GRANULARITY HIERARCHIES IN CONCURRENCY CONTROL

by

Michael J. Carey

Memorandum No. UCB/ERL M83/1

14 January 1983

ELECTRONICS RESEARCH LABORATORY

Granularity Hierarchies in Concurrency Control

Michael J. Carey

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

ABSTRACT

This paper shows that granularity hierarchies may be used with many types of concurrency control algorithms. Hierarchical versions of a validation algorithm, a timestamp algorithm, and a multiversion algorithm are given, and hierarchical algorithm issues relating to request escalation and distributed databases are discussed as well. It is argued that these hierarchical algorithms should offer improved performance for certain transaction mixes.

1. Introduction

Considerable work in the area of concurrency control for both centralized and distributed database systems has focused on locking algorithms [Gray75, Ries77, Mena78, Rose78, Gray79, Ries79a, Ries79b, Ston79, Kort82]. In addition to studying alternative locking protocols and their correctness, some researchers have examined issues associated with selecting the appropriate level of *granularity* for partitioning a database into lockable units [Gray75, Gray79, Ries79a, Ries79b, Kort82]. It was found that a database can be organized as a hierarchy of lockable units, called a *lock hierarchy*, and that locking protocols for such a hierarchy can be developed [Gray75, Gray79, Kort82]. It was also found that, under some typical transaction mixes, a lock hierarchy offers increased system performance [Ries79a, Ries79b].

Since that time, a wide variety of new concurrency control algorithms have been suggested in the literature (see [Bern81] for a summary). Many of these algorithms are based on mechanisms other than locking, such as timestamps [Bern78, Reed78, Thom79, Bern81] or validation [Kung81, Ceri82]. Virtually all of these proposals ignore the granularity issue, modeling a database simply as a homogeneous, unstructured collection of fixed-size objects. Timestamp-based

This work was supported by the Air Force Office of Scientific Research Grant AFOSR-78-3596 and the Naval Electronic Systems Command Contract NESC-N00039-81-C-0569.

algorithms have been criticized for this very reason [Gray81]. In this paper, it is shown that the concept of granularity hierarchies can be generalized for use outside the domain of locking.

The remainder of this paper is organized as follows: Section 2 reviews the notion of hierarchical locking. In section 3, hierarchical versions of a validation algorithm, a timestamp algorithm, and a multiversion algorithm are presented. Issues involved in designing hierarchical versions of algorithms involving escalation and distributed databases are discussed in section 5. Section 6 presents arguments suggesting that generalized granularity hierarchies should lead to performance improvements, i.e., that the results reported for locking [Ries79a, Ries79b] should hold for other algorithms as well.

2. Hierarchical Locking

In locking algorithms, a transaction wishing to access some item in the database must first lock the item. A key performance question is: How big should the lockable items (or *granules*) be? To maximize potential concurrency for small transactions, many small granules are best, and to minimize locking overhead for large transactions, a few large granules are best. The notion of *hierarchical locking* was introduced to circumvent this performance tradeoff.

In hierarchical locking [Gray75, Gray79, Kort82], the database is viewed as a hierarchy of granules. When a transaction sets a lock on an item at a given level of the hierarchy, it is implicitly locking all its descendents as well. Small transactions obey a locking protocol whereby they set intention locks at higher levels of the hierarchy before setting access locks at a lower level. Large transactions can then avoid setting many lower-level locks. The result is a small increase in locking overhead for small transactions, a penalty which is hopefully offset by a large decrease in locking overhead for large transactions.

3. Generalized Granularity Hierarchies

In this section we present several hierarchical concurrency control algorithms based on mechanisms other than locking. For a granule g, the notation *parent*(g) will refer to the granule immediately above g in the hierarchy. The notation *children*(g) will refer to the set of granules right below g in the hierarchy. The notation *descendents*(g) will refer to the set of all descendents of g in the hierarchy. Finally, the notation *ancestors*(g) will refer to the set of all ancestors of g in the hierarchy. Granules at the bottom level of the hierarchy

will be referred to as *leaf* granules.

