1989

A Model for Adaptable Systems for Transaction Processing

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

John Riedl

Report Number:
89-900

A MODEL FOR ADAPTABLE SYSTEMS FOR TRANSACTION PROCESSING

Bharat Bhargava
John Riedl

CSD-TR-900
August 1989

A Model for Adaptable Systems for Transaction Processing[1][2]

Bharat Bhargava (IEEE Senior Member)
John Riedl (IEEE Student Member)

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
(317)-494-6013

[1]This research is supported in part by grants from AIRMICS, AT&T, NASA, NSF, UNISYS and a David Ross Fellowship.

[2]To appear in the December 1989 issue of IEEE Transactions on Knowledge and Data Engineering.

Abstract

There is a need for adaptable systems that can change as requirements change, both in the long and the short term. Adaptability can reduce software costs and improve reliability and performance. Adaptability includes design techniques that support enhancement, retargeting to new projects, restructuring during different phases of a mission, and changing algorithms as external conditions change. These techniques are particularly important for projects with stringent performance or reliability requirements. We model algorithmic adaptability, and develop techniques for switching between classes of schedulers in distributed transaction systems. RAID is an experimental system implemented to support experimentation in adaptability. We study adaptability features in RAID, including algorithmic adaptability, fault-tolerance, and implementation techniques for an adaptable server-based design. Adaptability is an essential tool for the future to manage escalating software costs and build high-reliability, high-performance systems.

Contents

1 Introduction

Adaptability and reconfigurability are needed to deal with the performance and reliability requirements of a system. Current distributed systems provide a rigid choice of algorithms for database software implementation. The design decisions are based on criteria such as computational complexity, simulations under limited assumptions, and empirical evidence. The desired life cycle of a system is at least several years. During such time new applications surface and the technology advances, making earlier design choices less valid. In addition, during a small period of time (within a 24 hour period), a variety of load mixes, response time requirements and reliability requirements are encountered. An adaptable distributed system can meet the various application needs in the short-term, and take advantage of

advances in technology over the years. Such a system will adapt to its environment during execution, and be reconfigurable for new applications.

Adaptability is particularly important for mission critical and distributed systems. Mission critical systems push current hardware and software technology to the limits, to meet extreme fault-tolerance or real-time performance requirements. Distributed systems must be adaptable to support heterogeneous hosts and to provide graceful degradation during failures. Computer networks support a larger and more diverse user community than centralized systems. Hosts of many different types running different software will be required to support this community. The underlying distributed system must be adaptable to a wide range of hardware and software. Furthermore, failures that would require complete shutdown of a centralized system can often be tolerated with only slight performance degradation in a distributed system. The distributed system software must be designed to be adaptable to a wide range of modes of operation, corresponding to the wide range of possible failure modes. For instance, [BB89] describes an algorithm for responding to failures by dynamically adjusting quorum assignments. As a failure continues, more and more quorum assignments are modified. Finally, when the failure is repaired those quorums that were changed can be brought back to their original assignments. By dynamically adapting to the failure the availability of data in the system is increased, at a cost that is only incurred during failure or recovery.

The principal advantages of adaptability are in the areas of reliability, performance, and software enhancement and maintenance. Adaptability provides for reliability through a design that is tolerant of site and communication failures. Adaptability improves performance because the system can adjust its transaction processing algorithms for optimum processing of the current mix of transactions. Adaptability simplifies software enhancement and maintenance through a design that is oriented from the start towards incorporating new ideas and approaches. Subsystems can be replaced without affecting other parts of the system, and the design of each subsystem supports the implementation of new algorithms. With maintenance costs climbing as high as 80% of life-cycle costs for many systems [Ale86, Weg84], a design that supports future changes is becoming an essential part of software development.

Flavors of Adaptability. We classify adaptability into four broad categories: structural static, structural dynamic, algorithmic and heterogeneous. Structural static adaptability consists of software engineering techniques for developing system software that can adapt to requirements changes over its life-cycle. Structural static techniques are categorized differently if they refer to layered software or unlayered software. Layered software supports *vertical adaptability*, the ability to replace a layer without affecting the other layers. Some layered software also supports *horizontal adaptability*, the ability to replace components within a layer[Gog86]. Unlayered software can also take advantage of structural static adaptability. Fault-tolerance adaptability in hardware is classified into circuit level and module level. *Circuit level adaptability* in hardware corresponds to software techniques that check for errors

within a software module, using redundancy [Avi76] or correctness criteria [Ran75]. *Module level adaptability* in hardware corresponds to techniques that isolate errors to within a single software module and allow the rest of the system to continue processing despite partial failures. Failure modes at the module level are often assumed to be *fail-stop*, which means that either the module does not produce a result or it produces the correct result.

Structural dynamic adaptability, usually called reconfiguration, is restructuring a running system in response to failures or performance requirements. Reconfiguration includes site failure protocols, network partitioning protocols, and other techniques for reorganizing the way the sites in a distributed system work together [PW85, chapter 5]. For instance, performance reconfiguration includes dynamically changing the communications structure of processors in a non-shared memory multiprocessor in response to changing tasks. Reconfiguration also has longer term benefits. New hardware can be easily integrated into a running system, and the software can be ported to new architectures with different interconnections between processors.

Algorithmic adaptability is a set of techniques for dynamically changing from the execution of one algorithm for a module to a different algorithm. For instance, a transaction system can change to a new concurrency controller, or a distributed system can change to a new site failure algorithm. Algorithmic adaptability can take place in one of three ways. *Temporal adaptability* refers to changes in algorithms over time. These methods generally have a brief conversion period after which operation continues with the new algorithm. The adaptability theory developed in this paper (section 2) is principally temporal adaptability. *Per-transaction adaptability* consists of methods that allow each transaction to choose its own algorithm. Different transactions running at the same time may run different algorithms based on their requirements. *Spatial adaptability* is a variant of per-transaction adaptability in which transactions choose the algorithm based on properties of the data items they access. Spatial adaptability is an advantage in cases in which properties of different algorithms are desired for different data items.

Heterogeneity is the form of adaptability that deals with the issues involved in distributed computing using many different types of computing systems [PW85, chapter 6]. The simpler problems of heterogeneity include establishing physical connection between the machines, and resolving data type differences (e.g., size of integers, byte order). More difficult problems include getting diverse database systems to work together [EH88], and developing software that takes advantage of the particular strengths of each machine. Heterogeneity is an important problem for the future, as missions requiring several different types of computing engine will become more common [KK86]. Systems that can incorporate heterogeneous hardware and software components will also have the advantage of being able to incorporate new technologies more easily. Solving the problem of heterogeneity involves many of the problems in algorithmic and reconfiguration adaptability, and is aided over the long term by the software design techniques of structural static adaptability.

Techniques for Adaptability. Adaptable software can be supported by an infrastructure of software tools. Support for modular design can make it easier to implement adaptability. The principle behind modular design is that abstract entities in the software design should be represented as physical entities in the implementation. For instance, separate activities should be represented as separate processes. In many operating systems, such as Unix, this means that the activities are represented as separate address spaces, each with a single thread of control. Shared resources are placed in one of the address spaces, and accessed from the other address spaces via messages or remote procedure calls. Since messages and remote procedure calls usually cost an order of magnitude more than local accesses, the performance of the entire system suffers. The problem is that the operating systems only offer a single process abstraction. More flexible process abstractions make sharing of resources such as memory and files efficient and easy. For instance, some operating systems support multiple threads of control within a single address space [CZ85, Ras86].

Powerful remote communication primitives can also support adaptability. For instance, efficient multicast encourages the development of applications that can be spread among hosts in many different ways. Using logical multicast addresses the application does not have to worry about the location of the destination. Servers can relocate without informing their clients. Service requests directed to the logical address can be answered by any of a group of replicated servers [CZ85].

New languages also offer support for adaptability. Object-oriented languages provide data abstraction to clearly define the interface to the programmer. Furthermore, the syntax and semantics of object references are the same whether the message is to an object in the same address space, in a different address space on the same computer, or even on a different computer. A smart compiler can exploit this flexibility by clustering objects into processes according to reference patterns, so most object invocations can be simple procedure calls [BHJ*86]. In fact, the object can by dynamically moved as long as the interface is not affected.

Models of adaptability are another tool. In this paper we develop the sequencer model for subsystems of adaptable transaction systems (section 2). The sequencer model formalizes three basic approaches to algorithmic adaptability: generic state, converting state, and suffix-sufficient state. With the model the problem of adaptability is reduced to mapping the subsystem to a sequencer, choosing one of the adaptability approaches, and implementing it. Software engineers have studied design techniques that enhance structural static adaptability. Models for other types of adaptability are needed.

Using Adaptability. Incorporating adaptability into a system requires an understanding of how and when adaptability will be used. Structural static adaptability is used every time the software is changed. Any software that will include maintenance in its life-cycle should incorporate techniques to make maintenance easier. Reconfiguration adaptability is necessary in systems for which availability is important or reconfiguration is common.

Reconfiguration adaptability may also be necessary for mission critical systems that must use different configurations to respond to different situations. Algorithmic adaptability is most important in systems that are pushing hardware performance to the limit, or that require very high levels of fault tolerance. Algorithmic adaptability is useful for responding to environmental changes that are predictable, or that last long enough to amortize the cost of the adaptation. Finally, support for heterogeneity is important in any system that has a long life-cycle or that will run on a large network of computers. In choosing when adaptability can be applied there is a tradeoff between the costs and benefits. Experimental work is needed to evaluate adaptability in practical environments.

Outline. In this paper we outline an approach to developing reconfigurable transaction systems software. Using these methods the algorithms of a transaction system can be switched without waiting for existing transactions to terminate. Among the contributions of this paper are a formalization of this conversion process, the specification of criteria sufficient to guarantee correctness during conversion, and a description of our ongoing implementation effort in adaptability. The paper focuses on algorithmic adaptability, but also discusses our implementation work on other types of adaptability.

