A Tailorable Conflict Manager For Flexible Concurrency Control

§ Michael Flanagan, § Fred Curtis, § Alan Fekete, ¶ Frans Henskens and § John Rosenberg

§ Basser Department of Computer Science, F09,
University of Sydney, NSW 2006, Australia.
¶ Information Systems Group, Department of Management
University of Newcastle, NSW 2308, Australia

Keywords: transaction management, concurrency control, advanced transaction models

1. Introduction

For twenty years, the transaction has been acknowledged as the central abstraction in preventing concurrent applications from corrupting the contents of a database, through errors such as lost update, dirty read or unrepeatable read [1]. The original concurrency control algorithm, strict two-phase locking with shared and exclusive locks, is still widely used in practice, since it is simple to implement and guarantees serializability. Many alternative algorithms have been proposed and, in commercial systems these include variants of key-range locking to avoid phantoms, and escrow reads to improve throughput on hotspot data, as discussed in [2]. New algorithms continue to appear. These algorithms are usually evaluated by simulation rather than being implemented. For example, a constrained shared lock has been proposed in [3].

Besides algorithms which offer alternative implementations for the traditional transaction semantics (ACID properties), there have been many new models proposed, for use in advanced application domains where cooperation is needed between concurrent activities. A detailed survey of these new ideas is found in [4]. Each new model needs one or more algorithms to provide concurrency control.

Traditionally, the choice of transaction model and even concurrency control algorithm in a DBMS has been made when the system is designed. The systems offers a fixed set of transaction management primitives, such as begin-transaction, or commit; also the lock manager has a fixed set of lock modes and unalterable rules for dealing with conflicts. For example, the lock manager described in [2] is hardwired so that a process blocks when another holds a conflicting lock. This is unable to deal with nested transactions or timestamp-based algorithms. This paper describes a system based on a different view. We offer a system architecture where the choice of transaction model and concurrency control algorithm can both be made at run-time; indeed different algorithms can be used simultaneously on different parts of the database.

The value of flexible concurrency control in a DBMS can be seen in two different dimensions. First, within a single transaction model, such as classical ACID transactions, it makes sense to construct a database with a simple concurrency control algorithm and later use data-type specific information to upgrade the algorithm for those items that are hotspots in an attempt to increase throughput. There is a substantial body of theory available to guide this process [5]. Second , the different transaction models are each useful in their own application domain; if a system supports only one model, then either its use will be restricted to the domain where that model is valid, or else the application programmers will need to waste time in finding work-arounds. Our work was actually motivated by the requirements of persistent stores, which support persistent programming languages; here the store replaces a conventional file system, so its need for broad support of many different application domains is even more clear.

The key idea of our system is that there is a "conflict manager" that fills the role of a traditional lock manager, but contains an interpreter for a small stack-based language. When a particular algorithm is chosen to provide concurrency control for some part of the data, the user sends strings written in the language to the conflict manager, and binds them to certain function names. Later when access to the data is needed, the functions are executed in the conflict manager. This results in a range of outcomes such as blocking the requestor, allowing it to continue, or even sending a signal to a waiting process. Since many traditional algorithms are based on classes of conflicting locks, our system contains a "fast-path" so that a table of conflict rules may be expressed in a particularly simple fashion.

