Multi-Node Multi-Level Transactions[1]

Lory D. Molesky and *Krithi Ramamritham*
Computer Science Technical Report 94-30
Department of Computer Science
Lederle Graduate Research Center
University of Massachusetts
Amherst Ma. 01003-4610
March 1994

Keywords: *multi-level transactions, transaction modeling, extended transactions, database recovery, shared-disk database system, shared-memory database system.*

Abstract

Multi-Level Transactions (MLT) offer a useful framework for increasing transaction concurrency in database systems. In this paper, we develop various approaches to implementing MLT in a *shared disk multi-node system.* By combining correctness and implementation specifications, we present concrete insights into the resulting tradeoffs between the degree of concurrency, the complexity of recovery, and runtime overheads in multi-node MLT. The main challenges include efficiently supporting subtransaction inverses on a multi-node system when coupled with performance enhancing mechanisms, such as fine-granularity locking, coarse-grain cache coherency, and coarse-grain persistence mechanisms. For example, in a two level implementation of multi-node MLT, serializability is achieved with respect to objects, and cache coherency is achieved with respect to pages. The migration of a page containing uncommitted objects may violate subtransaction atomicity and cause inter-transaction abort dependencies. These problems can be overcome by adjusting the object-to-page mapping policies, and by enforcing specific stability properties (whether an object or operation is logged, is stable logged, or is persistent) at page migration. These and other tradeoffs are investigated in a logical fashion by methodically studying (1) mechanisms invoked in response to significant events (such as *Commit* and *Abort*), (2) policies for object-to-page mapping, and (3) stability policies. The resulting implementation options are formally characterized by tailoring the ACTA formalism to include specifications of these implementation oriented considerations, in addition to (the traditionally specified) transaction model requirements.

[1]Partially supported by the National Science Foundation under grant IRI-9109210

Contents

1 **Introduction** 1

2 **MMLT** 2

2.1 Transaction Model 2

2.2 System Model 4

2.3 Correctness Requirements and Performance Related Properties 5

3 **Implementation-Oriented Components of MMLT** 6

3.1 Basic Formal Concepts 7

3.2 Stability 8

3.3 Significant Events 10

4 **Synthesis of MMLT Implementations** 11

4.1 Supporting Subtransaction Atomicity in MMLT – Maintaining Persistence Spheres 12

4.2 MMLT Implementations 14

4.2.1 $WS = PS$ 15

4.2.2 $WS \neq PS$ 17

4.3 Other Performance Related Options 19

5 **Summary and Conclusions** 19

1 Introduction

In this paper, we examine issues raised when Multi-level Transactions (MLT) [2, 18, 21, 22] are implemented in a shared disk multi-node system. MLT exploits the semantics of high-level operations in order to increase concurrency. Thus, in addition to providing fine granularity locking, concurrent updates are possible by exploiting the semantics of *object specific* operations, such as increment and insert. Moreover, the use of inverse operations simplifies recovery in MLT. These performance advantages of MLT for a single node system are demonstrated in [12]. Our work is motivated by a desire to reap the additional performance advantages that can be gained by implementing MLT in a multi-node system. The benefits include the potential for higher transaction throughput that can be obtained via the parallel execution of transactions and operations. However, implementing MLT in a multi-node environment raises several new questions including:

- How does the support for object specific operations and their inverses affect a multiple-node implementation?
- What are the implications of using page-grain I/O mechanisms in the presense of transactions that perform object-level operations?
- How does committing a transaction on one node affect transactions on other nodes?
- What are the implications of allowing uncommitted pages (containing uncommitted objects or portions of them) to migrate between caches?

Even a cursory examination of these questions reveals that when implementing MLT on multiple nodes, tradeoffs arise involving the degree of concurrency, the flexibility of mapping objects to pages, the complexity of recovery, and runtime overhead costs. For example, in MLT, recovering from the abortion of transactions that invoke object specific operations is effectively supported with operation or subtransaction inverses (also called as *compensating operations*), for which subtransaction atomicity is essential. One way to enforce subtransaction atomicity, is to use the persistence spheres mechanism[22]. In a *force* strategy, all updates made by a transaction are made persistent at transaction commit time, while in a *no-force* strategy, a checkpoint operation is used to periodically make (some set of) subtransactions persistent. In these contexts, the use of persistence spheres removes the need to make every subtransaction of *every* transaction persistent. However, in multi-node MLT, pages updated by uncommitted transactions may migrate between nodes. This increases the cost of determining persistence spheres in a multi-node system thereby increasing transaction commit processing overheads. Fortunately, by proper choice of the object-to-page mapping policy, these commit processing overheads can be significantly reduced, but, this will reduce the flexibility of storing objects in pages. Also, another problem still remains: if pages are allowed to migrate without logging or making stable some information about the changes made by a committed subtransaction, cascading transaction aborts can occur when nodes fail. But, specific *stability* policies can guarantee the absence

of these aborts, at the cost of increased transaction run-time overheads. *Stability* policies specify whether undo or redo information for objects (or operations) are logged, stable logged, or made persistent when the page containing the object (or effects of a subtransaction) migrates. For instance, one can combine stable logging to support the undo mechanism with either logging or persistence to support the redo mechanism. If none of these mechanisms is used, that is, changes are not logged or made persistent prior to page migration, the best runtime performance is achieved but the transactions involved are subject to inter-node cascading aborts.

Given the many tradeoffs that exist, our goal is to precisely characterize the implementation options for addressing these issues in the context of the different components of multi-node MLT (MMLT). Specifically, by methodically studying (1) the mechanisms invoked in response to significant events (such as commit and abort), (2) the policies for object-to-page mapping, and (3) the stability policies, the resulting implementation options are formally characterized by tailoring the ACTA formalism [5, 6] to include specifications of these implementation oriented considerations, in addition to (the traditionally specified) transaction model requirements. Our approach of combining correctness and implementation specifications yields concrete insights into the tradeoffs mentioned earlier.

The rest of this paper is structured as follows: The next section motivates the need for MMLT by contrasting and comparing it to MLT, and discusses correctness and performance related issues. Section 3 covers the implementation oriented components of MMLT, which include components that deal with durability issues, object to page mapping policies, and the significant events associated with MMLT. Section 4 uses the components and mechanisms discussed in section 3 to synthesize specifications of implementations of MMLT. Within the confines of the MLT transaction model, these specifications vary based on the stability policies and their impact on recovery, and the object-to-page mapping policies. Section 5 concludes the paper by discussing related work and summarizes the salient aspects of the possible implementations of MMLT.

2 MMLT

In this section, details of the MMLT transaction model and system model are presented. Correctness and performance related properties of a database system are then discussed.

2.1 Transaction Model

In MMLT, the structure of the transaction model is multi-level. Our protocols can be generalized to an arbitrary number of levels, but for ease of discussion, in the rest of this paper we assume a two level model consisting of a top level (level 2) transaction, object operations (subtransactions) in the middle level (level 1), and page operations on the bottom level (level 0). This structure is illustrated in figure 1 (permissible operations on pages are read and write). Subtransactions correspond to operations on objects, and these object operations are implemented by a series of page operations. For ease of explanation we begin by studying subtransactions which access a

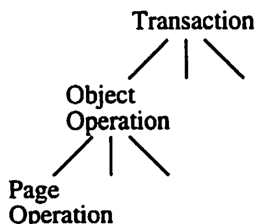

Figure 1: Transaction Model

single object, but subsequently (in section 4) this assumption is relaxed. For objects, we focus on an *arbitrary* operation model, which is an extension of the *r/w* (read/write) operation model. Under the r/w operation model, subtransactions perform reads and writes on objects, while under the arbitrary operation model, object-specific operations such as increment, modify, and insert are possible.

Associated with each operation on an object is an inverse operation which logically undoes the effect of the operation. Inverses allow more concurrency for object-specific operations. Consider *inc*(*ob*) (increment on an object *ob*). Without inverses, an exclusive object lock would typically be held for the duration of the transaction updating object *ob*. Thus, concurrent updates by different transactions will not be allowed. With an inverse operation (*dec*(*ob*)), concurrent updates can be allowed, since, if the incrementing transaction aborts, the abort can be effected by simply performing *dec*(*ob*) (that is, without affecting other non-conflicting updaters). Thus, instead of an exclusive lock on object on *ob*, an *operation-specific* lock on *ob* would be held.