3.1. Hierarchical Validation

One popular alternative to locking is validation [Kung81, Ceri82], also referred to as certification or optimistic concurrency control. In this section, a hierarchical version of the serial validation algorithm of [Kung81] will be presented.

```
procedure validate(t);
begin
  valid := true;
  foreach t_rc in T_rc(t) do
    foreach x_r in readset(t) do
      foreach x_w in writeset(t_rc) do
        if x_r = x_w then
          valid := false;
        fi;
      od;
    od;
  od;
  if valid then
    commit writeset(t) to database;
  else
    restart(t);
  fi;
end;
```

Figure 1: SV Algorithm.

The serial validation (SV) algorithm requires that the readsets and writesets of all transactions be recorded, and that a deferred update strategy be used for commit processing. When a transaction t wishes to commit, it is subjected to a validation procedure in a critical section of code. Let $T_{rc}(t)$ be the set of recently committed transactions, those which commit between the time when t starts executing and the time at which t enters the critical section for validation. Transaction t is validated if $readset(t) \cap writeset(t_{rc}) = \phi$ for all transactions $t_{rc} \in T_{rc}(t)$. If t is validated, its updates are applied to the database; otherwise, it is restarted. The serial validation algorithm is given in Figure 1.

It is fairly easy to extend this algorithm for use with a granularity hierarchy. For hierarchical serial validation (H-SV), the read and write sets of a transaction will be sets of granules. Short transactions may specify these sets in

terms of small granules, and large transactions may specify them in terms of granules higher up in the granularity hierarchy. As in SV, these sets are used for commit-time conflict testing, with transaction t being validated if $readset(t) \cap writeset(t_{rc}) = \phi$ for all transactions $t_{rc} \in T_{rc}(t)$. In testing for possible conflicts under H-SV, the algorithm must recognize that a granule g_1 has some data in common with another granule g_2 if $g_1 = g_2$, $g_1 \in ancestors(g_2)$, or $g_2 \in ancestors(g_1)$. The H-SV algorithm is given in Figure 2. It is fairly easy to prove the correctness of this new algorithm.

```
procedure validate(t);
begin
  valid := true;
  foreach t_rc in T_rc(t) do
    foreach g_r in readset(t) do
      foreach g_w in writeset(t_rc) do
        if g_r = g_w or g_r ∈ ancestors(g_w) or g_w ∈ ancestors(g_r) then
          valid := false;
        fi;
      od;
    od;
  od;
  if valid then
    commit writeset(t) to database;
  else
    restart(t);
  fi;
end;
```

Figure 2: H-SV Algorithm.

H-SV Theorem: The hierarchical version of SV is correct in the sense that serializability is guaranteed.

Proof Sketch: The SV algorithm is known to be correct [Kung81]. Thus, it suffices to show that H-SV only commits transactions which would be committed by SV. This may be shown as follows:

When a transaction t requests access to a granule g, it is requesting permission to access some or all of the granules in $descendents(g)$. The H-SV algorithm will restart t if any granule g_r in its readset either contains, equals, or is a sub-granule of another granule g_w in the writeset of any recently committed transaction. In the first two cases, the newly-written leaf granules associated

with g_w in the hierarchy will definitely overlap with the leaf granules associated with g_r under SV, so t will be restarted under SV as well. In the latter case, where g_r is a sub-granule of g_w, the newly-written leaf granules potentially overlap with those of g_r, so t might be restarted under SV as well. If none of these cases occur, no overlap of leaf granules exists, and both H-SV and SV will permit t to commit. ▪

3.2. Hierarchical Timestamps

Another popular alternative to locking in the literature is timestamp-based concurrency control [Bern78, Reed78, Thom79, Bern81]. In this section, a hierarchical version of the basic timestamp ordering algorithm of [Bern81] will be presented. For simplicity, we present a version which applies to single site database systems. Also, we assume that all write requests are processed together at commit time, which simplifies the considerations involved in making the algorithm work with two-phase commit, as otherwise some scheduling would be required to prevent transactions from reading objects for which a write request has been processed but the associated deferred update [Gray79] has not yet taken place. Extending the algorithm to overcome these simplifications is straightforward.

