A Systematic Approach to System State Restoration during Storage Controller Micro-Recovery

Sangeetha Seshadri* Lawrence Chiu† Ling Liu*

*Georgia Institute of Technology
801 Atlantic Drive GA-30332
{sangeeta,lingliu}@cc.gatech.edu

†IBM Almaden Research Center
650 Harry Road CA-95120
{lchiu}@us.ibm.com

Abstract

Micro-recovery, or failure recovery at a fine granularity, is a promising approach to improve the recovery time of software for modern storage systems. Instead of stalling the whole system during failure recovery, micro-recovery can facilitate recovery by a single thread while the system continues to run. A key challenge in performing micro-recovery is to be able to perform efficient and effective state restoration while accounting for dynamic dependencies between multiple threads in a highly concurrent environment. We present Log(Lock), a practical and flexible architecture for performing state restoration without re-architecting legacy code. We formally model thread dependencies based on accesses to both shared state and resources. The Log(Lock) execution model tracks dependencies at runtime and captures the failure context through the restoration level. We develop restoration protocols based on recovery points and restoration levels that identify when micro-recovery is possible and the recovery actions that need to be performed for a given failure context. We have implemented Log(Lock) in a real enterprise storage controller. Our experimental evaluation shows that Log(Lock)-enabled micro-recovery is efficient. It imposes < 10% overhead on normal performance and <35% overhead during actual recovery. However, the 35% performance overhead observed during recovery lasts only six seconds and replaces the four seconds of downtime that would result from a system restart.

1 Introduction

Enterprise storage systems serve as repositories for huge volumes of critical data and information. Unavailability of these systems results in losses amounting to millions of dollars per hour [1], bringing organizations to a grinding halt.

Most existing work in the domain of storage system availability addresses failures of the storage media (such as disks) and recoverability from these failures [2, 3, 4]. However, failures at the firmware layer that result in service loss remain largely unaddressed. At the same time, the software at the firmware layer of a storage system has evolved tremendously in terms of functionality. Modern storage controllers are highly concurrent embedded systems with millions of lines of code [5, 6]. As a result of this complexity, recovering from controller failures is both difficult and expensive.

While system availability requirements are constantly being driven higher, failure recovery time is increasing due to increasing system size, higher performance expectations, virtualization and consolidation. Since software failure recovery is often performed through system-wide recovery, the recovery process itself does not scale with system size [6, 7, 8].

How can failure recovery be made scalable? Partitioning the system into smaller components with independent failure modes can reduce recovery time. However, it also increases management costs and decreases flexibility, while still being susceptible to sympathetic failures. On the other hand, refactoring the software into smaller independent components in order to use techniques such as micro-reboots [8] or software rejuvenation [9] requires sizable investments in terms of development and testing costs, unacceptable in the case of legacy systems. An alternative approach is to be able to perform fine-granularity recovery or *micro-recovery*, without re-architecting the system. Under this approach, failure recovery is targeted at a small subset of tasks/threads that need to undergo recovery while the rest of the system continues uninterrupted.

Enabling fine grained recovery can be challenging, especially in legacy systems, and the following issues must be addressed:

- ***Evaluating recovery success:*** What are the failures that can effectively and efficiently be recovered from, using micro-recovery?
- ***Determining recovery actions:*** What are the recovery strategies and recovery actions that must be performed in order to restore the system from an error state to an error-free state?
- ***Identifying dependencies:*** Given the large number of dynamic dependencies possible in a highly concurrent system, what is the scope of fine-granularity recovery?

- ***Enhancing recovery success and efficiency:*** How can we enhance the system to facilitate better recovery success and efficiency?

We address the first three questions, focusing on the challenges of tracking and restoring system state during micro-recovery, evaluating the possibility of recovery success and determining recovery actions.

We make two unique contributions in terms of effective state restoration during micro-recovery. First, by analyzing the system state space, we identify the set of events and system states that affect state restoration from the perspective of micro-recovery. We introduce the concepts of *Restoration levels* and *Recovery points* to capture failure and recovery context and describe how to flexibly evaluate the possibility of recovery success. Based on the restoration levels and recovery points, we introduce *Resource Recovery Protocol (RRP)* and *State Recovery Protocol (SRP)*, which provide rules to guide state restoration.

Our second contribution is Log(Lock), a practical and lightweight architecture to track dependencies and perform state restoration in complex, legacy software systems. Log(Lock) passively logs system state changes to help identify dependencies between multiple threads in a concurrent environment. Utilizing this record of state changes and resource ownership, Log(Lock) provides the developer with the failure context necessary to perform micro-recovery. Recovery points and their associated recovery handlers are specified by the developer. Log(Lock) is responsible for tracking dependencies and computing restoration levels at runtime.

We have implemented and evaluated Log(Lock) in a real enterprise storage controller. Our experimental evaluation shows that Log(Lock)-enabled micro-recovery is both efficient (<10% impact on performance) and effective (reduces a four second downtime to only a 35% performance impact lasting six seconds). In summary, micro-recovery with Log(Lock) presents a promising approach to improving storage software robustness and overall storage system availability.

2 Log(Lock): Design Overview

This section gives an overview of the Log(Lock) system design. We first describe the problem statement that motivates the Log(Lock) design. Using examples, we highlight the unique characteristics of micro-recovery in the context of highly concurrent storage controller software. Then we outline the technical challenges for systematic state restoration during micro-recovery. Finally, we briefly describe the system architecture of Log(Lock).

2.1 Motivation

In this section, we motivate the need for a flexible and lightweight state restoration architecture using a highly concurrent storage controller. The storage controller refers to the firmware that controls the cache and provides advanced functionality such as RAID, I/O routing, synchronization with remote instances and virtualization. In modern enterprise-class storage systems, the storage controller has evolved to become extremely complex with millions of lines of code that is often difficult to test. The controller code typically executes over an N-way processing complex using a large number of short concurrent threads (∼20 million/minute). While the software is designed to extract maximum concurrency and satisfy stringent performance requirements, unlike database transactions it does not adhere to ACID (atomicity, consistency, isolation and durability) properties. This software is representative of a class expected to sustain high throughput and low response times continuously.

With this architecture, when one thread encounters an exception that causes the system to fail, the common way to return the system to an acceptable, functional state is by restarting and reinitializing the entire system. While the system reinitializes and waits for the operations to be redriven by a host, access to the system is lost contributing to downtime. As the system scales to larger number of cores and as the size of the in-memory structures increase, such system-wide recovery will no longer scale [6, 8].

Many software systems, especially legacy systems, do not satisfy the conditions outlined as essential for micro-rebootable software [8]. For instance, even though the storage software may be reasonably modular, component boundaries, if they exist, are loosely defined and components are stateful. Under these circumstances, the scope of a recovery action is not limited to a single component.

The goal of micro-recovery is to perform recovery at a fine granularity such as at the thread-level, while determining the scope of recovery actions dynamically, based on dependencies identified at runtime. The key challenges in performing micro-recovery are identifying dependencies based on failure and recovery context, determining recovery actions and restoring the system to a consistent state after a failure.

2.2 Examples

We present three real examples from a storage controller software. We demonstrate how the semantics and success of fine-grained recovery are determined by failure context and the interactions of threads.