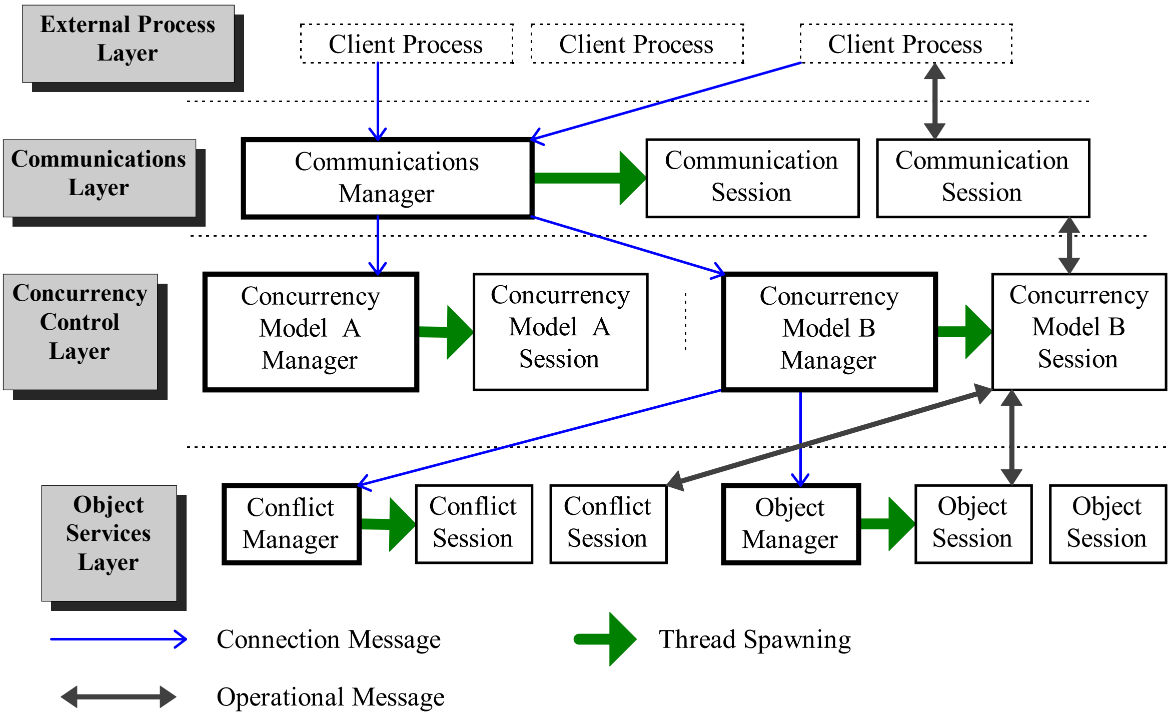
The system we describe here is part of a project in concurrency control for persistent systems. The persistent store has been implemented, as has the conflict manager. We have installed and tested Two Phase Locking, Multi-granularity Locking and Notification Locking algorithms. The first two provide a classical transaction model, while the latter supports cooperative transactions. Thus we have demonstrated our system's flexibility.

The prior work most closely related to ours is the Kala system [6], which also supports a range of concurrency control mechanisms. However, Kala provides a fixed collection of powerful primitives to allow and disallow sharing of versions, rather than a language executed at run-time, and so Kala is less flexible than our proposal. The ASSET system [7] which is based on the ACTA formalism, and a similar system by Georgakoupolous [8] both allow for variation in the transaction model. In these systems, a transaction model is defined by presenting dependencies: for example, that T1 cannot commit until after T2 has committed. A dependency of this sort will arise in certain cases based on access to objects by the transactions. Both systems [7] and [8] schedule operations to enforce whatever constraints have been specified in the transaction model. These systems operate at a higher, more declarative level than ours does. These systems can describe various transaction models, but they do not contain the stack based language which allows our system to also execute different algorithms within a single transaction model (For example neither [7] nor [8] suggests a way to model Multi-Granularity Locking).

The rest of the paper is structured as follows. Section 2 describes the overall architecture of our system, and especially the interaction patterns between the conflict manager and the rest of the system. Section 3 outlines the calls that are supported, and take place as an application is running. Section 4 briefly describes the stack-based language, in which the algorithm is expressed. Section 5 discusses briefly how the conflict manager can be programmed to follow the multi-granularity locking algorithm. A more detailed description of the conflict manager interface and language, together with an example, may be found in [9].

Published in: Proceedings, 1996 International Conference on Intelligent Information Systems, Washington DC, USA, ISMM/IASTED, 1996.

Fig. 1. System Layers and Messages



2. System Overview

The system is designed to provide flexible concurrency control over data residing in an object repository, or *store* (the term object is used in a more generic manner than in object oriented paradigms). The system provides run time selection of concurrency control requirements by providing a layered structure in which different tasks can be selected, within certain layers, to provide the run time support for the chosen concurrency scheme. The system is divided into layers as depicted in Figure 1. The major components which make up the system are the Communications Manager, the Concurrency Managers, the Object Manager and the Conflict Manager.

The various components exist concurrently as multi-threaded tasks which communicate with each other via a message passing protocol. When an external process connects to the system to request data services its causes new threads to be spawned within the various layers of the system. These new threads combine to provide dedicated service to the external process. Thus each external process connects to what is effectively a vertical slice through the system. The term “manager” is used to refer to the task responsible for initialising and spawning new threads in each layer of the system. The threads which are spawned to handle client process services are referred to as “sessions”. Thus the “Conflict Manager” and “Conflict Session” implement the Conflict Management Module. The responsibilities and activities of each of the system components are described below.

