total time needed to execute the transaction by summing over all program paths. Currently we model only execution time and do not take into account I/O costs. We believe this is not too strict a limitation, as in this case our target platform will have gathered the expected objects into local memory.

In each case the graph generated at compile time will be used during the runtime *on the actual data* to providing the runtime with knowledge of how the program will act in the future. The shape graph was annotated with aspects of the analyzed program which would be useful to the runtime system. In our next section we discuss these methods in detail.

5 Predictive Schedulers

The problem of transaction and lock scheduling is the most fundamental problem in concurrency control in databases that attracted a vast amount of research [3, 16, 19].

Yet, little work has been done on predictive schedulers. This is mainly due to the fact that eliciting and collecting information about future transactions is a non-trivial task, especially if this has to be done in a generic way that is not tailored to any specific application.

Shape analysis is of great aid here as it can provide information about the future execution and needs of the currently running transactions. Yet, exploiting this information is far from trivial. This is because finding the optimal schedule is known to be NP-hard even for the offline version of the problem when all events (i.e., transactions) in the system are known in advance [16]. Additionally, there is no general-case online algorithm that would approximate the optimal offline solution within some constant bound.

In this paper, we seek not to devise a completely new scheduler but rather to enhance commonly used schedulers, such as 2PL [3] by taking advantage of the partial future information that is provided by the shape analysis. Our approach is to augment existing OODBs with the prediction mechanisms of Section 4 and the scheduler extensions presented below. Specifically, we propose three separate enhancements of the the 2PL scheduler: deadlock handling, early lock release, and adaptive preclaiming. It is important to emphasize that while presented separately in this paper for the sake of clarity, in practice they are integrated into the same scheduler.

These techniques are particularly effective when locks are coarse-grain because for fine-grain locking, the runtime overhead of bookkeeping is high and only a small fraction of the database is locked, problems such as data contention are rare, which does not leave much to improve upon. However, fine-grain locks are unusual in practice because most object-oriented database systems group objects into pages and assign locks on per-page rather than per-object basis. In the presentation, we assume for the sake of clarity that each object has an associated lock. Yet, all the techniques that we discuss at the level of individual objects can be applied at the level of object pages.

5.1 Interaction between the Program, Scheduler, and Shape Analysis

Transparency is an important goal in the design of integrated database systems as the programmer would rather avoid learning a new programming model and rewriting existing database programs. In the method we propose the programmer only has to annotate the program with statements indicating the beginning and end of each transaction. Typically, such statements already exist in an OODB program so that no programmer's effort is required at all. This information is used by both the shape analysis, as described in Section 4.3, and the scheduler.

All other information about the objects, transactions, and locks can be derived from the program automatically. In particular, there is no special program interface for releasing locks. This is important because database systems that are enforcing some level of transaction isolation, do not trust applications to release locks. Yet, some additional annotations may turn useful, e.g., to indicate that there is no need to acquire a lock for a particular object and to account for this object in shape analysis. Since such (possibly useful) optimizations are not essential for the methodology we have developed, we do not consider them in this paper.

The interaction between the scheduler and shape analysis is somewhat more complicated even though this complexity is hidden from the programmer. To start with, they need to agree about unique identifiers for transactions and transaction types. Essentially, the runtime shape analysis has to convey the information about future object accesses of a transaction to the scheduler. This is done by invoking the FUTUREACCESSES method provided by the scheduler's API at least once for each transaction, when the transaction begins. It is possible that as the transaction proceeds, the shape analysis will have more precise information about future accesses and it will notify the scheduler by invoking the FUTUREACCESSES method again. The number of such invocations depends on the granularity of shape analysis as discussed in Section 4.3.

5.2 Deadlock Handling

In two-phase locking and other similar locking protocols, transactions need to wait when requested locks cannot be granted immediately. Thus, a set of transactions, each holding some locks and requesting an additional one, may end up being mutually blocked. Such cyclic wait situations are commonly known as *deadlocks*. There are several extensions of the basic two-phase locking protocol for handling deadlocks; those can be broadly divided into two categories: deadlock detection and deadlock prevention techniques. We now briefly describe the techniques and show how program analysis can be used to enhance them.

