A Recovery Conscious Framework for Fault Resilient Storage Systems

Sangeetha Seshadri*, Ling Liu* and Lawrence Chiu†

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

IBM Almaden Research Center
650 Harry Road
San Jose, CA – 95120
lchiu@us.ibm.com

Abstract

This paper presents a recovery-conscious framework for improving the fault resiliency and recovery efficiency of highly concurrent embedded storage software systems. Our framework consists of a three-tier architecture and a suite of recovery conscious techniques. In the top tier, we promote fine-grained recovery at the task level by introducing recovery scopes to model recovery dependencies between tasks. At the middle tier we develop highly effective groupings of “recovery scopes” into “recovery groups” based on system and workload characteristics. We study how to distribute recovery scopes between recovery groups and schedule recovery groups effectively in a multi-core storage system through a careful tuning of system parameters. At the bottom tier, we advocate the use of recovery-conscious scheduling instead of performance oriented scheduling to provide high recovery efficiency without sacrificing system performance. An important question to address in this tier is under which combinations of resource pools and recovery groups does recovery-consciousness help to stabilize performance during recovery while still providing acceptable performance during normal operation. Our techniques have been implemented on a real industry-standard storage system. Experimental results show that the right choice of recovery-sensitive parameters is critical and our techniques are effective, non-intrusive and can significantly boost system resilience while delivering high performance under a variety of system configurations.

Introduction

Today enterprises and even end users are dealing with unprecedented amounts of digital information creating new opportunities and challenges for mass storage and on-demand storage services. Enterprise systems and services riding the crest of these new trends are placing increasing performance and availability (moving close to 7 nines) demands on storage systems. At the same time, with software failures and bugs becoming an accepted fact, fast and efficient recovery has become more important than ever for many modern storage systems.

With the growing popularity of multi-core architectures, legacy storage controller systems are compelled to adapt to rapidly advancing hardware and increasing system size. On the one hand
high, availability requirements require recovery time to be driven down, while on the contrary, increasing system size stemming from performance expectations is causing failure recovery time to increase. This can attributed to the fact that the recovery process itself often does not scale with the system size, since these systems typically perform system-wide recovery where the entire system is halted, reinitialized and checked for consistency before lost work can be redriven.

Under these circumstances, an effective way to reduce the recovery time of system failures and provide a more scalable recovery process is to perform fine-grained recovery [1]. With this approach only failed tasks perform recovery while the rest of the system continues to function uninterrupted. However, ensuring that the effects of fine-grained recovery percolate to the level of system availability while guaranteeing good performance is challenging. System resilience and recovery efficiency depend on a number of factors including the ability to implement fine-grained recovery, the scope of a recovery action taking into account complex dependencies between components as well as tasks and resource availability for normal operation even during failure recovery. Finally, in order to allow our techniques to be retrofitted into existing systems we must ensure feasibility of techniques in terms of development time and cost and minimize changes to the software architecture.

With these observations in mind, we develop a recovery conscious framework for multi-core architectures and a suite of techniques for improving the failure resiliency and recovery efficiency of highly concurrent embedded storage software systems. The main contributions of our recovery conscious framework include (1) a task-level recovery model, which consists of mechanisms for classifying storage tasks into `recovery scopes' based on both programmer specified and system-defined recovery dependencies; all tasks that must undergo recovery simultaneously fall into the same recovery scope. (2) a recovery-conscious mapping and realignment of `recovery scopes' (identified from the previous step) into `recovery groups'. Recovery groups additionally take into consideration, parameters such as, system size, failure rates, recovery rates, workload distribution, performance overhead and availability requirements; and (3) recovery-conscious scheduling, which enforces some serializability of failure-dependent tasks, i.e., tasks belonging to the same recovery group, in order to reduce the ripple effect of software failures and improve the availability of the system.

Our earlier work[2] introduces the concept of recovery groups and recovery conscious scheduling. In [2] we present an experimental evaluation of three classes of recovery conscious scheduling algorithms: static, partially dynamic and dynamic. However, we assume that recovery scopes and recovery groups have a 1:1 mapping and that recovery scopes could directly be mapped to system resources. However, in reality, the number of recovery scopes can be very large and the workload may be unevenly distributed between recovery scopes. For example, Figure 1 shows the distribution of tasks between recovery scopes in an enterprise storage controller software. In this example, tasks that utilized the same global state information were classified into the same recovery scope. The figure shows that there may potentially be a large number of recovery scopes with uneven workload distribution. However, given a system with a limited number of processing cores, maintaining very high number of
recovery scopes may result in undue processor overhead while delivering no additional benefit. Therefore, an understanding of the various system parameters that affect performance and recovery efficiency in the presence of recovery-consciousness is vital to the overall effectiveness of the framework.

In this paper we focus on addressing the following critical issues:

1. How do system parameters, such as availability requirements, workload characteristics, failure rate, recovery rate, number of cores, scheduling strategy and number of recovery groups affect the efficiency and the effectiveness of our recovery conscious framework?

