Run-Time Parallelization of Sequential Database Programs

N. R. Soparkar[1]*
P. Krzyzanowski[2]

[1]Elect. Engin. & Comp. Sci.
University of Michigan
Ann Arbor, MI 48109 USA

H. V. Jagadish[2]
A. Asthana[2]

[2]AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974 USA

Abstract

In order to execute a database program written in sequential code efficiently on a parallel processor, we develop the use of transaction concurrency control paradigms to resolve data dependencies dynamically. The sequential code is divided into small units for execution, and these units are executed concurrently as separate "transactions." Our approach ensures that the concurrent execution of the smaller units is logically equivalent to the original sequential program. We present an order-preserving concurrency control strategy to execute concurrently the nested invocations that are generated by the parallelized execution of the database program. We present performance figures from a preliminary implementation to indicate the benefits of our strategy. Finally, we provide a rough analysis to gauge the overheads associated with our approach that would impact performance in a full-fledged implementation.

1 Introduction

Writing correct and efficient parallel database programs, as compared to sequential ones, is significantly harder. Moreover, there exists a large body of sequential code that could potentially be parallelized. Indeed, there opportunity in parallelizing sequential code that is not limited to database programming alone. The typical approach to detect and exploit inherent parallelism in sequential code is by the use of a parallelizing compiler. Such compilation statically analyzes the dependencies between portions of the sequential code. This is relatively straightforward if the database programming language is specifically amenable to parallelization (e.g., the query language SQL for a relational database). Such languages provide limited, though important, types of parallelism in their sequential code. However, in an object-oriented database environment, queries may be written in a general-purpose programming language such as C++ or Smalltalk. Such database programming code is less suitable for parallelization by a compiler. Examples of such code include data accesses that involve pointer de-referencing, or arrays that are accessed through indices (i.e., situations where the memory locations involved in the accesses cannot be ascertained at compile time). These difficulties encountered in parallelizing arbitrary sequential code suggest that the detection of dependencies should be attempted dynamically at run time (e.g., see [6]).

Similar issues arise in areas such as parallel discrete event simulation (e.g., [4]), and distributed shared memory systems (e.g., [7]). In the case of simulations, events are processed in order, and the processing of an event may cause additional events to be enqueued for some later position in the order. In a parallel system, each processor maintains such an event queue. Before a processor can process the current event, it must be ensured that there will not be a subsequent notification of an event that should have been processed earlier. However, if each processor waits for this assurance, there is limited parallelism. One solution adopted is to "warp" simulation time: processors make progress optimistically, and rollback when required. The parallel execution approach that we describe in this paper may be viewed as a generalization of this parallel simulation approach. The pertinent issues in distributed shared memory systems are also similar, but are not discussed here.

We use the database transaction management concepts of concurrency control and recovery to address the issue of dynamically parallelizing sequential database code. These concepts can be used to detect, and to maintain, dependencies within a program execution at run time, in a manner similar to the management of concurrent accesses to a database (e.g., see [3]). The idea is to divide the sequential code into small units of execution, each of which is treated like a transaction. These small units are allowed to execute concurrently provided that their execution is equivalent to a serial order that conforms to their order of appearance in the trace of a sequential execution of the original code. In principle, this allows the extraction of the maximum available parallelism at run-time — without the need to re-write code (although the code may need to be re-compiled). The price paid for the benefit of using run-time analysis is the cost overhead, in terms of time and processing expended, of ensuring the serializability of the executions in a specified order. For this purpose, we devise an relatively inexpensive concurrency control algorithm. We have implemented this algorithm on a main-memory, medium-grain MIMD parallel processor. We present preliminary performance numbers that exhibit the potential utility of our scheme.

*Work done at AT&T Bell Labs., Murray Hill.

CIKM '95, Baltimore MD USA
© 1995 ACM 0-89791-812-6/95/11..$3.50

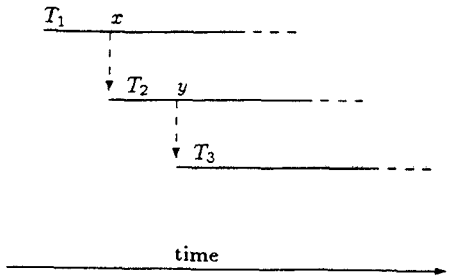

Figure 1: Method invocations and executions

2 The Source Code

While the ideas in this paper are applicable to any database programming language, the particular language we use for our exposition is C++ (e.g., see [9]). Below, we show how its object-oriented nature gives rise to a natural transaction unit, and we discuss how these units are invoked.

2.1 The Small Transaction Unit

The first issue that arises with respect to dividing the source code into smaller units, each of which is to constitute a transaction, is the granularity of this division. The smaller the size of a unit, the greater is the number of transactions, and consequently, the potential for parallelism. However, a large number of transactions also implies that the overhead due to concurrency control would be high.

The object-oriented nature of the code being written provides some guidance in the choice for the transaction size. Each method on an object is likely to be a self-contained piece of code that operates mainly on data local to the object. Distinct methods, on the other hand, operating on distinct objects, have a good chance of not having any dependencies. Therefore, initially we choose a method invocation as our unit of parallel execution (in Section 3 we shall modify this choice slightly). We treat each method (i.e., public member function) invocation, as a transaction.