This paper is divided into three major sections. In Section 2 we characterize the subsystems of a transaction system as sequencers of atomic actions. Based on this model we develop constructive methods for correctly switching between different algorithms for these subsystems. Section 3 uses the concurrency control subsystem as an example to demonstrate the applicability of the sequencer model. Section 4 describes the implementation of algorithmic, reconfiguration, and structural static adaptability techniques in the RAID distributed database system.

2 Methods for Adaptability

In this section we describe the sequencer model for algorithmic adaptability, which applies in a natural way to many subsystems of a transaction system. The model is based on Papadimitriou's model for concurrency control [Pap79], was introduced in [BR88], and appears here in abbreviated form. A primary advantage of this model is that it provides for a clean interface between subsystems. The model is based on the idea of a *sequencer*, which orders atomic actions in a transaction processing history according to some correctness criterion. A sequencer is a function that takes as input a series of actions of a history and produces as output the same actions, possibly in a different order. To be practical, a sequencer should be able to work on-line, in the sense that it should read the actions of the history in order and produce output actions before it has read the complete history. The classic example of a history sequencer is a locking concurrency controller. Actions are attempts to read or write database items, and the concurrency controller rearranges the actions using its lock

queues. Many of the subsystems of a distributed system can be modeled as sequencers. The advantage of a sequencer is that the history provides a simple interface to the rest of the system. The crucial problem in adapting between two algorithms for a sequencer is the transference of the state information from the old algorithm to the new one.

2.1 History Sequencers

Definition 1 *A* transaction *is a sequence of atomic* actions.

The purpose of a transaction system is to process transactions efficiently while maintaining concurrency atomicity and failure atomicity [Gra81].

Definition 2 *A* history *is a set of transactions and a total order on the union of the actions of all of the transactions. The actions of each transaction must be in the same order in the history that they are in their transaction, but may be intermingled with the actions of other transactions.*

We will use the notation $H \circ a$ to denote history H extended by action a. Likewise, $H_1 \circ H_2$ denotes history H_1 extended consecutively by the actions in H_2. A *partial history* is like a history except that it may only have a prefix of the history of some transactions. Partial histories represent systems that are in the process of running some transactions. Since this paper is focused on running systems, we shall use the term history interchangeably with the term partial history.

Most sequencers develop state information as they operate. For instance, a locking concurrency controller maintains queues of actions to determine the order in which actions should be executed. With incorrect or incomplete state information the concurrency controller will permit non-serializable executions. Adapting between two algorithms for a sequencer depends on correctly transferring this state information. The rest of this section suggests various ways in which the state information can be manipulated to permit the replacement of a running sequencer with a new sequencer without stopping transaction execution.

Let A and B be correct implementations of sequencer S. Let ϕ be a predicate on the output partial histories of S that returns true if the partial history is acceptable output from S. For instance, ϕ for concurrency controllers would be a function that determines whether the input partial history is a prefix of some serializable history.

Definition 3 *An* adaptability method *M is a process for converting from A to B without violating the correctness rules for either A or B. M starts with A running and finishes with B running. It may itself serve as sequencer for some part of the input history, and may perform arbitrary computations involving A and B during the conversion.*

Definition 4 *We say that an adaptability method M is* valid *for sequencer S if there are no histories that cause it to violate the correctness condition for sequencer S. More formally, suppose M is valid and let H be a partial history consisting in order of the sub-sequences H_A that could be the output of A, H_M that could be the output of M, and H_B that could be the output of B. Then $\phi(H)$ must be true.*

This is a general statement of the idea of validity for adaptability methods. Less general statements that avoid the need for ϕ are tempting, but can easily be reduced to the above form. In particular, defining validity to be output histories that could have been produced by some combination of methods A and B is less satisfactory, since the most efficient adaptability methods that we know cannot be proven correct in this case.

Note that predicates like ϕ are usually too expensive to be implemented. Practical adaptability methods such as those below may use ϕ in their correctness proofs, but should not depend on it for the actual adaptation.

Next we shall investigate three particular adaptability methods: generic state, converting state, and suffix-sufficient state, and prove them each valid. No one of the three methods is best for all situations, so in each we consider the advantages and disadvantages of the method.

2.2 Generic State Adaptability

One approach is to develop a common data structure for all of the ways to implement a particular sequencer. For network partition control, for instance, this data structure would contain information on the configuration of the network, the data available in the local partition, and the data items in this partition which have been updated since the partition occurred. Each algorithm for the sequencer is implemented to use this standard data structure. Under this strategy, switching to a new algorithm is done simply by starting to pass actions through an implementation of the new algorithm. Figure 1 depicts two concurrency control algorithms sharing the same data structure during conversion. There is a subtlety here, though. Many algorithms have conditions on the preceding state as part of their correctness requirements. For example, a locking concurrency controller can only guarantee serializability if no lock is held by more than one active transaction. Optimistic concurrency controllers, on the other hand, may permit multiple accesses to the same data item for improved concurrency. Thus serializability is not guaranteed if we switch from an optimistic concurrency controller to a locking concurrency controller, even if correct state information is available to both. Certain classes of sequencers are more amenable to adaptability. The best case, formalized by the following definition, is when all algorithms for a sequencer are guaranteed to satisfy the pre-condition for any other algorithm for the same sequencer.

Definition 5 *A sequencer S is called* generic state compatible *if any two algorithms A and B for S are guaranteed to produce acceptable output if B is run after A using A's generic*

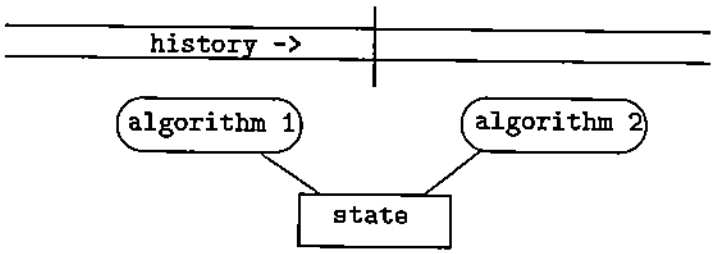


Figure 1: The generic state approach to sequencer adaptability.

state. Formally, if A produces as output history H_A and B with the generic state from A after producing H_A, produces history H_B then the history $H_A \circ H_B$ is acceptable output from S.

We note from the definition that algorithms for generic state compatible sequencers can be changed by just replacing the old algorithm with the new one:

Lemma 1 *Let S be a generic state compatible sequencer. Let M be the adaptability method for S that simply replaces an old algorithm with a new algorithm, using the same state. Then M is a valid adaptability method.*

Alternatively, a generic state adaptability method can be developed that works by aborting transactions to adjust the generic state information so that it could have been produced by the new algorithm. An important characteristic of sequencers for transaction systems is that regardless of the transactions that have already been committed it is always possible to adjust the currently executing transactions so that a new algorithm can correctly sequence them. In the worst case we can simply abort all active transactions, leaving the system in an initial state. Of course we are interested in situations in which few active transactions must be aborted. Adjusting the generic state has the advantage that it can work with sequencers that do not have the generic state compatibility property, but it requires additional effort in determining the set of transactions to be aborted. The correctness proof is similar to the above; most of the work lies in the definition of the state information to be passed to the new sequencer algorithm. In several special cases the conversion can be performed without aborting any transactions. For instance, when switching from one algorithm to a new algorithm that accepts a superset of the histories accepted by the old algorithm no transactions will have to be aborted.

The generic state method of adaptation has the advantages of simplicity and efficiency, but it only applies to a small class of sequencers. Furthermore, the requirement that all algorithms use the same data structure is restrictive. Some algorithms may be less efficient

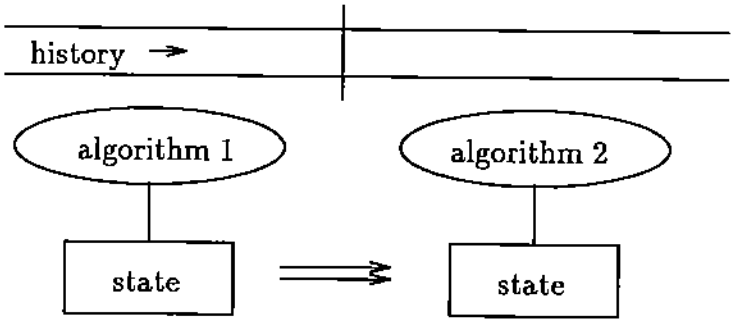


Figure 2: The state conversion approach to sequencer adaptability.

using the generic data structure, and the data structure may turn out not to be general enough for some future algorithm. For instance, hash tables of locks support locking algorithms in constant time per access. However, they do not contain enough information to support timestamp ordering concurrency control. Alternative data structures that support both locking and timestamp ordering are likely to be more expensive than constant time per access. Extending the generic state idea to adjusting the state by aborting transactions is more flexible, but still requires the existence of a single data structure to maintain the state information for all possible algorithms for a sequencer. The next section proposes a method that does some additional work during conversion so each algorithm can use its own data structure.

2.3 State Conversion Adaptability

In many cases the data maintained by different algorithms for a sequencer contains the same information in different forms. Sometimes it is not feasible to use the same data structures for all of these algorithms for reasons of efficiency or compatibility, but it may be possible to convert the data between the different forms. This suggests an adaptability method that works by invoking a conversion routine to change the state information to the format required by the new algorithm. Figure 2 depicts the conversion process from algorithm 1 to algorithm 2 by converting state information from one form to another. The new data structure may have to be modified to represent a situation that the new algorithm is able to correctly sequence.

The principal advantage of the state conversion adaptability method is that it does not require a single data representation for all algorithms. Each algorithm uses its own natural, efficient data structure. All that is needed to convert from algorithm A to algorithm B is a single routine that converts the data structures maintained by A to the data structures needed by B. This is summed up in the following lemma.