The Communication Manager is responsible for all communication with client processes. The Communication Manager is responsible for managing the connection process and selecting a concurrency manager of the type appropriate for the client process. All issues of communication such as pipe or socket management reside within this part of the system. All internal communication between the system components is handled via an internal message passing protocol.

The various Concurrency Model Managers each implement a different concurrency control scheme. Concurrency Model Managers register themselves with the Communication Manager and are activated when a client process requests a connection to the service corresponding to their registered name. The concurrency control manager and session are responsible for establishing a connection to the Conflict Manager and the Object Manager and providing any necessary initialisation, such as lock tables or functions for the Conflict Manager. The Concurrency Model Session then receives messages from the client process which are appropriate to the scheme being implemented. The Concurrency Model Session must translate the client requests into requests for data services or conflict services. This involves such actions as translating simple read and write requests into requests for locks and access to the Object Manager.

The Object Manager provides an interface to a simple object store similar to those which are commonly found in persistent systems. The store implements a type of object which is simpler in structure to objects found in object oriented systems. Objects consist of a number of bytes of uninterpreted data and a collection of references to other objects. Concurrency Model Managers make requests of the Object Manager to perform tasks such as reading, writing, creating, deleting objects and version management.

The objects within the store are given object identifiers (OIDs) which are unique throughout the lifetime of the store. The Object Manager provides caching and garbage collection services transparent to the other modules.

The Conflict Manager, which is the focus of this paper, provides a configurable conflict management service to the various Concurrency Model Managers. The Conflict Manager is initialised,

Published in: Proceedings, 1996 International Conference on Intelligent Information Systems, Washington DC, USA, ISMM/IASTED, 1996.

by a Concurrency Model Manager, with the necessary tables and functions for a given concurrency scheme. The conflict manager then receives requests from a Concurrency Model Session and interprets the requests using the installed conflict resolution procedures. The initialisation of the Conflict Manager, the processing of requests and the language used to code the required functions and tables are described in detail in this paper.

3. The Conflict Manager Interface

The role of the Conflict Manager is to provide services to the Concurrency Model Manager to handle tasks involving the determination of resource conflicts within the concurrency scheme. While the Concurrency Model Manager handles issues such as transactional structure, what should be locked, tagged or read and when such actions are performed, the Conflict Manager determines which associations cause conflicts or which requests cause the notification or blocking of other processes.

The Conflict Manager interface is divided into two sections. One section of the interface provides the facility for specifying a conflict scheme, i.e. initialising the Conflict Manager for a specific scheme. The other section of the interface accepts request messages which are interpreted using the tables and functions of the registered schemes.

The specification section of the Conflict Manager interface consists of functions for:

- gaining a connection to the Conflict Manager,
- installing code for the functions of a concurrency scheme
- querying the conflict manager for the value of scalars and identifiers for registered functions.

A description of the process of gaining a connection to the Conflict Manager involves a description of the task management policy and message passing protocols of the system. This falls outside the scope of this paper. Part of this initialisation process involves the specification of a file name which identifies a file containing code for the Concurrency Model Manager. This code specifies the functions and tables necessary to implement the conflict resolution of the desired scheme.

The functions used to query the values of scalars and function identifiers are:

```
ModeID CMGetModeID(SysTask cm, char *mode);
```

and

```
FunctionID CMGetFunctionID(SysTask cm, char
*f_name);
```

These two functions allow the Concurrency Model Manager to get numerical identifiers which are associated with the named functions or modes. This allows the Concurrency Model Manager to specify these modes or functions without the Conflict Manager needing to do time consuming string comparisons.

The messages described above are used by a Concurrency Model Manager during the initialisation phase. When these initialisation procedures are completed, individual Concurrency Model sessions (child tasks of the Concurrency Model Manager) can acquire a Conflict Manager connection and begin to perform association requests, free association etc. The messages which are used to perform the tasks are discussed below.

Messages may be sent from a concurrency model session to perform the following tasks:

- request an association on a resource
- free an association on a resource
- invoke a registered function to perform some action (e.g. commit and abort).