In the course of its execution, a method may itself invoke other methods. Each of these invoked methods is also considered to be a distinct (sub)transaction In this manner, we obtain a nesting within the transaction structure (e.g., see [8]). Such a nested transaction has the requirement that a parent transaction can complete successfully (i.e., commit) only after all of its children have completed similarly. Subtransactions are allowed to see the effects of the (uncommitted) updates of their parents up to the point of their invocation, but no updates that may have been performed after the subtransaction was invoked. This rule applies to any further levels of nesting as well.

Figure 1 depicts part of a typical nested transaction execution. At a point x in the execution, the transaction T_1 invokes a (sub)transaction T_2. Transactions T_1 and T_2 execute concurrently. During the execution of T_2, at a point y, the (sub)transaction T_3 is invoked by T_2. The important point to note is that T_3 is allowed to see the effects of T_1 up to point x, and the effects of T_2 up to point y, but not beyond. The reason for these views seen by (sub)transaction T_3 becomes clear if an equivalent sequential computation is considered. Such a computation is obtained by "folding back" T_3 into T_2, and T_2 into T_1, and thereby to obtain the serial execution that is logically equivalent to the concurrent execution that we described. Such an execution would have T_1 executed up to the point x, then T_2 up to the point y, then T_3, followed by the portion of T_2 after the point y, and finally, the portion of T_1 after the point x.

The goal of parallelizing sequential code correctly consists of having all the nested transactions execute in a manner that can be serialized into an order compatible with the original sequential program. In other words, the transactions at the top-level should be serialized in invocation order as defined by the program, and invoked subtransactions should be serialized in their invocation order within each transaction.

2.2 Dependencies

A method invocation in C++ is synchronous (i.e., the invoking procedure temporarily blocks after the invocation, and passes control to the invoked procedure; the invoking procedure resumes execution only after the invoked procedure completes its execution). A key modification we make to obtain parallelism is to make these function invocations asynchronous. That is, an invoking procedure is permitted to continue execution after having invoked a function (assuming that the values returned by the invoked function are not necessary to continue the execution).

In a sequential program, dependencies among method invocations can arise in different ways. First, control-flow in the program could be affected by the value returned by a previous invocation. Second, there may be explicit data dependencies in that the returned value of a function may be used to compute a parameter for a subsequent function. Third, hidden dependencies may be present due to pointer de-references, or a dependency on the input data. While the first two types of dependencies may be detected at compile-time, the third type may only be noticed at run-time. We address mainly of the last type of dependencies (although, the first two are subsumed by our approach). Our concurrency control technique may be regarded as being complementary to other available means to parallelize sequential code since it can be suitably modified for use with compilation-based techniques for parallelization.

2.3 An Example

The C++ code shown below (which does not include the variable declarations and class definitions) is designed to traverse a graph beginning with a set of M start nodes, and to mark each node in the graph with the number of the lowest numbered start node that can reach it.

```
i=1;
for (node in NODE) suchthat start_node(node) {
  mark_reachable(i,node) ;
  i++ ;
}


mark_reachable(int i, NODE* node)
{
  for (j=node->successor ; j != NULL ; j=j->next) {
    succ_node = j->nodename ;
    if (succ_node->mark == 0) {
      succ_node->mark = i ;
      mark_reachable(i,succ_node) ;
    }
  }
}
```

If the set of nodes reachable from a given node are all disjoint, it is possible to perform the marking in parallel. In general one cannot assume that this is the case for every input graph. As such, little parallelism can be extracted at compile time. Our approach to parallelizing such code is to attempt to run each invocation of the `mark_reachable` routine in parallel. Concurrency control is used to manage any conflicts that may arise (e.g., due to several source nodes that reach the same destination).

3 Dynamic Resolution of Dependencies

In this section, we describe our approach to the parallelization of sequential code in detail. We present a nested transaction model, and the concurrency control algorithms to be used in the context of parallelization.

3.1 Nested Transaction Model

Each method invocation that originates directly from the main program sequential code, is referred to as a "top-level" transaction. A top-level transaction is permitted to issue subtransactions (i.e., nested method invocations) during its execution. When a subtransaction completes its execution, it informs its invoker by an "acknowledgment" with respect to the invocation, and at the same time, it returns any computed values.

A transaction, or subtransaction, may itself execute some code between invocations of further subtransactions. Such portions of the execution, between successive pairs of subtransaction invocations, are also treated as subtransactions for purposes of concurrency control — even though there is no method invocation involved. Along with the top-level transaction, all the invoked subtransactions are permitted to execute in parallel, within the limits prescribed by the concurrency control.

Transactions execute atomically in that all the actions of a transaction execute, or none do. To maintain the correctness of the concurrent executions, the concurrency control must necessarily abort certain executions as described below. As in standard database terminology, an abort of a transaction refers to the undo of its effects. Since the sequential program code is written without anticipating a transaction-oriented execution, there can be no abort requests within the code itself. As explained below, if a transaction is aborted by the concurrency control, it is re-started repeatedly until a particular attempt executes to completion. A top-level transaction does not return a value to the main program until the concurrency control ensures that it will not need to be aborted (i.e., until the top-level transaction is committed).