2. Considering a task-level recovery granularity in a system with a large number of tasks, the number of recovery scopes can be huge. How do we determine the right granularity of recovery dependency tracking, given the relatively limited resource pools such as processor cores?

3. Finally, how do we map recovery scopes to recovery groups?

Based on our analysis we present recommendations that will guide the system designer in choosing the right settings of recovery groups, mappings and recovery strategy. Our techniques for fine-grained recovery have been implemented in a real-world enterprise class storage system with minimal architectural or design changes. Our techniques for handling failures in the system can be implemented incrementally. Based on analysis and experiments, we show that through good choice of recovery-sensitive parameters we can sustain high performance and recoverability as well as retain an advantage over non recovery-conscious techniques.

**Problem Statement**

Storage controllers are embedded systems that add intelligence to storage and provide functionalities such as RAID, I/O routing, error detection and recovery. Failures in storage controllers are typically more complex and more expensive to recover from, if not handled appropriately. Figure 2 gives a conceptual representation of a storage subsystem. In practice,
storage systems may be composed of one or more such nodes in order to avoid single-points-of-failure. The storage controller's embedded firmware provides the management functionalities for the storage subsystem and also controls the data cache. The system memory available within the controller's processor complex serves as program memory. The memory is accessible to all the processors within the complex and holds the job queues through which functional components perform work for host I/O requests. As shown in Figure 2, this processor complex has a single job queue and is an N-way SMP node. Any of the $N$ processors may execute the jobs available in the queue. The storage controller software typically consists of a number of interacting components, each of which performs work through a large number of asynchronous, short-running threads ($\thicksim \mu$secs). We refer to each of these threads as a `task'. Examples of components include SCSI command processor, cache manager and device manager. Tasks (e.g., processing a SCSI command, reading data into cache memory, discarding data from cache etc.) are enqueued onto the job queues by the components and then dispatched to run on one of the many available processors each of which runs an independent scheduler. Tasks interact both through shared data-structures in memory and message passing.

![Figure 2. Storage Subsystem Architecture](image)

With this architecture, when one thread encounters an exception that causes the system to enter an unknown or incorrect state, the common way to return the system to an acceptable, functional state is by restarting and reinitializing the entire system. We refer to this recovery strategy as system-level recovery. The necessity to embark on system-level recovery to deal with software failures is mainly due to the complex interactions between the tasks which may belong to different components. Due to the high volume of tasks (more than 20 million/minute in a typical workload), their short-running nature and the involved semantics of each task, it becomes infeasible to maintain logs or perform database-style recovery actions in the presence of software failures. Often such software failures need to be explicitly handled by the developer. However, the number of scenarios are so large, especially in embedded systems,
that the programmer cannot realistically anticipate every possible failure, given that often task interactions and execution paths are determined dynamically.

Many software systems, especially legacy systems, do not satisfy the conditions outlined as essential for micro-rebootable software[1]. For instance, even though the storage software may be reasonably modular, component boundaries, if they exist, are very loosely defined and the scope of a recovery action is not limited to a single component.

The discussion above highlights some of the key problems that need to be addressed in order to improve system availability and provide scalable recovery from software failures. Concretely, we must answer the following questions: (1) How do we identify the scope of a recovery action in a highly concurrent storage system? (2) What is the impact of fine-grained recovery on system availability and performance? (3) How do we ensure availability of the system during a recovery process, even with fine-grained recovery? What are important factors that will impact the recovery efficiency?

In addition to maintaining system performance while reducing the time to recovery, another key challenge in developing a scalable solution is to ensure that the recovery-conscious framework is non-intrusive and thus minimize re-architecting of the legacy firmware code.

**Contributions**

In this paper, we make the following contributions:

- We demonstrate that the granularity for effective recovery dependency tracking i.e. the number of recovery groups depends on the system size (number of cores in the system). As the system size increases it is beneficial to track finer-granularity recovery scopes through more recovery groups.
- The choice of the number of recovery groups is nearly independent of the task failure and recovery rates, where as system availability is more sensitive to recovery rate than failure rate.
- We recommend that the mapping of recovery scopes to recovery groups should take into consideration workload distribution between the groups and try to achieve load-balancing.
- We finally show that even with some inaccuracy in the selection of number of recovery groups and mapping of tasks, there is a conclusive advantage over performance oriented scheduling during failure recovery by being able to track recovery dependencies.

**Recovery-Conscious Framework**

In this section we give an overview of our recovery-conscious framework, which is designed for improving recovery efficiency and system fault resilience. Here, fault-resilience refers to the ability to reduce system recovery time and sustain good performance even during failure recovery. Figure 3 provides a schematic representation of our framework. The framework achieves its goals progressively through three consecutive stages detailed next. We describe tier 1 of our framework and overview tier 3 in the following sections. