Applying Method Data Dependence to Transactions in Object Bases[1]

Peter C.J. Graham and Michael E. Zapp and Ken Barker
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada
Canada R3T 2N2
{pgraham,zapp,barker}@cs.umanitoba.ca

Technical Report: 92-07

June 1992

[1]This research was partially supported by the National Science and Engineering Research Council (NSERC) of Canada under an operating grant (OGP-0105566).

Abstract

Concurrency control in object based systems is a new area of research that has only just begun to be addressed. Recent work has proposed using object level locking but this may be unnecessarily restrictive. Data level locking within an object increases concurrency but also introduces substantial lock overhead that makes the approach impractical. This paper proposes using a new approach applying *method level dependence analysis* techniques to object bases using a nested transaction model. Three algorithms are proposed. The first algorithm uses static analysis of methods invoked by *user transactions* only. The second performs runtime analysis of the nested methods invoked by the methods invoked by user transactions. Finally, a third algorithm proposes a hybrid technique that is compatible with both of these techniques and determines if the acquisition of locks is necessary at runtime or if the method can be invoked without the possibility of conflict. This algorithm is efficient, simple, and completely compatible with two–phase locking. The complexity of each algorithm and other areas of application of dependence analysis are also discussed.

1 Introduction

This paper makes an initial examination of the definition and uses of dependence in object bases with nested, atomic, method invocations. It adapts theory developed for optimizing compilers for application in the domain of object bases and provides a cohesive description of how that theory applies. Characteristics of the object base model are exploited to simplify the dependence analysis.

The primary goals of dependence analysis in object bases are:

1. To provide a less costly means of concurrency control than is provided by conventional schemes.
2. To provide less restrictive concurrency control.
3. To do this efficiently by using a combination of static and runtime analysis.
4. To gain insight into the semantics-related issues of dependences between concurrent method executions in an object base.

This paper contributes by providing algorithms for calculating dependence information in object bases and for scheduling user transactions using *method-level* dependence information. Further, some consideration is given to other uses of dependence information in such systems and a brief discussion of extension to other environments is provided. It does *not* directly address the orthogonal issues of deadlock handling and serializability.

Users who are unfamiliar with the basic concepts of dataflow and dependence analysis may wish to consult one or more of [ASU86, Ban88, Wol89, ZC90]. Information on object bases is also widely available [BM91, GK88, Kim90, KGBW90]. Work related to the model is also in the literature [AE92, HH91, RGN90, RE92] as is work on semantic issues [FO89, GM83].

The paper is organized as follows. The next Section discusses the pertinent features of our object base model and is followed in Section 3 by a discussion of the notation used. Section 4 explains how dependence may be defined in object bases. Section 5 presents three algorithms for scheduling transactions using dependence information, the last of which improves on conventional locking schemes by decreasing lock overhead. Section 6 draws some conclusions and discusses future work.

2 The Object Base Model

The following model of object bases and the transactions applied to them is used throughout this paper. The model makes reasonable assumptions and is general enough to allow the results obtained to be applied to other work in the area. A full description of the model is available in Zapp and Barker [ZB92].

The object base consists of a collection of existing objects that interact with one another by method invocations. It is assumed that objects are brought into existence *offline* using techniques not addressed in this paper. Each object maintains its own *persistent* data and is solely responsible for providing access to that data; through its methods interface. The set of all object data represents the contents of the object base. Objects commonly invoke methods in other objects to accomplish their work and this may be done in a directly or indirectly recursive way. No *real-life* assumptions about method invocation are made except to note that programming practice will tend to limit the method invocation patterns. The operations performed by a given object may be conveniently divided into those which access an object's

local data and those operations which "access" non-local data via method invocations on other objects.

Transactions applied against the object base consist of a collection of method invocations and possibly a set of local variable definitions used to store the results of method invocations. Invoked methods and the methods they may directly or indirectly invoke[1] are each executed atomically as *nested atomic transactions* [Mos85]. The collection of invoked nested method invocations from the first user initiated invocation to the final method invoked forms a *call tree*.[2] It is assumed that all transactions are correctly formed and execute on a consistent object base so a serial execution of the method invocations in a given transaction, in the absence of other transactions, will always produce a *new* consistent object base. This does not mean that a transaction's method invocations must be executed serially in the order requested, nor does it mean that only a single transaction may be processed at a time. The assumption is a standard one in database research and also provides a known *correct* execution order which is required for dependence analysis. A natural serial execution ordering also exists for the statements within individual methods.

Finally, it is assumed that a single global scheduler is assigned the responsibility of overseeing the execution of each object method invocation requested as a part of a transaction. This allows a single point of control where the derived dependence information for all transactions may be applied in scheduling their component method invocations. This assumption is reasonable since a global scheduler is also required for the management of the nested commits.

3 Notation

It must be possible to identify specific objects and methods and particular instances of methods arising from their invocations. Furthermore, it must be possible to determine from which transaction a given method was invoked. Therefore, some notation will prove useful.

The object base is assumed to consist of N objects $\{O_i, 0 \leq i \leq N-1\}$. A given object O_i consists of k_i methods $\{M_j, 0 \leq j \leq k_i - 1\}$ each of which consists of a series of statements that access the object's variables and a possibly empty set of method invocations. The j^{th} method of the i^{th} object is denoted $O_i.M_j$.

User transactions are submitted against the object base and are denoted UT^i. Each user transaction consists of a number of calls to methods within objects in the object base and may also include its own local variables.[3]

A method invocation of method j within object i from user transaction k is denoted $O_i.M_j^k$. If necessary, a specific order for a set of method invocations may be specified using the *happens-before* [Lam78] ('$\rightarrow$') relation. (e.g. $O_i.M_r^l \rightarrow O_j.M_s^m \rightarrow O_j.M_t^n$)

This paper adopts the convention of indicating method invocation using a procedure call syntax. Therefore, a method invocation (including those within a user transaction) may accept several parameters and optionally return an explicit result. (e.g. $R = O_i.M_j^k(arg1, arg2, \ldots)$)

When two entities 'E_i' and 'E_j' are found to be dependent, the notation $E_i \delta E_j$ will indicate dependence. In compiler theory, δ is a strictly ordered relation, but in this paper it will be clear from context when it is ordered or unordered.

[1] We adopt the convention that, unless otherwise explicitly stated, all method invocations may be either direct or indirect through an arbitrarily long sequence of other invocations.

[2] The tree induced by the pattern of method invocations resulting from one or more user transactions – analogous to a transaction tree with nested transactions [Elm92].

[3] Local variables in user transactions are used to store the intermediate results of method invocations.

4 The Notion of Dependence in Object Bases

Conservative assumptions must always be made with regard to dependences. If some entity is referenced within a conditional section of code it must be assumed that it is always referenced since it cannot be determined, prior to runtime, if that section of code will be executed. Similarly, if some entity *may* be referenced at the same time by two transactions then, based only on static dependence analysis, they must be serialized.

Dependence occurs at a variety of levels. In compiler construction dependence exists between operations, statements, basic blocks, and entire procedures. In object bases, dependence exists between objects, the methods within them, and references to the data within a given object.

Regardless of the level, dependence information is used primarily to determine a partial (potentially parallel) order in which the corresponding "entities" may be scheduled which will ensure results equivalent to the total (serial) order specified. Effectively, what is done is that potential Read/Write, Write/Read, and Write/Write conflicts between entities are detected and the partial order is constructed so these conflicts are avoided.[4]

There are at least two factors that make dependence analysis interesting in an object base:

- There are natural, hierarchical decompositions to the dependences that arise when methods invoke methods in other objects. Thus, although two objects are distinct (and therefore access different local data), they may be "dependent" if they both invoke conflicting methods within another object.

- Dependences within a single user transaction or object method may be judged according to the specified serial execution order of the corresponding code. It is generally assumed however that multiple user transactions may be scheduled in any serial order. This means that at this level of abstraction, there is no single *inherent* serial order by which a partial order for dependence analysis may be built. The transactions are unrelated by *any* partial order. Unfortunately, this does not mean that they may all be executed in parallel since some may invoke object methods which conflict with those others have invoked. This results in what will be referred to as a *method-level* conflict.

Definition 4.1 *Object-level dependence* occurs between object invocations when either the two invocations are to the same object (trivially) or the object methods in question both invoke methods from a common object. ■

An object-level dependence-based scheduling algorithm would operate at the same granularity as object level locking which is too restrictive.

Definition 4.2 *Method-level dependence* occurs when two methods within an object either access common object data or both invoke dependent methods from a common object. ■

Method-level dependence analysis provides information used in eliminating the situation where two *non-conflicting* concurrent method invocations must wait unnecessarily.

Definition 4.3 *Data-level dependence* occurs between individual data items *within* an object. ■

.

[4] Conflicting entities are related in the partial order and therefore will not be executed concurrently.

Since data is isolated within an object, only methods within that object may contain code which accesses that data. Dependences involving a given data item therefore only occur within the corresponding object. Hence, the execution of methods from different objects may always occur in parallel up until one or both call a method in another object. Since an object base exports only its method interface, all that can be done with data level dependence information is to improve the method level dependence information. This might include recognizing methods that fail to conflict given certain parameter conditions. For example, if some parameter has some value "r", the corresponding method is known to access data in a read-only fashion. Such a method invocation is guaranteed not to conflict with others that only read that data.[5]

These levels represent decreasing granularity and with it, increasing potential for concurrency but also greater overhead in dependence analysis.

5 Method-Level Dependence Analysis

As stated in the introduction, the primary goal of this paper is to examine *method-level* dependence and its use in scheduling transactions using our object base model.

5.1 Method-Level Dependence Within a Single Object

We wish to be able to *statically* examine two methods in some object ($O_i.M_{m_1}$ and $O_i.M_{m_2}$) and determine if their concurrent execution *could* result in a conflict which would produce incorrect results. A conflict can arise in two possible ways:

- The sets of object data referenced by the two methods may be non-disjoint.
- The sets of other methods invoked by the two methods may be non-disjoint. This may result in a conflict since such methods may contain conflicting data accesses.

It is simple to construct these two sets to test for disjointness. We define the following:

$\mathcal{DR}(O_i.M_j)$ *: the names of local data which method* M_j *may access.*

$\mathcal{MR}(O_i.M_j)$ *: the names of methods which* M_j *calls* directly *and those methods in the same objects that conflict with those* M_j *calls directly.*

$\mathcal{MR}^*(O_i.M_j)$ *: the logical transitive closure of* $\mathcal{MR}$ *when it is viewed as a selection function from the domain of all possible object methods.*

The set $\mathcal{DR}$ specifies the direct data references made by a method while the set $\mathcal{MR}^*$ represents the indirect (method) references by enumerating all *sub-methods* invoked. The sets $\mathcal{DR}$ and $\mathcal{MR}^*$ are referred to as a method's *reference* sets.

Using reference sets, the dependence $(O_i.M_{m_1} \delta O_i.M_{m_2})$ between two methods is detected using the test:

$$\mathcal{DR}(O_i.M_{m_1}) \cap \mathcal{DR}(O_i.M_{m_2}) \neq \phi$$

and

$$\mathcal{MR}^*(O_i.M_{m_1}) \cap \mathcal{MR}^*(O_i.M_{m_2}) \neq \phi$$

[5]Such data-level dependence information *can* be used to permit the operations within objects to be performed in parallel on a suitable parallel machine.

It is generally unnecessary to synchronize read accesses to data. Testing for $\mathcal{DR}$ set intersection is thus overly restrictive. It is simple to provide two sets, one specifying local data being read and the other data being written so that intersections between read accesses can be avoided. For the sake of simplicity, we will refer to only the single $\mathcal{DR}$ set. Any method with $\mathcal{MR}(O_i.M_j) = \phi$ will be called a *leaf method*.

The dependency test can be efficiently implemented by mapping elements of each set to positions in a bit string used to compactly represent that set. Presence of the element in a set is indicated by a '1' in the corresponding bit position in the string while a '0' indicates absence of the element. Set intersection is efficiently determined using a bitwise 'AND'.

Since all of the data items within an object are enumerable, this mapping can be easily done for the $\mathcal{DR}$ sets using well-known techniques used in conventional compiler dataflow analysis [ASU86]. Similarly, since all existing object methods in the object base can be enumerated, (and these are *all* the object methods which may be referenced by an instantiated object) the bitstring representing any $\mathcal{MR}^*$ set may also be constructed.

To determine if two methods conflict, the one-way mappings from methods and data-items to bit indices are sufficient. To derive the semantic information that a particular conflict arises due to potentially concurrent accesses to a given data object, the inverse mappings are required so that bit positions left with a '1' after ANDing may be mapped to their corresponding methods/data-items. All these computations on sets are done at *object compile time* and the sets are inexpensively stored along with their associated compiled objects. This means that the set computations introduce no added runtime overhead.

An example of the reference sets corresponding to the methods for the objects shown is illustrated in Figure 1. Each object is depicted as a sectioned box whose first section lists the object's local data and whose subsequent sections contain abstractions of its methods. The arcs in the diagram indicate method invocation paths and the $\mathcal{DR}$ and $\mathcal{MR}^*$ sets for each method in each object are shown. Thus, for example, the $\mathcal{MR}^*$ set for $O_{23}.M_1$ consists of $O_3.M_2$ which is called directly and $O_2.M_3$ which is called indirectly through $O_3.M_2$ and also $O_3.M_1$ and $O_2.M_1$ since they have data conflicts with $O_3.M_2$ and $O_2.M_3$ respectively. The $\mathcal{MR}^*$ set for $O_1.M_1$ is empty making it a leaf method. Finally, the $\mathcal{DR}$ sets for each method contain the set of object-local data that those methods reference.

5.2 Scheduling Using Static Dependence Information

The natural application of dependence information is in scheduling transactions so that those which are not dependent (i.e. do not conflict) may proceed in parallel. Here we consider an algorithm for scheduling non-conflicting method invocations for parallel execution using *only* the statically derived dependence information.

5.2.1 Single Transaction Scheduling

We first consider the problem of scheduling the object method invocations in a single user transaction using the statically derived dependences. The definition of dependence is extended to consider the interactions between methods from different objects. For convenience, those method invocations made *directly* from a user transaction are referred to as user method invocations (UMIs).

Given a pair of UMIs in a user transaction, they are first checked to see if they share a dependence within the user transaction itself (as in the following example where the result of