The multi-level operation hierarchy enables increased concurrency through a multi-level locking protocol. To perform an operation on an object, an operation-specific lock is first acquired on the object, then *page locks* are acquired on all the pages containing the object that is accessed by the operation. Note that, should a page which is stored on another node be accessed, the page must migrate to the accessing node prior to being locked. When the object operation is complete, the page locks are released. However, the object lock is released only when the transaction completes. Thus, in this locking-based approach, transactions employ 2PL (two-phase locking) on objects, while subtransactions use 2PL on pages.

Objects can be entirely contained within one page, or may span multiple pages. Multiple objects can be contained in one page. Object specific operations are performed on objects. However, these operations translate into reads and writes of pages holding the objects. Conventional computing systems are optimized for performing page granularity I/O. This mismatch between the granularity of abstract operations and physical I/O operations is the primary contributor to the complexity of concurrency control and recovery. For example, assume object *ob* spans multiple pages p and q, and object ob' is entirely contained in page p. If t^a on node a updates *ob* and then t^b on node b updates ob', node $a's$ cache will contain page q while node $b's$ cache contains p (figure 2 illustrates this scenario). In this situation, *inter-node cascading aborts* may

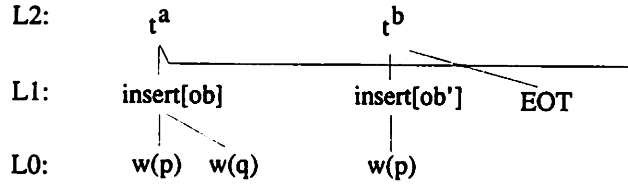

Figure 2: Object *ob* spans multiple pages p and q.

occur – i.e. if node b crashes, it will not only loose t^b's updates, but also some updates of t^a (those contained just in page p).

2.2 System Model

We consider a shared disk multiprocessor model where each node in the system has direct access to a single shared disk, as illustrated in figure 3. We assume transactions execute on a single node, and for a given transaction, operations execute sequentially. Each node contains a cache and local memory, which we collectively refer to as a cache. The cache comprises the volatile store, while the disk is the stable (permanent) store. Node failures are independent; all data in the node's volatile store is lost on a node failure. The log can reside in the cache or on disk, and thus can be volatile or permanent. We consider both types of logs in this paper.

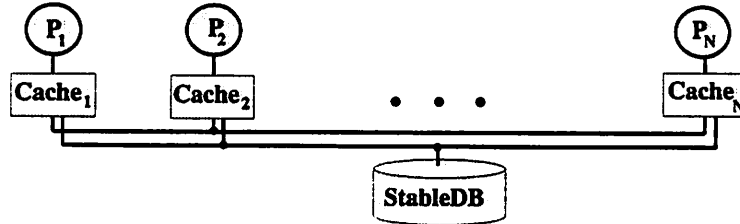

Figure 3: Multiprocessor Model

On a multiprocessor, a coherency protocol ensures that for all operations, the most recent updates are seen by a transaction. A data item is cached in a processor's cache prior to access by the processor. The cache line size (we assume this is a page) defines the unit of sharing between caches. Typically, this coherency protocol is implemented in hardware, implementing coherent read and write operations – reads may execute concurrently with other reads, but a write operation may not execute concurrently with a read or a write. $X^a(p)$ denotes that node a's cache holds exclusive access to a page p, while $S^a(p)$ denotes shared access. Consider any node $c \neq a$. Unless $X^a(p)$ is already true, the invocation of $w^a[p]$ (*write* to page p on node a) triggers $X^a(p)$, and negates any $X^c(p)$ that is currently true. Similarly, unless $S^a(p)$ or $X^a(p)$ is already true, the invocation of $r^a[p]$ (*read* of page p on node a) triggers $S^a(p)$, and negates any $X^c(p)$ that is currently true. $X^a(p)$ holds until some other node $c \neq a$ performs $op^c[p]$. Thus, X and S model the fact that the changes to pages are done in a mutually exclusive fashion. Note that *locked* pages cannot migrate, that is, should node a have a lock on a page p, for node b such that $b \neq a$, $X^b(p)$ cannot become true until the lock on page p has been released.

We assume that the processor/cache interface provides sufficient low-level *hooks* for the DBMS to define stability policies. Stability policies are used primarily to log or stable log updates prior to page migration. For example, by using a primitive which pins a page in cache, a transaction can ensure that an update is logged before the page could possibly migrate. We discuss object stability further in section 3.2 and stability policies are discussed in section 4.

Based on these implications of read and write, we consider how page level cache coherency affects *objects*. Of primary interest is under what conditions *object fragmentation* occurs. Object fragmentation is said to occur if the most recent state of an object is contained in multiple caches. Obviously, object fragmentation can occur only if an object spans multiple pages. The history $insert^a[ob]$; $insert^b[ob']$, based on figure 2, illustrates object fragmentation. Object ob, spanning pages p and q, becomes fragmented since after the second insert operation, page p is held exclusively in node b's cache ($X^b(p)$) while page q is held exclusively in node a's cache ($X^a(q)$). Note that fragmentation of ob can occur only as a side effect of some operation on a different object stored in one of the pages where ob is stored. Object fragmentation is disadvantageous since it makes subtransaction atomicity more difficult to ensure. In section 4, we discuss conditions under which object fragmentation can be avoided.

2.3 Correctness Requirements and Performance Related Properties

The selection of the components which implement MMLT is driven by the correctness requirements and the desired performance-related properties. While satisfying the correctness requirements is not optional, satisfying performance-related properties involves carefully weighing tradeoffs between them. The following are the primary correctness requirements for MMLT.

- *Serializability*: This is achieved via the multi-level two-phase locking protocol described in section 2.1.
- *Transaction Atomicity*: Transaction atomicity ensures that either all or none of the effects of a transaction affect the database.
- *Transaction Persistence*: Transaction persistence ensures that the effects of committed transactions will not be lost.

Transaction atomicity and persistence are achieved with the appropriate recovery and commit processing mechanisms. Alternatives for implementing these mechanisms and correctly supporting subtransaction inverses (as required by the MLT transaction model) in MMLT are discussed in detail in subsequent sections. In addition to the above requirements, the following performance-related properties are *desirable* in any implementation.

- *No Cascading Aborts.*
  The abortion of one transaction triggering the abortion of another transaction is called a cascading abort. In a multi-node system, inter-node cascading aborts may be caused

by node crashes. Based on application characteristics, in order to obtain better overall system performance, the database system implementor may be willing to risk inter-node cascading aborts.

- *Fast Normal Runtime Processing.*
  Because mechanisms that ensure that no cascading aborts require that changes done by subtransactions be logged or made persistent, these mechanisms have the side effect of degrading the normal runtime operation. In contrast, *fast normal runtime processing* denotes situations when low transaction processing overhead is achieved by avoiding these overheads.

- *Low Commit Processing Overheads.*
  Subtransaction atomicity enables the support of subtransaction inverses. In general, to achieve subtransaction atomicity, *Persistence Spheres* [22] must be computed. Computing persistence spheres can be expensive, unless they are kept small or are restricted to one transaction. This problem can be eliminated by proper choice of object-to-page mapping policies.

The above discussion of correctness requirements and performance considerations clearly shows that there are tradeoffs involved. For example,

- While subtransaction inverses provide for more concurrency and help in recovery, mechanisms designed to correctly support them incur commit processing delays.

- Requiring that no inter-node cascading aborts occur can degrade normal runtime operation since subtransactions are logged or are made persistent.

These tradeoffs are unique to multi-node systems. Our goal is to isolate the components of a database system involved in these and other tradeoffs, propose implementations which best handle these tradeoffs, and/or allows the implementation designer to select the properties to optimize in a database system.