To request an association a concurrency session sends the message:

```
Bool request_assoc(Resource res, AssocMode
mode)
```

This message requests that the resource name *res* be associated with the requesting task in the supplied *mode*. This causes the Conflict Manager to invoke the function specified with the name `requestAssoc`. Note that this action does not necessarily result in the creation of a lock in the conventional sense. It can result in some alternative type of association being created between the session and the resource, which could for example result in the notification of some other sessions. Also note that the “resource” is just a string; it need not be directly linked to any object in the store (although it often is so linked).

Similarly to the request_assoc message a concurrency session may send the message,

```
Bool release_assoc( Resource res, AssocMode
mode)
```

which will cause the invocation of the function registered as *releaseAssoc.* This message requests the Conflict Manager to remove the requesting task from the list of tasks which hold associations with the resource. Concurrency models may allow multiple tasks to hold compatible associations with a single resource so a *release_assoc* message will not necessarily leave a resource free of all associations.

To cause a transaction to commit or to perform some other action appropriate to the current concurrency control model, a task may send the message,

```
Bool invoke_function(FunctionID action,
    char *format, ...).
```

This message will cause the conflict manager to invoke the function with the given id. The ‘C’ function which acts as a stub for this message takes a variable argument list and a format string similar to the ‘C’ I/O functions. This allows a variable collection of arguments to be passed to the Conflict Manager function. The *invoke_function* message allows a scheme to implement such actions as:

- freeing all locks in the case of commit or abort in conventional transactions
- passing all locks onto the parent transactions on completion of a nested transaction
- etc.

The above mentioned messages allow concurrency models to register functions with the Conflict Manager and invoke functions. These functions are written in a small stack based language which will be described in the next section.

4. The Conflict Manager Language

The Conflict Manager language is a small stack based language similar to Forth or Postscript. The language is designed to be applicable to the development of small functions which determine simple conflict states and keep track of associations between tasks and resources.

The language includes facilities for

- function definition,
- table definition,
- scalar definition,

- process control,
- function invocation,
- flow control,
- list manipulation,
- association structure access
- table access,
- arithmetic operations and
- stack manipulation.

In addition to this the language has a collection of built in functions.

Function definition is achieved through the use of the *def* operator, which has the form:

<name><codeBlock> **def**

This associates the code in the codeblock with the supplied name. The name, which is preceded by a '/' character as in PostScript, is associated with a numerical identifier which may be used to uniquely and efficiently identify the function from outside the Conflict Manager. Within the Conflict Manager code the function is referred to by its ASCII name.

Scalar definition is similar. The *scalardef,* with the syntax:

<name><scalarList> **scalardef**

operator is used, as in the following example, to define an enumerated type. E.g.

/mode [/NONE /READ /WRITE /LASTMODE]
scalardef

Tables are defined with the *tabdef* operator which has the form:

<name> <entry$_{00}$> ... <entry$_{NM}$> <width> <height> **tabdef**

This operator takes a collection of table entries and a width and a height and creates a two dimensional array with the supplied name. This array may then be accessed using the language's table manipulation operators.

Process Control in the language involves the concept of a task which is a connection to a Conflict Manager session from a Concurrency Model session. Task management is performed using the operators *block* and *wake*. The *block* operator allows a Conflict Session to suspend the invoking task (the Concurrency Model Session to which it is connected) and label its suspension with a resource and an associated mode, so that it can later be awoken conditionally. The *wake* operator allows a Conflict Session to wake up another task. The structure of these operators is a follows:

<resource> <mode> **block**
<task> **wake**

Function invocation is performed by the *call* and *execTable* operators. The *call* operator takes a function id from the top of the stack and invokes the code associated with this function. As the language is stack based, parameters are passed and results returned by placing values on the stack before, and at the end of, invocation. The format of the *call* operator is simply:

<functionId> **call**

The *execTable* operator provides function invocation from a two dimensional lookup table. Its form is:

<index$_1$> <index$_2$> <tableID> **execTable.**

Another form of call is provided by the *callback* operator. When executed, the *callback* operator will invoke the callback function specified during registration of the scheme. The callback operator has the form:

<list> **callback**.

Flow control in the language is supported by the following operators:

<start> <inc> <stop> { *code_block* } **for**
{ code_block } **while**
{ code_block } **until**
<cond> { code_block } **if**
<cond> { btrue } { bfalse } **ifelse.**

The *for* operator steps through the integers from start to stop, using the incremental value inc, and executes the code block once for each integer after placing the integer on the top of the stack. The *while* and *until* provide loops by testing the top of stack and only executing the block of code if the value is true. *While* tests the stack before each invocation and until tests after, thus until must execute the block at least once. The *if* and *ifelse* operators execute the code segment conditionally depending on the value of the top of stack. *Ifelse* provides an alternative code segment to execute if the top of stack holds **false**.