3.2 Transaction Numbering and Serializability

Each atomically executed portion of the code is given an identifier that is recursively defined as follows. The successive portions of code in the main program demarcated by method invocations (i.e., the invocations that form top-level transactions) are given the successive identifiers 1, 2, 3, ... The first portion of a top-level transaction is given the identifier $i.1$ where i is the identifier of the portion of code in the main program that just precedes in the sequential execution order. Finally, consider a particular atomic unit of execution with an identifier i. The first method that it invokes, if any, is given the identifier $i0$ (i.e., i concatenated with the bit value 0). The subsequent, continuing, portion

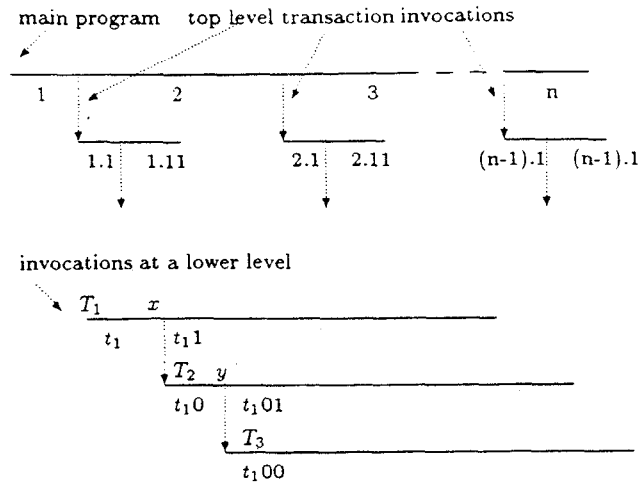

Figure 2: Scheme for identifier numbers

of the execution (that is *not* part of the method invoked), is regarded as a separate atomic unit of execution, and is given the identifier $i1$. Figure 2 depicts an example identifier numbering (including the code shown in Figure 1). Note that our scheme permits the identifiers for the code below the level of the main program to have a compact binary number representation.

We need to provide a total order on the identifiers. The order relation, $<$, is defined as follows (the intuition behind the definition is provided subsequently). For two distinct identifiers i_0 and i_1, $i_0 < i_1$ if, and only if, either:

- the number corresponding to the main program in i_0 is smaller than the same in i_1, or
- i_0 and i_1 arise from the same top-level transaction, and by suitably padding (one of) the identifiers with bit values of 0 (at the least significant bit positions) to make the identifiers equal in length — and referring to them as j_0 and j_1 modified from i_0 and i_1, respectively, we have either
  - j_0 is smaller than j_1 when these modified identifiers are regarded as numbers, or
  - the modified identifiers are indistinguishable, but i_0 has fewer bits than i_1.

Thus, for instance, the following total order among a possible set of identifiers is consistent with the above definition: $3.101 < 3.1011 < 3.11 < 4.110 < 4.111 < 4.1110$. We find it necessary to define the order relation on identifiers with differing number of bits in this manner since it is not possible, during a dynamic execution, to ascertain *a priori* the number of bits necessary in the largest identifier. Henceforth, we say that an identifier i_0 is lower or higher than an identifier i_1 according as $i_0 < i_1$ or $i_1 < i_0$.

Our numbering scheme provides unique identifiers, and we use the term *segment* to refer to each part of an execution that has a distinct identifier. Each segment of the code must be executed as one atomic block by sequentially executing it in its entirety. Note that if the entire program were to be executed sequentially on a uniprocessor, the different segments would execute in the increasing order of their identifiers. This may be seen by regarding the executions in the

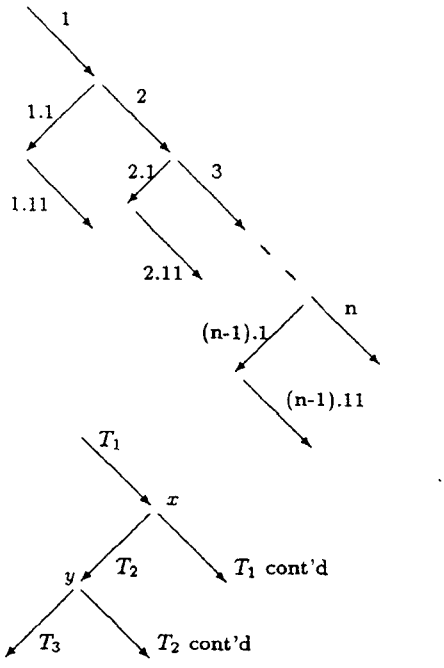

Figure 3: A binary invocation order tree

form of a binary tree of invocations, as depicted in Figure 3 (for the example from Figure 2). Each edge in the tree corresponds to a particular segment of the code, and the depth-first search order of the edges in the tree (i.e., in the order that the edges are first encountered in the search) corresponds to a sequential execution order of the segments. Therefore, our aim is not just to ensure serializability of the segments (e.g., see [3]), but also to ensure that an equivalent serial order of execution of the segments is the order that would be followed if the entire program were to be executed sequentially. That is, the segment executions should be serialized in the increasing order of their identifiers. .