3 Implementation-Oriented Components of MMLT

In this section we present the major implementation-oriented components of our approach – *durability*, *cache coherency*, and *significant events*. The specification and combination of these components is based on the ACTA formalism, the basic formal concepts of which are reviewed in section 3.1. Section 3.2 discusses durability issues by focusing on the object model and basic transaction processing mechanisms, emphasizing a classification of data objects based on their volatility and object type. The basic recovery mechanisms, *Logging*, *Stable Logging*, *Persistence*, *Undo* and *Redo*, and their effect on operations, objects, and pages are discussed. The combination of the volatility properties of database objects, the cache coherency protocol, and the stability policies at page migration determine the recovery requirements. Section 3.3

discusses the significant transaction events, such as *Commit* and *Abort*, and system events, such as *Crash* and *Mig* (page migration). Our approach to formally modeling the entire system hinges on assertions relating the events in the history. This is accomplished as follows: The events discussed in section 3.3 trigger the invocation of (one of many possible) mechanisms which further trigger other events.

3.1 Basic Formal Concepts

During the course of their execution, transactions invoke operations on objects as well as transaction management primitives, such as, *Commit* and *Abort*. We will introduce the other transaction management events incrementally. Transaction i is denoted t_i. s_i is a subtransaction of some transaction, $s \in subtrs(t)$ denotes that s is a subtransaction of t. ϵ_t denotes the transaction management event ϵ pertaining to t. $op_t[ob]$ denotes the event corresponding to the invocation of the operation op on object ob by transaction t. In general, multiple objects can be stored in one page, and a single object can span multiple pages. Page objects, corresponding to physical pages, are denoted p and q. Where convenient, when $ob \in p$ (object ob is contained in page p), we substitute $op_t[p]$ for $op_t[ob]$. In a multi-node system, a superscript applied to t_i, s_i, or op, (i.e. t_i^x, s_i^x, or op^x) denotes the node where t_i, s_i, or op executes. In MMLT, subtransactions have compensating operations: the inverse of $s_t^x[ob]$ is $s_t^{x^{-1}}[ob]$.

The concurrent execution of a set of transactions T is represented by H, the *history* [3] of the significant events and the object events invoked by the transactions in the set T. H also indicates the (partial) order in which these events occur. This partial order is consistent with the order of the events of each individual transaction t in T. H_{ct} (the current history) is used to denote the history of events that occur until a point in time.

DEFINITION 3.1: The predicate $\epsilon \rightarrow \epsilon'$ is true if event ϵ *precedes* event ϵ' in history H. It is false, otherwise. (Thus, $\epsilon \rightarrow \epsilon'$ implies that $\epsilon \in H$ and $\epsilon' \in H$.)

We will use $pre(\epsilon)$ and $post(\epsilon)$ to denote the preconditions and postconditions respectively of an operation or a transaction management event ϵ. *Dependencies* are a simple way to characterize relationships between transactions.

DEFINITION 3.2: $(condition \Rightarrow (Abort_{t_j} \in H))$ specifies that if *condition* holds t_j aborts.

Invoked operations that have neither committed nor aborted are termed *in progress* operations. An operation is committed only if the invoking transaction commits and it is aborted only if the invoking transaction aborts. The following definitions relate objects and transactions to pages.

DEFINITION 3.3: $pages(ob)$, defines the set of pages in which object ob is stored.

DEFINITION 3.4: $WS(s)$, the *Write Set of a subtransaction* s, is the set of pages which are updated during the execution of s.

DEFINITION 3.5: $WS(t)$, the *Write Set of a transaction* t, is a set of pages, defined as the union of the write sets of all the subtransactions of t:

$$WS(t) = \{ \bigcup_i WS(s_i) \mid s_i \in subtrs(t)\}$$

DEFINITION 3.6: $PS(s_i)$, the Persistence Sphere of a subtransaction s_i, is a set of pages defined as follows:

$$PS(s_i) = \{ \bigcup_k WS(s_k) \mid (s_i \, \mathcal{I}^* \, s_k)\},$$

where two subtransactions are related by the binary relation $\mathcal{I}$ if they have intersecting write sets:

$$(s_i \, \mathcal{I} \, s_j) \text{ iff } (commit(s_j) \notin H_{ct}) \wedge (commit(s_i) \notin H_{ct}) \wedge (WS(s_i) \cap WS(s_j) \neq \emptyset).$$

That is, $PS(s_i)$ is transitively affected by the write sets of all uncommitted subtransactions. Although the PS is defined in terms of subtransactions, it can also be defined for transactions (by substituting all occurrences of s with t in definition 3.6). The persistence sphere is used to ensure subtransaction atomicity when making pages persistent. For example, consider the history illustrated in figure 2. Suppose t^b commits using force at EOT (end of transaction). If only page p ($WS(t^b)$) is made persistent, then the atomicity of insert[ob] may be violated due to a node crash. This is because, since updates made by *insert*[*ob*] on p are stable while those updates on q are volatile, node crash losing page q would invalidate the use of the inverse $insert^{-1}[ob]$. To avoid this problem, the persistence sphere is used whenever pages are made persistent. In this example, when t^b commits using force, the $PS(t^b)$ ($\{p, q\}$) is made persistent. Persistence spheres are also used in a no-force scheme, but only when performing a checkpoint operation.

3.2 Stability

Transactions perform operations on objects. When a transaction performs in-place updates, the effects of an operation are captured in the state of an object. We specify important features of a database implementation via set operations. The *Stable Database* is an example of one of these sets:

DEFINITION 3.7: The Stable Database, $StableDB$, is a set made up of the stable state of the objects forming the database. It is the stable state that the DBMS would have if a crash occurred, before any log information is applied [11].

We consider object and operation *stability* (volatility) policies as one component in the specification of a database implementation. We specify stability of operations, objects, and pages with mechanisms frequently used to implement the permanence of actions: *logged*, *stable logged* and *persistent*. Only pages are made persistent (for performance reasons), and either objects, or operations on objects, can be logged. This assumption of *logical* logging supports fine granularity locking, and reduces the amount of data written to the log [3, 10, 14, 16]. Thus, if

a transaction updates one object in a page containing multiple objects, only the updated object is logged. Events correspond to the invocation of these mechanisms. In the following definitions of these events, ϵ^a is used to indicate that ϵ is performed on node a. Consider subtransaction $s \in subtrs(t)$:

DEFINITION 3.8: Logged – A subtransaction is *logged* if the subtransaction itself or its effects are entered in the database log. Logging the subtransaction s on an object ob corresponds to the event $Log\text{-}s^a(s[ob])$, while logging the new state of ob that reflects the effects of s on ob corresponds to $Log\text{-}ob^a(s[ob])$.

DEFINITION 3.9: Stable Logged – A subtransaction is *stable logged* if the subtransaction itself or its effects are saved in stable storage so that the effects of the subtransaction can be made to manifest themselves in the stable database. Stable logging the subtransaction s on an object ob corresponds to the event $SLog\text{-}s^a(s[ob])$, while stable logging the effects of s on ob corresponds to $SLog\text{-}ob^a(s[ob])$. Typically, stable logging is achieved by logging the subtransaction itself or its effects, and then flushing the log to stable store.

Thus, *stable* logging a subtransaction (or object) implies that the log entry will survive a crash, while no such guarantee is associated with logging (since logging is performed *in memory*). Also note that to facilitate undo, we log subtransaction inverses instead of logging *before images*.

DEFINITION 3.10: Persistent – A subtransaction is *persistent* if its effects are installed in the stable database. Since the $StableDB$ is assumed to contain pages, persistence is applied only to pages. The event $Persist\text{-}page^a(s[p])$ corresponds to making the effects of subtransaction s on page p persistent (by writing p to the $StableDB$). $Persist\text{-}page^a$ can also take a set of pages as an argument, i.e., $Persist\text{-}page^a(p_1, p_2, ..., p_n)$ indicates that pages p_1 through p_n are made persistent. The event $Persist\text{-}page^a(WS(s))$ corresponds to making the effects of subtransaction s persistent by making all pages updated by s persistent.

Since a page may contain multiple objects, making a page persistent may make multiple objects persistent. Moreover, some of these objects may be *uncommitted* (updated by subtransactions of uncommitted transactions). This granularity mismatch between object operations and persistence significantly impacts commit processing in MLT, and, as we discuss in section 4, has even a more serious impact on commit processing in MMLT.

Associated with each of these events is a predicate. For example, the predicate $Logged\text{-}s^a(s[ob])$ is true if $Log\text{-}s^a(s[ob]) \in H_{ct}$, i.e. $Log\text{-}s^a(s[ob])$ appears in the current history. Predicates $Logged\text{-}ob^a(s[ob])$, $SLogged\text{-}s^a(s[ob])$, $SLogged\text{-}ob^a(s[ob])$, and $Persists\text{-}page^a(s[p])$, are defined similarly.

The Redo and Undo mechanisms use the log in order to recover the database:

DEFINITION 3.11: Redo – The Redo mechanism "redoes" the effects of operations. This mechanism is implemented by either reexecuting the named operation, or reinstalling a copy

Event	Associated Mechanisms
$Commit(s)$	{Stable Logging, Persistence}
$Abort(t)$	{Undo}
$Crash^a$	{Undo, Redo}
$Mig(a,b,p)$	{Logging, Stable Logging, Persistence}

Figure 4: Implementation Mechanisms triggered by Significant Events

of the object containing the effects of the named operations. The event $Redo\text{-}s^a(s[ob])$ occurs when $s[ob]$ is redone. The event $Redo\text{-}ob^a(s[ob])$ occurs when the object which contains the effects of $s[ob]$ is reinstalled.

DEFINITION 3.12: Undo – The Undo mechanism "undoes" the effects of operations. This mechanism is implemented by executing the inverse of the named operation. When the event $Undo\text{-}s^a(s[ob])$ is triggered, $s^{-1}[ob]$ is executed.

In section 4, different stability policies associated with page migration are specified based on whether logging, stable logging, or persistence is used on the migrating pages. Then, based on associating each stability policy, with a node crash predicate (discussed next in section 3.3), the recovery requirements are specified with events associated with *Abort*, *Undo*, and *Redo*.

3.3 Significant Events

Depending on the desired performance characteristics, one may wish to implement MMLT using a different combination of implementation policies and mechanisms. These different implementations can be specified based on the choice of mechanisms which are triggered by the occurrence of *significant events*. For MMLT, these significant events consist of $Commit(t)$, $Commit(s)$, $Abort(t)$, $Abort(s)$, $Crash^a$, and $Mig(a,b,p)$. The implementation mechanisms which are triggered by these events are summarized in figure 4 and justified below.

For performance reasons, a couple of assumptions are made about the implementation of these mechanisms. Given the performance advantages of page grain I/O, only *page* persistence is used. To minimize the frequency of I/O, persistence of a subtransaction is not required until transaction commit. Thus, $Commit(t)$ triggers the commit of all its subtransactions, i.e.,

$$(Commit(t) \in H_{ct}) \Rightarrow \forall s \in subtrs(t), (Commit(s) \in H_{ct}).$$

By invoking the stable logging *or* persistence mechanisms when $Commit(s)$ occurs, subtransaction s is made durable.

$Abort(t)$ ($Abort(s)$) denotes that transaction t (subtransaction s) has aborted, eliminating the effects of t (s) from the database. A subtransaction abort is performed by executing its inverse, as implemented with the Undo mechanism. The abort of transaction t is achieved by aborting all $s \in subtrs(t)$. $Crash^a$ denotes that node a has crashed, causing all volatile data on node a to be lost and hence implies the abort of all uncommitted transactions. $Crash^a$ implies

the abort of all uncommitted transactions on node a. *Crash* can trigger both the Redo and Undo mechanisms.

When pages migrate in a multi-node system, we must consider the implications of node crashes. We capture these affects with the migration event *Mig* and with the *NodeCrash* predicate. Event $Mig(a,b,p)$ denotes that page p migrates from node a to node b. $Mig(a,b,p)$ satisfies the following invariant :

$$(pre(Mig(a,b,p)) \Rightarrow X^a(p)) \wedge (post(Mig(a,b,p)) \Rightarrow X^b(p))$$

That is, the migration of page p from node a to b occurs after a write to p is performed on node a and before a write to p is performed on node b. Page migration can trigger Logging, Stable Logging, or Persistence mechanisms.

The predicate $NodeCrash(c,a,b,p,t^a)$ indicates that node c has crashed, transaction t^a has not committed (prior to the crash), and page p migrates from node a to node b, remaining on node b until node c crashes [2].

DEFINITION 3.13: $NodeCrash(c,a,b,p,t^a) \Leftrightarrow (Mig(a,b,p) \rightarrow Crash^c) \wedge \neg(commit(t^a) \rightarrow Crash^c) \wedge (\nexists d, (Mig(a,b,p) \rightarrow Mig(b,d,p) \rightarrow Crash^c))$.

Consider t^a, transaction t executing on node a. When uncommitted data of t^a migrates to another node b, different recovery strategies will be necessary depending on whether the source node, a, or the destination node, b, crashes. These situations are captured by the predicates *SourceCrash* and *DestCrash*. *SourceCrash* is an instance of *NodeCrash*, binding the crashing node c to a, while *DestCrash* is also an instance of *NodeCrash*, binding the crashing node c to b.

DEFINITION 3.14: $SourceCrash(a,b,p,t^a) \Leftrightarrow NodeCrash(a,a,b,p,t^a)$.
$DestCrash(a,b,p,t^a) \Leftrightarrow NodeCrash(b,a,b,p,t^a)$.

Thus, in the context of an active transaction running on node a (t^a) in which uncommitted data (on page p) has migrated to node b, $SourceCrash(a,b,p,t^a)$ indicates that the source node (a) crashes, while $DestCrash(a,b,p,t^a)$ indicates that the destination node (b) crashes.

The events and predicates presented in this section comprise the building blocks of the specification of various MMLT implementations. In section 4, we discuss these implementations.

4 Synthesis of MMLT Implementations

In this section we propose various implementations of MMLT. In section 4.1, we first review existing techniques for supporting subtransaction atomicity in MLT (*subtransaction quiescence* and *persistence spheres*), and discuss the implications of adapting these techniques for MMLT.

[2](The notation $a \rightarrow b \rightarrow c$ is shorthand for $a \rightarrow b \wedge b \rightarrow c$.)

We find that, in the most general MMLT implementation, enforcing subtransaction persistence with persistence spheres can be expensive. To remedy this situation, in section 4.1 we identify conditions under which the persistence sphere does not need to be computed, that is, when the persistence sphere of a transaction is equal to the write set of the transaction ($WS = PS$). Section 4.2 proposes MMLT implementations based on whether or not $WS = PS$. These implementations are synthesized based on an object-to-page mapping policy and stability policies. At page migration, stability policies are used to guarantee the absence of inter-node cascading aborts. In section 4.3 we discusses other performance enhancing related options.

4.1 Supporting Subtransaction Atomicity in MMLT – Maintaining Persistence Spheres

When persistence is enforced in MLT, appropriate steps must be taken to correctly support subtransaction inverses. This is most effectively done by enforcing subtransaction atomicity. The basic methods for ensuring subtransaction atomicity in MLT can be applied to MMLT. However, given that transactions execute in parallel in MMLT, we must consider how implementing persistence may affect the performance of transactions on other nodes. To ensure subtransaction atomicity, the MLT implementation presented in [22] advocates using *subtransaction quiescence* and *persistence spheres*. Thus, motivated by correct support for subtransaction inverses, we now consider implementation mechanisms and transaction management primitives for supporting subtransaction quiescence and persistence spheres in MMLT.

Subtransaction quiescence means completing an active subtransaction. Quiescing subtransactions enables atomic subtransaction persistence. If only part of a subtransaction were made persistent, the semantics of the subtransaction inverse would be incorrect. The following definition states that if one page of a subtransaction is made persistent, all pages in the subtransaction's WS must also be made persistent.

DEFINITION 4.15: Subtransaction Quiescence:

$$\forall p \in WS(s),\ (post(Persist\text{-}page(p)) \Rightarrow \forall p' \in WS(s)\ Persists\text{-}page(p')).$$

Subtransaction quiescence can most effectively be supported by requiring subtransactions to use *strict* 2PL on pages (in contrast to [2]). Note that during the commit of a transaction t, to make the pages updated by t persistent, no locks can be held on these pages by *other* subtransactions. This guarantees that the contents of those pages do not *suddenly* change. Thus, when subtransactions use strict 2PL on pages, if page p had been updated by s, a committing transaction t (where $s \notin subtrs(t)$) cannot make p persistent until $Commit(s)$ occurs. The use of strict 2PL provides an efficient mechanism for supporting subtransaction quiescence in a multi-node implementation since inter-node communication is avoided. The use of strict 2PL on pages has a further advantage – pages updated by incomplete subtransactions will not migrate. This assertion is important for ensuring subtransaction atomicity in the context of node crashes.