Associated with each transaction t in the basic timestamp ordering (BTO) algorithm is a timestamp, $TS(t)$, issued at the time at which t begins executing. Associated with each data item x in the database is a read timestamp, $R\text{-}TS(x)$, and a write timestamp, $W\text{-}TS(x)$. These are the largest timestamps of any read or write request, respectively, that has been processed for x. A read request from t for x is rejected if $TS(t) < W\text{-}TS(x)$, and a write request from t for x is rejected if $TS(t) < W\text{-}TS(x)$ or $TS(t) < R\text{-}TS(x)$. Transactions whose requests are rejected are restarted, causing serialization to occur in timestamp order. The BTO algorithm is given in Figure 3.

To extend this algorithm for hierarchical use, each granule g will have read and write *summary* timestamps, $R_s\text{-}TS(g)$ and $W_s\text{-}TS(g)$, in addition to its actual read and write timestamps. Its read and write summary timestamp values will be:

```
procedure readReq(t,x);
begin
  if TS(t) < W-TS(x) then
    restart(t);
  else
    grant readReq;
    R-TS(x) := max(TS(t),R-TS(x));
  fi;
end;

procedure writeReq(t,x);
begin
  if TS(t) < R-TS(x) or TS(t) < W-TS(x) then
    restart(t);
  else
    grant writeReq;
    W-TS(x) := TS(t);
  fi;
end;
```

Figure 3: BTO Algorithm.

$$R_s\text{-}TS(g) = \max\{R\text{-}TS(G) \mid G \in g \cup \textit{descendents}(g)\}$$
$$W_s\text{-}TS(g) = \max\{W\text{-}TS(G) \mid G \in g \cup \textit{descendents}(g)\}$$

The actual read (write) timestamp for a granule g is the largest timestamp of any transaction for which a read (write) request for g has been granted. The summary read (write) timestamp for g is the largest timestamp of any transaction for which a read (write) request for g or any sub-granule of g has been granted. With these timestamps at each level of the hierarchy, the BTO algorithm requires two extensions. First, when a transaction t wishes to access a granule g, it must make sure that no granule in $\textit{ancestors}(g)$ has an actual timestamp that, when compared with $TS(t)$, violates the BTO ordering rules. This would mean that some transaction younger than t has already made a request that potentially conflicts with t's request. Second, the algorithm must propagate timestamp changes upwards in the hierarchy to keep the summary timestamp values accurate. The hierarchical version of BTO (H-BTO) is given in Figure 4. It is not difficult to prove the correctness of this new algorithm.

H-BTO Theorem: The hierarchical version of BTO is correct in the sense that serializability is guaranteed.

Proof Sketch: The BTO algorithm is known to be correct [Bern82a]. Thus, it suffices to show that H-BTO only permits accesses which would be permitted by BTO. This may be shown as follows:

When a transaction t requests access to a granule g, it is requesting permission to access some or all of the granules in $descendents(g)$. The H-BTO algorithm refuses read requests in two cases:

(1) $TS(t) < W\text{-}TS(G)$ for some $G \in ancestors(g)$

(2) $TS(t) < W_s\text{-}TS(g)$

The first case guarantees that, if some transaction younger than t has been allowed to write a granule which contains g (and thus to write g), t cannot read g. This would not be allowed under BTO. The second case guarantees that, if some transaction younger than t has been allowed to write some portion of g, t cannot read g. This is also disallowed under BTO. If neither (1) or (2) hold, H-BTO grants the request. Since this occurs only when no transaction younger than t has written g or any portion thereof, implying that no write timestamps of leaf granules associated with g in the hierarchy would exceed $TS(t)$ in the BTO algorithm, BTO would grant the request as well.

The H-BTO algorithm refuses write requests in two cases:

(1) $TS(t) < R\text{-}TS(G)$ or $TS(t) < W\text{-}TS(G)$ for some $G \in ancestors(g)$

(2) $TS(t) < R_s\text{-}TS(g)$ or $TS(t) < W_s\text{-}TS(g)$

The first case guarantees that, if some transaction younger than t has been allowed to read or write a granule which contains g (and thus to read or write g), t cannot write g. This would not be allowed under BTO. The second case guarantees that, if some transaction younger than t has been allowed to read or write some portion of g, t cannot write g. This is also disallowed under BTO. If neither (1) or (2) hold, H-BTO grants the request. Since this occurs only when no transaction younger than t has read or written g or any portion thereof, implying that no read or write timestamps of leaf granules associated with g in the hierarchy would exceed $TS(t)$ in the BTO algorithm, BTO would grant the request as well. ▪

To illustrate the roles played by the actual and summary timestamps in H-BTO, consider the simple hierarchy of Figure 5, where there are two lower-level

```
procedure readReq(t,g);
begin
  okay := true;
  foreach G in ancestors(g) do
    if TS(t) < W-TS(G) then
      okay := false;
    fi;
  od;
  if TS(t) < W_s-TS(g) then
    okay := false;
  fi;
  if not okay then
    restart(t);
  else
    grant readReq;
    R-TS(g) := max(TS(t),R-TS(g));
    R_s-TS(g) := max(TS(t),R_s-TS(g));
    while parent(g) exists do
      g := parent(g);
      R_s-TS(g) := max(TS(t),R_s-TS(g));
    od;
  fi;
end;


procedure writeReq(t,g);
begin
  okay := true;
  foreach G in ancestors(g) do
    if TS(t) < R-TS(G) or TS(t) < W-TS(G) then
      okay := false;
    fi;
  od;
  if TS(t) < R_s-TS(g) or TS(t) < W_s-TS(g) then
    okay := false;
  fi;
  if not okay then
    restart(t);
  else
    grant writeReq;
    W-TS(g) := TS(t);
    W_s-TS(g) := TS(t);
    while parent(g) exists do
      g := parent(g);
      W_s-TS(g) := max(TS(t), W_s-TS(g));
    od;
  fi;
end;
```

Figure 4: H-BTO Algorithm.

granules, X and Y, and one upper-level granule, XY. Suppose that $R\text{-}TS(X) = 8$, $W\text{-}TS(X) = 8$, $R\text{-}TS(Y) = 15$, $W\text{-}TS(Y) = 13$, $R\text{-}TS(XY) = 5$, and $W\text{-}TS(XY) = 5$. This implies that, while X and Y have been accessed since time 5, their parent granule XY has not been accessed as a whole since that time. The summary timestamp values will be $R_s\text{-}TS(X) = 8$, $W_s\text{-}TS(X) = 8$, $R_s\text{-}TS(Y) = 15$, $W_s\text{-}TS(Y) = 13$, $R_s\text{-}TS(XY) = 15$, and $W_s\text{-}TS(XY) = 13$. (Note that the actual and summary timestamp values are always the same for leaf granules, so it is not actually necessary to maintain them separately at the bottom level of the hierarchy.)

Now, suppose that a transaction t with timestamp $TS(t) = 10$ wishes to read X. H-BTO checks $W\text{-}TS(XY)$, finds that the request is okay so far, then checks $W_s\text{-}TS(X)$, finds that the request is indeed okay, and then grants the request, setting $R\text{-}TS(X)$, $R_s\text{-}TS(X)$, and $R_s\text{-}TS(XY)$ all equal to 20. Suppose instead that t had wished to read XY. H-BTO would have checked $W_s\text{-}TS(XY)$, found that the request violated the BTO ordering rules for reading because some sub-granule of XY had been written since time 10, and the request would have been rejected.

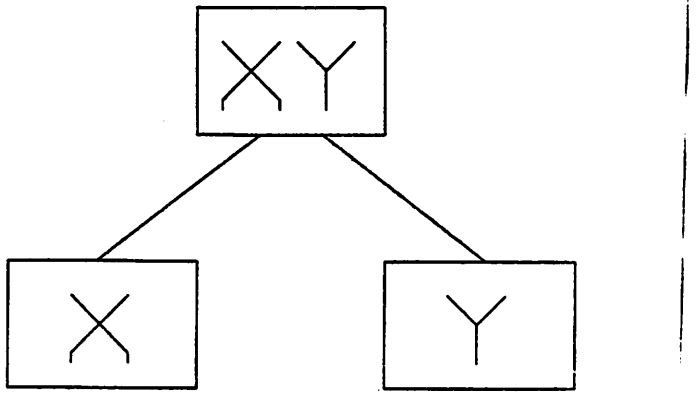

Figure 5: Simple Example Hierarchy.

3.3. Multiple Versions

There have been a number of recent papers proposing the use of multiple versions of data to increase potential concurrency [Reed78, Baye80, Stea81, Chan82, Bern82b]. Hierarchical versions of these algorithms are possible as well. In this section, a hierarchical variant of the multiple version algorithm of [Reed78] will be presented. For simplicity, we present a variant which applies to single site database systems. We assume that all write requests are processed together at commit time (for the same reasons as in basic timestamp ordering). We also assume that all versions are maintained for all time, ignoring garbage collection issues. Extending the algorithm to overcome these simplifications is not overly difficult, and doing so does not affect the ideas to be presented here. Finally, we treat version management and concurrency control separately in our description. This will prove helpful in developing a hierarchical variant of the algorithm.

The multiversion timestamp ordering (MVTO) algorithm is similar to the BTO algorithm in many ways. Associated with each transaction t is a timestamp, $TS(t)$, issued at the time at which t begins executing. Associated with each data item x in the database is a collection of versions, a set of (*time*,*value*) pairs indicating values of x and the (timestamp) times at which the values were assigned to x. If T is a timestamp, let $x[T]$ be the value of the most recent version of x written at time less than or equal to T. A read request from t for x will be granted using the value $x[TS(t)]$, and a write request from t for x, if granted, will result in the creation of a new version of x.

For concurrency control purposes, a read/write history, $H_{rw}(x)$, is maintained for each data item x. This history is a set of time intervals which correspond to versions of x. The starting time of each interval is the creation timestamp of the version, and the ending time of each interval is the largest read timestamp of the version. For example, $H_{rw}(x) = \{(3,6), (10,13)\}$ means that x has two versions, one created at time 3 and last read at time 6, and the other created at time 10 and last read at time 13. Histories in MVTO play the role which timestamps played in BTO, allowing the concurrency control algorithm to know when potential conflicts arise.

The MVTO algorithm grants all read requests using the appropriate version, extending the interval for the version read in $H_{rw}(x)$ as necessary. A write request from t for x is rejected if any interval in $H_{rw}(x)$ contains the time $TS(t)$. (Otherwise, transactions which previously read x between $TS(t)$ and the

end of the interval containing $TS(t)$ would have their reads invalidated by t's write.) If the write request is granted, a new version of x is created, and a new interval with starting and ending times of $TS(t)$ is added to $H_{rw}(x)$. Continuing with our previous example, a read request for x from a transaction t with $TS(t) = 7$ would be granted using the version of x created at time 3, and the history for x would be changed to $H_{rw}(x) = \{(3,7), (10,13)\}$. A write request from t for x would now be rejected if $3 < TS(t) < 7$ or $10 < TS(t) < 13$. If $TS(t) = 8$, however, the request would be granted, a new version of x would be created, and the history for x would be changed to $H_{rw}(x) = \{(3,7), (8,8), (10,13)\}$.

The MVTO algorithm is given in Figure 6. In the figure, the *extVers* operation is assumed to extend the version interval corresponding to $x[TS(t)]$ in $H_{rw}(x)$ if $TS(t)$ is larger than the ending time of the interval. The *newVers* operation is assumed to create a new interval in $H_{rw}(x)$, starting at $TS(t)$ and having length zero, recording the creation of a new version of x.

```
procedure readReq(t,x);
begin
  grant readReq;
  extVers(H_rw(x), TS(t));
end;

procedure writeReq(t,x);
begin
  if TS(t) in H_rw(x) then
    restart(t);
  else
    grant writeReq;
    newVers(H_rw(x), TS(t));
  fi;
end;
```

Figure 6: MVTO Algorithm.

To extend this algorithm for hierarchical use, each granule g will have a read/write *summary* history, $H_s(g)$, in addition to its actual history, $H_{rw}(g)$. This summary history will be:

$$H_s(g) = \bigcup \{H_{rw}(G) \mid G \in g \cup descendents(g)\}$$

Thus, $H_s(g)$ is the union of $H_{rw}(G)$ for all granules G having any data in common with g. This union operation may be interpreted graphically. The

read/write history for a granule can be thought of as a timeline, with the intervals in the history being line segments drawn on this timeline. The union of two or more histories is what would be produced by laying these timelines on top of each other, with the intervals in the resulting history being those intervals included in one or more of the histories being unioned. For example, the union of {(3,7), (10,13)} and {(1,2), (5,11) (15,17)} would be {(1,2), (3,13), (15,17)}.

These actual and summary histories are analogous to the actual and summary timestamps used in creating the H-BTO algorithm from the BTO algorithm. With these histories at each level of the hierarchy, the MVTO algorithm requires two extensions. First, when a transaction t wishes to write a granule g, it must make sure that no higher-level granules have actual histories with an interval containing $TS(t)$. Second, when a transaction t causes some history to be updated, the algorithm must propagate the change upwards in the hierarchy to keep the summary histories accurate. For the lowest level granules in the hierarchy, the actual and summary histories will always be equal (just as for timestamps in the H-BTO algorithm), so they need not be separately maintained for leaf granules.

The hierarchical version of MVTO (H-MVTO) is given in Figure 7. Is is assumed in the figure that the *newVers* operation creates a new interval in a history by taking the union of the history and the new interval, and that the *extVers* operation merges intervals when extending one causes it to overlap with another. It is not difficult to prove the correctness of this new algorithm.

H-MVTO Theorem: The hierarchical version of MVTO is correct in the sense that serializability is guaranteed.

Proof Sketch: The MVTO algorithm is known to be correct [Bern82a]. Thus, it suffices to show that H-MVTO only permits accesses which would be permitted by BTO. This may be shown as follows:

When a transaction t requests access to a granule g, it is requesting permission to access some or all of the granules in *descendents*(g). The H-MVTO algorithm always grants read requests, just as the MVTO algorithm does. The H-MVTO algorithm refuses write requests in two cases:

(1) $TS(t)$ is in an interval in $H_{rw}(G)$ for some $G \in ancestors(g)$

(2) $TS(t)$ is in an interval in $H_s(g)$

The first case guarantees that, if a granule containing g has an interval which contains $TS(t)$, indicating that the version of g to be written may have already been read by a younger transaction, t cannot write it. This would not be allowed under MVTO. The second case guarantees that, if some granule contained within g has an interval which contains $TS(t)$, indicating that some portion of the version of g to be written has already been read by a younger transaction, t cannot write it. This is also disallowed under MVTO. If neither (1) or (2) hold, H-MVTO grants the request. Since this occurs only when no portion of g has a version which was written before $TS(t)$ and read after $TS(t)$, implying that none of the read/write histories of the leaf granules associated with g in the hierarchy would have intervals containing $TS(t)$ under MVTO, MVTO would grant the request as well. ▪

4. Extensions

In this section, the issues involved in extending generalized granularity hierarchies for use with escalation and distributed databases are discussed.

4.1. Escalation

In hierarchical locking algorithms, a decision must be made about whether a transaction is to access leaf granules or higher-level granules. This decision can either be made statically, with transactions being given the responsibility for selecting the appropriate level of granularity at which to make their requests, or it can be made dynamically, by *escalation* [Gray75, Gray79]. In escalation, when a transaction crosses a pre-determined threshold in the number of locks requested for granules at one level of the hierarchy, it moves up a level, escalating its requests.