Lemma 2 *Let A and B be algorithms for sequencer S such that there is a conversion algo-*

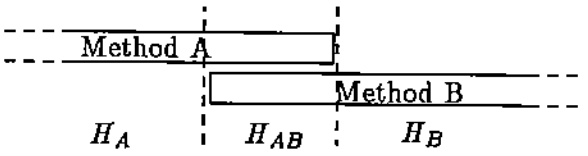


Figure 3: The structure of histories accepted by the suffix-sufficient adaptability method.

rithm from the data structure for A to the data structure for B as described above. Let M be the conversion method that converts from A to B by running the data conversion algorithm and then replacing A with B. Then M is a valid conversion method.

The state conversion adaptability method is flexible. It can be applied to sequencers that have a wide range of algorithms, and allows each algorithm to use the most efficient data structure for its own purposes. The major problem with the approach is that a conversion algorithm is needed between each pair of algorithms for the sequencer. This problem is exacerbated by the fact that correctness of the adaptation depends on correctness of the conversion algorithm. Thus to permit arbitrary adaptation for a sequencer for which n different algorithms have been implemented would require n^2 conversion algorithms and n^2 correctness proofs. One approach to alleviate this problem is to use a hybrid between the generic state and the state conversion methods. The old data structure is converted to a generic data structure which is then converted to the data structure for the new algorithm. This would reduce the implementation effort to $2n$ conversion algorithms and correctness proofs. The cost would be in possible information loss in the conversion to the generic data structure that might require additional aborts. An even greater improvement would be an adaptability method that depends only on the sequencer rather than on the particular algorithms involved in adaptation. The next section explores one such approach.

2.4 Suffix-sufficient State Adaptability

The basic observation used below is that an implementation of a sequencer will seldom have to refer to state information that is very old. This section presents a model within which proofs of such locality of reference are easy to construct for some sequencers.

The underlying idea is that during the adaptation process actions are permitted only when both the old and new algorithms for the sequencer permit them. The old algorithm A guarantees correctness of the "old" output history, and the new algorithm B permits actions to enter the "new" history only if B will be able to correctly complete the sequencing of their transactions. Figure 3 depicts the structure of these histories. The "old" history has a prefix H_A that is acceptable to method A. H_{AB} is a part common to both "old" and "new" histories that is acceptable to both A and B. The "new" history has a suffix H_B that is acceptable to B. During creation of the H_{AB} part of the history, algorithm B records enough state information to take over the sequencing job by itself. When this condition, called a

suffix-sufficient state, is detected by the adaptation method, algorithm A is stopped, and only algorithm B continues. Of course, this approach can only succeed if the algorithms for the sequencer are likely to permit actions in almost the same order. The higher the overlap between the old and new algorithms, the higher the degree of concurrency permitted during conversion. In most cases the algorithms for a sequencer permit substantially the same histories, but otherwise there will be a performance penalty. The difficulty lies in knowing when the conversion has terminated. We define a conversion termination condition to be a predicate on histories that guarantees that conversion can be terminated.

Definition 6 *A* conversion termination condition *for a sequencer S with correctness condition ϕ is a predicate ρ that determines whether adaptation is complete. More formally, let M be the conversion method that works by running both A and B for an interim period and then replaces A with B. If for any history $H = H_A \circ H_M \circ H_B$ such that H_A is the output of A, H_M is the output of M, H_B is the output of B and $\rho(H_A, H_M) \Rightarrow \phi(H)$, then ρ is a conversion termination condition for S.*

Remember that the theory behind the suffix-sufficient state conversion method is that if we wait long enough the new algorithm will have absorbed all of the important state information. ρ is a predicate that tells us when 'long enough' happens. The following lemma expresses the correctness of the suffix-sufficient adaptability method that completes after a conversion termination condition is satisfied.

Lemma 3 *Let ρ be a conversion termination condition for sequencer S and let A and B be algorithms for S. Let M be the adaptation method that works by running both A and B until ρ is satisfied (as above) and then replaces A with B. Then M is valid.*

For sequencers for which there exist conversion termination conditions that are easy to implement this theorem provides an adaptability method that works for any possible algorithm. This allows us to design the system in such a way that it is able to accommodate new algorithms as they are developed, and adapt to them dynamically in response to environmental conditions. An alternative would be to prove conversion termination theorems about adapting between each pair of algorithms. This is more flexible, but suffers from the disadvantage of the state conversion approach, i.e. a method must be proven correct and implemented for each pair of algorithms.

The main advantage of the suffix-sufficient state adaptability is its generality. Any new algorithm for the sequencer can be immediately incorporated. Neither the implementation of the algorithm nor the existing conversion code need to be changed to incorporate the new algorithm. A weakness of the suffix-sufficient state approach is that the conversion termination condition may not be guaranteed ever to be true. Even if the termination condition eventually becomes true we may spend a very long time with poor performance while trying to convert to the new algorithm. The next section suggests several ways by which we can achieve earlier termination of the conversion algorithm.

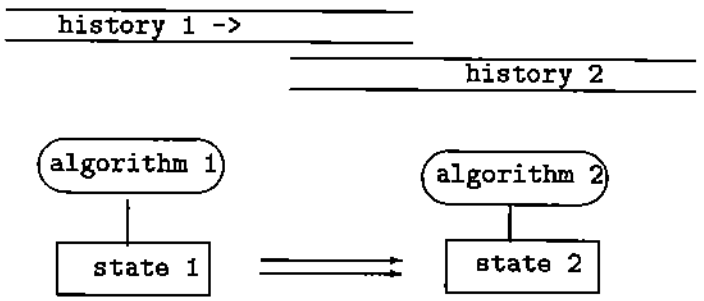


Figure 4: Suffix-sufficient state conversion.

2.5 Suffix-sufficient State Amortized

This section consists of improvements to the suffix-sufficient state adaptability method, that speed up the termination of the conversion process. Each of these ideas is a way in which state information can be transferred from the old algorithm to the new algorithm in parallel with transaction processing. These are mixtures of the suffix-sufficient state method and the state conversion method. Figure 4 depicts this conversion method. Algorithm 1 is being converted to algorithm 2, by absorbing history information. In addition, some state information is being directly transferred from algorithm 1 to algorithm 2. Thus, state information is simultaneously being absorbed through the current history and from state information about the old history. In the state conversion method transaction processing must halt while the state information is being transferred, but these new ideas simultaneously process actions and transfer state information. Rather than stopping transaction processing during conversion, the suffix sufficient state method amortizes the cost of conversion over the cost of processing new actions. These hybrid methods enhance the suffix sufficient state approach by guaranteeing eventual termination.

The simplest suggestion is to maintain a log of actions as they are processed. When the conversion process is started it proceeds as in the suffix-sufficient state method in that both the old and new algorithms are simultaneously run. However, in addition to the actions that we want the algorithms to sequence, we pass actions from the old history to the new algorithm. Since we will not know how many of the old actions must be seen by the new algorithm they should be passed to it in reverse order. Including these actions in its state information will permit the conversion process to terminate earlier. Note that the conversion termination function will have to be modified to incorporate this reverse history information.

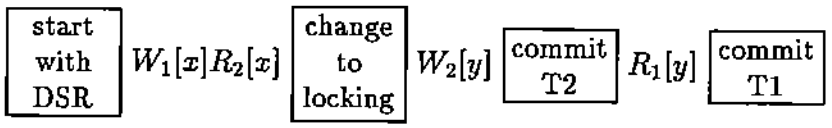


Figure 5: An example of an incorrect concurrency control decision caused by uncautious conversion.

Once again it is possible that some of these old actions will belong to active transactions which may have to be aborted if the action is not acceptable to the new algorithm. However, now the conversion is guaranteed to terminate, since eventually the entire history of all active transactions will be incorporated.

Rather than pass the raw actions from the old history, it is preferable to pass converted state information directly from the old algorithm. The idea is to develop techniques for incrementally converting the state information from one concurrency controller to another. By converting incrementally rather than all at once, as in the state conversion approach, the impact on throughput can be spread out. The advantage of this modification is that the state information in the old algorithm is usually small compared to the history information, so termination is likely to happen more quickly.

3 Concurrency Control: An Example for Adaptability

This section uses concurrency control as an example to demonstrate algorithmic adaptability techniques using the sequencer model. Concurrency control has been studied formally, and is well understood. For this reason it serves as a good example of how the adaptability techniques of section 2 can be applied to transaction systems. We do not suggest that concurrency control is the best subsystem for implementing adaptability. We are particularly interested in the application of adaptability to reliability issues. However, concurrency control is a good vehicle for explaining our ideas.

When concurrency control methods are switched while the system is running, special care must be taken to ensure correctness. Figure 5 is an example of how locking depends on the structure of the past history. In this example a concurrency controller implementing DSR had been running and it was removed from the system and replaced by locking without appropriate preparation. The final history is not serializable because transaction 1 read y after transaction 2, and transaction 2 read x after transaction 1. Although both concurrency controllers made locally correct decisions, the combination permitted a non-serializable history. This section applies the techniques of Section 2 to permit adaptable concurrency control

```
trans = record
    id:  tranid;
    read_actions:  action_list;
    write_actions:  action_list;
    waiting-for:  tranid;
    blocking:  tranid;
end;
```

Figure 6: Transaction-based generic data structure.

while preserving correctness.

We use three classes of concurrency controllers to present our ideas: two-phase locking (2PL) [EGLT76], timestamp ordering (T/O) [Lam78], and optimistic concurrency control (OPT) [KR81]. The version of 2PL that we are using implicitly acquires read locks when data items are read, implicitly acquires write locks during transaction commit, and releases all locks after commitment. T/O chooses a timestamp for each transaction when it starts, and aborts transactions that attempt conflicting actions out of timestamp order. OPT allows transactions to proceed without concurrency control until commitment, at which time it checks for conflicts between the committing transaction's read-set and committed transactions' write-sets, aborting the committing transaction if there is a conflict. All three of the methods buffer writes in a temporary work-space until commitment. To simplify the discussion we will assume that all transactions are either uncommitted (active) or committed during adaptation. The conversion algorithms will wait until transactions that are in the process of committing terminate.