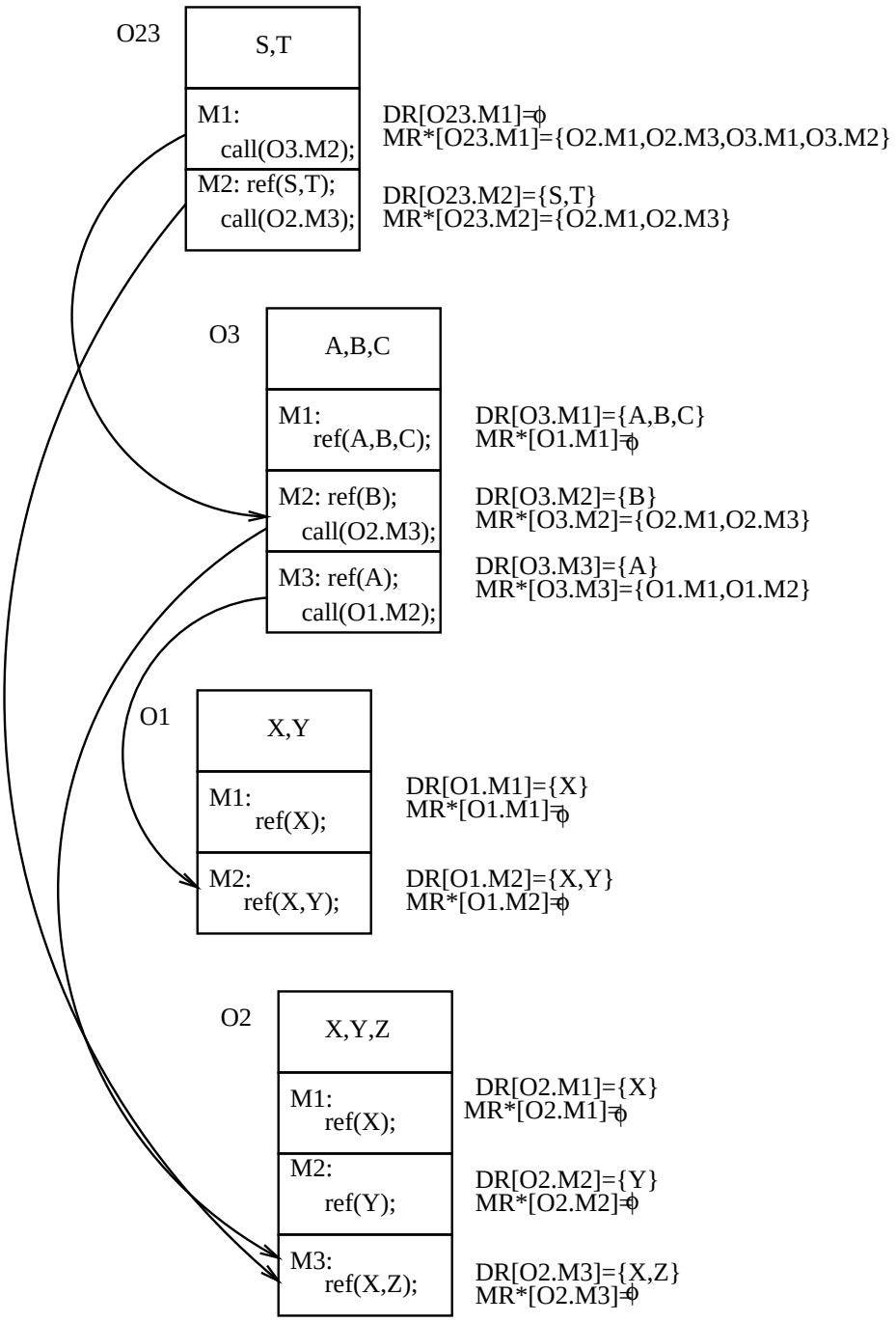

Figure 1: Construction of Reference Sets

one method call is used as an argument to the other).

$$UT^i = \{VAR\ R; \ldots; R = O_j.M^i_{m_j}(); O_k.M^i_{m_k}(R); \ldots\}$$

If they do not share a dependency at this level then their $\mathcal{MR}^*$ sets are checked to see if they are disjoint.

$$\mathcal{MR}^*(O_j.M^i_{m_j}) \cap \mathcal{MR}^*(O_k.M^i_{m_k}) = \phi$$

If they are not, then the method invocations are serialized since they both invoke at least one method which may conflict with a method invoked by the other. If the $METHOD\_SETS$ are disjoint, then the user invoked methods are checked to see if they belong to the same object (i.e. $j = k$). If they do, then the $\mathcal{DR}$ sets must also be checked since both of these methods could be accessing the same object-local data.

$$\mathcal{DR}(O_j.M^i_{m_j}) \cap \mathcal{DR}(O_k.M^i_{m_k}) = \phi$$

If the $\mathcal{DR}$ sets are found to be disjoint then the method invocations may be executed in parallel since they are neither directly nor indirectly dependent. Otherwise, the invocations are dependent and must be serialized.

It is not necessary to analyze the entire set of UMIs before scheduling any of them.[6] We can precompute the partial order defined by the definition-use patterns strictly within the user transaction itself (i.e. arising due to operations on *transaction local* data) and then consider scheduling sets of those UMIs which have no such inter-dependences. Using such a greedy approach the UMIs are initiated as early as possible.

The first UMI may be submitted for execution immediately. The reference sets of each scheduled UMI are recorded by the scheduler. By checking the reference sets of the next UMI to be scheduled against those of the active ones, the scheduler can decide if a conflict is possible and therefore if unrestricted execution of the new UMI is possible. When a conflict is detected, the scheduler can queue the new UMI for execution after the conflicting one has completed its execution. This can be implemented by associating the new UMI with conflicting ones using dependence arcs from the running to the waiting UMIs.

Conflicts may also exist between *queued* UMIs. This must be accounted for by comparing the next UMI with not just the executing but also the waiting ones adding dependence arcs as necessary. When considering two conflicting UMIs the one which first appears in the user transaction must be scheduled first. (This is required to ensure execution results that are equivalent to the sequential execution defined by the user transaction.) When a UMI completes, its reference sets are removed from the scheduler's tables so it no longer blocks subsequent UMIs. At the same time, any UMIs known to be waiting solely upon the completion of that UMI (i.e. those pointed to by a single dependence link from the finished UMI) may be initiated.

Benefits may be realized by not using a greedy scheduling algorithm. If the scheduler *does* examine each UMI before scheduling it, greater potential concurrency may be achieved at the expense of delayed invocation. This approach is more applicable to the multiple transaction case where the invocation delay can be tolerated since work on behalf of other transactions will likely be available to be done.

The use of dependence arcs is illustrated in Figure 2 where arcs reflecting the dependencies between UMIs are shown using dashed lines. A greedy scheduling algorithm is assumed in this example, where UMI_1 is started first, UMI_2 is then queued because of its dependence with UMI_1, UMI_3 is started, etc.

[6] In other words, it is unnecessary to *pre-compute* the dependence relationships for all pairs of UMIs.