Deadlock detection approaches attempt to detect the deadlock situation if it occurs and then to break the cycle by aborting one or more transactions. The detection algorithms are generally based on the notion of a *waits-for graph* (WFG), which is a graph $G = (V, E)$ whose nodes are the active transactions, and in which an edge of the form (t_i, t_j) indicates that t_i waits for t_j to release a lock that it needs. There is a deadlock in the execution if and only if there is a cycle in WFG.

Maintaining WFG throughout the execution is considered expensive, which was the motivation for alternative deadlock prevention methods. These methods do not explicitly maintain WFG but rather detect "dangerous" situations that can possibly lead to a future deadlock and abort at least one of the conflicting transactions based on some heuristics, such as *wait-die* or *wound-wait*. Situations are identified as dangerous in an efficient but simple-minded way: for example, if there is a conflict between a pair of transactions and one transaction has a smaller identifier than the second one. Such deadlock prevention strategies impose a smaller overhead of additional testing operations compared with deadlock detection but may cause significantly more transaction aborts, many of which are in states that do not lead to deadlock.

If all locks that are needed for a transaction are known in advance when the transaction starts, it is possible to achieve almost "perfect deadlock prevention" that avoids aborts altogether without sacrificing concurrency. One such method is based on the notion of *resource-allocation* graph, in which both currently executing transactions and currently assigned locks are vertices. A directed edge from transaction T_i to lock L_i implies that either T_i waits for L_i or T_i will request L_i in the future. A directed edge from lock L_i to transaction T_i implies that L_i is being held by T_i. The lock scheduler maintains this graph and uses it as follows: a transaction T_i that requests L_i waits as long as granting L_i to T_i would create a cycle in the resource-allocation graph.

It should be noted that while this method is considered important from theoretical point of view, it is never used in practice, mainly because it is difficult to elicit information about the locks that a given transaction is going to request in the future. Another reason is that resource-allocation graph can have several times as many nodes as WFG and significantly more edges due to accounting for future lock requests. Thus, maintaining this graph and detecting cycles in it is considered too expensive.

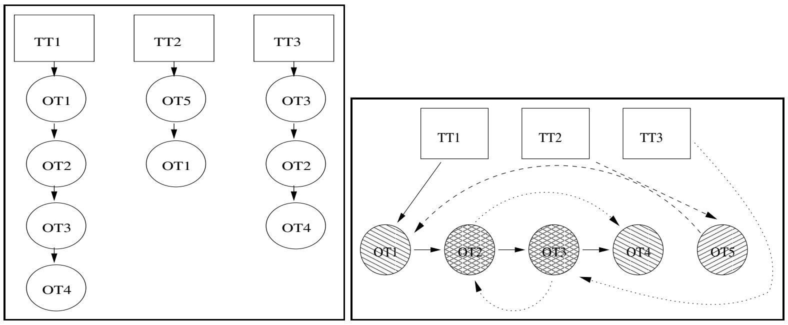

Fig. 4. Transaction object lock order and type graph.

Information obtained from program analysis as described in Section 4, can be used to facilitate deadlock handling in several ways. First of all, when constructing WFG, we can efficiently identify and prune dependencies that cannot be part of the deadlock cycle based on the type information. In this way, we reduce the size of WFG and make cycle detection faster, thereby eliminating the major deficiency of deadlock detection approaches. To illustrate how our technique works, let us consider the following example: transaction of type TT_1 first locks an object of type OT_1, then an object of type OT_2, an object of type OT_3 and finally an object of type OT_4. Transaction of type TT_2 locks objects of type OT_5 and OT_1 in this order. Transaction of type TT_3 locks objects of type OT_3, OT_2, and OT_4 (see Fig. 4). Observe that a cycle in WFG can be created only by transactions of types TT_1 and TT_3, and only due to waits on objects of types OT_2 and OT_3. Thus, transactions of type TT_2 and dependencies between transactions of TT_1 and TT_3 due to objects of type OT_4 do not need to be inserted into WFG.

To capture this intuition, we make use of the type shape graphs described in Section 4.2. Section 4.3 explains how to construct a type shape graph for each transaction type. In order to facilitate deadlock detection, graphs from different transaction types are merged into a single graph in the following way: all nodes in the graphs that correspond to the same object type are combined into a single node. For example, Figure 4 shows the merged graph for the example above.