Figure 1 shows two code snippets: R1 increments the number of active users before performing work

```
R1: /* Increment number of Users */
    lockWrite( &numActiveUsersLock);
    numActiveUsers ++;
    unlockWrite( &numActiveUsersLock);
    ...
    ...
    /* Decrement number of Users */
    lockWrite( &numActiveUsersLock);
    numActiveUsers --;
    unlockWrite( &numActiveUsersLock);

R2: /* Start background tasks if no users active */
    lockRead ( &numActiveUsersLock);
    if ( numActiveUsers == 0 ) {
     Start performing background tasks.
    }
    unlockRead( &numActiveUsersLock);
```

Figure 1: Lost Update Conflict

```
R3: /* Get cache track to write to fast-write cache */
    startSCSICmd();
    └→ processRead();
       └→ getCacheTrack();
          └→ getTempResource() {
                ...
                PANIC
```

Figure 2: Resource Ownership Conflict

and in R2, a background job is triggered when there are no active users in the system. When a panic (user defined or system failure/exception) occurs during the execution of region R1, then assume that the micro-recovery strategy is to reattempt execution of region R1. The recovery action must ensure clean relinquishing of resources such as the lock *numActiveUsersLock*. It is also important to ensure that the system state is consistent since corruption of the counter can either cause the background jobs to never be triggered or to be triggered in the presence of active users. In Example-1, while it is permissible for other threads to read the value of the *numActiveUsers* count at anytime provided the *numActiveUsersLock* has been released, the system must ensure that if and only if a thread fails after performing an increment operation on the count, a decrement operation is performed during recovery. On the other hand, if the failure was caused during the execution of region R2, an idempotent background task that is not critical, the recovery strategy may be to just abort the current execution of the background task. However, recovery must ensure that the lock *numActiveUsersLock* has been released.

Figure 2 shows the processing of a write command. In the event of encountering a failure, state restoration must ensure that temporary resources obtained from a shared pool are freed correctly in order to avoid resource leaks or starvation. It may

```
R4: /* Update Metadata Location */
    lockWrite( &MetadataLocationLock);
    MetadataLocation = XX;
    unlockWrite( &MetadataLocationLock);
     ...
```

Figure 3: Dirty Read Conflict

also require that certain cache tracks are checked for consistency, depending upon the point of failure. However, for a resource such as a buffer or empty cache track obtained from a shared pool, restoring the previous contents is not necessary.

Figure 3 shows a thread that updates a global variable indicating the metadata location, such as for checkpoint activity. In the event of a failure caused due to a failed location, the thread may have the opportunity to modify the location without notifying other threads or causing inconsistency, provided no other thread has already consumed the value. However if that is the case, the system may have to resort to recovery at a higher level.

These examples highlight the fact that consistency requirements for state restoration vary with failure context. For example, in the case of a counter generating unique numbers, the only requirement may be that modifications are monotonous. For a shared resource, the state remains consistent as long as there are no resource leaks that could eventually lead to starvation and system unavailability. Unlike a transactional system, where similar problems are addressed, the semantics of the state and failure may render certain types of conflicts irrelevant from the perspective of system recovery. This emphasizes the need for a flexible state restoration architecture that is also lightweight and efficient, thereby allowing the system to sustain high performance.

2.3 Failure Model

Our work is targeted at transient failures in the system, especially failures where the developer now uses system restart as a method to take the system from an unknown or faulty state to a known state. A number of such failure scenarios occur in storage controller software and may apply equally well to other software systems. We present some examples from our analysis of storage controller failures. Bad input from administrator or user, insufficient error handling, deadlocks, a faulty communication channel, unhandled race conditions, boundary conditions, and timeouts are some examples of such failures seen in storage controllers. The system restart mechanism is used often because the system has insufficient information, for example, when reacting to an asynchronous event or when dealing with an unknown state or receiving an unexpected stimulus.

For example, consider a failure scenario where a write operation to disk fails because a driver from a third party vendor returns an unidentified error code due to a bug in its code. In this case, since writes are buffered in a fast write cache and the actual write to disk is performed asynchronously, dropping the request is not an option. Another example is a configuration issue that appeared early in the installation process that may have been fixed by trying various combinations of actions that were not correctly undone. As a result the system finds itself in an unknown state that manifests as a failure after some period of normal operation. Such errors are difficult to trace, and although transient may continue to appear every so often.

Some transient failures can be fixed through appropriate recovery actions that may range from dropping the current request to retrying the operation or performing a set of actions that take the system to a known consistent state. Some other examples of such transient faults that occur in storage controller code are: (1) An unsolicited response from an adapter - An adapter (a hardware component not controlled by our microcode) sends a response to a message which we did not send - or do not remember sending; (2) Incorrect Linear Redundancy Code (LRC): A control block has the wrong LRC check bytes, for instance, due to an undetected memory error; (3) Queue full condition: An adapter refuses to accept more work due to a queue full condition. In addition, there are other error scenarios such as violation of service level agreements. The 'time-out' conditions are also common in large scale embedded storage systems. While the legacy system grows along multiple dimensions, the growth is not proportional along all dimensions. As a result hard-coded constant timeout values distributed in the code base often create unexpected artificial violations. For a more detailed classification of software failures, please refer to [6].

2.4 Technical Challenges

With software recovery, state restoration actions depend on the actions of the failed thread and its interactions with state and shared resources.

Threads in the system interact in two fundamental ways: (1) reading/writing shared data and (2) acquiring/releasing resources from/to a common pool. Threads also interact with the outside world through actions such as positioning a disk head or sending a response to an I/O. Often these actions cannot be rolled back and are referred to as *outside world processes (OWP)* [10]. In such a system, state restoration and micro-recovery must consider the sequence and interleaving of the actions of concurrent threads that gives rise to the following conflicts:

- **Dirty Reads (Write-Read Conflict):** Data written by the failed thread has already been consumed by another thread.
- **Lost Updates (Write-Write Conflict):** Rolling back the failed thread may cause the updates of other threads to be overwritten or lost.
- **Unrepeatable Reads (Read-Write Conflict):** The value of the shared state variable required by the failed thread has already been overwritten.
- **Resource Ownership :** The failed thread may continue to be in the possession of resources from a shared pool or may be holding a lock resulting in resource leaks or starvation issues.

The above taxonomy is derived from that used to describe concurrency control concepts in transaction processing systems [11]. For a given failure, the set of recovery actions that need to be performed to return the system to a consistent state may vary depending upon the failure and the occurrence of one or more of the above conflicts. Note that for application state, the intention is not to deterministically replay the events before the failure, or recover the application state to exactly as it was at the instant of failure. Rather, the goal is to restore the system to an error-free state. In fact, the recovery strategy may itself explicitly rely on non-determinism to remove transient failures. For example, Rx [12] demonstrates an interesting approach to recovery by retrying operations in a modified environment using checkpointed system states for rollbacks.