List manipulation is performed by the use of the *makelist*, *addhead*, *addtail*, *head*, *tail* and *joinlist* operators. In addition to these operators the language provides operators for iterating over a list. The simple operators have the following syntax:

<i_1>... <i_n> n **makelist** ⇒ [i_1...i_n]
[i_1...i_n] [j_1...j_m] **joinlist** ⇒ [i_1...i_n, j_1...j_m]
[i_1...i_n] <j> **addhead**⇒ [j, i_1...i_n]
[i_1...i_n] <j> **addtail** ⇒ [i_1...$i_{n,}$ j]
[i_1, i_2...i_n] **head** ⇒ [i_2...i_n] I_1
[i_1,...i_{n-1}, i_n] **tail** ⇒ [i_1...i_{n-1}] i_n

Makelist converts the top n items on the stack into a single list item. *Addhead*, *addtail* and *joinlist* allow incremental list construction by adding elements to the beginnings and ends of lists or concatenating two lists.

To facilitate iteration over a list of elements the language provides the operator lfor, land and lor. The syntax for these operators is:

[i_1..i_n] { code_block } **lfor**
[i_1..i_n] { code_block } **land**
[i_1..i_n] { code_block } **lor**.

The *lfor* operator iterates over a whole list by placing subsequent elements from the list onto the stack then executing the code block. The *land* and *lor* operators provide short cut evaluation of a predicate over the contents of a list. The *land* operator executes the code block once for each element of the list (after placing the element on the top of stack as in *lfor*) until the code block returns a false value on the top of stack or the list is consumed. If the block of code returns true for all list elements the final value true is left on the top of stack otherwise a value of false is returned. The *lor* operator performs similarly with disjunction instead of conjunction.

The language supports a built in type called an association. This type represents a binding between a task (Concurrency Model session) and a resource which is tagged with a mode. Thus it contains the three fields: owner, resource, mode. The language has the following operators for manipulating this structure

<o:owner><r:resource><m:mode> **makeassoc** ⇒ <(o,r,m):
lock>

<(o,r,m):assoc> **assocowner** ⇒ <o:owner>

<(o,r,m):assoc> **assocres** ⇒ <r:resource>
<(o,r,m):assoc> **assocmode** ⇒ <m:mode>.

These operators allow the construction and separation of associations.

The table access functions provided by the language allow the implementation of two dimensional arrays. The operators used for this are:

<tableid> <row> <col> **tget** ⇒ <table element>
<tableid> <row> <col> <element> **tput**

Operators also exist for the manipulation of lists as single element arrays. These are *lget* and *lput* which behave similarly to their two dimensional counterparts.

The language also contains operators which are similar to and in many cases the same as those provided by Postscript. They include: *dup*, *add*, *sub*, *neg*, *exch*, *rol*, *ndup*, *pop*, <, > etc.

To facilitate the specification of a conflict management scheme in the language, the Conflict Manager has several reserved names. These names are

- mode - the name of the scalar definition used to identify valid association modes
- releaseAssoc - the name of the function called in response to the *release_assoc* message.
- requestAssoc - the name of the function called in response to the *request_assoc* message.
- maxTable - the name of a table of association modes used by the predefined function *MaxMode*.

The language includes a number of predefined functions which have predefined function identifiers. These functions perform common tasks which are expected to be needed in most systems. These functions are called through the use of the *call* operator in the same way as registered functions but all their function identifiers fall within a reserved range which will never be returned as the identifiers of registered functions. Some of these functions manipulate a built-in table shared among all Conflict Sessions, which stores all of the registered associations between resources and tasks.

The two functions for storing and removing associations from the built-in table are:

<lock> *storeAssoc* **call**
<lock> *deleteAssoc* **call**

The *deleteAList* function is called as follows:

[$lock_1$...$lock_n$] *deleteAList* **call**

It examines each association record in the supplied list and removes the referenced association from all internal tables, thus freeing the resource from this association.

The holds_list and blocked_list functions:

<res><mode> *holds_list* **call** ⇒ [$lock_1$...$lock_n$]
<res><mode> *blocked_list* **call** ⇒ [$lock_1$...$lock_n$]

return a list of all locks held on, or blocked during request of, resource 'res' in mode 'mode'. The mode may be replaced by the reserved mode **any_mode** to get a list of all locks held on 'res'.