Our methodology for pruning WFG nodes and edges is based on the following theorem:

Theorem 1. *If transactions $T_1, T_2, \ldots, T_n$ create a cycle in WFG at runtime, then the static object types that belong to the type shape graphs of transaction types $Type(T_1)$, $Type(T_2), \ldots, Type(T_n)$ create a cycle in the merged type shape graph. Furthermore, every edge that is part in the WFG cycle is due to wait on the object whose type is a node in the type shape graph cycle.*

It should be emphasized that while this technique is very general, it is inherently conservative due to being based on purely static compile-time program analysis. In other words, more scrupulous and dynamic analysis could prune more parts of WFG. We can lose precision at several stages: when the function that represents the transaction has many branches (as explained in Section 4.3) and when reducing an object shape graph to a type shape graph. Yet, this methodology can significantly decrease the size of WFG in many existing applications because most transactions traverse objects in the same order of types. For example, in the OO7 application described in Section 6, a transaction never accesses a low level construct called atomic part before accessing a higher level composite part which contains the same atomic part. Yet, we improve this methodology further by complementing it with runtime analysis that takes into account the dynamic information about the execution.

Program analysis not only facilitates deadlock detection but it also makes perfect deadlock prevention feasible. Specifically, having the information about future object accesses as described in Section 4.4 allows us to set up the edges in the resource-allocation graph that represent future lock requests. Admittedly, this method may sometimes yield a conservative estimate because shape analysis can deduce only a superset of the actual objects to be accessed. However, once created at the beginning of a transaction, future request edges can be incrementally removed as transaction proceeds and more precise knowledge about transaction execution is gained. The interaction between the lock scheduler and shape analysis as described in Section 5.1, allows such incremental updates.

Program analysis can also make algorithms based on the resource-allocation graph more efficient: similarly to WFG, we can reduce the size of the resource-allocation graph by using the information extracted from type shape graphs. Finally, we can use hybrid deadlock detection-prevention schemes. For example, we can use deadlock detection as long as the deadlock rate is low and switch to deadlock prevention if the deadlock rate exceeds a predefined threshold.

5.3 Early Lock Release

All of the existing variations of the classical two-phase commit protocol can be classified as strict or non-strict. In strict protocols, all locks are held until the end of transaction, while in non-strict protocols, locks can be released if the transaction no longer needs to access the object [3]. It is generally considered that an early release of a write lock may pose a problem because other transactions may obtain such a lock and read the new object value that has not been committed yet and may never be committed in the case of an aborted or failed transaction. However, early release of a read lock is highly desirable as it makes the object accessible to other transactions and improves the concurrency of the execution.

Yet, most practical systems are using strict protocols because implementing an early release of read locks is far from straightforward. The main reason for this is the challenge in detecting that the transaction has finished accessing the object. In order to perform such a detection without requiring the programmer to add annotations, the scheduler has to predict future object accesses by the transaction. Note that this is exactly where the shape analysis proves useful as we discussed in Section 4.4.

Another problem may arise if a transaction T_1 unlocks an object O_1 and then acquires a lock for another object O_2: if another transaction acquires write locks for both O_1 and O_2 and commits between the two operations of T_1, the two transactions cannot be serialized. To address this issue, the classical non-strict two-phase locking acquires all locks that the transaction requires prior to releasing the locks that are no longer needed. Again, this might require the scheduler to predict the future accesses of a transaction. Furthermore, preclaiming of locks (i.e., acquiring all the locks up front at the beginning of a transaction) can hurt the concurrency, especially if the transaction is long (see Section 5.4 for a discussion of preclaiming).

To eliminate the need of preclaiming, *altruistic locking* has been proposed [18]. Informally speaking, the general idea behind altruistic locking is that if a transaction T_1 releases a lock for O_1, then any other transaction T_2 that acquires a lock for O_1 before T_1 terminates, can acquire only locks released by T_1.[2] The rationale here is to prevent T_2 from accessing an object that may be required by T_1 in the future. However, altruistic locking is still conservative because an access of T_2 to an object that has not been released by T_1 does not necessarily lead to a problem.

It may appear that simply disallowing T_2 to access any object that may be required by T_1 in the future will solve the problem. Unfortunately, this is not the case: if T_2 modifies O_2 and then another transaction T_3 accesses first O_2 and then another object that is required by T_1 in the future, the execution is not serializable.