Checkpointing for fault-tolerance is a well known technique [10, 12, 13, 14] that has also been applied to deterministic replay for software debugging [15, 16, 17]. However, checkpointing techniques are mostly targeted at long-running applications [10] such as scientific workloads [13], or applications where the system can tolerate the overhead imposed by checkpointing [12, 14]. A number of unique challenges in the case of storage controller software make checkpointing infeasible: Unlike long-running applications, storage controllers have a high rate of short ($< 500\mu$secs) concurrent threads and are designed to support extremely high throughput and low response times. Given the highly concurrent nature of controllers, both quiescing the system in order to take the checkpoint, as well as logging the tasks in order to re-execute work beyond the checkpoint is expensive in terms of time and space - especially since system state includes large amounts of metadata and cached data. Next, communication with OWPs such as hosts and media cannot be rolled back and hence invalidates checkpoints. Finally, due to the complexity of the code, not all failures will be amenable to micro-recovery, making checkpointing

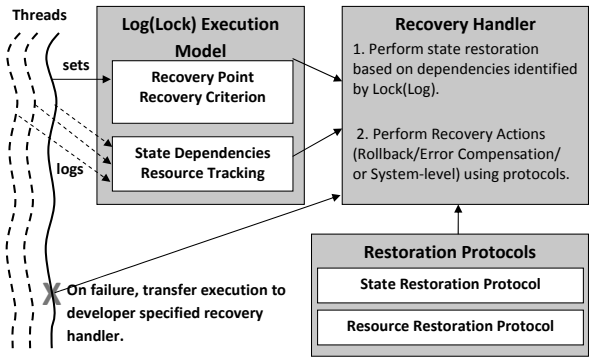

Figure 4: Log(Lock) Architecture Overview

too heavy weight.

State restoration and conflict serialization is also of interest to transactional systems [18]. Transactional databases use schemes like strict 2-phase locking (2PL) to guarantee conflict serializability [19]. However, such techniques can increase the length of critical sections (i.e. durations of locks) and are inefficient for the highly-concurrent storage controller environment. Moreover, we show in Section 2.2 that, recovery actions are determined based on both the context and semantics of failure and a "one size fits all" serializability, while simplifying recovery procedures, can constrain the recovery process.

2.5 System Architecture

The Log(Lock) architecture provides support for state restoration during micro-recovery. To achieve this goal, Log(Lock) tracks resources and state dependencies relevant to a thread that has incorporated recovery handlers for micro-recovery.

Figure 4 presents an overview of our system architecture and describes the roles played by the Log(Lock) execution model and restoration protocols. The figure shows a system with concurrently executing threads where the thread depicted by a solid line incorporates micro-recovery mechanisms. In order to facilitate micro-recovery, the thread sets recovery points during execution, where each recovery point is associated with a recovery criterion. The recovery criterion specifies the conditions that must be satisfied by the failure context in order to use the recovery point as a starting point for recovery. Using the Log(Lock) architecture, the thread (depicted by a solid line) enabled with micro-recovery mechanisms indicates state and resources that are relevant to recovery. Log(Lock) then begins logging all relevant changes and dependencies, based on the actions of both this thread and other concurrent threads (depicted by dotted lines).

In the event of a failure, control transfers to a developer specified recovery handler. The handler performs state restoration actions by utilizing the resource tracking and state dependency information provided by the Log(Lock) execution model, in consultation with the restoration protocols. It also decides on an appropriate recovery strategy such as rollback, error compensation or system-level recovery. The implementation of the Log(Lock) dependency tracking component must ensure efficiency during normal operation while the recovery protocols ensure consistency of state restoration during failure recovery. Below, we summarize the four primary design objectives of Log(Lock):

- **Incremental:** Allow micro-recovery to be applied incrementally to handle failures depending upon effectiveness of a fine-grained approach.
- **Lightweight and Non-intrusive:** Minimize impact on system performance and modifications to legacy software functional architecture.
- **Dynamic:** Handle dynamic dependencies.
- **Flexible:** Allow application developers the flexibility to treat different failures differently without enforcing a "one size fits all" consistency requirement, allowing a larger number of failures to be handled correctly at a fine-granularity.

In the next two sections, we first describe the concepts of 'restoration levels' and 'recovery points' and present the restoration protocols. Then, we present the Log(Lock) execution model and illustrate application of the protocols through example scenarios.

3 State Space Exploration

In this section, we model failure scenarios and recovery contexts using a state space analysis approach. Our approach is based on the intuition that in a concurrent system, global state and shared resources are often protected by locks or similar primitives.

This section is divided into two parts. In the first part, we model system events, state transitions and interleaving of concurrent threads and demonstrate the discrete state space and recovery scenarios. We introduce the concepts of *Restoration Level* and *Recovery Criterion*, that help match a failure context to a recovery strategy. In the second part, we systematically identify the set of recovery strategies that can be applied to each failure scenario and present two protocols for state restoration. The **Resource Recovery Protocol (RRP)** defines the steps to handle resource ownership conditions and the **State Recovery Protocol (SRP)** sets forth the rules to perform state restoration.

3.1 Modeling Thread Dependencies

Let $\mathcal{T} = \{T_i | 1 \leq i \leq n\}$ define a system with n concurrent threads. Let $\mathcal{X}_i(t)$ denote the sequence of

Table 1: Valid States for Thread T_i

Notation	Description
T_iS	T_i initial state
T_iR	T_i holds a read lock
T_iW	T_i holds an exclusive write lock
T_iU	T_i has released the lock
T_iF	T_i is in failed state
T_iA	T_i acquired a resource
T_iRe	T_i released a resource
T_iE	T_i performed an externally visible action

states of thread T_i up to time t. The schedule $\mathcal{S}(t)$ at time t is the interleaving of the sequence of actions in $\mathcal{X}_i(t)$ for each thread T_i. Let v denote a globally shared structure protected by a lock. Table 1 shows the list of valid states for a thread.

The system implements micro-recovery at a thread granularity. Any failure that cannot be handled by micro-recovery is resolved using a system-level recovery mechanism (e.g. software reboots).

The state space for system execution consists of all legitimate schedules $\mathcal{S}(t)$. System states that represent the failed state of one of the executing threads are relevant from the perspective of micro-recovery. To simplify the subsequent discussion, we apply the following rules to reduce the state space:

- We consider the interactions between only two threads T_1 and T_2.
- We only consider system states where the last state of thread T_1 is T_1F.
- Only T_1 encounters a failure. Failures of thread T_2 are symmetric and can be treated similarly.
- Read or write actions performed by T_2 before any such actions by T_1 are ignored.
- We assume that the system can recover from only a single failure. Failure during recovery results in system-level failure recovery.
- The externally visible action is equivalent to a 'commit action' that cannot be rolled back.

Occurrences of the following patterns in the schedule $\mathcal{S}(t)$ are of interest and relevant to the selection of a recovery strategy by thread T_1. Let $\rightarrow$ denote the "happened before" relation [20].