3.1 Generic State Adaptability

We propose two separate data structures for generic state adaptability of concurrency control. Both of these data structures maintain timestamps of past actions, and support many different concurrency control methods. The first data structure is a list of the actions of recent transactions, grouped by transaction. The second lists the recent actions performed on each data item, accessed by data item. In this section we discuss the advantages and disadvantages of each data structure in terms of the time and space required for 2PL, T/O and OPT.

Figure 6 shows the transaction-based data structure. Each transaction includes a list of timestamped accesses to data items, a list of transactions that are waiting for this transaction to terminate, and the number of the transaction for which this transaction is itself waiting.

```
data_item = record
    item:  item_id;
    read_actions:  action_list;
    write_actions:  action_list;
end;
```

Figure 7: Data item-based generic data structure.

If individual transactions access a large number of data items the action lists should have additional structure to speed up lookups. For instance, a separate pointer can be kept to the write actions, a simple rule such as transpose or move-to-front can be used to keep hot items near the front [BM82], or in the extreme case the actions could be organized in a search tree [AHU74]. For the common case of transactions with just a few actions, however, a simple unorganized list will be most efficient. Actions of committed transactions must be maintained to support techniques such as OPT. To bound the growth of required storage, old actions should be periodically purged. Transactions that need to examine previously purged actions to determine whether they can commit must be aborted, so choosing the correct actions to purge is important. The most straight-forward way to purge actions is in FIFO order, although in some cases application-specific knowledge may suggest a better choice. One general improvement is to move actions towards the front of the FIFO list when they are accessed, so heavily accessed actions are purged later.

Figure 7 shows the data item-based data structure. It is similar to the data structures maintained by version-based concurrency control methods [Ree83], except that it maintains only timestamps and not values. Each data item has separate timestamped lists for read and write actions. The actions lists are maintained in order of decreasing timestamp to improve performance. Note that ordering the actions in this manner does not require extra work since the actions will occur in decreasing order naturally. For performance reasons the lists should be kept in main memory. Old actions are purged as before, again with the stricture that transactions that need to examine purged actions are aborted. The data items should be organized in memory in a search tree or a hash table. In the performance discussion below, we will assume that a hash table similar to conventional in-memory lock tables is used for the data items, with the actions chained in decreasing timestamp order from each data item.

Performance Now we will consider the performance of the two data structures on 2PL, T/O, and OPT concurrency control. The transaction-based data structure must scan the list of actions to search for conflicts with each new action. Scanning will require time

proportional to the number of actions of transactions that may conflict. For 2PL, only active (uncommitted) transactions need to be considered, so the time will be proportional to the number of actions of active transactions. For OPT only committed transactions need to be considered, but this is likely to involve considerably more actions. T/O must consider both committed and uncommitted transactions, but only has to check transactions that have been assigned a higher timestamp. The timestamp of a transaction will be the timestamp of the first data access by the transaction. Since transactions with higher timestamps are younger, they are likely to have relatively few data accesses, so T/O is likely to have fewer actions to scan than either 2PL or OPT. An implementation could improve performance by organizing the transaction list so that T/O can directly access transactions with higher timestamps.

The data item-based data structure is more efficient, since the head of the action list is the only item that needs to be checked for potential conflict in each case. 2PL just checks if the transaction that performed the head action is still active. T/O checks if the head action has a larger timestamp than the current transaction's timestamp. OPT checks if the write action at the head of the list has a larger timestamp. In each case, the check can be done in constant time if the read and write action lists are maintained separately for each data item.

Storage Any difference in memory requirements can be significant since the amount of available storage determines how long actions can be maintained, which in turn determines the number of aborts caused by potential conflicts with purged actions. This factor becomes especially important when long transactions are running, since long transactions are more likely to have conflicts with old actions. The storage required for the two data representations is about the same, in that they maintain the same set of actions. The transaction-based structure uses somewhat less space because it does not use a search structure. Furthermore, the data item-based structure must maintain a separate data structure to purge actions of transactions that eventually abort. If the data item-based method uses an adaptable hash table that changes its size as the number of entries increase or decrease, though, it should require no more than a factor of two additional storage. Storage requirements can be reduced further by using a search tree instead of the hash table.

The data item-based structure wins in performance. If memory is scarce, the search tree implementation can be used. The principal advantage of the transaction-based structure is that it closely resembles the readset and writeset information already kept by the transaction manager, and hence can be implemented easily.

Aborting Transactions Some transactions may need to be aborted during conversion, to adjust the generic state so that it is suitable for the new algorithm. For instance, in converting from T/O to locking, lock conflicts must be resolved by aborting some active transactions. The section on converting state adaptability presents algorithms for choosing

```
for l in lock_table do begin
    l.t.readset := l.t.readset + l.item;
    release_lock(l);
end;
```

Figure 8: An algorithm for converting from 2PL to OPT concurrency control.

transactions to abort during conversion. These algorithms can also be used for generic state adaptability.

3.2 State Conversion Adaptability

We now demonstrate the use of the converting state approach to adaptable concurrency control with several examples of algorithms for converting the data structures. All of the examples require time at most proportional to the union of the sizes of the read-sets of active transactions. In several cases it is necessary to abort some active transactions to avoid potential future problems.

2PL to OPT OPT chooses serialization order based on commit time. It enforces this by causing a transaction T to abort if T reads an item before some committed transaction wrote that item. OPT checks the read sets of active transactions against the write sets of committed transactions, and aborts if there is conflict. In converting from 2PL to OPT, write sets for previously committed transactions are not needed, because 2PL already guarantees that any active transactions performed conflicting reads after committed transactions finished writing. (Otherwise, the committed transaction could not have obtained the write locks.) So, to convert from 2PL to OPT, we convert the read locks into readsets, release the locks, and restart processing. Figure 8 shows the algorithm in a simple Pascal-like language. In the figure, if l is a lock, $l.item$ is the item being locked, and $l.t$ is the transaction that holds the lock. The conversion takes time proportional to the number of read-locks.

OPT to 2PL Converting from OPT to 2PL requires that some transactions be aborted. We say that a transaction has an *outgoing dependency edge* if in the dependency graph the transaction has an edge that forces it to serialize before some other transaction. The algorithm depends on a general property of locking concurrency control:

Lemma 4 *In converting to 2PL concurrency control, it is sufficient to guarantee that there are no outgoing dependency edges from active transactions.*

Any such outgoing edges would be to a previously committed transaction, since active transactions have not yet written. In a sense, these are 'backwards' edges, since they force an active transaction to serialize before an already committed transaction. Locking never permits such edges, and may serialize transactions incorrectly if they exist.

Proof. Suppose there is a cycle in the conflict graph. Choose the smallest such cycle. It must include at least one transaction that committed under 2PL (else the old concurrency control method failed to preserve serializability). Consider the situation just before the last such transaction, T, committed (creating the cycle). We know from the hypothesis that at conversion time there were no outgoing edges from T to any transaction. Further, 2PL will prevent such edges from being created after conversion, until T is committed and has released its locks. Therefore, T does not have any outgoing edges, and cannot be a part of the cycle. □

The significance of this lemma is that violations of the locking protocol among previously committed transactions (e.g., which committed using OPT concurrency control) will not cause future serializability violations. If we restrict our attention to adaptability methods that convert to pure 2PL, the lemma is necessary as well as sufficient. This is because if the active transaction writes the data item that caused the backward edge it will create an unserializable execution. Note that an equivalent property does not hold for T/O, because T/O allows active transactions to have both incoming and outgoing dependency edges.

Using the lemma, it is easy to convert from OPT to 2PL: just abort any active transactions with backward edges. An easy way to identify backward edges is to run the OPT commit algorithm on active transactions, and abort those that fail. Note that these transactions would have been aborted eventually by the OPT algorithm anyway. Then, we assign read-locks to the active transactions based on their readsets, and continue processing. There can be no lock conflicts, since the operations are all reads at this point. This conversion method works for converting from any validation technique to 2PL, although Kung's validation [KR81] must be used to abort transactions with backward edges.

Conversion from any method to 2PL Lemma 4 leads to a general method for converting from any concurrency control method to 2PL, if we have access to past history information. The idea is to reprocess the history from the most recent action that was coactive with some currently active transaction to the present, aborting any transactions that violate the locking rules. Earlier actions can be ignored, since they cannot cause outgoing dependency edges from active transactions. We use a data structure called an *interval tree* to maintain the time history of the locks for each data item. The interval tree provides O(log n) lookup and insert of non-overlapping time intervals. Each time interval represents a period when a lock was held on the data item. When an action attempts to insert an overlapping time interval into one of the trees, some transaction must be aborted. Any resolution rule that guarantees non-overlapping intervals in all of the trees is acceptable. The simplest such rule is to abort transactions that try to insert actions that cause overlaps.

```
for t in active_trans do begin
    for a in t.actions do begin
        if a.writeTS > t.TS then abort(t)
        else get_lock(t, a.item);
    end;
end;
```

Figure 9: An algorithm for converting from T/O to 2PL concurrency control.

The advantage of this technique is that it works in general to convert from any method to 2PL. The disadvantage is that it has to re-process what may be a substantial portion of the recent history. In general, special case algorithms such as those above will be more efficient when they are available.

T/O to 2PL Once again we can use lemma 4 to construct a conversion algorithm that aborts active transactions with 'backward' edges to committed transactions. In this case backward edges are easy to identify, since they will be represented by data items whose write timestamp has changed since an active transaction read them. The algorithm is given in figure 9. In the figure, active_trans is the set of active transactions, t.actions is the set of actions in transaction t, and TS is a timestamp.

These algorithms give the flavor of the advantages and disadvantages of converting state adaptability. The algorithms are each simple and efficient, but there are many of them.