In this work, we propose a solution based on the notion of *causal dependency* [12]: transaction T_1 causally precedes transaction T_2 (denoted as $T_1 \stackrel{\text{hb}}{\rightarrow} T_2$ if either a) T_2 is initiated after T_1 by the same client, or b) T_2 acquires a lock that T_1 has released, or c) there is another transaction T_3 such that $T_1 \stackrel{\text{hb}}{\rightarrow} T_2$. Our *causality-aware* scheduler is the standard non-strict two-phase locking with the following extension: it precludes the situation when there are two transactions T_1 and T_2 such that $T_1 \stackrel{\text{hb}}{\rightarrow} T_2$ and T_2 holds a lock for O_1 that may be requested by T_1 in the future. If a transaction requests a lock and granting the lock may lead to such a situation, the request is blocked until T_1 acquires a lock on O_1 or terminates.

Theorem 2. *Causality-aware scheduler generates only executions that are one-copy serializable.*

Techniques for tracking causality, such as assigning increasing logical timestamps to transactions, are well known and have been extensively studied through the literature. However, the application of the knowledge of causal dependencies for locking schedulers appears to be new. Furthermore, by using type shape graphs we can exclude from our consideration some transaction types like we did for deadlock handling as presented in Section 5.2. We discuss the performance of the proposed scheme in Section 6.2.

[2] More precisely, altruistic locking extends the scheduler interface by adding a "donate" operation. This operation signifies that the transaction does not need the object any longer while the actual unlocking is done at the end of the transaction.

5.4 Adaptive Lock Preclaiming

While the standard two-phase locking protocol acquires a lock when the object is accessed for the first time, there is a group of schedulers called *conservative* two-phase locking [3, 19] that *preclaim* all potentially required locks up front when the execution of a transaction begins. [3] explores the tradeoff between the two approaches. In summary, gradual lock acquisition works better when data contention is low and transactions are long whereas preclaiming is more suitable for the applications with high data contention and short transactions.

Note that preclaiming requires the knowledge of future accesses which can be obtained only by programmer annotations or tools like shape analysis. Furthermore, advanced knowledge of future accesses allows us to devise adaptive hybrid schemes. By using shape analysis we can estimate the future data contention level across the transactions that have already started and decide whether to use conservative or standard two-phase locking. Furthermore, predicting execution times makes it possible to preclaim locks for short transactions but assign locks gradually to longer ones.

Finally, by using shape analysis combined with the information about the execution history, we can sort objects by their popularity, i.e., the degree of concurrency in accessing the object. Observe that popular objects are typically accessed for a shorter time. This provides the rationale for preclaiming: If a transaction first accesses a popular object O_1, and then a less popular object O_2, it will be more efficient to acquire locks on the both objects simultaneously. The full algorithm is presented in Figure 5.

Upon starting a transaction
  $S =$ set of all objects that will be locked with probability $> \alpha$
  sort S by object popularity in the ascending order

Upon requesting a lock for object $S[i]$
  verify that preclaiming will not create a possibility for an induced deadlock
  acquire locks for all objects $S[1], \ldots, S[i]$ that have not been acquired yet

Upon raising the probability of a lock L not in S
  recalculate S and acquire L if necessary and does not create a possibility for a deadlock

Fig. 5. Adaptive lock preclaiming based on object popularities

6 Experiments

We have developed a simulation test-bed in order to test the performance of the techniques described in Section 5 and their effect on data contention. Using the same simulation technique, we ran experiments with two different applications: a prototype of a car reservation system that we developed, and the standard OO7 benchmark [6] for object-oriented databases. In order to perform tests for varying data contention conditions, we have designed a synthetic workload generator that produces a sequence of operations to be invoked at certain times along with the parameters to be passed to those operations. We have also implemented an artificial client that replays a previously generated sequence of operations. The main venue of our experiments was to compare the predictive lock scheduler with the standard schedulers. To this end, we have implemented a strict two-phase locking (S2PL) and optimistic schedulers. The overall test-bed architecture is depicted in Figure 6.