To make a subtransaction persistent in MLT ([22]), the persistence sphere of the subtransaction must be made persistent:

Definition 4.16: Subtransaction Persistence Rule:
$post(Persist\text{-}page(WS(s))) \Rightarrow Persists\text{-}page(PS(s))$.

To commit a transaction, we could adopt any combination of (*steal, no-steal*) with (*force, no-force*) [11]. Given the possibility of multiple objects per page, where some objects may be uncommitted and others committed, we do not consider no-steal as a viable option. Because, to make page p persistent under no-steal would require that all transactions which have updated p must have committed. Thus, in general, enforcing no-steal in a multinode context would require some set of nodes to reach a commit consistent state. Reaching this commit consistent state could impose significant delays on other transactions. Given this, we focus on force/steal and no-force/steal policies next.

With a force policy, objects are made persistent at transaction commit time, alternatively, with no-force, the appropriate redo log information is written to stable store at commit time. With no-force, in order to bound recovery time, some form of checkpointing (flushing cache slots dirtied by database updates to stable store) should be adopted. Thus, with force, objects are made persistent at commit time, and with no-force, objects are made persistent periodically, as part of the checkpoint operation. In either case, we must ensure subtransaction quiescence and enforce the subtransaction persistence rule.

For both force and no-force, two basic strategies can be adopted for enforcing the subtransaction persistence rule. The first strategy is a *brute force* approach. Whenever persistence is to be performed, *all* volatile database objects on *all* systems are made persistent. Of course, prior to this activity, all active subtransactions must complete, ensuring subtransaction quiescence. The advantage of the brute force approach is that the persistence sphere need not be computed (since we are making the maximal sphere persistent). There are at least two disadvantages to the brute force approach, both of which can delay the normal runtime operation of the database. First, all nodes in the system must synchronize to reach a state where no node has cached dirty database objects. Second, generally, more I/O must be performed. With force, applying the brute force approach means that, whenever a transaction t commits, a prerequisite for making WS(t) persistent will be requiring that all other nodes must first reach a subtransaction quiescent state, then they must flush all dirty database objects to stable store. Under no-force, applying the brute force approach means that, after system-wide (all nodes) subtransaction quiescence is achieved, a system-wide, cache consistent checkpoint operation is taken.

The second strategy available for enforcing the subtransaction persistence rule attempts to minimize the number of database objects written to stable store. Under force, when transaction t commits, only those pages in $PS(t)$ are written to stable store. For force, to $Commit(t)$, $PS(t)$ is computed by computing the transitive closure of the WS's (as defined in 4.16). Computing $PS(t)$ at commit time requires inter-node communication, which can also delay the execution of transactions on other nodes. Furthermore, a prerequisite for computing $PS(t)$ is to maintain $WS(t)$ for each active transaction. The $WS(t)$ is best maintained *on the node executing* t. Note

that, since transactions execute on a single node, a given WS is only updated by one node (although there are likely to be multiple readers due to the construction of the PS).

Consider the second strategy applied to a no-force scheme. A fuzzy checkpointing scheme is an example of this case. In fuzzy checkpointing, all dirty database objects which have not been flushed since before the previous checkpoint are written to the *StableDB* (the fuzzy set). To implement this scheme, the $WS(t)$ for all active transactions must be maintained, just as for the force case. Once the fuzzy set has been determined, the PS of this set must be determined. This could involve the significant inter-node communication, and computation at other nodes may be required.

This discussion indicates that, in general, maintaining subtransaction atomicity can be time consuming and can incur substantial overheads. Hence strategies must be devised to minimize these impediments to efficient implementations of MMLT. For instance, if we knew that the persistence sphere of a transaction t contained exactly the same pages as t's write set ($WS(t) = PS(t)$), then ensuring the subtransaction persistence rule would be trivial. Sufficient conditions for complying with $WS(t) = PS(t)$ follow:

Definition 4.17: $\forall t' \neq t,\ \forall s \in \boldsymbol{subtrs}(t),\ \forall s' \in \boldsymbol{subtrs}(t'),$
$$((WS(s) \cap WS(s')) = \emptyset) \lor (WS(s) \supseteq WS(s')) \Rightarrow WS(t) = PS(t).$$

That is, for all pairs of subtransactions s and s' (of transactions t and t'), if either s or s' have non-intersecting WS's, or $WS(s')$ is a subset of $WS(s)$, then $WS(t) = PS(t)$. These conditions can be satisfied by proper object-to-page mappings. Next, we propose implementations of MMLT, based on whether on not $WS(t) = PS(t)$, as defined by the object-to-page mapping policy, and are synthesized by combining an object-to-page mapping policy with an stability policy.

4.2 MMLT Implementations

We propose two different types of implementations of MMLT, (1) $WS = PS$, and (2) $WS \neq PS$. $WS \neq PS$ allows the most general object-to-page mapping, but requires more complex stability policies and commit processing strategies than $WS = PS$. When $WS = PS$, we can avoid the overheads associated with computing the PS, maintaining the PS, or performing subtransaction-consistent savepointing. We refer to this feature of $WS = PS$ as achieving *low internode interference*. Note that a single database implementation can concurrently process transaction t_i under $WS = PS$, while processing transaction t_j under $WS \neq PS$. This construction is facilitated by having two *classes* of transactions and requiring $WS = PS$ transactions to have disjoint WS's from $WS \neq PS$ transactions. Recall that, in all these implementations, subtransaction quiescence is enforced by using strict 2PL.

The choice of stability policy is orthogonal to the transaction class. Stability policies which ensure *No Inter-node Cascading Aborts* utilize combinations of *Logging, Stable Logging, and Persistence* to support Undo and Redo. However, by allowing inter-node cascading aborts,

Fast Normal Runtime Processing can be achieved by not using *Logging, Stable Logging, or Persistence* (called *Nothing* [3]

The following example illustrates how page migration can cause inter-node cascading aborts. Consider a stability policy which avoids cascading aborts with Redo Logging and Undo Stable Logging. $Insert^a$ is a subtransaction of t^a, $Insert^b$ is a subtransaction of t^b, and page p contains ob and ob':

$$Insert^a[ob];\ mig(a,b,p);\ Insert^b[ob'].$$

Consider the case when node b crashes immediately following $mig(a,b,p)$. If no redo information for $Insert^a(ob)$ is available, t^a must be aborted. The stability policy can ensure redo logging, e.g., if $Insert^a[ob] \rightarrow Log\text{-}s(Insert^a[ob]) \rightarrow\ mig(a,b,p)$. With a redo logging stability policy, a consistent state can be reached after the crash of b by performing $Redo(Insert^a[ob])$. Next, consider the case when node a crashes immediately following $Insert^b(ob')$. Without sufficient undo information, the crash of a will require the abort of t^b. The stability policy of Undo Stable Logging ensures that this undo information is available, e.g., $Insert^a[ob] \rightarrow SLog(Delete^a[ob]) \rightarrow Insert^a[ob]$ will enable $Undo(Insert^a[ob])$ during recovery.

4.2.1 $WS = PS$

We propose two implementations with $WS = PS$, called MMLT-WS_1 and MMLT-WS_2. The first (simple) implementation, MMLT-WS_1, makes the following assumption:

- Subtransactions update at most one page. ($\forall s,\ |WS(s)| \leq 1$).
- A page is locked for the duration of a subtransaction.

The first assumption is a sufficient condition for $WS = PS$ (recall definition 4.17). Since subtransactions update at most one page, the effects of a subtransaction will never be fragmented, allowing for a simple implementation of the stability policies. In MMLT-WS_2, we relax the assumption that subtransactions update at most one page:

- Subtransactions update at most one object.
- Subtransactions can update multiple pages, but the object-to-page mapping is subject to the following restriction:
  - multiple objects may be stored in a page iff an object is contained completely within a page.
    That is, if object ob spans multiple pages, $PSET = \{p_1, p_2, ...p_n\}$,
    then no other object ob' is contained in any page of $PSET$.
- Subtransactions use strict 2PL on pages to ensure subtransaction quiescence.