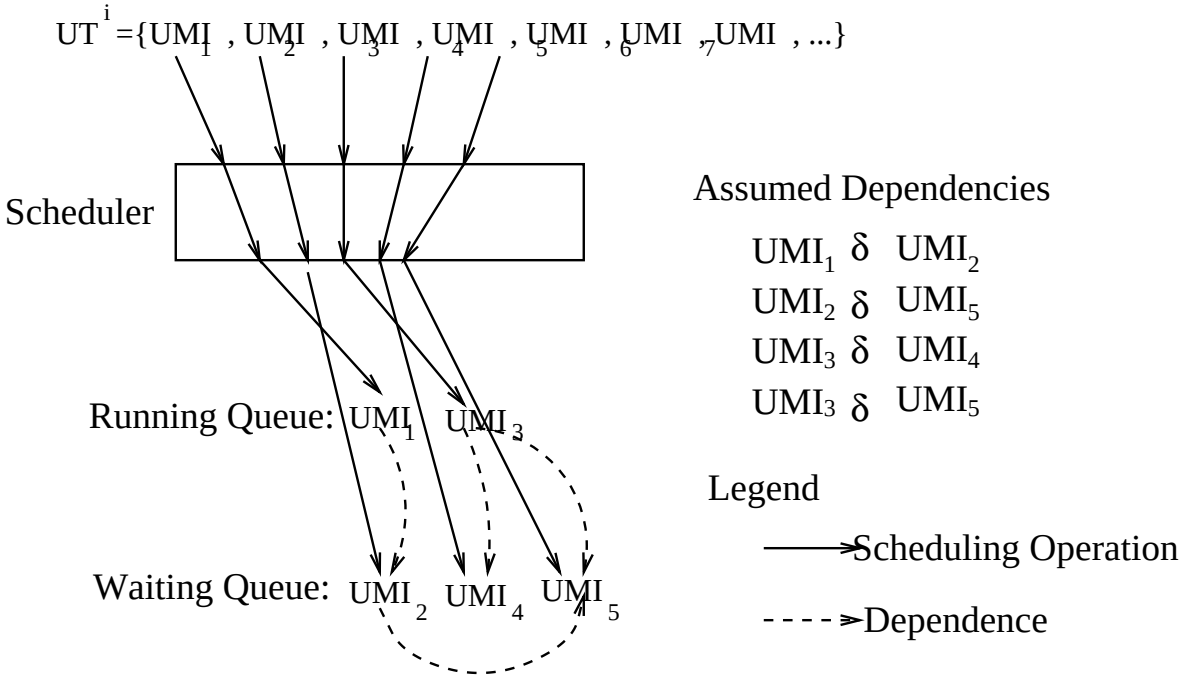

Figure 2: Dependency Links between UMIs

5.2.2 Extension to Multiple Transaction Scheduling

Scheduling multiple user transactions concurrently is a simple extension of the single transaction case if centralized scheduling is used. The same information kept for single transactions must be kept for *all* user transactions submitted. The scheduler tests to see if UMIs from different user transactions conflict by constructing appropriate dependence arcs and scheduling in a manner similar to that for the single transaction case. A correct execution ordering will therefore be obtained.

The only required change in the scheduling algorithm involves waiting UMIs and introduces *bidirectional* dependence arcs in addition to unidirectional ones. Within a single transaction, a serial execution order for the method invocations is defined. Between multiple transactions however, no such ordering exists. Therefore waiting UMIs from *different*, conflicting transactions may be scheduled in either order (presumably depending on the order in which they become "ready" to run). This lack of explicit order is what requires the introduction of bidirectional dependence arcs.

If the next UMI to be scheduled conflicts with only waiting UMIs from *other* transactions, it may proceed and unidirectional dependence arcs are added from the set of queued, conflicting UMIs. If it conflicts with both waiting and running UMIs then it is queued and unidirectional dependence arcs from the running UMIs and *bidirectional* dependence arcs to the other waiting UMIs are added. The unidirectional dependence arcs reflect cases where execution ordering is required while the bidirectional ones indicate potential conflicts but do not require specific orderings.[7] A queued UMI may be scheduled once all its unidirectional dependence arcs are satisfied. At this time, any outstanding bidirectional arcs are converted to unidirectional arcs

[7] This introduces potential serialization problems if two pairs of conflicting UMIs from the same two transactions are scheduled in different orders. (e.g. $O_1.M_1^i \rightarrow O_1.M_2^j$ and $O_2.M_1^j \rightarrow O_2.M_2^i$ where $O_1.M_1 \delta O_1.M_2$ and $O_2.M_1 \delta O_2.M_2$.)

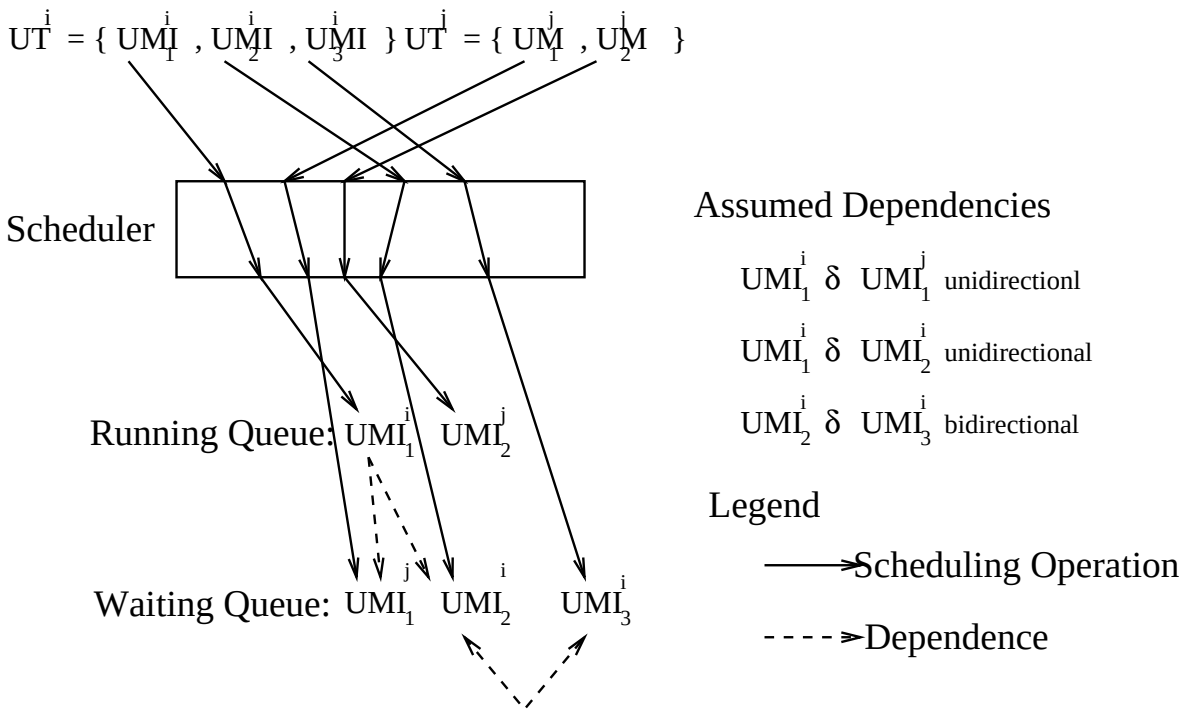

Figure 3: Dependency Links between UMIs in Different Transactions

from the newly scheduled transaction to those still waiting. An example of the use of dependence arcs in the multiple transaction case is illustrated in Figure 3. In this example, a particular (unstated) scheduling order is assumed. Since UMI_3^i is scheduled after UMI_2^j has been queued, it is queued and a bidirectional dependence arc between the two is created.

5.2.3 Performance and Algorithm Analysis

This algorithm is superior to an equivalent *object-based* dependency algorithm because concurrent accesses to non-conflicting methods are permitted. (Dependence is based on sets of conflicting methods within objects rather than on the objects themselves.) Since the granularity of the dependence analysis is decreased, the opportunity for concurrency is increased.

This suggests that the algorithm should perform better than object-based locking. However, despite having more knowledge concerning dependence patterns than simple object locking, this algorithm may perform worse when the $\mathcal{MR}^*$ sets are non-disjoint but, because of the timing of the most likely execution sequences, conflicts will not normally occur.[8] If potential conflicts are common but actual conflicts are rare, this algorithm will perform worse than object locking if the cost of locking is relatively low.

Since there may be a large number of concurrent user transactions, it is likely that actual conflicts *will* occur. Nevertheless, the overhead associated with serializing object method invocations occurring near the root of the call tree resulting from conflicts near the leaves is very high. This is illustrated in Figure 4 where an unfortunate user transaction invokes two methods $O_j.M_k$ and $O_m.M_n$ that must be executed serially due to $\mathcal{MR}^*$ conflicts arising from leaf level calls to $O_x.M_y$ and $O_x.M_z$, respectively.

[8] Any approach that uses only *static* analysis, must serialize method invocations under *worst-case* assumptions.

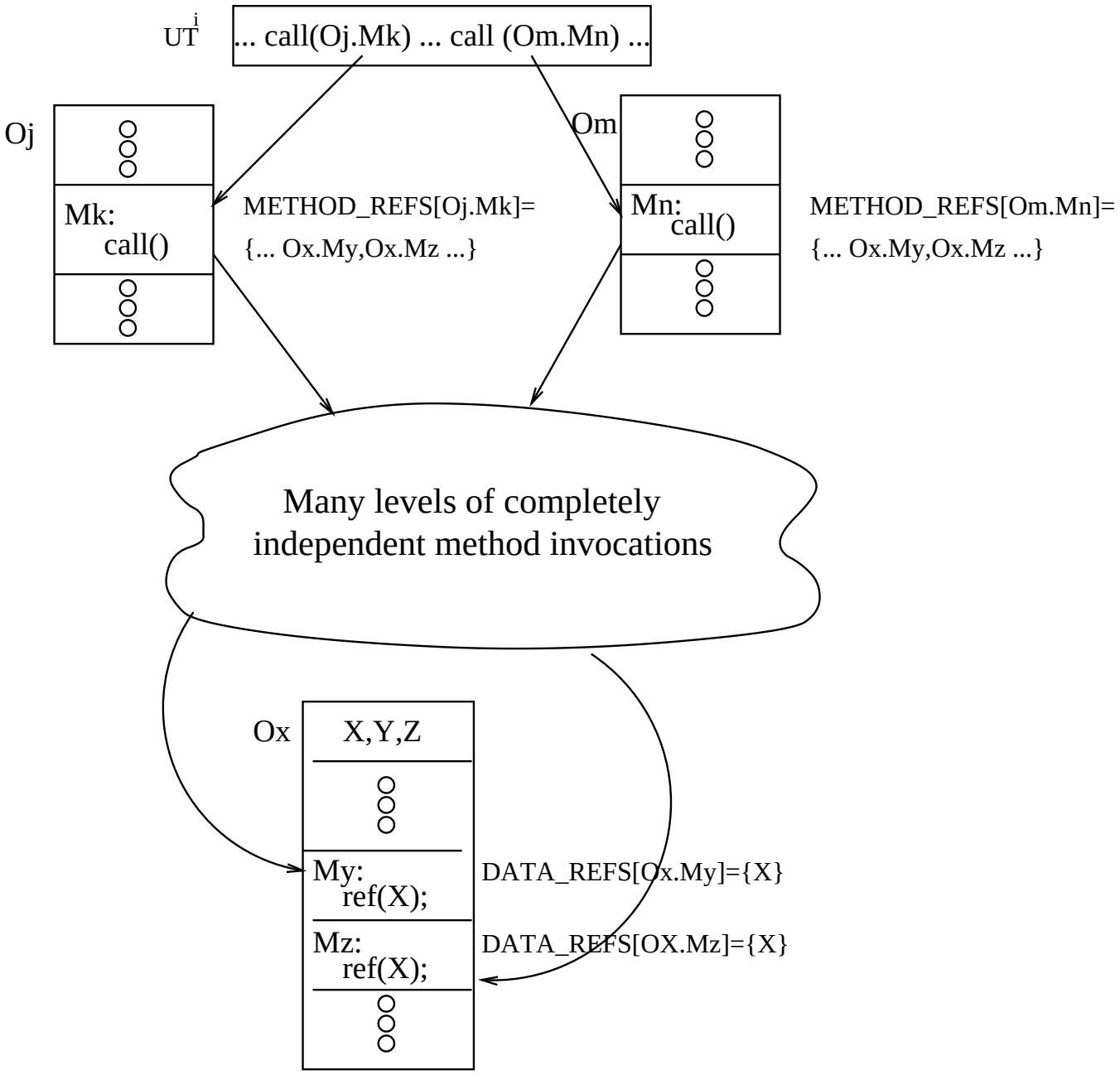

Figure 4: Call Tree with Leaf Conflict

Finally, this algorithm does not address the problem of potential deadlocks or, more importantly, the question of the serializability of multiple user transactions. It should be noted that due to the atomicity of nested transactions recovering from deadlocks is simplified.