3.3 Using Concurrency Control

We make certain assumptions to help describe the use of concurrency control paradigms for the parallel executions. Each segment of the code accesses at most one object. If a segment requires to access additional objects, it can do so only through further method invocations, which are then treated as separate segments, and assigned their own identifiers. Each segment is required to execute as a single uninterrupted thread.

For each object in the database, we maintain a list consisting of identifiers of the segments that have successfully accessed the object. We refer to each element of a list as a *lock*, and the order of the elements is the order in which the the accesses occurred — with the most recent access being the last element.

The state of an object just before, and just after, a segment accesses it are referred to as the *before-image*, and the *after-image*, respectively, for the segment. The effect (w.r.t. the state of the accessed object) of aborting a segment is reinstating its before-image. Below, we discuss how aborts are effected, and how such "shadow" images are maintained.

Consider a segment with an identifier i_0, that needs to access an object O. The following algorithm is executed after the completion of any ongoing access to O. First, the identifier i of the last lock on the list for O is examined — the identifier would be that of the most recent segment that successfully accessed O. The algorithm checks whether it is safe to permit the segment requesting the access to proceed by comparing the identifiers i and i_0. If $i < i_0$, or there is no lock in the list, then a lock with identifier i_0 is appended to the list for O, and the segment with identifier i_0 is allowed to access O (after the requisite shadow information is recorded as described below).

If the comparison described above yields $i > i_0$, then the segment with the identifier i must be aborted (and if necessary, executed again later). Furthermore, the lock i is deleted from the list for object O. Thereafter, the same comparison, with any accompanying aborts and deletions from the list, are effected for the next lock on the list. The algorithm iterates in this manner until either $i < i_0$ for the identifier i of the last lock on the list for O, or the list is empty. Since each list is finite, the segment with the identifier i_0 gets access to the object O eventually. Note that our approach is similar to executions of a "timestamp" order scheme with a wound-wait restart policy (e.g., see [5, 3]) — with the identifiers regarded as the timestamps. The difference is that in our scheme, the segment with a higher number is aborted, and a segment is committed only if all lower numbered segments are committed (commitment is described below).

We use database concurrency control theory (e.g., see [3]) to prove that the above algorithm is indeed correct in our context. At a given point in the execution, consider a set of segments that have either completed executing or are currently executing. Define a ***serializability graph*** (SG) for the execution (e.g., see [3]) with nodes consisting of the segments labeled by their identifiers. A directed edge (i, j) is created in the SG if, and only if, an object was accessed by the segment numbered i before the same object was accessed by the segment numbered j, and neither segment was subsequently aborted. The execution of the segments is serializable since each segment executes as a single uninterrupted thread. Therefore, from the theory of concurrency control (e.g., see [3]), the SG must be acyclic.

It is more important to establish the correctness of our approach in the context of equivalence to the sequential execution of the original code. This would be established if, after all segments are successfully executed, each object can be shown to have been accessed by segments in increasing order of their identifiers. The following result helps in this regard.

Theorem 1. *For a completed execution of a set of segments under the concurrency control, the identifiers provide a topological sort order of the nodes of the SG for the execution.*

Proof: Consider an edge (i, j) in the SG for the execution. The edge indicates that segment i accessed an object O before the segment j did, and hence, the lock corresponding to i must precede the lock corresponding to j in the list for O. Now, the concurrency control ensures that for each list for an object in a completed execution, the locks in the list are in an increasing order of the identifiers. Hence, $i < j$ must hold, thereby implying that the identifiers provide the required topological sort order. □

Note that deadlocks and starvation do not occur in our scheme since there is a total order on the segment identifiers, and a lower identifier segment does not wait indefinitely on a higher identifier segment.

3.4 Effecting Aborts

A segment may sometimes have to be aborted and re-started, as mentioned above. To facilitate this, a segment creates a copy of an object when its lock is added to the list for the object.The segment updates the object in place as it executes. If the segment is forced to abort, then the copy can be used to recover the state of the object as it was before the segment commenced execution. If and when the segment commits, the shadow copy is discarded.

A segment with a higher identifier may be permitted to see the effects of a segment with a lower identifier. This may happen if both segments access a common object, or as a consequence of the results returned back by the lower numbered segment. The former situation is recognized from the lock lists, and the latter situation by the concurrency control that passes the returned results from the lower numbered segment to the higher numbered segment. In such situations, if the lower numbered segment is aborted, then so must the higher numbered segment. That is, as a consequence of a segment abort, there may be cascading segment aborts. Therefore, when a segment aborts, it may necessary to effect other aborts, and restart the aborted executions again. Whenever a set of aborts occur together in this manner, for each object, the aborts are effected in the decreasing order of the identifiers in the list — consistent with the most recent access being aborted first. Due to space constraints, we do not discuss the low-level abort and recovery algorithms for the concurrency control in detail.