The *max_mode* function

<res> *max_mode* **call** ⇒ <max_mode>

uses the lock maximization table registered with the name *maxTable* to determine the upper bound of the modes of all locks on resource 'res'. This function is used in schemes such as Multi Granularity Locking to determine if a requested lock is compatible with the locks already granted.

The language includes predefined functions which make use of parent/child relationships between tasks using the conflict manager. These functions include:

<t_1><t_2> *is_ancestor* **call** ⇒ <boolean>
True if t_1 is ancestor of t_2
<transaction> *parent* **call** ⇒ <task_id> Parent of transaction

These functions are applicable to schemes such as nested transactions where there is a family tree structure which relates all tasks which use the lock manager.

The function *task_locks*

<task> <mode> *task_locks* **call** ⇒[$lock_1$...$lock_n$]

returns a list of all locks held by the task with identifier 'task' which hold a resource in the given mode. Again the reserved mode **any_mode** may be used in place of a specific mode and will result in the return of a list of all the locks help by 'task' in any mode.

The next section discusses how the language features and functions described above, may be used to implement a Multi-granularity locking scheme.

5. An Example: Multi-granularity Locking

The multi-granularity locking example discussed here is a very simple one. It assumes a system consisting of two levels of objects. When a client locks more than a given threshold number of subobjects then the parent object is locked. More details (including substantial code fragments) can be found in [10].

5.1. Conflict Manager Functions

To implement Multi-granularity locking we must register a group of functions with the conflict manager.

The function Gproc is called from a jump table to grant a lock. It makes use of the built in function *store_lock* which adds a lock to the granted queue for a given resource. It assumes that a lock structure exists on the argument stack which contains the necessary lock details. The text of the Gproc function is:

```
Gproc {
    STORE_LOCK call
    false
}
```

It leaves **false** on the stack which will be used in other code as an indication that the lock request did not block.

The Bproc function, as with Gproc, is called from a jump table when the requested mode conflicts with the currently granted mode of a resource. It uses the predefined function BLOCK to cause the current thread to suspend execution until it is awoken using the WAKE function. The BLOCK function takes a resource and a lock mode so that threads may be selectively reactivated depending on the resource and mode provided when they executed a BLOCK call. Bproc extracts the resource and mode from the lock structure which it assumes is on the stack. Bproc and Gproc must assume the same stack contents when invoked as they are both called from the same jump table. The text of Bproc is:

```
   Bproc {
      dup          // copy the lock
structure on the TOS
      lockres exch // get resource and
swap with
                      // lock structure on
TOS
      lockmode     // TOS now holds res,
mode
      BLOCK call
      true
   }
```

The Gproc function returns the value **true** to indicate that blocking took place.

The GetLock function is used to process a lock request. It first determines the maximum lock mode under which the requested resource (r_res) is currently locked by calling the predefined function MAX_MODE. It then uses this value and the value of the requested mode as indices to the LockRequestTable (shown figuratively below) to select either Gproc or Bproc depending on the lock compatibility. If a block occurs the function called from the LockRequestTable will leave the value **true** on the stack which will cause the while loop to repeat when the thread is awoken. In this manner the process will continue until the lock is successfully acquired. As GetLock will be called from outside the interpreter (i.e. in direct response to a Concurrency Model Manager request) it makes use of the external argument symbols r_owner, r_res and r_mode. The text of the GetLock function is:

```
   GetLock {
      {
         // create lock structure as arg
for Gproc or Bproc
         r_owner r_res r_mode makelock
         // find maximum current mode
         r_res MAX_MODE call
         r_mode
         // index jumptable using
requested and maximum modes
         LockRequestTable exectable
      } while
   };
```

In the above code the symbol *LockRequestTable* would be replaced by the identifier returned when the table was registered. The registering of this table is discussed below.

The FreeLock function is used to wake any processes which are blocked waiting for the given lock. The processes will then compete to acquire the lock. The text of the FreeLock function is:

```
   FreeLock {
      dup DELETE_LOCK call
      lockres ANY_MODE BLOCKED_LIST call
         { lockMode wake }
      lfor
   }
```

The function assumes a lock structure is on the stack indicating which lock is to be freed. It duplicates this lock structure (dup) then frees it using the built in operator *delete_lock*. The function then uses the built in operator lockRes to extract the resource name from the lock structure and wakes any tasks which are in the list of tasks blocked on this resource. Note this implementation is inefficient in that many processes may be wakened while only one may successfully get the lock. Note also that this method does not support fifo granting of lock requests. If fifo granting is desired, or if efficiency is required, the wake operator could be applied only to the head of the list returned by BLOCKED_LIST.