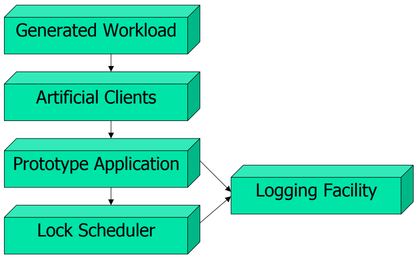

Fig. 6. The testbed implementation

6.1 Prototype of a Car Reservation System

Consider a database system for online car reservations that can be placed through Web requests. For the sake of an example, assume that the database contains three different parts: a large partition of reservations (A), a large partition of available cars (B), and the rest of data that contains, e.g., rental rules and terms (C). There are three different types of transactions: Frequent user transactions update already existing individual reservations and place new ones. To this end, they need to lock A. There are also infrequent traversing and maintenance transactions. A maintenance transaction updates the maintenance information for the fleet of cars owned by the company, thereby locking B for a very long time. A traversing transaction computes a tentative assignment of available cars to existing reservations. This transaction locks C, then A, and finally B. The following problem can occur if the scheduler incrementally assigns locks by only looking at the currently granted ones: A traversing transaction obtains locks for C and A. Then a maintenance transaction starts and it is granted a lock for B. The traversing transaction now cannot obtain a lock for B and it has to wait until the maintenance transaction terminates. Meanwhile, many user transactions are blocked because they cannot access A. If the scheduler knew to take future events into account, it would delay the maintenance transaction access to B in order to let the traversing transaction finish and release the lock it holds on A.

We defined three different sets of parameters for the purpose of testing. These parameters determine the frequency and the duration of locks for each transaction. The values of these parameters for each of the configurations are given in Table 2.

We repeatedly ran a simulation of our prototype 100 times for each configuration. Tables 3 and 4 summarize the results of our experiments. Table 3 shows execution times

Table 2. Parameters used in different experiments

	Configuration 1	Configuration 2	Configuration 3
User transaction rate	*300/sec*	*30/sec*	*30/sec*
User transaction duration	*30ms*	*30ms*	*30ms*
Mainten. transaction duration	*60000ms*	*2000ms*	*2000ms*
Traversing transaction in A	*30ms*	*30ms*	*200ms*
Traversing transaction in B	*30ms*	*30ms*	*500ms*
Traversing transaction in C	*500ms*	*500ms*	*500ms*
Traversing transaction period	*1min*	*1min*	*1min*
Mainten. transaction period	*6min*	*2min*	*2min*

in milliseconds for the S2PL, predictive, and optimistic schedulers as well as the proper execution time of the transactions. The predictive scheduler significantly outperforms the S2PL scheduler for the first configuration while being slightly better for the second and third configuration. This improvement is solely due to the adaptive preclaiming technique described in Section 5.4 because no lock can be released early in this application. The optimistic scheduler did not perform well in our experiments because of the high number of conflicts. For all configurations, the traversing transaction was either completely starved or took a very long time to complete. Furthermore, the abort rate was high for the optimistic scheduler as shown in Table 4.

Table 3. Comparison of transaction execution times in various configurations and for different schedulers

	Configuration 1			Configuration 2			Configuration 3		
	Traversal	User	Mainten.	Traversal	User	Mainten.	Traversal	User	Mainten.
Execution	560	30	60000	560	30	2000	1200	30	2000
S2PL	7577	3423	60024	860	43	2010	1241	43	2010
Predictive	6829	43	60024	876	40	2010	1242	43	2010
Optimistic	starved	54	60024	4327	40	2010	starved	40	2010

Table 4. Comparison of transaction abort rates in various configurations and for different schedulers

	Configuration 1			Configuration 2			Configuration 3		
	Traversal	User	Mainten.	Traversal	User	Mainten.	Traversal	User	Mainten.
Optimistic	many	15	0	6.33	0.07	0	many	0.07	0

6.2 The OO7 Benchmarks

Several standard benchmark sets for object-oriented databases, such as OO1 [7], and OO7 [6], have been designed to facilitate the testing of experimental database design techniques in realistic settings. We opted to conduct our experiments with the OO7 benchmark since it is the most complex in terms of the database structure and supported operation set. Additionally, the OO7 benchmark exhibits a rich object structure that lends itself well to program shape analysis. Since the original OO7 benchmark were written in C++ while our shape analysis implementation works for Java, we extended a Java port of the OO7 benchmarks [1].