In terms of overhead, the scheduler must compare every UMI's reference sets against those of all the *outstanding* (scheduled or queued) UMIs. The cost of this depends on both the number of outstanding UMIs and the total number of objects in the object base (which determines the complexity of performing each comparison). Thus, assuming that the average number of outstanding UMIs is 'k' and that the number of objects is 'n', the average case complexity is $O(k \cdot n)$ bit operations. The space required by the algorithm is of the same complexity – but the unit of measure is bits not operations on bits.

5.3 An Improved Algorithm

The performance of the algorithm just described suffers because it makes scheduling decisions about nested method invocations *before* they are actually invoked. This means that execution time knowledge of other concurrent activity cannot be used in scheduling. A scheduler could apply the statically obtained information when *every* method invocation is scheduled rather than just when UMIs are scheduled. Knowing which methods within an object may conflict with one another, the scheduler could then allow only one invocation of a conflicting method to be executing at a time. If the overhead of this additional checking is less than that of obtaining locks, the net performance will increase.

5.3.1 Single Transaction Scheduling

If a method invocation $O_i.M_j$ is scheduled, the scheduler checks to see if it will conflict with any currently executing method invocation. The $\mathcal{DR}$ set associated with $O_i.M_j$ is compared to those of the currently executing methods. Only currently running methods from the *same* object (O_i) may conflict since data is local to each object and invocations of other methods are handled when they are scheduled. Thus, $O_i.M_j$ must not be executed if there exists an active invocation of a method $O_i.M_k$ such that:

$$\mathcal{DR}(O_i.M_j) \cap \mathcal{DR}(O_i.M_k) \neq \phi$$

This guarantees that no data access conflicts arise. Later, when $O_i.M_j$ invokes another method, the process will be repeated for the nested method being invoked. Hence, the scheduler deals with every method invocation when it is occurs and thereby can use information about what is currently executing.

5.3.2 Extension to Multiple Transaction Scheduling

The multiple transaction case is handled in the same way as the single transaction case. The only difference is that the set of running method invocations is larger since there may be many transactions making invocations concurrently. A test for the intersection of $\mathcal{DR}$ sets from *different* user transactions must be performed:

$$\mathcal{DR}(O_i.M_j^p) \cap \mathcal{DR}(O_i.M_k^q) \neq \phi$$

5.3.3 Performance and Algorithm Analysis

This algorithm decreases the likelihood of method invocations being queued unnecessarily. Unfortunately, while this algorithm does decrease unnecessary waiting it does not improve performance significantly under all possible conditions. This is due to the increase in overhead associated with checking the $\mathcal{DR}$ sets for *every* method invocation. If this checking is inexpensive, performance will be good. Normally, objects should be reasonably well structured and therefore the set of global data shared by their methods should be small. This means that the $\mathcal{DR}$ sets should also be small and hence, in most cases, they will be tested inexpensively.

In terms of overhead, this algorithm must compare the reference sets of every method invocation against those of all *outstanding* (scheduled or queued) method invocations. The cost of this depends on both the number of outstanding method invocations and the total number of objects in the object base. Thus, assuming that the average number of outstanding method invocations is 'l' and that the number of objects is 'n', the average case complexity is $O(l \cdot n)$. In general, $l \gg k$ so the overhead of this algorithm is much higher than that of the first. Once again, the space complexity of the algorithm is the same as the time complexity.

5.4 Hybrid Scheduling with Locks

We now develop a *hybrid* [9] algorithm which employs static dependence data to improve the performance of dynamically obtained locks by minimizing the overhead of their use.

5.4.1 Single Transaction Scheduling

In a manner similar to the last algorithm, the statically derived $\mathcal{DR}$ sets may be used to detect data dependence between methods in a given object. Applying this information statically, the methods in each object can be subdivided into groups of *conflicting* methods. Each such group of methods is assigned a single lock which must be obtained before invocation (*when necessary* – see below).[10] The scheduler can make use of its information about which methods are currently active and what their $\mathcal{MR}^*$ sets are to indicate whether or not a lock must be obtained. In this way the overhead associated with unnecessary lock allocation and deallocation may be avoided.

By examining the $\mathcal{MR}^*$ sets, the scheduler can tell if there may be concurrent execution of conflicting UMIs. If the scheduler detects there will not be any such conflicts, then it can invoke those UMIs in such a way that they do *not* need to obtain their locks. In this case, the dependence information has allowed UMIs to execute without unnecessary lock overhead. If the scheduler detects that there may be conflicts, it can still permit concurrent execution but must require that each method obtains its lock(s). In this case, lock overhead is still incurred but if the *potentially* conflicting method invocations do not actually conflict, the amount of concurrency is not reduced as it would be using the previous algorithms.

5.4.2 Extension to Multiple Transaction Scheduling

As with the previous algorithms, dealing with multiple transactions does not significantly affect the scheduling algorithm. It does increase the number of potentially concurrent method invocations and with them the likelihood of potential conflicts. If the number of concurrent user transactions is very large then decreased savings in lock overhead will be realized since the use of locking is pessimistically required.

[9] Dependency analysis is combined with locking.

[10] These locks and the operations upon them may be created automatically when the code is compiled.