- **Dirty Read (DR):** $T_1W \rightarrow T_2R \rightarrow T_1F$.
- **Lost Update (LU):** $T_1W \rightarrow T_2W \rightarrow T_1F$.
- **Unrepeatable Read (UR):** $T_1R \rightarrow T_2W \rightarrow T_1F$.
- **Residual Resources (RR):** $(T_1R \rightarrow T_1F) \wedge (T_1U \nrightarrow T_1F)$ or $(T_1W \rightarrow T_1F) \wedge (T_1U \nrightarrow T_1F)$ or $(T_1A \rightarrow T_1F) \wedge (T_1Re \nrightarrow T_1F)$.
- **Committed Dependency (CD):** $T_1W \rightarrow T_2R \rightarrow T_2E \rightarrow T_1F$ or $T_1W \rightarrow T_2W \rightarrow T_2E \rightarrow T_1F$ or $T_1R \rightarrow T_2W \rightarrow T_2E \rightarrow T_1F$.

To determine the right strategy for recovery, it is important to determine which of the above conflicts have occurred and are relevant to recovery.

Restoration Level: The restoration level $\mathcal{R}_i(t)$ of a thread T_i at instant t, is a 5-tuple $\langle DR, LU, UR, RR, CD \rangle$ indicating the occurrence of dirty reads, lost updates, unrepeatable reads, residual resources and committed dependencies in $\mathcal{S}(t)$.

Recovery Point: A recovery point p_i in thread T_i represents an execution point to which control is transferred at the end of a recovery procedure. A default recovery point defined for all threads is the initial system state.

Recovery Criterion: Each recovery point p_i is associated with a recovery criterion $\mathcal{C}_i$ which is a 4-tuple $\langle DR, LU, UR, RR \rangle$ that represents the set of criteria for dirty reads, lost updates, unrepeatable reads and residual resources, that the system state should satisfy before recovery can be attempted using p_i. For the default recovery point, all elements of the recovery criterion are defined as "don't care".

CD does not figure in the recovery criterion since this information is used only to choose between alternate recovery strategies in the recovery handler. We discuss the use of CD conditions during recovery in the state recovery protocol in Section 3.2. In our current design, recovery points and their associated recovery handlers are identified by developers and are associated to an execution context. When a thread leaves a context, the associated recovery points go out of scope. Within a single execution context, multiple recovery points may be defined, any of which could potentially be used during recovery. Then the appropriate recovery point for the current failure scenario is chosen by the logic in the recovery handler. In the developer-specified recovery handler, the feasibility and correctness of restoring the failed system state using a recovery point, is determined using the resource and state recovery protocols described next. Once the valid recovery points have been identified from the available choices, the selection of an appropriate recovery point and recovery strategy may be a decision depending upon factors such as the amount of resources available for recovery and the time required to complete recovery.

3.2 Restoration Protocols

We consider the following possible recovery strategies: (1) Rollback; (2) Roll-forward style recovery or error compensation; (3) System-level recovery [21]. Of these the rollback and error compensation strategies may be applied to the failed thread only (single-thread recovery) or to multiple threads including the failed thread (multi-thread recovery). The following

protocols are based on the assumption that committed dependencies cannot be rolled-back.

Resource Recovery Protocol (RRP): System state can be restored to recovery point p_i only if $\mathcal{R}_i(t)$ meets $\mathcal{C}_i$ on the RR criterion. Otherwise, the thread must first attempt to release or acquire resources to meet the criterion.

The state recovery protocol (SRP) specifies the recovery strategies applicable for different failure and recovery contexts. The rationale behind the SRP rules is that an occurrence of DR, LU or UR events imply that an interaction with other concurrent threads in the system have occurred. When the restoration level does not meet the recovery criterion and interactions with other threads have occurred, then single thread recovery is no longer sufficient. Next, the success of multi-thread recovery depends on the occurrence of an externally visible action and whether the dependency has already been committed. Concretely, the rules of state recovery are:

State Recovery Protocol (SRP): 1. To perform single-thread recovery and restore state to recovery point p_i, $\mathcal{R}_i(t)$ should meet $\mathcal{C}_i$ on every element of $\mathcal{C}_i$.
2. If $\mathcal{R}_i(t)$ does not meet $\mathcal{C}_i$ on DR, LU, UR conditions and CD occurs in $\mathcal{S}(t)$, then only error compensation or system-level recovery can be attempted.
3. If $\mathcal{R}_i(t)$ does not meet $\mathcal{C}_i$ on DR, LU, UR conditions and CD has **not** been observed in $\mathcal{S}(t)$, then only multi-thread rollback, error compensation or system-level recovery is possible.

4 Log(Lock) Execution Model

In this section, we present a concrete execution model of Log(Lock), that utilizes the state space analysis presented in the previous section. We show how to decide recovery strategies and how restoration levels can be tracked practically. Although the discussion in this paper focuses on a thread-level recovery granularity, the Log(Lock) architecture can easily be extended to a more coarse granularity of micro-recovery such as at a task or component level.

In a complex legacy system such as a storage controller, not all failures can be handled efficiently through fine-grained recovery - either because the failure and recovery code may be too complex, or system-level recovery may be a more effective recovery technique, or simply because there may be insufficient development and testing resources. Therefore, our approach first involves identifying candidates for fine-grained recovery based on the analysis of failure logs and the software itself. The executing instance of each candidate is known as a **recoverable thread**. Recall that, for each recoverable thread multiple recovery points and associated recovery criterion may be defined. In the event of a failure, control is transferred to the recovery handler (Section 2.5).

4.1 Tracking State Changes

Log(Lock) is based on the intuition that all shared state and resources are protected by locks or similar synchronization primitives. Tracking lock/unlock calls can therefore guide the understanding of system state changes and provide the information required to identify the restoration level at the instant of failure. At the same time, by tracking these calls on resources and applying the resource recovery protocol, we can prevent deadlocks or resource starvation issues. In order to compute restoration levels and perform system state restoration, Log(Lock) maintains the following:

Undo Logs: Undo logs are local logs maintained by each recoverable thread for the following purposes: (1) Track the sequence of state changes within a single thread; (2) Track the creation of recovery points and (3)Track resource ownership. In general, the Undo logs can be used to encode any information required by a thread's recovery handler. In our current implementation, Undo-logging activities and maintenance of the Undo logs are left to the developer.

Change Track Logs: In order to track conflicts between concurrent threads, Log(Lock) maintains Change Track Logs for each lock. The Change Track Log is used to: (1) Track concurrent changes to shared structures and (2) Track commit actions.

Both the Undo Log and Change Track Logs are maintained only in main memory and are verified for integrity using checksums. In our implementation, the change track log is implemented as a hashtable indexed using the pointer to the lock as key. Unlike database logs or checkpoints for state restoration, these logs do not need to be flushed to stable storage. If a failure crashes the system causing it to lose or corrupt the logs, then we must perform a system-level restart to restore the system to a consistent, functional state and no longer require the software's state restoration logs from before the failure.