3.5 Commit Processing

The list of locks for an object could get arbitrarily large as the concurrent executions proceed. Therefore, there is a need to discard some locks, and the corresponding shadow information, regularly. We refer to this process of discarding as the commitment of the segments since, as is the case for database transaction management, the execution of a transaction is committed irretrievably once the shadow information for it is discarded. It is safe to commit a segment only if it is ensured that no subsequent abort for it will be necessary. The following result helps in this regard.

Lemma 1. *The execution of a segment with an identifier i_0 does not need to be aborted if every segment with an identifier i, such that $i < i_0$, is committed.*

Proof: There are two ways in which the execution of a segment with an identifier i_0, that accessed an object O, may need to be aborted. The first case is when a segment with a identifier i such that $i < i_0$ attempts to access the object O after the execution of the segment with the identifier i_0. The second case occurs if a segment with a lower identifier from the same transaction is aborted, and as a consequence of having used its results, the segment with the identifier i_0 needs to be aborted. Either case cannot occur since all lower identifier segments are assumed to be committed. □

We assume a shared-nothing architecture in which each object resides in exactly one processor with no replication across the processors. We assume that methods invoked on a particular object are executed at its corresponding processor. Note that it is possible, for instance, for a top-level transaction to be a function call f_1 on an object O_1 at processor A, for function f_1 to invoke a function f_2 on object O_2 at processor B, and for function f_2 to issue yet another function f_3 on object O_3 at processor A; it is even possible that O_3 is the same as O_1. For the inter-processor communications, we make the sole assumption that the messages between processors maintain order for any given source-destination pair. For instance, we allow for the possibility that processor A first sends a message m_1 to processor B and then a message m_2 to processors C, but the m_2 may reach processor C earlier than m_1 reaches processor B. In fact, another message m_3 issued by processor C after it receives m_2, may reach processor B before m_1 does.

We now describe our commitment protocol for a top-level transaction. Since the objects accessed by a top-level transaction may be on different processors, the commitment of all its, possibly non-local, segments must be coordinated. This is done by traversing the binary tree of invocations in a depth-first search order. A commit token is sent to a segment by its invoking processor when the segment is encountered in the "forward" direction of the depth-first search — as reflected by the next higher identifier. The commit for a segment is effected (by discarding the corresponding lock and shadow information) when the commit token is received. Acknowledgments are sent back in the "reverse" direction of the depth-first search.

The main program segments are committed by a *coordinator* process, executing at a single, dedicated processor, that counts-up through the main program segment identifiers. The coordinator issues a commit token to a top-level transaction only after it receives an acknowledgment for the successful commitment of the preceding top-level transaction.

The coordinator knows about the top-level transactions since, when initiated, a top-level transaction it sends the coordinator a message informing it of the initiation. The coordinator keeps this information in a queue, and issues a commit token at the appropriate time. Finally, after commitment, a top-level transaction sends an acknowledgment back to the coordinator. We expect that centralized coordination will not be a bottleneck for most realistic programs and levels of parallelism (see also Sections 4 and 5).

Theorem 2. *Once a segment is committed by the commitment protocol, its abort cannot be required subsequently by the concurrency control.*

Proof: The use of a coordinator process that counts-up through the top-level transaction identifiers ensures that the segments of only one top-level transaction at a time are under the process of commitment. Moreover, it ensures that the order of commitment between the top-level transactions is in the increasing order of their identifiers. Within each top-level transaction, commitment occurs in the order of increasing identifiers since our protocol follows the order of invocation through the tree of invocations. Therefore, the segments are committed in the increasing order of the identifiers, and the result follows from Lemma 1. □

4 Experimental Evaluation

We describe a preliminary implementation and experiments to provide a rough assessment of our strategy. We made some simplifying assumptions which generally degraded performance; a detailed implementation ought to provide better results. First, we limited the length of an identifier to a fixed size (i.e., segments that were beyond a certain depth in the invocation tree had the same identifier as their invoking segment, and we executed them sequentially). Second, we maintained only one lock at a time for an object (i.e., if there was a lock on an object, then the requesting segment had to await the release of the lock).

4.1 System Configuration

We used the SWIM system [1, 2] for our experiments (although the generality of our approach implies that it can be used for any multiprocessor system). SWIM may be regarded as an "intelligent memory" used to accelerate object-oriented data storage and manipulation. It comprises of several (up to a few dozen) small memory units, called Active Storage Elements (ASEs), embedded in a communication network. An ASE can be configured to manage objects of a particular class by loading its on-chip microcode memory with the appropriate microcode to execute member functions associated with that class. A member function is invoked on a specific object by sending a message to the ASE managing it. This message must identify the particular object of interest, the specific function to be executed, and the values for any parameters that may be required. Any response from the ASE is also in the form of a message. In addition to the semantic fit in terms of object-oriented storage, the key feature of importance is that SWIM can be considered a medium-fine-grain multiprocessor with a very low inter-processor communication cost; barring contention, a message of two 32-bit words takes only 3 clock cycles, from the registers of one ASE to the registers of another ASE. All our performance measurements were on a SWIM system used as the parallel object-manipulation back-end to a SUN workstation that executed the top-level transaction calls. The particular SWIM system we used had 16 ASEs. One was dedicated for commit processing. The other ASEs were used to execute the segments in parallel.