3.3 Suffix-Sufficient State Adaptability

Suffix-sufficient state adaptability converts between arbitrary concurrency controllers without the need for a generic data structure or for special conversion algorithms. The difficulty with suffix-sufficient state adaptability is the need for a conversion termination criteria that is powerful enough to allow early termination, and yet reasonably easy to implement. This section exhibits one such conversion termination condition for concurrency control, including a proof of correctness.

The following conversion termination condition permits adaptation for all concurrency controllers that accept subsets of the digraph-serializable histories, or DSR [Pap79]. DSR includes all known practical concurrency controllers. Using this termination condition, suffix-sufficient state adaptability is possible between any two concurrency controllers in this class.

Let M be the suffix-sufficient state conversion method of Section 2.4, and let $H = H_A \circ H_M \circ H_B$. Recall that ρ is a function that determines when M is done with the job of conversion, and must be specified for each sequencer.

Theorem 1 *Method M is a valid adaptability method for concurrency control methods contained in DSR under the conversion termination condition* $\rho(H_A, H_M) \Longleftrightarrow$

1. *All transactions started in* H_A *complete in* H_A *or* H_M*, and*
2. *There is no path in the merged conflict graph from a transaction in* H_B *to a transaction in* H_A*.*

The first part of the restriction function is simple and intuitive. The intermediate part of the history, H_M, is present to ensure that the transactions that were started under method A complete under method A. Thus H_M must extend until these transactions have all completed in order to guarantee the serializability of $H_A \circ H_M$. Part 2 of ρ is also simple but is less intuitive. The insight in the proof (below) is that if histories $H_A \circ H_M$ and $H_M \circ H_B$ are constructed carefully, their conflict graphs will merge to produce the conflict graph for the entire history $H_A \circ H_M \circ H_B$. Part 2 of ρ is a sufficient condition for this merged conflict graph to be acyclic, which demonstrates that the entire history accepted by the adaptable concurrency controller is serializable.

Proof. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be the conflict graphs for $H_A \circ H_M$ and $H_M \circ H_B$, respectively. The merged conflict graph is $G = (V, E)$ where $V = V_1 \cup V_2$ and $E = E_1 \cup E_2$. It is easy to see that G is the conflict graph for $H_A \circ H_M \circ H_B$, since it includes all of the transactions and all of the conflict edges. In order to prove that the conversion is correct we must prove that the entire history is serializable or, equivalently, that the entire history has an acyclic serializability testing graph ([Pap79] discusses STGs). We shall constructively exhibit an acyclic STG by showing that the conflict graph G is acyclic.

Suppose, for purposes of contradiction, that G has a cycle. Since A and B are known to be correct concurrency controllers the cycle cannot be entirely contained in $H_A \circ H_M$ or $H_M \circ H_B$. This means that the cycle must contain at least one transaction from H_A and one from H_B. Call these transactions T_A and T_B respectively. We can choose names for the other transactions in the cycle and write it starting from T_A as

$$\begin{aligned} T_A = T_1 \ &\rightarrow T_2 \rightarrow T_3 \rightarrow \cdots \rightarrow T_{m-1} \rightarrow T_m = T_B \rightarrow \\ &T_{m+1} \rightarrow \cdots \rightarrow T_{n-1} \rightarrow T_n \rightarrow T_1 = T_A. \end{aligned}$$

Notice that the second half of the cycle

$$T_B = T_m \rightarrow T_{m+1} \rightarrow \cdots \rightarrow T_{n-1} \rightarrow T_n \rightarrow T_1 = T_A$$

is a path from H_B to H_A, contradicting part 2 of the definition of ρ from Theorem 1.

This contradiction means that G must be acyclic. Since G is an acyclic STG for $H_A \circ H_M \circ H_B$, the history must be serializable. Since the conversion method only permits serializable histories it is valid. □

3.4 Related Work

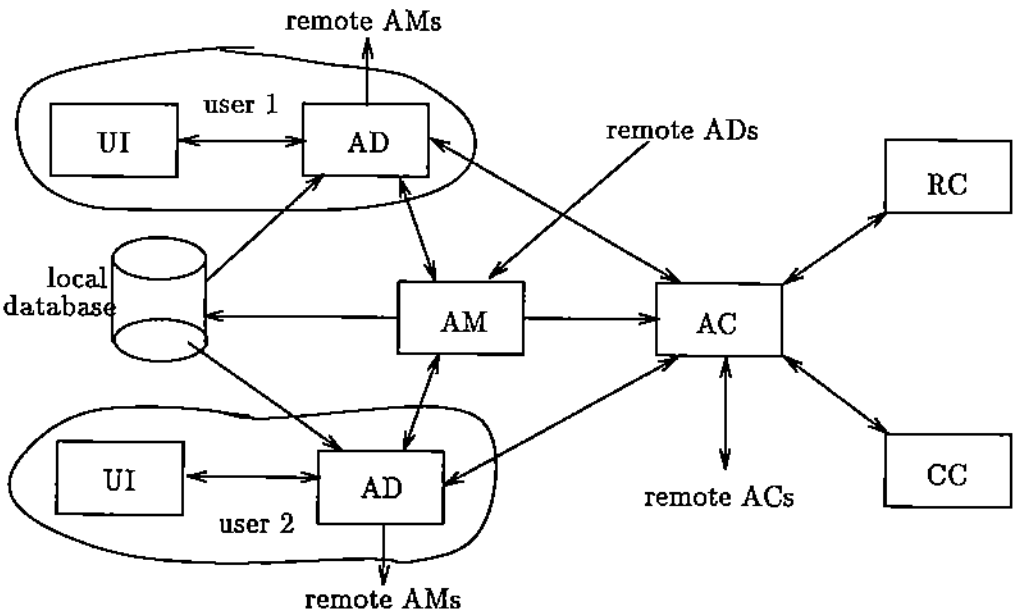
Adaptable concurrency control has been studied in the past primarily with the view of developing hybrid concurrency control methods that combine the virtues of the optimistic (OPT) and locking (2PL) protocols (OPT and 2PL are described in section 3) [Lau82, SL86, BM84]. We shall call this type of adaptability *per-transaction adaptability*. Some of these techniques also extend to *spatial adaptability*, in which accesses to parts of the database require locks, while accesses to the rest of the database run optimistically.

These techniques all fall under our category of generic state adaptability, because they rely on merging the information needed by locking and optimistic. A unique feature of these methods is that they are able to simultaneously support both concurrency control methods, with individual transactions choosing which to use. This flexibility is possible because the generic state used is always kept compatible with either method. In general this approach is unsatisfactory, because individual concurrency controllers perform better when they have the freedom to manipulate the data structure as they wish. For the particular case of locking and optimistic, however, it works quite well, because they have similar constraints on concurrency.

4 Engineering Adaptability in RAID

This section describes the ideas for adaptability in the RAID system. RAID is an experimental distributed database system [BR89] being developed on SUNs under the UNIX operating system (Figure 10). RAID is structured as a server-based system. Each major functional component of RAID is implemented as a server, which is a process interacting with other processes only through the communication system. Currently there are six major subsystems in RAID: User Interface, Action Driver, Access Manager, Atomicity Controller, Concurrency Controller, and Replication Controller. We focus on adaptability in concurrency control, network partitioning (in the Atomicity Controller), and reconfiguration (in the Replication Controller). We also describe how the structure of RAID enhances adaptability, including discussions of inter-server communication, the capability to merge several servers into the same process, and server relocation. In our work on RAID we have studied many facets of adaptability. The rest of this section details our successes, but first we would like to mention some of the lessons we have learned from our mistakes.

One of the earliest decisions we made in RAID was that the flow of control information among servers would be fixed, and that different implementations of the servers would all support the same interface. We have learned that it is difficult to support many different types of server implementations with the same control flow. In particular, the RAID concurrency control implementation is designed for validation. It supports optimistic concurrency control well, but works less well for pessimistic methods such as locking. The easiest solution is to run RAID with a control flow that is the superset of the control flows needed for every



Legend				
	AC	Atomicity Controller	UI	User Interface
	CC	Concurrency Controller	AM	Access Manager
	RC	Replication Controller	AD	Action Driver

Figure 10: Raid Site Structure.

possible server implementation. This approach would have high overhead for the common case, though. An alternative is to allow the flow to change dynamically depending on the set of algorithms running at a given time. In this model RAID would become a set of library routines that support the development of distributed systems through services such as naming, logging, etc. A disadvantage is that an implementation of a server for one configuration is unlikely to work with any other configuration because the interfaces are different. Yet another possibility is to add support for a subset of the possible control flows, and allow the system to choose control flows based on the set of active servers. In RAID we have identified two additional control flows that would especially improve adaptability. First, the Action Driver should be able to pass actions directly through the concurrency controller to better support pessimistic concurrency control methods. Second, the Replication Controller server should be extended to handle partial replication, and the Action Driver should also be able to pass individual actions through the Replication Controller. When running an optimistic concurrency controller the entire set of actions would be passed to it in a single message, as in the current system. When running with full replication the Action Driver would not send requests to the Replication Controller.

There is one further set of control flows that would be useful. Some algorithms for certain servers work by contacting remote servers. Currently, most remote communication is channeled through the Atomicity Controller. An alternative would be for servers of the same type on different sites to communicate directly. Thus, the Concurrency Controller would be able to communicate with other Concurrency Controllers on remote sites to make its decisions. One disadvantage is that either the interfaces between servers of the same type must be the same for all implementations, or all sites must run the same implementation of each server at the same time. The former is unlikely to work, and the latter restricts heterogeneity. Another disadvantage is that communicating with remote sites for every action is likely to be expensive. The expense can be decreased by implementing the servers with specially designated distributed state supported by shared distributed memory.