[3] Typically, transaction atomicity is implemented with logging, but this is not required. The work on recovery in MLT ([22]) using a DB cache ([8]) discusses an alternative. The combination of a no-steal policy and a stable store (in addition to the *StableDB*) enables transaction atomicity without logging.

These assumptions imply $WS = PS$ (recall definition 4.17).

We now discuss the details of the formal specification of MMLT-WS$_1$ and MMLT-WS$_2$. Based on stability policies, we can now consider alternatives for supporting *No Inter-node Cascading Aborts* or *Fast Normal Runtime* for such objects. When p migrates from node a to node b, all uncommitted but complete operations (s) performed by node a could be logged or made persistent, and/or the inverse (s^{-1}) could be stable logged. We formally specify different stability policies below where t and s satisfy the following:

$$\neg(commit(t^a) \rightarrow Mig(a,b,p)) \;\wedge\; (s \in subtrs(t^a)) \;\wedge\; (s^a[p] \rightarrow Mig(a,b,p)).$$

That is, on node a, all subtransactions s of t which operated on page p prior to the migration of p from a to b are considered. When these pages migrate, the preconditions specified below are satisfied.

Stability Policy	*Condition*
Nothing:	$pre(Mig(a,b,p)) \Rightarrow True.$
Redo Logged:	$pre(Mig(a,b,p)) \Rightarrow Logged\text{-}s^a(s[p]).$
Redo Persistent:	$pre(Mig(a,b,p)) \Rightarrow Persists\text{-}page^a(s[p]).$
Undo Stable Logged:	$pre(Mig(a,b,p)) \Rightarrow SLogged\text{-}s^a(s^{-1}[p]).$

The reason for (just) these possibilities will be made clear now as we examine the implications of the various stability policies when a source or destination crash occurs. Suppose the source node (a) crashes after uncommitted operations have migrated. If pages containing uncommitted operations migrated from the source node a, the database will be in an inconsistent state. In order to return the database to a consistent state, all uncommitted operations should be undone. If the stability policy is *Undo Stable Logged*, after $Crash^a$, undo information will be available to eliminate the effects of uncommitted operations on migrated pages. Without stable logged undo information (the *Nothing* alternative), in order to eliminate the effects of t^a, any t^b which updated p (after p had migrated) would have to be aborted These effects of the stability policies are stated below:

Undo Stable Logged: $SourceCrash(a,b,p,t^a) \Rightarrow$
$$(\forall t^a,\ \forall s^a[p] \in subtrs(t^a),\ (s^a[p] \rightarrow Mig(a,b,p)) \Rightarrow Undo\text{-}s(s^a[p]) \in H)).$$

Nothing: $SourceCrash(a,b,p,t^a) \Rightarrow$
$$(\forall t^a,\ t^b,\ (\exists s^a[p] \in subtrs(t^a),\ s^b[p] \in subtrs(t^b),\ (s^a[p] \rightarrow Mig(a,b,p) \rightarrow s^b[p]) \Rightarrow (Abort^b(t^b) \in H))).$$

If the destination node (b) crashes after uncommitted operations of t^a have migrated, updates performed by t^a may be lost. If the stability policy is *Redo Persistent*, they will not be lost, if it is *Redo Persistent*, these updates can be redone. Otherwise, if no redo information is available on node a, the destination crash will trigger an abort of t^a. These effects of the stability policies are stated below:

	Redo Stability	**Undo Stability**
No Cascading Aborts	{RedoLogged, Redo Persistent}	Undo Stable Logged
Fast Normal Runtime	Nothing	Nothing

Figure 5: Database System Requirements and Associated Mechanisms for $WS = PS$. Since $WS = PS$, *Low Commit Processing Overhead* is achieved for both combinations.

Redo Persistent: $DestCrash(a, b, p, t^a) \Rightarrow True.$

Redo Logged: $DestCrash(a, b, p, t^a) \Rightarrow$
$(\forall t^a, \forall s^a[p] \in subtrs(t^a), (s^a[p] \rightarrow Mig(a, b, p)) \Rightarrow$
$(Redo\text{-}s(s^a[p]) \in H)).$

Nothing: $DestCrash(a, b, p, t^a) \Rightarrow$
$(\forall t^a, (\exists s^a[p] \in subtrs(t^a), (s^a[p] \rightarrow Mig(a, b, p)) \Rightarrow (Abort^a_{t^a} \in H))).$

Figure 5 summarizes the system requirements and associated mechanisms for $WS = PS$.

Although MMLT-WS_1 and MMLT-WS_2 have different object-to-page mapping policies, the implications of these policies are similar. Like MMLT-WS_1, object fragmentation does not occur in MMLT-WS_2. Recall that object fragmentation of *ob* can occur only as a side effect of some operation on a different object stored in one of the pages where *ob* is stored. Since any object *ob* spanning multiple pages will not share these pages with other objects, it is not possible for an operation on a different object *ob'* to cause only some of those pages, and thus part of the object, to migrate. Thus, object fragmentation can not occur in MMLT-WS_2. In summary, this section presented MMLT implementation options in the case where restrictions are placed on the object-to-page mapping. These object-to-page mappings achieve low internode interference and simplify the stability policies.

4.2.2 $WS \neq PS$

We now allow the following:

- Arbitrary object-to-page mapping policies.
- A single subtransaction can update multiple objects.

However, this generality causes a more complex implementation, and the resulting higher computation and communication overheads may degrade overall system performance. Under $WS \neq PS$, both commit processing and the stability policies are more complex. Given that we have discussed the complexities of commit processing in section 4.1, we focus on the stability policies in this section.

Although object fragmentation does not occur in the two implementations presented for $WS = PS$, object fragmentation can occur $WS \neq PS$, requiring more complex stability policies

to ensure cascadeless aborts. To avoid cascading aborts in $WS \neq PS$, the stability policies specified for $WS = PS$ are extended by augmenting the Redo Logged and Undo Stable Logged stability policies to log the objects operated on by subtransaction (in addition to logging subtransactions) [4].

To illustrate the necessary extensions to the stability policies when object fragmentation occurs, in the remainder of this section, we assume that object ob spans exactly two pages, p and q, and object ob' is stored in page p (see figure 2). To illustrate the problems caused by object fragmentation, consider the case where $s^a[ob_{\{p,q\}}]$ is executed on the source node, p migrates, then $s'[ob'_p]$ is executed on the destination node. We use $ob_{\{S\}}$ to denote the object spanning pages contained in set S. If the source or destination node crashes, the atomicity of $s^a[ob_{\{p,q\}}]$ is destroyed. If the source node crashes after p migrates, we must undo $s^a[ob_p]$, and if the destination node crashes, we must redo $s^a[ob_p]$. In the formal specifications of the stability policies, transaction t and subtransaction s (on node a) satisfy the following:

$$\neg(commit(t^a) \rightarrow Mig(a,b,p)) \wedge (s^a \in subtrs(t^a)) \wedge \exists q\ (s^a[ob_{\{p,q\}}] \rightarrow Mig(a,b,p)).$$

That is, on node a, all subtransactions s of t which operated on an object spanning multiple pages prior to the migration of p from a to b are considered. These pages are allowed to migrate according to the preconditions specified below:

Stability Policy	*Condition*
Redo Logged:	$pre(Mig(a,b,p)) \Rightarrow Logged\text{-}s^a(s^a[ob_{\{p,q\}}]) \wedge Logged\text{-}ob^a(s^a[ob_p])$.
Undo Stable Logged:	$pre(Mig(a,b,p)) \Rightarrow SLogged\text{-}s^a(s^{a^{-1}}[ob_{\{p,q\}}]) \wedge SLogged\text{-}ob^a(s^a[ob_q])$.

Thus, the Redo Logged policy is extended to log the *portion* of the object which did migrate, while the Undo Stable Logged policy is extended by making persistent the portion of the object which did *not* migrate. For Redo Logged, logging the portion of the object which did migrate is indicated by $Logged\text{-}ob^a(s^a[ob_p])$, while for Undo Stable Logged, stable logging the portion of the object which did *not* migrate is indicated by $SLogged\text{-}ob^a(s^a[ob_q])$. If the use of subtransaction inverses is impossible due to the loss of part of an object (caused by a node crash) for which the inverse was defined on, correct recovery would require inter-node cascading aborts. This is avoided here by redo logging and stable undo logging.