Log(Lock) provides four basic primitives to a recoverable thread:

- *startTracking(lock)*: Start tracking changes to the structure protected by *lock*.
- *stopTracking(lock)*: Stop tracking changes to the structure protected by *lock*.
- *getRestorationLevel(lock)*: Compute the restoration level for the structure protected by *lock*.
- *getResourceOwnership(lock)*: Get ownership information (including lock ownership) for the

```
/* Recovery Criterion for R1: No residual resources */
        Owner = getResourceOwnership(&numActiveUsersLock);
/* Acquire ownership in write mode for consistent recovery*/
        if( Owner == ReadMode) {
                unlockRead(&numActiveUsersLock);
                lockWrite(&numActiveUsersLock);
        } else if(!Owner)
                lockWrite(&numActiveUsersLock);
        level = getRestorationLevel(&numActiveUsersLock);

        if ( level indicates dirty reads or lost updates ) {
                /* Indicates write completed */
                numActiveUsers -- ;
        } else {
    /* No other operations or write may not have completed */
                Replace old value using the Undo log;
        }
        unlockWrite( &numActiveUsersLock);
/* State restore complete. Jump to new execution point */
Jump to R1;
```

Figure 5: State Restoration Using Log(Lock)

structure protected by *lock*.

All the above primitives are explicitly inserted into the code by the developer. The *startTracking* call is used to trigger change tracking for shared state and resources protected by the *lock* parameter. These accesses are identified by trapping lock/unlock calls. When the recoverable thread determines that the logs for a particular structure are no longer required, it explicitly issues a *stopTracking* call. In the event of a failure, the system transfers control to the designated recovery handler. The recovery handler can utilize the *getRestorationLevel* and *getResourceOwnership* primitives to determine the current restoration level and resource ownership and then invoke recovery procedures appropriately. The restoration level is determined by examining the undo and change track logs.

4.2 Recovery Using Restoration Protocols

The goal of our state restoration approach is to return the system to a correct, functional and known state by performing localized recovery and state restoration actions. The recovery actions are targeted at only a small subset of the threads in the system and a small region of the total system state that has been identified as affected by failure-recovery. Figure 5 shows pseudo code for state restoration using the restoration protocols and the Log(Lock) architecture for the scenario shown in Figure 1. Assume that, the recovery criterion associated with recovery point R1 specifies that resources (*numActiveUsersLock*) acquired after the recovery point should be released and does not care about occurrences of DR, LU or UR events. As shown in the Figure 5, the *getResourceOwnership* primitive is used to determine ownership of the *numActiveUsersLock* resource. Then, if the restoration level indicates that a DR or LU event has occurred, that would imply that the thread has successfully completed incrementing *numActiveUsers* in the first place. Then in order to rollback the failed thread execution correctly to recovery point R1 without losing the work done by other threads, a matching decrement operation would need to be performed. If however the change track logs indicate that no other thread has consumed data written by the failed thread, it could imply that the failed thread either did not complete its increment operation or was the last thread to update the value of *numActiveUsers*. In that case, the recoverable thread could use its undo log to undo its changes, if any. The developer of this recovery handler is expected to have used the Undo log interfaces to store the old value prior to modification. Once state restoration is complete, execution is transferred to recovery point R1.

Similarly, in the case of the example in Figure 2, assume that the recovery criterion only specifies the constraint on releasing the temporary resource acquired after the recovery point. Therefore, the *getResourceOwnership* primitive is used to obtain the current ownership status of the temporary resource. If the resource is held by the thread, in order to rollback to recovery point R3, the resource must be cleanly relinquished. The pseudo code for this example and the next is not shown due to lack of space.

In the case of the failure scenario shown in Figure 3, the recovery criterion for recovery point R4 would be that no resources acquired after the recovery point (such as lock *MetadataLocationLock*) should be held by the thread and that no DR or LU events should have occurred. If the restoration level indicates that no other thread has already consumed this value (i.e., no DR or LU events have occurred), then the changes of the failed thread can be undone safely by replacing with the values in the Undo log. However, if the value is likely to have been consumed by another thread (i.e. DR or LU occurred), then the restoration level does not meet the recovery criterion for R4. So, in accordance with SRP, the error cannot be handled using single-thread recovery. Depending upon the support for multi-thread recovery (provided the CD event has not occurred) recovery may require rollbacks of multiple threads. If however, CD has occurred, then system-level recovery or error-compensation is performed.

4.3 Implementation Details

Undo logs go out of scope i.e., can be purged when a recoverable thread completes execution. Similarly, change track logs for a lock are purged when the recoverable thread issues a *stopTracking* call. How-

ever, unlike undo logs, change track logs cannot be purged immediately since these centralized logs may be shared by multiple recoverable threads. In that case, the log entries corresponding to the purging thread are only marked for purging and are actually purged when the last recoverable thread using the log issues a *stopTracking* call on that lock.

Multi-thread recovery i.e., applying state restoration and recovery to more than one thread, can typically handle more failure scenarios compared to single-thread recovery. However, multi-thread recovery is complex to implement. Moreover, multi-thread recovery may result in a domino effect [22] (also referred to as cascading aborts) potentially resulting in unavailability of resources and unbounded recovery time[6]. A simpler and more effective technique would be to limit recovery to a single thread and ensure recovery success through other mechanisms such as dependency tracking and scheduling. Recovery conscious scheduling [6] describes an approach where dependencies between concurrent threads are identified and dependent threads serialized. This approach can help limit the number of concurrent dependent threads and increase single-thread recovery success.

5 Experiments

We have implemented the Log(Lock) architecture for system state restoration and micro-recovery on an industry standard, high-performance storage controller and applied Log(Lock) to a variety of state and resource locks. In this section, we present our evaluation of Log(Lock) with respect to performance, failure recovery and scalability. We next describe our experimental setup, evaluation metrics, experimentation methodology and results.

We identified state and resource instances that are changed or accessed rapidly through the observation periods, based on instrumenting the system (Table 2). We also identified representative failure scenarios by analyzing bug reports, failure logs and code. Using these scenarios as candidates for micro-recovery and state restoration, we evaluate Log(Lock) efficiency and effectiveness. In summary, our results show that:

- The Log(Lock) architecture imposes negligible overhead and sustains high performance ($< 10\%$ impact) under a variety of workloads, even while tracking rapidly changing state (nearly 15K times/second) for significant durations.
- We observe an extremely high rate of recovery success (>99%), i.e., percentage of time restoration levels meet recovery criterion. This high rate of recovery success makes it evident that micro-recovery with Log(Lock) can be a promising approach to system recovery from transient failures.
- The Log(Lock) approach exhibits significant improvement in availability, replacing a four second downtime without micro-recovery with only a 35% performance impact lasting six seconds with Log(Lock).

5.1 Experimental Setup

We implemented the Log(Lock)-based state restoration architecture in an enterprise-class high performance, highly concurrent embedded storage controller. The system consists of a 4-way processor complex (4 3.00 GHz Xeon 5160 processors with 12 GB memory running IBM MCP Linux) running the controller software over a simulated backend. The controller implements persistent memory (non-volatile storage) for write caching. Simulating the backend allows flexibility in terms of experimenting with different configurations such as read/write latencies and error injection. The back end configuration varied between 50-250 disks of 100GB each with the maximum read and write latencies of the disk set to 20 ms. The memory footprint of our implementation of the Log(Lock) architecture was less than 48KB. The host functionality was performed from a different system (2 1.133 GHz Pentium III processor with 1 GB memory, RHLinux 9) connected to the storage complex through a high-bandwidth (2 GB) fiber channel interconnect.