The process model also restricts adaptability. In RAID, incoming messages trigger special lightweight processes (LWPs) that perform a task, and then return control to the main loop to receive another message. Since the processes run synchronously they cannot block, and even if multiple servers are merged into a single address space there is still only a single thread of control. A better implementation would be based on remote procedure calls (RPCs) [BN84] with invocation of asynchronous lightweight processes as needed. The LWPs can block on I/O or messages and different servers are supported by different LWPs. Multiple LWPs can work on the same server if semaphores are used for access control to the server data structures. A language with strong typing would be preferred to avoid problems with access violations in the shared address space. The RPC model would provide a single reference style for both local and remote references. This would make the task of supporting merged servers simpler. Furthermore, the ability to have multiple LWPs implementing the same server increases the potential parallelism in RAID, which will be especially important on

multiprocessors.

4.1 Concurrency Control

RAID [BR89] uses a concurrency control method called *validation.* Validation works by collecting timestamps for actions while a transaction is running and then distributing the entire collection of timestamps for concurrency control checking after the transaction completes. Each site checks for local concurrency conflicts, and then the sites agree on a commit or abort decision. The local conflicts can be detected by checking the transaction against the history of committed transactions using methods ranging from locking to timestamp-based to conflict-graph cycle detection.

The RAID concurrency controllers support the three types of adaptability discussed in section 3. The global concurrency controller is composed of separate Concurrency Control servers on each site, communicating through the Atomicity Controller. Each Concurrency Controller maintains a copy of the transaction-based generic state described in section 3.1. History information in the generic state is periodically purged by setting a logical clock forward and discarding all actions older than the new clock time. Transactions that have some action that is older than any action in the history for that data item have to be aborted. Converting to a new concurrency controller can be done by just switching to the new algorithm, as long as the state is acceptable to it. We are working on writing routines to guarantee that the state is acceptable. We are implementing suffix-sufficient state adaptability using the generic data structures. A new concurrency controller is started in parallel with the old one, and decisions are made jointly during an intermediate period. This concurrency controller runs in the same process as the old one, and the two are invoked synchronously. When the conversion termination condition from section 3.3 is satisfied, the old concurrency controller is terminated since the state is now acceptable to the new concurrency controller. In the future, more efficient versions of the RAID concurrency controller will be implemented that directly maintain lock tables and conflict graphs. Then, we will use the conversion algorithms of section 3.2 to convert the data structures.

In validation concurrency control the only requirement on each local concurrency controller is that it correctly check the transactions that are sent to it for serializability. This means that the techniques of Section 3 can be applied to the local concurrency controllers individually without need to coordinate with other sites. So it is possible to run a version of RAID in which each site is running a different type of concurrency controller, chosen based on the local environment. Thus validation can also be used to support heterogeneous database systems, each of which is running its own concurrency controller. The only requirement is that each of the transaction managers preserves the timestamp information for transactions as it executes them. This information is passed to each of the local database systems which check it for validity.

We wish to make the system adaptive, so it automatically responds to changes in its

environment and workload. We have developed a prototype expert system that determines when to switch to a new concurrency control algorithm [BRW87]. The expert system uses a rule database describing relationships between performance data and algorithms. The rules are combined using a forward reasoning process to determine an indication of the suitability of the available algorithms for the current processing situation. Based on the current environment, it chooses an appropriate algorithm for the environment, along with an indication of how much better the new algorithm is than the currently running algorithm. The expert system also maintains a confidence (or "belief") value in its reasoning process. This is used to avoid decisions that are susceptible to rapid change, or that are based on uncertain or old data. If the advantage of running the new algorithm is determined to be larger than the cost of adaptation, the expert system recommends switching to the new algorithm.

4.2 Network Partitioning

Network partitioning control is the task of maintaining consistency in a distributed system despite some sites not being able to communicate with other sites [DGS85]. There are many solutions to this problem, falling broadly into the classes of optimistic and conservative methods [DGS85]. The RAID reconfiguration method cannot yet handle network partitions (see section 4.3). A future version of RAID will be set up to run either a majority partition network partition algorithm or an optimistic algorithm. The majority partition algorithm dynamically determines the majority partition during multiple partitions and merges [Bha87]. The algorithm recognizes situations in which a small partition can guarantee that no other partition can be the majority, and thus declare itself the majority partition. The optimistic algorithm changes to a mode in which transactions run as normal, but are only able to semi-commit until the partitioning is resolved. Both of these partition control algorithms are good sometimes, but neither is best for all conditions. Let us see how the techniques of Section 2 can be applied to respond to changing environmental conditions.

The state conversion adaptability method can be used to explicitly change between the two partition control methods. Suppose RAID is running the optimistic partitioning control algorithm because only brief network partitionings are likely. During a certain period the probability of very long partitionings becomes high, perhaps because of electrical storm activity or repair work. The system begins to set up the majority partition method, although the optimistic method must still take over if there is a partitioning. Setup involves initializing data structures for the partition and merge information on each site. Once the majority partition method is ready to handle a partitioning, a two-phase commit protocol is used to switch from the optimistic method to the majority partition method. There is a small window of vulnerability during the conversion, corresponding to blocking during termination of two-phase commit [Ske82], but after the conversion the system runs just as if it had started with the majority partition method.

We plan to implement a data structure which contains enough information for either method to be used. This data structure supports generic state adaptability. When a partitioning occurs the optimistic method is used for the first few minutes, or until the partitioning is determined to be of long duration by some other criterion. Then a conversion algorithm is applied which rolls back any transactions which made changes that are not consistent with the majority partition rule, which is then used for the duration of the partitioning. This method has the advantage of permitting adaptability even during a partitioning, but requires more state information to be maintained.

Voting protocols are especially convenient for adaptability. Recent work on voting has extended to protocols that dynamically change the number of votes assigned to each data copy during a partitioning [BGS86]. Herlihy generalizes to non-voting quorum methods [Her87]. Rather than specifying quorums to be a majority of votes, Herlihy provides for explicitly listing sets of sites that form read and write quorums. [BB89] also supports adaptable quorums. Quorums that have not been changed during a failure can be used after the failure is repaired. The effect is that the system dynamically adapts to the failure as objects are accessed, with more severe failures automatically causing a higher degree of adaptation.

Both voting and the more general quorum protocols are examples of converting state adaptability. They both rely on an algorithm that runs when adaptability is desired to convert from one data structure to another. An interesting feature of these protocols is that only the data structures are converted; the same transaction processing algorithms are used after conversion. Thus, this adaptability is entirely data-driven.

4.3 Reconfiguration

Another aspect of adaptability in distributed systems is the *reconfiguration* problem [PW85, chapter 5]. Reconfiguration is the process of adding or deleting sites from a distributed system without violating consistency. When a site leaves the system, either because of a failure or an administrative decision, its transactions must be terminated. The data can be brought up to date using a multi-phase commit protocol in such a way that the rest of the system can continue processing transactions [Ske82]. When the site rejoins the system its data must be brought up to date. Adaptable systems can dynamically incorporate new sites into the system, and provide graceful degradation as sites leave the system.

Recovery after a failure involves two complementary problems: reconstructing an old, but consistent, state, and reaching an up-to-date state. First, the servers must be instantiated and must rebuild their data structures from the recent log records. Actions are sent from the Access Manager to the recovering server, and replayed by the server to establish the necessary state information. Second, the servers must collect information from active servers about the final status of transactions that were involved in commitment before the failure.

To keep track of out-of-date data items, RAID maintains commit-locks during failure. The Replication Controller keeps a bitmap that records for each other site which data items

were updated while that site was down. When the site recovers, it collects the bitmaps from all other sites and merges them. Then the recovering site marks all of the data items that missed updates as stale, and rejoins the system. Now, the recovering site can process transactions, fetching fresh copies of stale data from other sites as needed. RAID uses a two-step process to refresh stale copies. During the first step, some stale copies are refreshed automatically as transactions write to the data items. After 80% of the stale copies have been refreshed in this way (for free!), RAID issues copier transactions to refresh the rest of the stale copies. Experiments show this to be an effective way to efficiently maintain fault-tolerance [BNS88]. We have a complete implementation of this technique in a stand-alone system called mini-RAID, and everything except copier transactions implemented in the full RAID system.

4.4 Distributed Commit

Multiple-phase distributed commit algorithms successfully terminate transactions on all sites in a distributed system. Three-phase algorithms tolerate arbitrary site failures without causing blocking, at the cost of an extra round of messages [SS83]. RAID currently uses two-phase commit, but using the converting state adaptability method we can convert between two and three phase commit algorithms while processing transactions.

The fundamental rules controlling the commit protocols are:

messages Messages are received and sent from/to one or more sites during each transition.

commitable state A state is commitable if all other sites have replied 'yes' to the transaction and the state is adjacent to a commit state. All other states are called non-commitable.

one-step rule All sites in the distributed system are within one transition of all other sites in the commit protocol.

non-blocking rule A commit protocol is non-blocking if and only if no commitable states are adjacent to non-commitable states.

In RAID, the one-step rule is enforced despite failures by requiring that all transitions be logged before they can be acknowledged to other sites. The non-blocking rule is maintained with a special termination protocol that understands both two-phase and three-phase commit protocols.