4.2 Example Experiments

We ran several different queries on our system. The time for each was measured on the host SUN workstation, using the standard UNIX `time` command. The performance benefit varied greatly depending most significantly on the amount of parallelism inherent in the given sequential code. The overhead due to our concurrency control algorithms were small. Below, we discuss small, but illustrative, examples of database programs that were parallelized by our technique.

4.2.1 Example 1

Consider a program that is perfectly parallelizible, comprising of a single loop iterating over a collection of objects evenly distributed between the 15 ASEs. In each iteration of the loop, some local processing was performed on an object, and no other objects were updated or referenced. This is a program that any reasonable parallelizing compiler should be able to parallelize, and we present performance numbers here to quantify the run time overhead introduced by concurrency control. We used objects that were 256 bytes each in size. Each method invoked on an object required approximately 5500 instructions to execute.

Executed sequentially, as written, the program took 53.76s ("real" time) to execute. Executed in parallel, with perfect parallelism (without the overheads of concurrency control — as though detected by a parallelizing compiler), the code took 3.96s to execute. Therefore, on 15 processors, the "perfect" speed-up achievable for this application on our system showed a factor of 13.58. This factor is not a perfect 15 because of the skew in dispatch times for the multiple parallel tasks and the time spent by the host doing a small amount of the sequential processing.

Using the concurrency control algorithm to parallelize the sequential program at run-time, the time required for execution was 4.03s, giving a speed-up of a factor 13.34. This provides a rough estimate for the small overheads of commitment, shadowing, and locking. This speed-up is very close to the best possible parallel execution, and in this case the overhead for concurrency control were negligible.

4.2.2 Example 2

Consider a more realistic program in which there is potential for data contention, and little parallelism can be obtained at compile time. The program involved traversing a graph of objects whose connectivity was defined only at run time (i.e., the connectivity was provided as input data). In fact, each method invocation included a connectivity list for the object as an argument. The program performed a pattern search over the data in the objects, and the pattern and destination of the results were required to be a function of the invocation order. This program was similar to the example code fragment discussed in Section 2. On account of dynamic data dependencies, we cannot expect any parallelism to be detected by a compiler.

In terms of performance, a sequential execution took 107.33s. We also gauged performance for a blindly optimistic parallel execution, which, as may be expected, provided incorrect results. However, it did indicate the overheads of commitment and shadowing as described for the example above. The time taken for such an execution was 10.04s. With the concurrency control in place, the time taken was 15.73s, a speed-up of 6.8 times over the sequential execution (and correct results were obtained!).

Our preliminary studies exhibit that, at least for some situations, using the concurrency control strategies as described in this paper are likely to yield good results. We mention some ways to reduce the overheads for our scheme; the details have been omitted due to space restrictions. First, if it is known *a priori* that during some phase of an execution, a method will not invoke any further methods (e.g., this holds for a segment of the sequential code that initializes variables), then locks, shadowing, and commit processing are not needed for that phase. Second, shadowing overheads may be reduced by creating a shadow for an object just once for any top-level transaction — the trade-off being that rollbacks may be costly. Note that since the very same sequence of events may cause repeated aborts, a policy may be enforced to execute a top-level transaction sequentially if too many aborts are encountered. Third, to reduce the number of messages, our scheme may be modified to issue only a single commit token and acknowledgment pair for any participating processor for a given top-level transaction.

5 A Simple Analysis

Of necessity, experimental verification of our proposed technique is limited. We present a crude analysis to gauge the importance of various factors governing parallelism in order to estimate how our approach will scale-up.

Figure 4 depicts a typical top-level transaction in which each (sub)transaction requires t time units to execute (not including the time to execute the invoked subtransactions). Each such t-sized part, which we refer to as a *portion*, is expected to execute sequentially on a single processor. Assume that each non-leaf (sub)transaction invokes a subtransaction at c equally spaced points in its execution, and let the depth of such invocations be d. (The $\frac{t}{c+1}$-sized executions are the segments within each portion.)