Our workload was generated using a randomized synthetic workload generator which took the following inputs: read/write ratio, block size and queue depth (i.e. maximum number of outstanding requests from the host). The experiments presented in this paper utilized three distinct read/write ratios: 100% writes, 50%-50% mix of reads and writes and 100% reads. Block size was set to 4 KB and queue depth varied between 16 and 256.

5.2 Metrics

Our experiments evaluate efficiency and effectiveness of the Log(Lock) architecture. Efficiency and effectiveness depend on the following parameters: (1) rate of access to shared state or resources and (2) duration of a recoverable thread. Increasing each of these parameters results in an increase in the log size, logging overhead and the probability of conflicts.

Efficiency refers to the impact of Log(Lock) on system performance. To measure performance, we utilize two metrics: *throughput* (IOs per second or IOps) and *latency* (seconds/IO).

Effectiveness refers to the ability of the state restoration architecture to reduce the recovery time and positively impact the availability of the system.

Concretely, it refers to the probability of recovery success with the Log(Lock) architecture and the impact on system recovery time.

Effectiveness is measured using the following metrics: (1) *recovery success*, i.e. the percentage of time the restoration level meets the recovery criterion for single thread recovery, and (2) *recovery time*, i.e. the time required to restore the system to a consistent state after encountering a failure. Note that in the experiments reported in this paper we focus on single thread recovery while evaluating recovery success. While our Log(Lock) approach can also be applied to multi-thread recovery, as described in Section 4.3, multi-thread recovery can be costly in terms of coding effort, resource consumption and recovery time. Instead, we assume that a technique such as recovery conscious scheduling [6] can help reduce the need for multi-thread recovery and improve the success of single thread recovery.

5.3 Methodology

In order to evaluate Log(Lock), we first identify state and resource instances in the software for tracking. We instrumented the system to identify top locks in terms of access and contention. Table 2 shows the top five locks in terms of number of accesses and contention. The table shows the semantics of the lock (i.e. the state or resource protected), the number of CPU cycles lost to contention, number of occurrences of contention ($>$ 2000 CPU cycles), number of accesses to the lock and the average number of lock acquisitions per IO. Frequently acquired locks are indicative of state that is accessed or modified often. For example, Table 2 shows that the fiber channel lock accessed nearly 10 times per IO is a good candidate for evaluating the efficiency of Log(Lock). Contention, while indicative of longer durations of holding locks, also shows a higher probability of accesses by concurrent threads. As Table 2 shows, the percentage of accesses resulting in lock contention is low as a result of the highly concurrent design of the controller. Thus, for short durations of tracking we expect high recovery success.

To evaluate effectiveness, we first measure the recovery success for the candidates identified from Table 2. We measure recovery success across locks with different rates of access and varying duration of tracking. To evaluate the impact on recovery time, we identify candidates for state restoration based on analysis of the software, failure logs and defects.

We present evaluation of the efficiency of our Log(Lock) architecture as compared to the original system, henceforth referred to as *baseline*. The baseline implementation does not perform state restoration or fine-grained recovery. Instead, it uses a highly efficient system level recovery mechanism that checks all persistent system structures such as non-volatile data in the write cache for consistency, reinitializes software state and redrives lost tasks. Note that no hardware reboot is involved.

Table 2: Lock Access over 75 minutes

Lock	Contention Cycles (Count)	Number of accesses	Locks/IO
Fiber channel	2654991 (578)	137196747	10.34
IO state	219969 (76)	90122610	6.79
Resource	608103 (100)	63482290	4.78
Resource state	124965 (52)	30040757	2.26
Throttle timer	79848 (11)	113316	0.0085

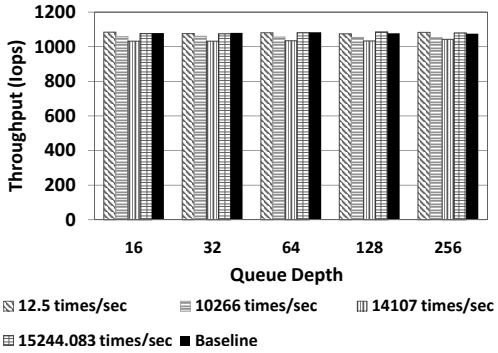

Figure 6: Rate vs Throughput (100% Writes)

An alternative approach to Log(Lock) is to implement schemes such as strict 2-phase locking (2PL), commonly used in transactional systems. Essentially, these protocols require locks to be held for the entire duration of a recoverable thread. However, due to the high degree of concurrency in the system and the implementation of lock timeouts, such a scheme when implemented in our storage controller software caused lock timeouts and failed to bring up the system. Therefore, throughout this evaluation section, we primarily use the baseline system for comparison.

5.4 Efficiency of Log(Lock)

In order to measure efficiency, we compare the performance of the Log(Lock) architecture with the baseline system during failure-free operation.

5.4.1 Effect of Frequency of State Change

As described in Section 5.2, as the rate of accesses to a state variable or resource being tracked increases, the logging overhead increases. The workloads used for this experiment consisted of 100% write IOs and the data is averaged over 10 runs of 10 minutes each. The queue depth is represented on the x-axis. For this experiment, we chose four locks from Table 2, representative of a range of access rates, ranging from 12.5 times/second to 15244 times/second. The

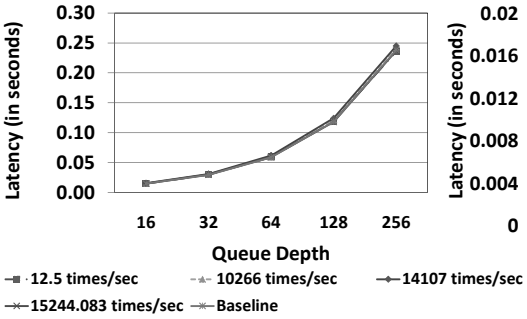

Figure 7: Rate vs Latency (100% Writes)

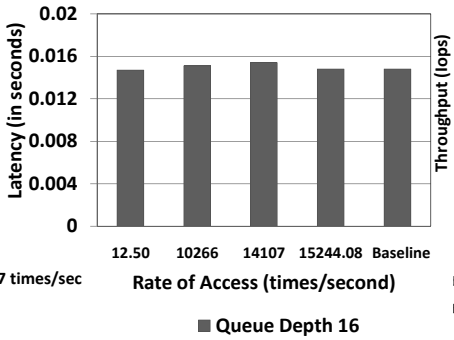

Figure 8: Latency

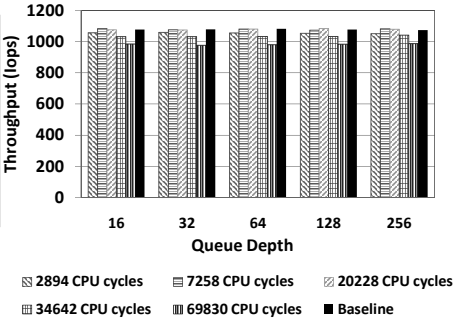

Figure 9: Duration of Tracking vs Throughput (100% Writes)

Table 3: % Duration of Tracking vs Latency

Queue Depth	(Duration of tracking in CPU Cycles)				
	2894	7258	20228	34642	69830
	% Increase in latency over baseline				
16	2.03%	0.68%	0.00%	4.05%	9.46%
32	1.69%	0.34%	0.34%	4.39%	10.47%
64	2.72%	0.34%	0.51%	4.76%	10.71%
128	2.54%	0.85%	0.00%	5.08%	9.32%
256	2.10%	0.00%	0.42%	2.94%	8.82%

Table 4: % Overhead (other workloads)

Queue Depth	Workload 1		Workload 2	
	Through-put	Latency	Through-put	Latency
16	0.43%	0.47%	0.08%	∼0.00%
32	0.25%	∼0.00%	0.78%	0.75%
64	0.24%	0.39%	0.13%	∼0.00%
128	0.29%	0.39%	0.79%	0.75%
256	0.25%	0.00%	0.12%	0.19%

duration of tracking was 2600 CPU cycles on average (and standard deviation 265 CPU cycles).