The OO7 Database In this section, we summarize those details of the OO7 database description in [6] that are relevant to our experiments. The benchmark models a database for CAD/CAM/CASE applications. A key component of the database is a set of *composite parts*. Each composite part corresponds to a design primitive such as a register cell in a VLSI CAD application.

At a lower level, each composite part has an associated graph of *atomic parts*. Intuitively, the atomic parts within a composite part are the units out of which the composite part is constructed. For example, if a composite part corresponds to a procedure in a CASE application, each of the atomic parts in its associated graph might correspond to a variable, statement, or expression in the procedure. One atomic part in each composite part's graph is designated as the "root part."

Composite parts are grouped into *base assemblies*.[3] For example, in a VLSI CAD application, an assembly might correspond to the design for a register file or an ALU. Base assemblies are further grouped into *complex assemblies*, which can be part of upper level complex assemblies. Cycles between assemblies are not allowed. Therefore, the overall organization can be visualized as a set of assembly hierarchies, each hierarchy being called a *module*.

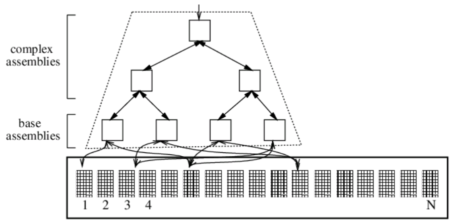

Fig. 7. OO7 module structure

Figure 7 depicts the full structure of an OO7 module. The hierarchy scale is configurable with respect to several parameters. In our experiments, we worked with a single module because all operations provided by the benchmarks act on a single module so that having multiple modules does not create any interesting concurrency issues. Other relevant parameters are the number of composite parts per module, the number of composite parts per base assembly, the number of levels in the assembly hierarchy, and the

[3] Some parts may be shared across multiple assemblies while other parts may belong to a single assembly. This is unlike atomic parts that are never shared by multiple composite parts.

number of child assemblies per complex assembly. Since the number of children nodes (child assemblies or composite parts) per assembly has a great impact on the degree of data contention, we varied it in our experiments while the other two parameters were fixed. Table 5 summarizes the values of these parameters that were used in the experiments.

Table 5. OO7 database configuration

# composite parts per module	500
# composite parts per base assembly	3–27
# levels in the hierarchy	3
# assemblies per complex assembly	1–5

Each object in the database has a number of attributes, including the integer attributes `id` and `buildDate`. `id` is a distinct identifier assigned to each entity to distinguish it from other entities while `buildDate` specifies the last time when this part or assembly was modified.

Benchmark Operations and Their Concurrency Patterns The designers of the OO7 benchmarks provided a rich set of operations to manipulate the database. However, many of these operations are equivalent as far as concurrency goes. For example, a search for a composite part by any of two different attributes takes about the same time and requires the same locks. In fact, concurrency was not the focus of the OO7 design: each operation as a whole was considered a separate transaction while all lock assignments were handled by the underlying OODBMS. On the contrary, we need to consider the details of lock assignment by the concurrency manager, even if it is transparent for the application. In our experiments, we used the following three operations that represent different operation classes from concurrency standpoint:

- Querying an arbitrary composite part:

  The operation selects a random base assembly and a random composite part which is contained in this assembly. Thus, it needs to acquire a read lock for the base assembly and then a read lock for the part. If the base assembly is accessed directly using some index structure, then no other locks are required. Another way to reach the assembly is to traverse the assembly hierarchy from the root choosing a child assembly at each node by some arbitrary criteria. In this case, the operation also requires read locks for all assemblies on the path from the root to the base assembly of interest.
- Reorganizing an arbitrary composite part:

  Like the previous operation, this operation selects a random base assembly and a random composite part which is contained in this assembly. Then, it obtains a write lock for the part and performs a long update which involves recreation and reorganization of all atomic parts within this composite part.

– Updating an attribute of several related objects:
  As a typical representative of this class, we took an operation that updates the `buildDate` of a composite part and its parent base assembly. The operation starts with obtaining a read lock for an arbitrary assembly, chooses an arbitrary part of this assembly, obtains a write lock for the assembly and updates it, and finally acquires a write lock for the part and performs an update on it.

The submission rate for the query, reorganization, and update transactions was 10, 600, and 3 transactions per minute, respectively. The execution times were 2ms, 2000ms, and 2ms as shown in Table 6.