The Atomicity Controller tracks each transaction in a state-transition diagram. Adaptability adds transitions from certain states in the two-phase commit protocol to other states in the three-phase commit protocol and vice versa. The complete set of transitions supported in RAID is depicted in figure 11. The start states (Q), abort states (A), and commit states (C) are equivalent between the two diagrams. Dashed lines in the figure represent transitions

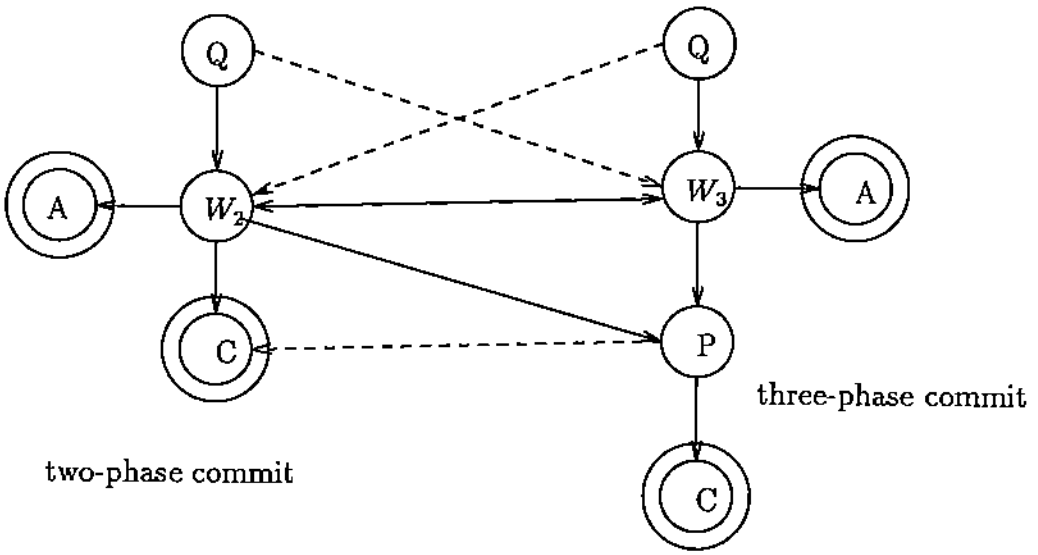


Figure 11: Adaptability transitions between two-phase and three-phase commit protocols.

to or from equivalent states that may be useful in implementations. Conversions can only happen from one of the non-final states Q, W_1, W_2, or P. We will only consider transitions that do not move upwards in the state transition graph, since upward transitions slow down commitment. The start states Q are equivalent, so transitions $Q \rightarrow W_2$ and $Q \rightarrow W_3$ are trivial. The prepared state P can move to either commit state, since they are equivalent. W_3 can only adapt to W_2, since the non-blocking property requires that W_3 not be adjacent to a commit state, and all other transitions are upward. Note that this transition does not save an entire round of messages, because W_2 is still one round away from commit. However, the coordinator can overlap the conversion request $W_3 \rightarrow W_2$ with the first round of replies from the slaves. Thus, slaves that are still in Q will move directly to W_2, while slaves that are already in W_3 take an extra transition to W_2. The transitions from W_2 can also go in parallel with a round of commitment. If the coordinator has collected all 'yes' votes it may directly issue the transition $W_2 \rightarrow P$. However, if the coordinator is still waiting for some votes it may issues the transition $W_2 \rightarrow W_3$ in parallel with collecting the rest of the votes. Then, when all of the votes are in and all sites have moved to W_3 the coordinator can issue the $W_3 \rightarrow P$ transition.

We are beginning experiments with a stand-alone implementation of the Atomicity Control module, using this adaptability technique. After we have gained experience from the experiments we will implement commit protocol adaptability in RAID. In the stand-alone system each site keeps separate finite state automata for the state diagrams for 2PC and 3PC. Adaptability transitions are always started by the coordinator, possibly in response to a request from a slave. When an adaptability transition is received by a slave it changes to the new finite state automaton, and changes its state to the new state requested by the

- if any site is in state C, commit
- if any site is in state Q or A, abort
- if any site is in state P, commit
- if all sites are in W_2 or W_3, including the coordinator, abort
- if all sites are in W_2 or W_3, but the master is not available:
  - if some site is in W_3 and no other partition can be active, abort
  - if no W_3 or some other partition may be active, block

Figure 12: Centralized termination protocol for combined two-phase and three-phase commitment.

coordinator. Messages between pairs of sites are ordered by sequence numbers, and each transition, including adaptability transitions, has a separate message identifier.

The termination protocol is similar to the normal three-phase termination protocol, except that the non-blocking rule can only be applied in a partition if at least one site in W_3 is present, thus guaranteeing that no other site has committed by the one step rule. The complete protocol is given in figure 12. Notice that the three-phase commit protocol is an advantage in the case where some failures will consist of only site failures.

This conversion approach can be generalized to adaptability between centralized and decentralized commit protocols. Let W_C and W_D be the wait states for the centralized and decentralized protocols, respectively. To convert from two-phase centralized to two-phase decentralized, the coordinator sends a $W_C \rightarrow W_D$ transition to all slaves. Each slave then sends its votes to all other sites, which then run the usual decentralized protocol starting from W_D. If the coordinator has already received some votes before initiating the conversion, it can include the list of sites that have already voted in the conversion request. These sites do not have to repeat their votes to all other sites. Note that the one step rule keeps the coordinator from committing until after all slaves have acknowledged the transition to W_D. The conversion from decentralized to centralized works in much the same manner. The primary difficulty is in ensuring that only one slave attempts to become coordinator, which can be solved with an election algorithm [Gar82].

Commitment is different from many of the other protocols used in distributed systems in that each transaction can run using a different commit method. If the time to complete transaction commitment is small, this characteristic can be used to convert between commit algorithms by just using the new protocol for new commit instances. If it is important for the

entire system to convert to the new protocol at the same time one of the methods mentioned above can be used to convert in-progress commit protocols. Spatial conversion is also possible, in which the commit protocol can be varied for particular data items. Data items are tagged with a "number of phases" indicator. Each transaction records the maximum of the number of phases required by the data items it accesses, and uses the corresponding commit protocol. This is more useful than allowing each transaction to choose its own commit protocol, since it provides the ability to tailor the availability characteristics of the data items to their failure patterns. Data items requiring higher availability ask for an additional phase of commitment. If, instead, each transaction chose its own commit protocol, the blocking status of the data item would be determined by the mix of transactions that happened to be accessing it at a given time.

4.5 Inter-server Communication

The RAID communication system supports adaptability through a high-level, location-independent message system that maintains a clearly defined interface between the servers, and through a naming system that includes alerter messages when servers fail, recover, or relocate. RAID communication is layered on LUDP, which is a datagram facility that we have implemented on top of UDP/IP to support arbitrarily large messages. Above LUDP are the lower-level RAID communications functions, including communication with the oracle for naming services and basic location-independent inter-server communication. Finally, the RAID layer provides transaction-oriented services such as "send to all Atomicity Controllers". The layers use an integrated memory management mechanism to avoid buffer copying, with each layer processing the header that pertains to it, and advancing a pointer to the next header. However, individual layers are functionally isolated from other layers, and can be replaced independently. Furthermore, the high-level RAID interface is designed to support vertical replacement to take advantage of more efficient transport mechanisms in the lower level. For instance, we are changing the implementation of the "send to all Atomicity Controllers" primitive to make use of a kernel-level multicast mechanism that we have implemented [BMRS89].

The RAID oracle is a server process listening on a well-known port for requests from other servers. The two major functions it provides are *lookup* and *registration*. The oracle maintains for each server a *notifier list* of other servers that wish to know if its address changes. Notifier support makes the oracle a powerful adaptability tool, since it can be used to automatically inform all other servers when a server relocates or changes status.

The high-level communication services define the interface between servers. Servers only communicate through these services, so new implementations of servers can be substituted freely as long as they use the same communication routines. Thus, the communication system provides for adaptability by separating the function of a server from its interface, in the same way that specification and implementation of abstract data types are separate.

4.6 Merged Server Configurations

The process structure of RAID is adaptable in the sense that RAID servers can be grouped into processes in many different ways [KLB89]. Server-based systems suffer from performance problems because communication between the separate address spaces becomes a bottleneck. In RAID, merged servers communicate through shared memory in an order of magnitude less time than servers in separate processes. Each merged server is composed of a main loop that receives all messages and dispatches them to the correct internal server, which processes the message as usual. After processing the message the internal server returns control to the main loop which reads the next message. Messages between two servers in the same process are queued on an internal message queue. When the main loop checks for available messages, it first dispatches internal messages before blocking to wait for external messages.

The merged server implementation does not change the design of RAID. Each subsystem is structured as a server, communicating only via messages. Maintaining this design enhances the merged server approach because the servers do not depend on hidden side effects. Thus, the servers can be linked together in any combination safely.

RAID can run in different configurations depending on available hardware or on external conditions. RAID transaction management is performed by the Atomicity Controller, Concurrency Controller, Access Manager, and Replication Controller. These four servers are usually merged into a single Transaction Manager process for performance reasons. In addition, each user has her own User Interface and Action Driver servers, usually merged into a single user process. However, on a multiprocessor a RAID site might separate transaction management into two separate processes. One process could contain the Atomicity, Concurrency, and Replication Controllers, while a second could contain the Access Manager. In this way transaction processing could proceed in parallel on separate processors. If a new processor becomes available the Replication Controller could be relocated to an external process (see section 4.7). Alternately, if enough users invoke User Interface/Action Driver processes the Access Manager could be relocated into the Transaction Manager process to free a processor for a User Interface/Action Driver server. Furthermore, when a new implementation of a server is being debugged it can be run as a separate process to increase fault isolation. Once it is more stable, it can be merged with other servers to increase performance. The strict communication interface coupled with merged servers provides RAID with a technique for adapting to many different environments.

There are several alternative methods for improving communication performance without sacrificing the adaptability of the server design. The crucial problem is reducing latency, thus reducing commit time which decreases the probability of abort and improves transaction throughput. We studied alternatives, including using a special-purpose local IPC mechanism [BMRS89], using a language such as Objective C that uses the same syntax for local and remote object references, and implementing the servers as separate lightweight processes within the same address space [KLB89]. We felt that the best solution would be based on

an object-oriented language with the same syntax for local and remote object references and with explicit support for asynchronous execution within a process. We had no such language available, though, and implementing such a language would be too large a task. Our current approach has many of the advantages of the language-based approach, although the programmer must enforce the object properties without help from a compiler.

4.7 Server Relocation

Another aspect of RAID adaptability is the ability to move servers to new locations, using process migration techniques [PM83]. Reliability is enhanced because servers or entire virtual sites can be moved from hosts before upcoming failures (e.g., periodic maintenance, partial failure that will eventually cause crash). For instance, a site might move to a host more isolated from other RAID sites to decrease the chance of a single failure wiping out the entire system. Relocation for performance generally is in response to changing load conditions over relatively long periods of time. Alternately, the site may move closer to active terminals or to other RAID sites to decrease communications costs. We are in the process of implementing server relocation.