Figure 6 shows the throughput with varying access rates under different queue depths. The numbers show that even for high access rates, the Log(Lock) approach has negligible impact on performance. The lock with access rate 14107 times/sec (the resource pool lock) was tracked for 2429 CPU cycles and results in a 4.5% drop in throughput. We attribute this to the occurrence of nested lock conditions in that particular code path, causing the system to be sensitive to even the small delay introduced by Log(Lock).

Figure 7 shows the variation of latency with queue depth for different access rates. The curves for the various access rates almost completely overlap showing that across configurations, the impact of Log(Lock) on latency, even for high access rates, is negligible. The observation that the latency increases with queue depth is a queuing effect commonly observed in systems [23] and is independent of Log(Lock). Figure 8 zooms into the points for queue depth 16 to give the reader a closer look at the data. As in the case of throughput, latency increases by ∼4% for the resource pool lock and is attributed to the occurrence of nested lock situations in the code path. The important message from Figures 6 and 7 is that Log(Lock) tracking can sustain high performance even while tracking rapidly modified/accessed state or resources.

5.4.2 Effect of Duration of Tracking

Figures 9 and Table 3 show the variation of system performance with different durations of tracking. The durations were measured in terms of number of CPU cycles between the *startTracking* and *stopTracking* calls, averaged over 10 runs of 10 minutes each. The independent parameter queue depth is shown on the x-axis. The data represents the performance for candidate locks from Table 2 that were tracked for different durations ranging from 2894 CPU cycles to 69830 CPU cycles (IO state for 2894 and 69830 CPU cycles, timer, fiber channel and resource pool for 7258, 20228 and 34642 CPU cycles respectively). The numbers were chosen to be representative of a range of tracking durations. Since no functional code was modified, rather than varying the duration of a single lock, different locks were instrumented to obtain this range. The rate of access of each lock varied as shown in Table 5.

From Figures 9 and Table 3 we observe that, the performance of the system with Log(Lock) is comparable to the baseline system across various queue depths. For the IO state lock (a lock in the IO path), when the duration of tracking was increased from 2894 CPU cycles to 69830 CPU cycles, the throughput dropped by 8.85% and response time increased by 9.76% on average compared to baseline. This drop in performance can be attributed to two factors: (1) occurrence of more conflicts with increase in duration of tracking and (2) increased possibility

of encountering nested lock conditions, which are sensitive to the delay introduced by tracking. In the case of the resource lock, a tracking duration to 34642 CPU cycles resulted in a drop of only 4.3%, which is nearly identical to the performance with a tracking duration of only 2429 CPU cycles, as shown in Section 5.4.1. We conclude that, though the overhead of tracking is a function of both the frequency and duration of tracking, it is more significantly impacted by the semantics of the lock being tracked and the efficiency of the code path involving the lock.

5.4.3 Performance with Other Workloads

Table 4 show the throughput and latency with four other workloads. The figures compare the performance of a system powered by Log(Lock) and the baseline system under varying queue depths for the following workloads: Workload-1 (100% read, disk latency 1ms), and Workload-2 (50% read, disk latency 1ms). Data from tracking the fiber channel lock (15244 times/sec for 20228 CPU cycles each) is shown. Overall, the impact on performance was $<$ 1% in all cases. These results reiterate the observation that Log(Lock) is lightweight and sustains high performance for a range of workloads.

Examining the object code for our implementation showed that in the event of a lock being tracked, fewer than 200 assembly instructions were added to the code path. Assuming one instruction executes per CPU cycle, even at a frequency of 15244 times/second, on a 3.00 GHz processor, this amounts to a time overhead of less than 1% (assuming that the size of the state being saved to undo logs is small). Also, note that storage controller code by itself is aggressively optimized to sustain high throughput, minimize the duration of locks in the I/O path and avoid nesting of locks to a large extent. Unlike checkpoints, which require a large amount of state to be copied to stable storage, our techniques copy small amounts of relevant state and information in memory only. The combination of all these factors results in the Log(Lock) system being able to sustain high performance despite an extremely high frequency of access to shared state and resources. In conclusion, we believe that the scenarios where performance will be impacted by tracking are when there are multiple levels of nesting with frequently accessed locks, increasing sensitivity to tracking delay. However, we expect that these situations are uncommon in well-designed systems.

5.5 Effectiveness of Log(Lock)

The next set of experiments are focused on evaluating the effectiveness of a micro-recovery framework with Log(Lock) in improving system recovery.

5.5.1 Recovery Success

The first metric of effectiveness is recovery success i.e., the percentage of time the restoration level meets the recovery criterion at the end of execution of a recoverable thread. This metric demonstrates the opportunity for micro-recovery in the system and evaluates if the system can effectively utilize Log(Lock)-based state restoration. Table 5 shows the recovery success for locks of varying semantics, rates of access and duration of tracking. The IO state lock was tracked for two types of recoverable threads, for a duration of 2894 CPU cycles in one and 69830 CPU cycles in the other. Hence data for this lock appears twice in Table 5. For each lock, the recovery criterion, the number of tracking threads per second, the rate of access, duration of tracking and recovery success are shown. The restoration level in each case was obtained by calling the *getRestorationLevel* method before *stopTracking*, and recovery success was computed as the percentage of time the restoration level met the recovery criterion. As Table 5 shows, our storage controller exhibits a high rate of recovery success for a range of locks, even with high rates of access. We conclude that, for failures involving the restoration of these instances of state and resources, fine-grained recovery presents an effective recovery strategy.

5.5.2 Recovery Time

To illustrate the impact of Log(Lock)-based micro-recovery on the overall recovery time and availability of the controller software, we injected transient failures that disappeared on retry. The failures required restoration of the IO state to its previous value and a retry of the function. For the Log(Lock) system, the recovery criterion for IO state was set as shown in Table 5. Once the failure was injected, the thread verified if the restoration level at the time of recovery met the recovery criterion, before attempting state restoration and retry. The tracking duration was equivalent to the set up with 69830 CPU cycles.