Table 6. Comparison of transaction execution times for different schedulers

	Time	S2PL	Early release	Adaptive preclaiming	Both preclaiming and early release
`buildDate` update	2ms	92	48	116	58
Query composite part	2ms	15	9	5	5
Reorganization	2000ms	2041	2018	2010	2010

Performance of Locking Schedulers In Table 6 we compare the average transaction time in milliseconds for strict 2-phase locking (S2PL) and our predictive scheduler. Furthermore, in order to understand the contribution of each individual mechanism, we ran the predictor with only early release activated, adaptive preclaiming activated, and both. As it can be seen, early release of read locks improved the average times of all transactions. In contrast, adaptive preclaiming significantly reduced the times for the query transaction but slightly increased the times for the `buildDate` update transaction which required two locks. This tradeoff is desirable for the OO7 application because short queries are more common than longer updates and the difference in their times is immediately perceived by the user.

Perhaps even more important than the difference in average times is the difference in the standard deviation. Figure 8 shows the times for the short query transaction in a section of a typical execution using a pessimistic strict 2 phase-lock scheduler. We can clearly see sharp peaks when the reorganization transaction blocked the `buildDate` update transaction, which in turn blocked the query transaction. Figure 9 presents execution times of the predictive scheduler for the same section in the same workload. Not only has the overall mean transaction time been significantly reduced but there are also fewer peaks and those peaks are not as high.

Execution Time Prediction In Section 4.4 we outlined our method for determining the expected execution time of a transaction. The precision of estimation is important for the adaptive preclaiming mechanism used in the experiments that we described above. However, the standard OO7 benchmark does not create sufficient diversity in the duration of transactions of the same type. To create a better diversity, we modified

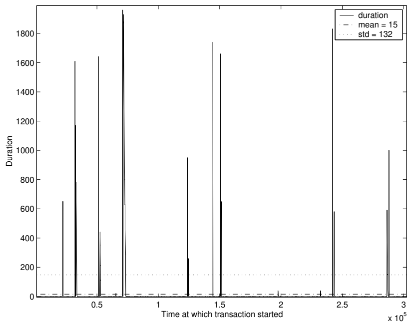

Fig. 8. A typical execution section for the S2PL scheduler

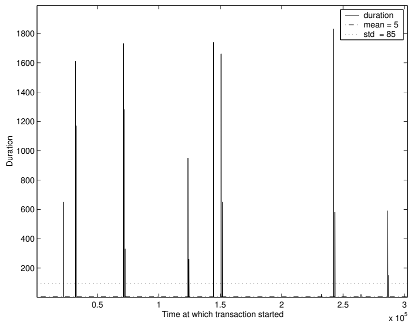

Fig. 9. A typical execution section for the predictive scheduler

the OO7 benchmark to create randomly sized composite parts having between 20 and 1000 atomic parts. We then executed 2187 random search and traversal queries on the database.[4] The traversal query transaction visited a section of the database in order to perform maintenance on a set of atomic parts.

In this experiment, we compared the accuracy of our predicted times to the actual runtime of the transaction. As the predicted times were dependent on the actual hardware used, we were mostly interested in the relationship between the predicted values and the actual execution times.

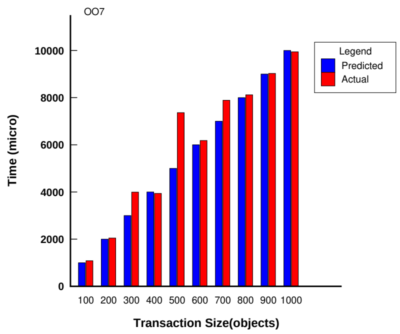

Fig. 10. Predicted time compared to actual execution times

Figure 10 shows our results for the OO7 traversal transaction. The results were sorted by transaction size and the overall times were averaged over 3 runs. In almost all cases the predicted time matches the actual time very closely. However, in two circumstances (300, 500) the times diverged. On closer inspection, several transactions took

[4] Based on the number of Base Assemblies in the benchmark tiny configuration.

over 10 times the median. We believe this to be due to outside operating system issues and beyond our control.