The problem of server relocation is to transfer the data, executable code, and environment of the server to a new location. The data may be substantial, and may require conversion between different data representations. The environment includes system data structures such as information on open files. Environmental state can be particularly hard to transfer because of differences between the hosts. For instance, they may have different naming schemes.

The RAID approach to relocation is based on the server recovery mechanism (see section 4.3). Relocation is planned by simulating a failure of the server on one host, and recovering it on a different host. The RAID server recovery method is responsible for starting the Unix process on the designated host, preparing the system data structures by issuing system calls to open files and initialize network connections, and establishing the current server data structures.

During server recovery other servers detect the failure through timeouts, and are notified of the new address from the oracle. An improvement is to arrange for messages to be automatically delivered to the new recipient during the relocation. This change does not affect correctness, but decreases the cost of relocation incurred by other servers. We have studied four approaches:

- Leave a stub server at the old address to forward messages until the new address has been distributed to all servers.

- Have the sender check the address at the oracle before deciding that a server has failed. If the oracle reports a new address, the sender can try that address before beginning failure processing.

- Use a completely location-independent transport system. For instance, ethernet multicast addresses could be used.

- Actively inform all other servers of an impending relocation before it takes place. A stub server at the new location enqueues messages during relocation.

In RAID we use a combination approach in which a stub version of the new server is instantiated and registered with the oracle immediately, and the sender checks the address with the oracle before declaring a timeout. These two modifications make it likely that in the absence of failures the sender will discover the relocation before detecting the failure.

We studied two alternative methods for relocating servers: general process relocation techniques, and having the servers provide procedures for copying their data structures to a new instantiation. General process relocation techniques copy the entire address space from the old server to the new server. In strongly typed systems the data can be converted to a new representation if type information is available at runtime. Otherwise, relocation is possible only if the two systems use the same data representation. Kernel data structures are more difficult to copy, since duplicating kernel state depends on understanding differences in semantics between the two systems. If the systems are completely homogenous the data can be reconstructed on the destination. Otherwise, it maybe necessary to maintain a dummy process on the original host to process certain types of kernel requests that cannot be successfully replicated on the destination. For our purposes this is unsatisfactory since impending failure of the original host is a likely cause for relocation. Having the servers provide special-purpose routines for copying their data structures and for establishing kernel state has the advantage that only the necessary user data structures are copied, and that type information is available for the user data structures so it can be copied to heterogeneous environments. Also, differences between the sites may be taken into account in establishing kernel data structures. The disadvantages are that the server programmer must provide not only code for maintaining his data structures during normal processing and rebuilding them during failure, but also separate code for copying the data structures during relocation. Furthermore, the server programmer has the responsibility of handling failures during the copying. On balance, then, using the server recovery methods for relocation has the benefits of handling failures during relocation easily and of requiring less programmer effort, at some cost in performance during relocation.

5 Conclusions and Further Work

Results. We believe that adaptability is an achievable goal. Systems designed from the beginning to be responsive to change will be suitable for a wider range of problems, and may have better performance and reliability than systems that are not designed for adaptability. However, adaptability is a difficult problem, especially when the system is considered as a

whole, rather than in parts. Research is needed to develop techniques for adaptability, and to understand its impact on performance and reliability.

In this paper we have developed concepts for modeling adaptability for a transaction system. The contributions of the paper are a model of an adaptable subsystem and several methods for adapting between different algorithms for one of these subsystems while the system is running. We also discussed implementation techniques that support adaptability, based on providing an implementation framework within which our adaptability techniques can be applied. Our methods generalize previous approaches to adaptability.

Further Work. One of the difficulties with adaptability techniques is that the advantages of converting to a better algorithm for a sequencer may be dominated by the cost of the conversion. We are currently developing a model of the costs and benefits of adaptability to determine in which situations the benefits outweigh the costs. Some of the factors that must be considered are:

- Costs of Adaptability
  - expense of conversion protocol
  - aborted transactions during conversion
  - decreased concurrency during conversion
- Benefits of Adaptability
  - increased availability because more suitable recovery algorithms are running
  - improved overall system performance
  - flexibility to meet specific performance goals (e.g. maximize throughput)
  - fewer transactions will be aborted after conversion

The experimental work will be used to validate this analytical work, and to provide values for various parameters of the model.

There are numerous choices of algorithms for concurrency control [BG81], network partition management [DGS85], transaction commit/termination [SS83], database recovery [Koh81], etc. It has been found that certain algorithms for each of the above subsystems cooperate well to reduce bookkeeping and to increase the efficiency of the implementation [Bha84]. For example, optimistic concurrency control methods work well with optimistic network partition treatment, log based database recovery mechanisms, and integrity checking systems in a distributed environment. We will experiment with various combinations of algorithms to understand their interactions.

We expect this research to lead towards necessary and sufficient conditions for adaptability and reconfigurability of a complete transaction system. We have established general methods for adapting a transaction system. These methods have been successfully applied to concurrency and atomicity control, and hold promise for many other subsystems.

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Adison-Wesley, Reading, MA, 1974.

[Ale86] Nikitas A. Alexandridis. Adaptable software and hardware: problems and solutions. *IEEE Computer*, 19(2), February 1986.

[Avi76] A. Avizienis. Fault-tolerant systems. *IEEE Transactions on Computers*, C-25(12):1304–1312, December 1976.

[BB89] Bharat Bhargava and Shirley Browne. *Adaptability to Failures using Dynamic Quorum Assignments.* Technical Report CSD-TR-886, Purdue University, June 1989.

[BG81] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *Computing Surveys*, 13(2):185–221, 1981.

[BGS86] Daniel Barbara, Hector Garcia-Molina, and Annemarie Spauster. Polices for dynamic vote reassignment. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986.

[Bha84] Bharat Bhargava. Performance evaluation of reliability control algorithms for distributed database systems. *Journal of Systems and Software*, 3:239–264, July 1984.

[Bha87] Bharat Bhargava. Transaction processing and consistency control of replicated copies during failures. *Journal of Management Information Systems*, 4(2):93–112, October 1987.

[BHJ*86] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-12(12), December 1986.

[BM82] Jon Louis Bentley and Catherine McGeoch. Worst-case analysis of self-organizing sequential search heuristics. In *Proceedings of 20th Allerton Conference on Communication, Control, and Computing*, pages 452–461, University of Illinois, Urbana-Champaign, October 1982.

[BM84] D. Z. Badal and W. McElyea. A robust adaptive concurrency control for distributed databases. In *Proceedings of IEEE INFOCOM*, pages 382–391, 1984.

[BMRS89] Bharat Bhargava, Enrique Mafla, John Riedl, and Bradley Sauder. Implementation and measurements of an efficient communication facility for distributed database systems. In *Proceedings of the 5th IEEE Data Engineering Conference*, Los Angeles, CA, February 1989.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BNS88] Bharat Bhargava, Paul Noll, and Donna Sabo. An experimental analysis of replicated copy control during site failure and recovery. In *Proc. of the 4th IEEE Data Engineering Conference*, pages 82–91, Los Angeles, CA, February 1988.

[BR88] Bharat Bhargava and John Riedl. A model for adaptable systems for transaction processing. In *Proceedings of the 4th IEEE Data Engineering Conference*, pages 40–50, Los Angeles, CA, February 1988.

[BR89] Bharat Bhargava and John Riedl. The RAID distributed database system. *IEEE Transactions on Software Engineering*, June 1989.

[BRW87] Bharat Bhargava, John Riedl, and Detlef Weber. *An Expert System to Control an Adaptable Distributed Database System.* Technical Report CSD-TR-693, Purdue University, May 1987.

[CZ85] David. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[DGS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3), September 1985.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[EH88] Ahmed K. Elmagarmid and Abdelsalam Helal. Supporting updates in heterogeneous distributed database systems. In *Proc. of the 4th IEEE Data Engineering Conference*, pages 564–569, Los Angeles, CA, February 1988.

[Gar82] Hector Garcia-Molina. Elections in a distributed computing system. *ACM Transactions on Computer Systems*, C-31(1):48–59, January 1982.

[Gog86] Joseph A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2), February 1986.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations. In *Proc of the VLDB Conference*, Cannes, France, September 1981.

[Her87] Maurice Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2):170–194, June 1987.

[KK86] S. Kartashev and S. Kartashev. Guest editor's introduction: Design for adaptability. *IEEE Computer*, 9–15, February 1986.

[KLB89] Charles Koelbel, Fady Lamaa, and Bharat Bhargava. Efficient implementation of modularity in RAID. In *Proceedings of the USENIX Workshop on Experiences with Building Distributed (and Multiprocessor) Systems*, Ft. Lauderdale FL, October 1989. To appear.

[Koh81] W. H. Kohler. A survey of techiques for synchronization and recovery in decentralized computer systems. *AMC Computing Surveys*, 13(2):149–183, June 1981.

[KR81] H.T. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.

[Lau82] G. Lausen. Concurrency control in database systems: a step towards the integration of optimistic methods and locking. In *Proceedings of the ACM Annual Conference*, pages 64–68, October 1982.

[Pap79] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[PM83] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, 1983.

[PW85] Gerald J. Popek and Bruce J. Walker. *The LOCUS Distributed System Architecture*. The MIT Press, 1985.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[Ras86] Richard F. Rashid. Threads of a new system. *Unix Review*, 4(8):37–49, August 1986.

[Ree83] D.P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[Ske82] D. Skeen. Nonblocking commit protocols. In *Proc of the ACM SIGMOD Conference on Management of Data*, pages 133–147, Orlando, Florida, June 1982.

[SL86] Amit P. Sheth and Ming T. Liu. Integrating locking and optimistic concurrency control in distributed database systems. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986.

[SS83] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, SE-9(3), May 1983.

[Weg84] P. Wegner. Capital-intensive software technology. *IEEE Sofware*, 1(3):7–45, July 1984.