Escalation may be employed in systems based on any of the hierarchical algorithms presented in this paper in the same manner that it is employed in systems based on locking. The concurrency control subsystem monitors requests made for granules at the current level of the hierarchy, and if this number becomes larger than some threshold number, translates them into requests for granules at the next level up in the hierarchy. This can be repeated in systems with more than two levels of granules. All three algorithms presented

```
procedure readReq(t,g);
begin
  grant readReq;
  extVers(H_rw(g), TS(t));
  extVers(H_s(g), TS(t));
  while parent(g) exists do
    g := parent(g);
    extVers(H_s(g), TS(t));
  od;
end;

procedure writeReq(t,g);
begin
  okay := true;
  foreach G in ancestors(g) do
    if TS(t) in H_rw(G) then
      okay := false;
    fi;
  od;
  if TS(t) in H_s(g) then
    okay := false;
  fi;
  if not okay then
    restart(t);
  else
    grant writeReq;
    newVers(H_rw(g), TS(t));
    newVers(H_s(g), TS(t));
    while parent(g) exists do
      g := parent(g);
      newVers(H_s(g), TS(t));
    od;
  fi;
end;
```

Figure 7: H-MVTO Algorithm.

in this paper will accommodate such a protocol with no modifications being required.

4.2. Distributed Systems

One of the conclusions resulting from the work of Ries [Ries79a] is that distributed systems may benefit even more from hierarchical concurrency control algorithms. Hierarchical versions of distributed concurrency control algorithms are easily developed, the only new issue being how to deal with granules in the hierarchy whose descendents are located at several sites.

In the hierarchical algorithms presented here, as in hierarchical locking, a request for access to a granule g requires two things to be done: The concurrency control state of granules in $ancestors(g)$ must usually be checked before the request can be granted, and, if the request is granted, some information must be propagated to these granules as well. In a distributed system, if some portion of the concurrency control information for $ancestors(g)$ resides on a site other than the site where g is stored due to data partitioning or replication, this may involve messages which would not be required in a non-hierarchical version of the algorithm. Such additional messages could result in a severe performance problem, so they should be avoided.

There are at least two ways to avoid the problem of additional messages. One approach is to use a central site algorithm for handling distributed concurrency control [Mena78, Bern81], where a single site is responsible for handling all concurrency control requests. Another approach is to use a primary copy algorithm [Ston79, Bern81], and to assign data to sites in such a way that $ancestors(g)$ and g always have the same primary site. For example, one may choose to have a two-level hierarchy, with relations being the higher level of granularity and pages within a relation being the lower level, assigning primary sites on a per-relation basis. Other distributed algorithms may also be applicable (see [Bern82a] for a survey of scheduler location strategies for distributed concurrency control algorithms).

5. Granularity and Performance

The effects of locking granularity and the use of lock hierarchies on database management system performance was the subject of a fairly extensive simulation study [Ries77, Ries79a, Ries79b]. Briefly, [Ries79b] showed that if transaction access patterns are random in nature, the expected tradeoff between parallelism and locking overhead indeed arises between small and large transactions, and that performance is indeed improved by using a lock hierarchy.

The role of any concurrency control algorithm is to prevent transaction conflicts by ensuring that each transaction sees a consistent view of the database. Each concurrency control algorithm for which we have developed a hierarchical version shares the following properties:

(1) The algorithm can be viewed as a scheduler which monitors requests from transactions and responds with *okay*, *block*, or *restart* [Papa79, Bern82a].

(2) In the absence of restarts, the amount of CPU time used by the algorithm to process the set of read and write requests for a transaction t is proportional to the number of requests that t makes.

(3) The level of transaction parallelism achievable is proportional to the number of granules into which the database is partitioned.

(4) For a hierarchical version of the algorithm, the CPU cost of processing a request for a granule is proportional to its depth in the hierarchy times the CPU cost of processing the request in its non-hierarchical counterpart.

These concurrency control algorithm properties hold for locking as well as for the other algorithms discussed in the paper. In fact, it seems clear that these are the properties which lead to the result, reported in [Ries79a, Ries79b], that hierarchical locking outperforms non-hierarchical locking under transaction mixes consisting of small and large transactions with random access patterns. Since the algorithms described here share these properties with locking, it seems quite plausible that their hierarchical counterparts will offer superior performance under such mixes of small and large transactions.

6. Conclusions

This paper has shown that the notion of a lock hierarchy is generalizable, indicating that a granularity hierarchy may be used with many types of concurrency control algorithms. Hierarchical variants of a validation algorithm, a timestamp algorithm, and a multiversion algorithm have been developed, and hierarchical algorithm issues relating to request escalation and distributed databases have been discussed as well. Finally, it has been argued that these hierarchical algorithms will offer improved performance for some mixes of small and large transactions.