Table 7 summarizes our results. We found a relatively strong correlation (0.64) between the predicted times and the actual run times. This will be true whenever the runtime of the transaction is dependent on navigation cost of the data structure. Again it should be noted that the transaction cost will usually be tied to the number of the objects used by the transaction code and not the number of objects in the database.

Table 7. Predicted time compared to actual execution times

Source	Sample Size	mean	median
Predicted	2187	4945	4800
Actual	2187	5214	4677

Our initial results show this technique to be promising. However, two caveats must be pointed out. First, since the shape graph contains all paths that a program may take, the analysis maybe overly conservative in estimating the total expected time. Secondly, if the majority of the transaction's total work is navigating the structure, the prediction time will be on the order of the execution time. Since our predictor effectively visits the data structure elements in a similar way to the original transaction.

7 Conclusions

We have presented several novel techniques for automatically increasing concurrency in object oriented database systems. In this paper, we have proposed using shape analysis for database programs. Using program analysis we can provide a succinct representation of the future accesses of a program fragment even across dispatched method calls in object oriented programs. Knowledge of the future accesses permits the simpler algorithms for early-lock release, data contention, and deadlock avoidance/detection. We have demonstrated our technique using our own car-reservation simulation and the OO7 benchmark adapted to use multiple clients in order exercise the currency scheduler. Our techniques increased concurrency and have lowered the mean time to complete the workloads.

While we showed that our predictive scheduler is beneficial for the above applications, the power of prediction is bound to be inherently limited. Identifying individual cases when performance can be hurt because of the poor prediction accuracy is part of our future research. In particular, it would be desirable to devise a heuristics that would determine online whether the predictive scheduler mechanisms should be used.

In the future, we plan to investigate execution time prediction and lease scheduling. As the gap between traditional programming languages and database programming languages continues to diminish, applying program analysis to database problems will be a fruitful area of research.

References

1. Ozone oodb. Technical report, www.ozone-db.org, 2001.
2. D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA Object-Oriented Programming Systems, Languages, and Applications*, San Jose, California, Oct 1996. ACM.
3. P. Bernstein, V. Hadzilacos, and H. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.
4. Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *OOPSLA*, pages 35–465, Denver, CO, Nov 1999. ACM.
5. Cahoon and McKinley. Data flow analysis for software prefetching linked data structures in java. In *International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, Sep 2001. ACM.
6. Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):12–21, 1993.
7. R. G. G. Cattell and J. Skeen. Object operations benchmark. *TODS*, 17(1):1–31, 1992.
8. F. Corbera, Rafael Asenjo, and Emilio L. Zapata. New shape analysis techniques for automatic parallelization of C codes. In *International Conference on Supercomputing*, pages 220–227, Rhodes, Greece, Jun 1999. ACM.
9. Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Symposium on Principles of Programming Languages POPL*, pages 1–15, St. Petersburg, Florida, Jan 1996. ACM.
10. Nils Knafla. *Prefetching Techniques for Client/Server, Object-Oriented Database Systems*. PhD thesis, University of Edinburgh, 1999.
11. Kristian Kvilekval and Ambuj Singh. Prefetching for mobile computers using shape graphs. In *LCR 2002*, Washington DC, Mar 2002. ACM.
12. L. Lamport. Time, Clocks and the Ordering of Event in a DistributedSystem. *Communications of the ACM*, 21(7):558–565, 1978.
13. Sun Microsystems. *Java Data Objects*, 2003.
14. Dor Nurit, Rodeh Michael, and Sagiv Mooly. Detecting memory errors via static pointer analysis. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 27–34, New York, NY, Jun 1998. ACM.
15. ODMG. *Object Query Language*, 2003.
16. C. Papadimitriou. *The Theory of Concurrency Control*. Computer Science Press, Rockville, MD, 1986.
17. Erik Ruf. Effective synchronization removal for Java. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI2000),*, Vancouver, British Columbia,, Jun 2000. ACM.
18. K. Salem, H. Garcia-Molina, and J. Shands. Atruistic locking. In *Transactions on Database Systems*. ACM, 1994.
19. Gottfried Vossen, Gerhard Weikum, and Jim Gray (Editor). *Transactional Information Systems: Theory, Algorithms, and Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.
20. R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Proc. of CC 2000: 9th Int. Conf. on Compiler Construction*, Berlin, Germany, Mar 2000. Springer-Verlag.