Redo Logged: $DestCrash(a,b,p,t^a) \Rightarrow$
$$(\forall t^a,\ \forall s^a[ob_{\{p,q\}}] \in subtrs(t^a),\ (s^a[ob_{\{p,q\}}] \rightarrow Mig(a,b,p)) \Rightarrow (Redo\text{-}ob(s^a[ob_p]))).$$

Undo Stable Logged: $SourceCrash(a,b,p,t^a) \Rightarrow$
$$(\forall t^a, \forall s^a[ob_{\{p,q\}}] \in subtrs(t^a),\ (s^a[ob_{\{p,q\}}] \rightarrow Mig(a,b,p)) \Rightarrow (Redo\text{-}ob(s^a[ob_q]) \rightarrow Undo\text{-}s(s^a[ob_{\{p,q\}}]))).$$

[4] We do not consider Redo Persistence as a viable policy when $WS \neq PS$, since, based on the studies of MLT [22, 12], system performance degrades substantially due to the frequent computation of $PS(s)$.

	Redo Stability	Undo Stability
No Cascading Aborts	Redo Logged	Undo Stable Logged
Fast Normal Runtime	Nothing	Nothing

Figure 6: Database System Requirements and Associated Mechanisms for $WS \neq PS$. In $WS \neq PS$, *High Commit Processing Overheads* are incurred.

Thus, to recover from *SourceCrash*, the page of the object which did not migrate is restored prior to the application of the inverse operation. To recover from *DestCrash*, the page of the object which migrated is restored. This recovery strategy is similar to physiological logging [10] and the the ARIES [14, 15] style recovery of performing page oriented redo followed by either physical or logical undo. In section 5, we discuss the implications of adopting physiological logging in MMLT and compare MMLT with ARIES. Figure 6 summarizes the system requirements and associated mechanisms for $WS \neq PS$.

4.3 Other Performance Related Options

Physiological logging [10] can be adopted as a compromise between logical and physical logging, replacing our logical logging assumption. One of the benefits of physiological logging is that redo can be page oriented, while, as in logical logging, undo can be implemented with inverse operations. The basic strategy for recovery with physiological logging is that redo is performed first, repeating page operations recorded in the log, then logical undo is performed for any uncommitted operations. In the context of MMLT, the stability policies would log pages or objects for redo, and (logical) log operations for undo. Given this, to make transaction **t** persistent, instead of requiring $PS(t)$ to be persistent (as done in sections 4.2.1 and 4.2.2), $WS(t)$ can be made persistent and $(PS(t) - WS(t))$ stable logged. In the event of a crash, the recovery process can make $(PS(t) - WS(t))$ persistent prior to executing any inverse operations.

ARIES [16] addresses issues involved when a no-force policy is assumed and page locks are retained after a transaction as committed. These issues are further studied in a performance analysis of coherency control policies through lock retention [7]. Also, the taxonomy of concurrency and coherence control presented in [19] surveys lock retention strategies. These studies discuss the inherent tradeoffs between lock retention and the complexity of recovery. Lock retention can be incorporated as part of the implementation options for MMLT. This will not change the specification of the stability policies.

5 Summary and Conclusions

In this paper, we have identified and critically examined the implementation options available when implementing MLT on multiple nodes. By exploiting the semantics of high-level operations,

MLT increases concurrency of operation execution. By implementing MLT in a multi-node system, additional performance advantages can be gained via the parallel execution of transactions and operations. Due to differences between single and multi-node MLT, the implementation mechanisms and policies originally designed for MLT needed to be revisited to enable an efficient implementation of MMLT. For example, MLT uses persistence spheres and subtransaction quiescence [22] to maintain subtransaction atomicity during commit processing. However, in a multi-node system, a direct application of these techniques can cause high commit processing overheads, in terms of the delay on the committing node and the interference caused on other nodes. Furthermore, in a multi-node system, inter-node cascading aborts can occur due to page migration.

Our examination of the available implementation options for MMLT was guided (constrained) initially by making specific assumptions about transaction model correctness requirements and standard performance enhancing mechanisms. The main challenge in constructing an efficient implementation of MMLT is the efficient support of subtransaction inverses, for which subtransaction atomicity is essential. In order to arrive at an efficient implementation, we have assumed standard performance enhancing mechanisms such as coarse-grain (page) persistence and coarse-grain (page) cache coherency.

In MMLT, subtransaction atomicity is supported by using subtransaction quiescence, by an appropriate commit processing strategy, and (under certain implementation options) by proper choice of the stability policy. Subtransactions use *strict* 2PL on pages to guarantee subtransaction quiescence. The policies and mechanisms for supporting commit processing and stability properties are dependent on the object-to-page mapping policy, which in turn determines whether $WS = PS$ or $WS \neq PS$. For both these object-to-page mappings, either the force or the no-force commit processing strategy can be adopted. Furthermore, with either of these commit processing strategies, inter-node cascading aborts can be avoided with specific stability properties.

A summary of the implementation options for supporting the subtransaction atomicity requirement and achieving specific database performance related properties is presented in figure 7. It should be noted that we are summarizing complex options in a nutshell, and hence this summary is complemented by the details of these options discussed in the previous two sections. In the figure, the ellipses denote database policies. Associated with each policy is some performance related property, denoted with the *italicized* text. Specific subsets of these policies are required to support subtransaction atomicity. In the figure, the unshaded ellipses denote policies which *do not* play a role in supporting subtransaction atomicity, such as **No Stability Policies**. Two arrows connected by an `OR` arc indicate that one policy or the other must be selected to satisfy the source requirement or policy. Arrows without an `OR` arc point to policies which are mandatory to satisfy the requirement or policy. For example, the **Subtransaction Atomicity** requirement is supported by using **Subtransaction Quiescence** and either the **WS = PS** or **WS ≠ PS** object-to-page mapping policy. If **WS = PS** is selected, (implying a *Restricted*

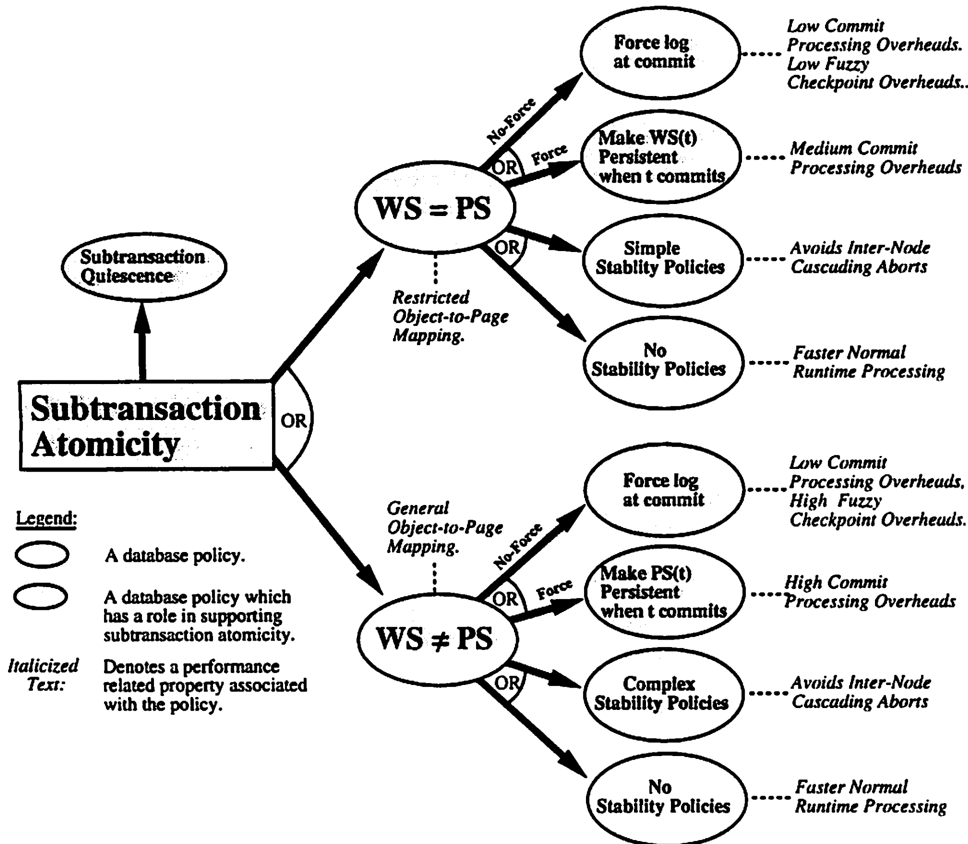

Figure 7: MMLT Implementation Options for Supporting Subtransaction Atomicity.