The EndProc function is used to clean up after a transaction has committed or aborted. It simply gets the list of locks held by the current transaction using the predefined function TASK_LOCKS and then frees all locks in the list using the FreeLock function.

```
   EndProc {
      r_trans ANY_MODE TASK_LOCKS call
      { FreeLock call } lfor
   }
```

5.2. Concurrency Model Manager overview

Rather than present the Multi-granularity Model Manager code, we merely outline how the code works.

Firstly the abovementioned Conflict Manager functions must be registered. This is done by passing a character string to the function *register_function* and storing the result in a variable of type FunctionID. Next the tables for lock maximization and the jump table called *LockRequestTable* are registered using the *register_table* function. The two tables are passed textually, with entries so that in the lock maximization table, the entry corresponding to the row for mode S and the column corresponding to IX has value SIX, to indicate that when a transaction has both an S and an IX lock on an item, the effect is the same an SIX lock. Similarly, in the lock request table, the row corresponding to S and the column corresponding to IX has entry Bproc, to indicate that when a transaction requests an S lock on an item already locked (by another transaction) in IX mode, the requestor must be blocked.

Having registered all required functions and tables the Concurrency Model Manager can register the scheme with the conflict manager by a call to the new_control function.

In addition to registering the functions and tables needed for conflict management the code for the Concurrency Model Manager must include functions to handle connections, disconnections and requests for reads and writes to objects. The main message loop of the Concurrency Model Manager uses a two level object hierarchy with simple reads and writes of second level objects. If more than a certain number (threshold) of locks are requested for a group of siblings then a lock is taken out on their common parent..

Acknowledgements

We are grateful for financial support from the Australian Research Council under grant A49232246.

Conclusions

We have presented a system design that can support flexible, and even dynamic, choice of concurrency control algorithm and transaction model. The key idea is to have a programmable conflict manager that maintains “locks” that are associations between a transaction and a resource name. The conflict manager can interpret a small stack-based language whose details are in this paper. When a concurrency control scheme is chosen, one can register appropriate functions to obtain and release locks. Later, when an application is running, these functions are executed, which results in transactions being blocked, allowed to proceeed, or woken up, as specified by the concurrency control algorithm. The system has been implemented; in [10] its flexibility is demonstrated by showing how several different algorithms are expressed.

References

[3] D. Agrawal and A. El Abbadi, “Locks with Constrained Sharing” in

Proceeedings ACM PODS: 85-93, April 1991

[7] A. Biliris, S. Dar, N.Gehani, H. Jagadish, K. Ramamritham, “ASSET: A system for supporting extended transactions” in *Processdings ACM Sigmod: 44-54, May 1994.*

[4] A. Elmagarmid (ed), *Database Transaction Models for Advanced Applications*, Morgan Kauffman 1992.

[10] M. Flanagan, *Concurrency Control for a Persistent Object Store*, Ph.D. thesis, Department of Computer Science, University of Sydney, January 1996.

[9] M. Flanagan, F. Curtis, A. Fekete, F. Henskens and J. Rosenberg, “A Tailorable Conflict Manager for Fl;exible Concurrency Control”, Basser Department of Computer Science Technical Report, University of Sydney, Australia, 1996.

[1] K.Eswaran, J. Gray, R. Lorie, I. Traiger, “The Notion of Consistency and Predicate Locks in Database Systems” in *Comm. ACM 19(11): 624-633, November 1976.*

[8] D. Georgakopoulos, M. Hornick, P. Krychniak, F. Manola, “Specification of Extended Transactions in a Programmable Transaction Environment” in *Proceedings International Conference on Data Engineering,* 1994.

[2] J. Gray and A. Reuter, *Transaction Processing*, Morgan Kaufmann 1993.

[6] S. Simmel and J. Godard “The Kala Basket” in *Proceedings OOPSLA* 1991.

[5] W. Weihl, “Local Atomicity Properties: Modular Concurrency Control for AbstractData Types” in *ACM TOPLAS 11(2): 249-282, April 1989.*