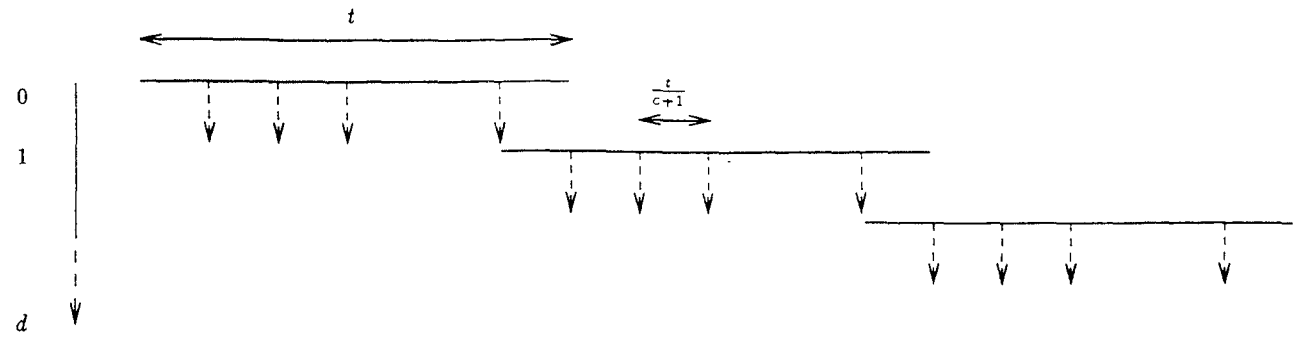

Figure 4: Idealized model of a nested transaction

The time required for the sequential execution of one top-level transaction is $t\frac{c^{d+1}-1}{c-1}$. If there are C concurrently active top-level transactions at any time, their sequential execution would require $Ct\frac{c^{d+1}-1}{c-1}$ time. Unrealistically assuming that there are enough processors available to exploit all the parallelism, that there are no dependencies, and that no aborts or waits occur, the execution of a transaction would appear similar to the depiction in Figure 1. The makespan time for such an optimistic parallel execution of a single top-level transaction, and consequently, C top-level transactions, is $t(1+\frac{cd}{c+1})$. Therefore, with C transactions executing concurrently, the best possible speed-up factor is approximately $\frac{Cc^d}{d}$ (assuming that c is sufficiently large — i.e., that the transactions have a large degree of inherent parallelism). Observe that the speed-up possible is greater than a factor of C since there is intra-top-level-transaction parallelism due to the subtransactions, in addition to inter-top-level-transaction parallelism.

Now let us consider the various significant factors that could adversely affect the attractive optimistic parallel execution time. Except where noted otherwise, the issues are each considered independently in the following analysis.

Shadowing and locks. Each time a portion executes, it must create shadow data, and also create locks. Let us assume that the time taken for shadowing is s per object, that is done per portion, and disregard the comparatively negligible time taken to create a lock. Assuming that shadows are created at each portion, the time taken to execute a segment increases to $t_s = t + (c+1)s$, and this may be approximated to $t_s = t + cs$ for a sufficiently large value of c. (If shadows and locks are created at each segment, the same analysis can incorporate the added complexity by considering an additional depth of the nesting.)

Inherent dependencies. We assume that the only dependencies that may occur in the parallelized executions occur in the form of contention for data. That is, we assume that each segment within a particular top-level transaction can complete its execution without having to wait for the values returned by the execution of another segment of the same top-level transaction. (We may reasonably expect that many of these unaccounted for dependencies could be detected by a parallelizing compiler.) The following analysis only deals with the question of data contention.

Let a denote the probability of a contention between two portions that request access to a common object. That is, for the concurrent execution of C top-level transactions, $aC\frac{c^{d+1}-1}{c-1}$ is the probability that a portion will actually contend with another active portion during its lifetime. Therefore, the probability that a contention is *not* encountered by a portion is $(1 - aC\frac{c^{d+1}-1}{c-1})$. We assume that a is small enough to safely permit disregarding the probability that more than two portions contend for the same object. For a sufficiently large value of c, we may regard the probabilities for contention and non-contention as being aCc^d and $1 - aCc^d$, respectively.

We now estimate the effective dilation in the execution time for a portion due to the concurrency control overheads. Note that since each such execution involves shadowing and the creation of locks, instead of t, we use t_s as estimated above. Consider a portion that requests access to an object, and is faced with data contention. In such a situation, if the portion needs to await the completion of an access on the requested object, then the duration of the wait may be estimated to be $\frac{t_s}{2(c+1)}$, which is half the time taken for a portion to execute. Now, let us consider the situation that a segment numbered j of a portion is to be executed, and that it encounters contention with a segment numbered i of another portion. Assuming that there are no cascading aborts, the following occur with a probability of $aC\frac{c^{d+1}-1}{4(c-1)}$ each.

- $i < j$, and i executes before j. In addition to the execution time for j, a wait is encountered, and the effective time for the execution of the portion containing segment j becomes $t_1 = t_s + \frac{t_s}{2(c+1)}$.
- $i < j$, and j executes before i. Segment j must be aborted, and it takes s units of time to reinstate the before-image for j. Assuming that a re-execution of segment j is initiated directly after i completes execution, the effective time taken to execute successfully the portion containing segment j becomes approximately $t_2 = t_s + s + \frac{t_s}{c+1} + \frac{t_s}{c+1}$ (i.e., the sum of the times taken for the successful execution, the abort of segment j, the execution of i, and the re-execution of segment j).
- $j < i$, and j executes before i. Only the execution time for the portion with segment j needs to be accounted, and that is $t_3 = t_s$.
- $j < i$, and i executes before j. The contention induces the overheads of a wait for the execution of segment i to complete, followed by an abort of i, and hence, the effective time taken for the portion to execute is $t_4 = \frac{t_s}{2(c+1)} + s + t_s$.