Figures 10 and 11 show the variation of throughput and latency respectively over time. The points of failure injection are marked in the figures. The throughput and latency shown are for a workload with 100% write IOs, queue depth 64 and disk latency 20 ms. The Log(Lock) architecture is compared to system-level recovery (abbreviated as SLR) in the case of the baseline system. Recall that SLR is implemented entirely in software and involves restarting the controller process and verifying data structures and cache data for consistency before redriving IO transactions. Overall, during failure-free operation, the average throughput and latency respectively with Log(Lock) is $708 IOps$,

Table 5: Recovery Success with the 100% Write Workload

Lock	Recovery Criterion	Tracking Calls (times/sec)	#Access (times/sec)	Duration CPU cycles	Recovery Success
Fiber channel	No Residual Resources	3666	15244	20228	100%
IO state	No DR, LU or UR	2500	10266	2894	99.88%
Resource pool	No Residual Resources	10	14107	34642	100%
Resource state	No Residual Resources	5	6675	4806	100%
Throttle timer	No Residual Resources	10	12.59	7258	100%
IO state	No DR, LU or UR	2444	10045	69830	99.38%

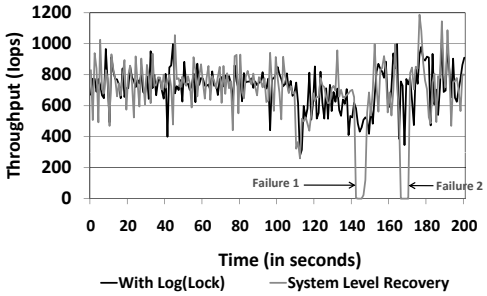

Figure 10: Throughput with Error Injection

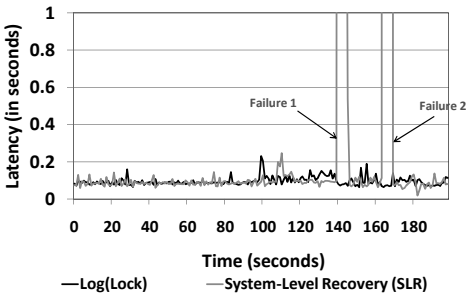

Figure 11: Latency with Error Injection

0.0946 sec/IO and $710 IOps$, 0.0912 sec/IO for the baseline system.

Log(Lock)-enabled micro-recovery imposes a 35% performance overhead lasting six seconds during recovery. However, system-level recovery results in 4 seconds downtime and it takes an additional 2 seconds to begin sustaining high performance. It is important to remember that as the size of the system and in-memory data structures increase, the recovery time for SLR is bound to increase. This, along with the opportunity for micro-recovery illustrated by the high recovery success shown in the previous experiment, further promote the case for micro-recovery in high performance systems like the storage controller.

6 Related Work

Our work is largely inspired by previous work in the area of transactional systems, software fault tolerance and system availability. Hardware redundancy and software redundancy [24], rejuvenation [9] or fault isolation approaches such as isolating VMs from the failure of other VMs [14] are complementary to our techniques and are already deployed in our setups. Since these approaches are targeted at handling failures at a different level they focus on a coarser granularity of recovery compared to our techniques. Failure-oblivious computing [25] introduces a novel method to handle failures - by ignoring them and returning possibly arbitrary values. This technique may be applicable to systems like search engines where a few missing results may go unnoticed, but is not an option in storage controllers where ignoring failures or returning arbitrary values could lead to data corruption.

Application-specific recovery mechanisms such as recovery blocks [22], and exception handling [26] are used in many software systems. Constructs such as try/throw/catch [27] can be used to transfer control to an exception handler and a similar exception model is used by our implementation. However such exception handling constructs alone are insufficient for performing micro-recovery which requires richer failure context information. The goal of the Log(Lock) architecture is to provide this context information and provide the developer with a set of guidelines to decide the precise way in which the system should be restored given the failure context.

Logging of access patterns has been used for deterministic replay [15, 16, 17] during debugging. However, in micro-recovery, there is no requirement to perform deterministic replay. Also, the purpose of logging access patterns in Log(Lock) is to identify recovery dependencies between concurrent threads.

7 Conclusion

We have presented Log(Lock), a practical and flexible architecture for tracking dynamic dependencies and performing state restoration without rearchitecting legacy code. By exploring system state space, we formally model thread dependencies based on both state and shared resources, capturing failure contexts through different 'restoration levels'. We develop recovery strategies in the form of restoration protocols based on recovery points and restoration levels. A comprehensive experimental evalua-

tion shows that Log(Lock)-enabled micro-recovery is both efficient and effective in reducing system recovery time.

Even with retrofittable mechanisms such as micro-recovery, we emphasize that failure recovery should be a design concern. One approach to reducing recovery time would be to design the software using components with independent failure modes (e.g. client-server interactions) or use a state space based approach where transitions to functional states can be identified even from a failure state.

Our effort in designing scalable failure recovery continues along a number of directions. One of our ongoing efforts is to reduce the need for programmer intervention in defining recovery actions. We are also interested in deploying and evaluating Log(Lock) in other high performance systems.

References

[1] D. Scott, "Assessing the costs of application downtime.," *Gartner Group, Stamford, CT*, 1998.

[2] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID hierarchical storage system," in *SOSP*, 1995.

[3] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *SIGMOD Rec.*, vol. 17, no. 3, 1988.

[4] "Cdp buyers guide," *Available at http://www.snia.org/tech activities/dmf/docs/CDP Buyers Guide 20050822.pdf*, 2005.

[5] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "IRON file systems," *SOSP*, 2005.

[6] S. Seshadri, L. Chiu, C. Constantinescu, S. Balachandran, C. Dickey, L. Liu, and P. Muench, "Enhancing storage system availability on multi-core architectures using recovery conscious scheduling," in *USENIX FAST*, 2008.

[7] D. Scott, "Making smart investments to reduce unplanned downtime.," *Tactical Guidelines Research Note, Gartner Group, Stamford, CT*, 1999.

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot–a technique for cheap recovery," *OSDI*, 2004.

[9] N. Kolettis and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *FTCS*, 1995.

[10] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[11] J. Gray and A. Reuter, *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann, October 1992.

[12] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating bugs as allergies — a safe method to survive software failure," in *SOSP*, Oct 2005.

[13] E. N. Elnozahy and J. S. Plank, "Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 2, pp. 97–108, 2004.

[14] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems," in *USENIX ATC*, 2007.

[15] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: a lightweight extension for rollback and deterministic replay for software debugging," in *USENIX ATC*, 2004.

[16] M. Ronsse and K. D. Bosschere, "Recplay: a fully integrated practical record/replay system," *ACM TOCS*, vol. 17, no. 2, pp. 133–152, 1999.

[17] M. Russinovich and B. Cogswell, "Replay for concurrent non-deterministic shared-memory applications," in *PLDI*, 1996.

[18] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM TODS*, vol. 17, no. 1, pp. 94–162, 1992.

[19] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[20] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, 1978.

[21] L. L. Pullum, *Software fault tolerance techniques and implementation*. Norwood, MA, USA: Artech House, Inc., 2001.

[22] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software*, 1975.

[23] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Prentice Hall PTR, 1982.

[24] J. Gray, "Why do computers stop and what can be done about it?," in *Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[25] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee, "Enhancing server availability and security through failure-oblivious computing," in *OSDI*, 2004.

[26] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh, "Using rescue points to navigate software recovery," in *SP*, 2007.

[27] B. Stroustrup, *The Design and Evolution of* C++. Addison-Wesley, 1994.