Several related avenues of research remain. First, it would be interesting to study the performance of hierarchical versions of concurrency control algorithms, comparing their performance to that of their non-hierarchical counterparts, under some set of realistic assumptions about transaction access patterns. Second, decentralized or voting algorithms have been suggested as superior alternatives to primary site and primary copy algorithms for robust distributed database systems. It would be interesting to see if decentralized or voting

versions of hierarchical algorithms can be developed without introducing unreasonable message overheads.

Acknowledgements

The author wishes to thank Mike Stonebraker for his helpful comments, suggestions, and support.

References

[Baye80] Bayer, R., Heller, H., and Reiser, A., "Parallelism and Recovery in Database Systems", ACM Transactions on Database Systems 5(2), June 1980.

[Bern78] Bernstein, P., Rothnie, J., Goodman, N., and Papadimitriou, C., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", IEEE Transactions on Software Engineering 4(3), May 1978.

[Bern81] Bernstein, P., and Goodman, N., "Concurrency Control in Distributed Database Systems", ACM Computing Surveys 13(2), June 1981.

[Bern82a] Bernstein, P., and Goodman, N., "A Sophisticate's Introduction to Distributed Database Concurrency Control", Proceedings of the Eighth International Conference on Very Large Data Bases, September 1982.

[Bern82b] Bernstein, P., and Goodman, N., "Multiversion Concurrency Control Theory and Algorithms", Technical Report No. TR-20-82, Aiken Computation Laboratory, Harvard University, June 1982.

[Ceri82] Ceri, S., and Owicki, S., "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases", Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks, February, 1982.

[Chan82] Chan, A., Fox, S., Lin, W., Nori, A., and Ries, D., "The Implementation of An Integrated Concurrency Control and Recovery Scheme", Proceedings of the ACM-SIGMOD International Conference on Management of Data, March 1982.

[Gray75] Gray, J., Lorie, R., Putzulo, G., and Traiger, I., "Granularity of Locks and Degrees of Consistency in a Shared Database", IBM Research Report No. RJ1654, September 1975.

[Gray79] Gray, J., "Notes On Database Operating Systems", in Operating Systems: An Advanced Course, Springer-Verlag, 1979.

[Gray81] Gray, J., "The Transaction Concept: Virtues and Limitations", Proceedings of the Seventh International Conference on Very Large Databases, September 1981.

[Kort82] Korth, H., "Deadlock Freedom Using Edge Locks", ACM Transactions on Database Systems 7(4), December 1982.

[Kung81] Kung, H., and Robinson, J., "On Optimistic Methods for Concurrency Control", ACM Transactions on Database Systems 6(2), June 1981.

[Mena78] Menasce, D., and Muntz, R., "Locking and Deadlock Detection in Distributed Databases", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 1978.

[Papa79] Papadimitriou, C., "Serializability of Concurrent Updates", Journal of the ACM 26(4), October 1979.

[Reed78] Reed, D., "Naming and Synchronization in a Decentralized Computer System", PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1978.

[Ries77] Ries, D., and Stonebraker, M., "Effects of Locking Granularity on Database Management System Performance", ACM Transactions on Database Systems 2(3), September 1977.

[Ries79a] Ries, D., "The Effects of Concurrency Control on Database Management System Performance", PhD Thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1979.

[Ries79b] Ries, D., and Stonebraker, M., "Locking Granularity Revisited", ACM Transactions on Database Systems 4(2), June 1979.

[Rose78] Rosenkrantz, D., Stearns, R., and Lewis, P., "System Level Concurrency Control for Distributed Database Systems", ACM Transactions on Database Systems 3(2), June 1978.

[Stea81] Stearns, R., and Rosenkrantz, D., "Distributed Database Concurrency Controls Using Before-Values", Proceedings of the ACM-SIGMOD International Conference on Management of Data, March 1981.

[Ston79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", IEEE Transactions on Software Engineering 5(3), May 1979.

[Thom79] Thomas, R., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", ACM Transactions on Database Systems 4(2), June 1979.