Therefore, with data contention, the effective execution time for a large value of c is

$t_d = (1 - aCc^d)t_s + (\frac{aCc^d}{4})(t_1 + t_2 + t_3 + t_4)$

which simplifies to

$t_d = (1 - aCc^d)t_s + aCc^d(t_s + \frac{s}{2} + \frac{3t_s}{c+1})$.

Further approximations yield $t_d = t + cs(1 + aCc^{d-1})$ indicating that shadowing, and deeply nested transactions, are significant factors that would degrade performance.

Notice that our analysis for data contention has been quite pessimistic for two reasons. First, since the transactions would be generated to a large extent in the increasing order of their identifier numbers, the order of accesses on the objects is less likely to cause aborts. Second, with a limited number of processors, it should be expected that the processors could be assigned to the portions in increasing identifier orders.

Commitment overheads. If all the subtransactions invoked are to an external processor, then for each segment, there is an overhead of two messages to account for the commit token and its corresponding acknowledgment. Note that the messages related to the function invocation and return are present even for the sequential case, and therefore, they are not overheads. Thus, the number of overhead messages related to committing the C concurrently executing top-level transactions, is $2C\frac{(c^{d+1}-1)(c+1)}{c-1}$ (i.e., twice the number of portions involved in the execution), and that may be approximated to $2Cc^{d+1}$ messages for a large value of c. The time taken for the traversal of these messages should be added to the expected time for parallel execution. This is a pessimistic estimate since the number of external invocations may be few due to careful data placement strategies. Also, a limited number of processors implies that several messages may actually be local to a processor. The significance of this communication overhead depends on the specific parallel architecture used, and the cost of sending small messages in this architecture. If the time taken to send a small message is m, then the overhead time taken to process the commitment for the executions is $t_c = 2mCc^{d+1}$ for a large value of c (barring contention for the communication resources). Other than communication overheads, the time expended on the computation related to commit processing is likely to be negligibly small, and hence, we disregard them.

Limited processors. If each segment could execute in parallel, there could be $C\frac{c^{d+1}-1}{c-1}$ such segments executing in true parallelism. That is, approximately Cc^d processors would be needed for a large value of c. If only N processors are available, and $N < Cc^d$, then we may hope to achieve a linear speed-up if the overheads due to scheduling are small.

Assuming that all the necessary processors are available, our analysis suggests that instead of the optimistic $t(1+\frac{cd}{c+1})$ time taken to execute the C transactions in parallel, the overheads to manage the concurrency would increase the expected parallel time to $t_d(1 + \frac{cd}{c+1}) + t_c$ where the following approximate values hold for sufficiently large c,

$t_d = t + cs(1 + aCc^{d-1})$, which is the dilated portion execution time, $t_c = 2mCc^{c+1}$, which is the delay caused by communications due to commitment overheads, s = time taken to create shadow information, and a lock, for an object, and m = message traversal time for a commit, or its acknowledgment.

The main observations from the above analysis are as follows. First, note that the overhead for concurrency control grows exponentially as the depth of the invocation tree, and polynomially as the number of parallel execution threads. Therefore, deeply nested invocations tend to degrade performance sharply. Second, the overhead due to shadowing, which has been included within the execution time for each segment of a transaction in the above analysis, may be significant because the concurrency control algorithm adopted may create shadows frequently. This overhead is dependent on the amount of shadow information that must be created, which is an application-specific consideration. Third, the message communications overhead is a simple additive term, although it has a multiplicative factor that is sensitive to the degree of parallelism, and the depth of the invocations. The message overhead is architecture-dependent in that the time for the traversal of the messages between the processors, is an important factor.

6 Conclusions

A novel use of concurrency control to perform dependency resolution for a database program written in sequential object-oriented code, thereby to permit parallel executions, was described in this paper. This was achieved by managing the parallel executions of portions of the code in the manner that concurrent transactions are handled in database systems. By mapping method invocations in a sequential object-oriented program into a nested transaction model, we showed how to implement order-preserving concurrency control for the parallel executions generated. Also, we presented a preliminary analysis and some performance numbers that exhibit the potential benefits of our technique. Our approach to dynamic parallelization of sequential database programs would have good potential for significant performance pay-offs in multiprocessor systems when it is implemented in a more complete manner.

References

[1] A. Asthana, H. V. Jagadish, J. A. Chandross, D. Lin, and S. C. Knauer. A high bandwidth intelligent memory for supercomputers. *Proceedings Third International Conference on Supercomputing*, May 1988.

[2] A. Asthana, H. V. Jagadish, and S. C. Knauer. An intelligent memory transaction engine. In *International Workshop on Database Machines, Deauville, France*, June 1989.

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

[4] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10), October 1990.

[5] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.

[6] M. Lam. Coarse-grain parallel programming in Jade. *ACM SIGPLAN Notices*, 26(7):94–105, July 1991.

[7] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[8] J. E. B. Moss. Nested transactions: An introduction. In B. Bhargava, editor, *Concurrency Control and Reliability in Distributed Systems*, pages 395–425. Van Nostrand Reinhold, 1987.

[9] B. Stroustrup. *C++ Programming Language*. Addison-Wesley, Reading, MA, 1987. 2nd ed.