5.4.3 Performance and Algorithm Analysis

The use of static dependence information to refine dynamic locking behaviour is clearly a good solution to the scheduling problem. Since the cost of obtaining locks is high [Moh90] this algorithm offers significant overhead reductions. Furthermore, using this algorithm allows the serializability of user transactions using two-phase locking [BG81]. In this environment, two phase locking synchronizes method-invocations by explicitly detecting and avoiding conflicts between those methods. Using this hybrid algorithm, locking is not performed when it will not be necessary. That is, two methods which cannot conflict, need not perform locking. Also, since the hybrid algorithm only checks for dependence between UMIs rather than between all method invocations, its runtime overhead is significantly lower than that of the improved algorithm suggested. Finally, the hybrid algorithm supports mutually recursive method-invocations if *friendly* [Tan92] locks are provided. As with two-phase locking, there is nothing inherent in the scheduling algorithm which prevents the use of friendly locks. In fact, once a lock has been acquired by a method-invocation it is possible to have it inform the scheduler that it will be friendly to other invocations of the same method. Thus, not only is the recursion permitted, but it is efficiently implemented since locks will not have to be repeatedly obtained.

The time and space complexity of this algorithm is the same as the first, but a significant amount of lock overhead has been saved. Alternatively, the algorithm could also be applied as every method invocation is processed as was done in the second algorithm. This increases the complexity (as it did in that algorithm) but also improves performance by further reducing the number of times locks must be obtained.

6 Conclusions and Future Work

This paper has introduced the new concept of dependence analysis in object bases and discussed how it may be implemented and used to perform effective *method-level* transaction scheduling. Three scheduling algorithms were developed, the last of which is clearly a practical algorithm. This hybrid algorithm uses statically calculated dependence information to decrease the overhead associated with conventional locking schemes, including two phase locking. Since the algorithm is compatible with two-phase locking, the issue of serialization is also addressed [11] and since our object base model assumes nested atomic method invocations, the problem of deadlock arising due to uncontrolled waiting may be addressed using conventional techniques.

The compile-time overhead of dependence analysis is negligible since most of the work will be done by any good optimizing compiler and the run-time overhead will be reduced significantly compared to existing locking methods. Thus, the suggested technique is both practically and theoretically interesting. It should be noted that these techniques are similar to providing partial predeclaration of read and write sets as might be done in a conventional database system [BHG87].

Future work requires a technique to devise analytic methods which compare dependence-based and other scheduling algorithms. This will provide answers to questions such as the level and frequency at which dependence analysis should be performed so that decisions as to how to apply dependence information to the scheduling problem can be made. This will allow quantitative results to be reported in the future. Aside from developing analytical methods, research continues in several other directions:

[11] Other methods for assuring serializability in nested object-oriented databases is an open area of research which we are currently investigating.

• **Data-Level Dependence** – Once method-level dependence is defined, it is relatively straightforward to extend our model to include dependence at the data item level. This has the benefit of increased potential parallelism but will incur higher analysis costs and, in the case of the hybrid algorithm, higher locking costs. It will be interesting to see if applying data dependence information gathered by the object code compiler can speed transaction processing further than the method-level techniques discussed.

• **Other Uses of Dependence Information** – The availability of dependence information is of use in areas besides scheduling. For example, by applying statically-determined knowledge of method reference patterns, it may be possible to predict when deadlocks may occur. This information could be used to tune deadlock handling methods for improved performance. Other possible uses include verification of appropriate concurrent behaviour within and between transactions, detection of hot-spots in object code, and debugging concurrent transactions.

• **Operation in a Distributed Environment** – In a distributed environment the ability to apply dependence analysis to decrease locking would be a great advantage. Since the hybrid scheduling algorithm is compatible with centralized two-phase locking it is likely extendable to any distributed environment where two phase locking is applicable.

References

[AE92] D. Agrawal and A. El Abbadi. A Non-Restrictive Concurrency Control for Object Oriented Databases. In *Proceedings of the International Conference on Extending Database Technology*, pages 469–482, 1992.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[BG81] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[BM91] E. Bertino and L. Martino. Object-Oriented Database Management Systems: Concepts and Issues. *IEEE Computer*, 24(4):33–47, 1991.

[Elm92] A.K. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[FO89] A.A. Farrag and M.T. Özsu. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, 1989.

[GK88] J.F. Garza and W. Kim. Transaction Management in an Object-Oriented Database System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 37–45. ACM, 1988.

[GM83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, 1983.

[HH91] T. Hadzilacos and V. Hadzilacos. Transaction Synchronisation in Object Bases. *Journal of Computer and System Sciences*, 43(1):2–24, 1991.

[KGBW90] W. Kim, J.F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, 1990.

[Kim90] W. Kim. Object-Oriented Databases: Definition and Research Directions. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):327–341, 1990.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[Moh90] C. Mohan. Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems. In *Proceedings of the 16th VLDB Conference*, pages 1–14, 1990.

[Mos85] J.E.B. Moss. *Nested Transactions – An Approach to Reliable Distributed Computing*. The MIT Press, 1985.

[RE92] R.F. Resende and A. El Abbadi. A Graph Testing Concurrency Control Protocol for Object Bases. In *Proceedings of the International Conference on Computers and Information*, pages 289–292, 1992.

[RGN90] T.C. Rakow, J. Gu, and E.J. Neuhold. Serializability in Object-Oriented Database Systems. In *Proceedings of the International Conference on Data Engineering*, pages 112–120. IEEE, 1990.

[Tan92] A.S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[Wol89] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, 1989.

[ZB92] M.E. Zapp and K. Barker. An Architecture and Model for Transactions in Object Bases. Technical report, University of Manitoba, Dept. of Computer Science, TR 92-8, July, 1992.

[ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison Wesley, 1990.