Object-to-Page Mapping), then either the **Force** or the **No-Force** strategy is selected and either **Simple** or **No Stability Policies** must be selected. Depending on the selected stability policy, this configuration would either *Avoid Inter-Node Cascading Aborts*, or provide *Faster Normal Runtime Processing.* Comparing the **WS = PS** and the **WS ≠ PS** configurations, the overheads for the **Force** and **No-Force** commit processing strategies are greater for **WS ≠ PS** due to the non-trival nature of computing persistence spheres. Furthermore, the stability policies which avoid inter-node cascading aborts are more complex for **WS ≠ PS**. In fact, the reason for the additional complexity of the stability policies associated with $WS \neq PS$ is that, to support subtransaction atomicity, object fragmentation must be addressed.

The object-to-page mapping policies can also be viewed as defining transaction classes, since a single database implementation can concurrently process transaction t_i under $WS = PS$, while processing transaction t_j under $WS \neq PS$. For both transaction classes, stability policies can be defined which either avoid cascading aborts, or allow cascading aborts but exhibit fast normal runtime behavior. Furthermore, for any of these implementation approaches, the DB cache method, retained locks, and physiological logging can be incorporated for better run-time performance and faster commit processing.

We have assumed that total leeway has been afforded to the stability policies with respect to logging. In reality, other factors influence whether logging is required. For example, it may

be desirable to perform redo logging in order to maintain an audit trail [10]. Furthermore, undo logging is essential for the application of operation inverses to achieve transaction atomicity. Thus, if other factors require redo and undo logging, the *additional* cost of enforcing cascadeless stability policies could be small.

Some of the mechanisms used in this paper are also related to the work done in ARIES [14, 15, 16]. ARIES/IM [15] discusses index management (on a single node), focusing on concurrency control and recovery of B^+-trees, and addresses how inverse operations on B^+-trees are supported. In another recent ARIES paper [16], the support for fine granularity locking in multi-node shared disk systems is discussed. Cascading aborts due to the migration of uncommitted data are avoided by enforcing stability properties at ownership change. Four different schemes for stability are identified – *simple*, *medium*, *fast*, and *super-fast*. At ownership change, the simple and medium schemes enforce persistence, the fast scheme enforces stable logging, and the super-fast scheme enforces (volatile) logging. However, the material presented in these papers is based on a *specific* implementa ion viewpoint. In contrast, based on MLT, we have presented a general methodology for supporting operation and subtransaction inverses in a multi-node system. By developing a broad spectrum of possible approaches to a multi-node database implementation of MLT and discussing the related benefits of the approaches, we have also shown which mechanisms contribute to a specific performance related requirement.

By explicitly modeling all database events and their associated implementation mechanisms, we were able to precisely characterize the significant relations between support for specific database system requirements and their implications for recovery. For example, by modeling database events such as *Commit*, *Abort*, and *Crash*, and properties of objects such as *Logging*, *Stable Logging*, and *Persistence*, we are able to capture the various inter-node dependencies induced by the object-to-page mapping policies and stability polices. The resulting insights into implementation tradeoffs have been facilitated through a precise characterization of policies and mechanisms by tailoring the ACTA formalism to specify both transaction model correctness requirements *and* implementation oriented properties. Some of the ideas incorporated in Flex transactions [4] have interesting parallels to our work. Like ACTA, Flex transactions allow the specification of transaction dependencies based on logical assertions between events. A "mixed system", one which allows both compensatable and non-compensatable operations, is also part of Flex transactions. Also related to the theme of combining transaction models is Lomet's work on MLR (multi-level recovery) [13], which unifies recovery for MLT and nested transactions. Although we have focused on multi-node MLT, our approach can also support multiple transaction models and operations with and without inverses. For example, based on the compensatability of an operation on an object, the specification of stability policies can involve making either the object or the operation stable.

We believe that our work suggests a general approach to modeling complex database systems – systems which may be implemented on multiple nodes and employ extended transaction models [9]. Our future work includes investigating this generalization. We have analyzed both

dependencies due to node crashes and commit dependencies which arise from normal execution. Extensions to our approach include integrating dependencies that can form from either the structure of extended transaction models, or from semantic notions, such as recoverability [1]. Our next step will be to implement MMLT on a multiprocessor testbed, in order to obtain actual performance measures of the various options available for the parallel transaction execution in the context of object specific operations. We have already begun the construction of our multiprocessor testbed (on a KSR [20] multiprocessor) in the context of related work on the design of efficient lock managers for shared-disk database systems [17]. Our performance studies on these lock manager designs have indicated that our formalisms, especially the stability policies, are useful in guiding the internal design process and for reasoning about recovery.

References

[1] B. R. Badrinath and K. Ramamritham. Semantics-based Concurrency Control: Beyond Commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992.

[2] C. Beeri, H. Schek, and G. Weikum. Multi-Level Transaction Management, Theoretical Art or Practical Need? *Lecture Notes in Computer Science*, 303:135 – 154, 1988.

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

[4] O. Bukhres, A. Elmagarmid, and E. Kuhn. Implementation of the Flex Transaction Model. *Bulletin of the IEEE Technical Committee on Data Engineering*, 16(2):28–32, June 1993.

[5] P. K. Chrysanthis. *ACTA, A Framework for Modeling and Reasoning about Extended Transactions*. PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, September 1991.

[6] P. K. Chrysanthis and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. *Proc. 1990 ACM SIGMOD International Conference on Management of Data*, pages 194–203, May 1990.

[7] A. Dan and P. S. Yu. Performance Analysis of Coherency Control Policies through Lock Retention. *Proc. 1992 ACM SIGMOD International Conference on Management of Data*, 21(2):114–123, June 1992.

[8] K. Elhardt and R. Bayer. A Database Cache for High Performance and Fast Restart in Database Systems. *ACM TODS*, 9(4):503–525, December 1984.

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

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[11] T. Haerder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[12] C. Hasse and G. Weikum. A Performance Evaluation of Multi-Level Transaction Management. *VLDB*, pages 55 – 66, 1991.

[13] D. Lomet. MLR: A Recovery Method for Multi-Level Systems. *Proc. 1992 ACM SIGMOD International Conference on Management of Data*, 21:185 – 194, June 1992.

[14] C. Mohan, D Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17:94–162, March 1992.

[15] C. Mohan and F. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. *Proc. 1992 ACM SIGMOD International Conference on Management of Data*, 21:371–380, June 1992.

[16] C. Mohan and I. Narang. Recovery and Coherency-control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared-Disk Transaction Environment. *VLDB*, 17:193–207, 1991.

[17] L. D. Molesky and K. Ramamritham. Autonomous Locking for Shared-Disk Database Systems. Technical Report 94–10, University of Massachusetts Dept. of Computer Science, February 1994.

[18] E. Moss, N. Griffeth, and M. Graham. Abstraction in Recovery Management. *Proc. 1986 ACM SIGMOD International Conference on Management of Data*, pages 72–83, 1986.

[19] E. Rahm. Concurrency and Coherency Control in Database Sharing Systems. *Technical Report, University of Kaiserslautern, Germany*, December 1991.

[20] Kendal Square Research. *KSR1 Principles of Operation.* KSR Research, Waltham, Mass., 1992.

[21] G. Weikum. Principles and Realization Strategies of Multi-Llevel Transaction Management. *ACM TODS*, 16(1):132 – 180, March 1991.

[22] G. Weikum, C. Hasse, P. Broessler, and P. Muth. Multi-Level Recovery. *Proc. 1990 ACM SIGMOD International Conference on Management of Data*, pages 109 – 123, 1990.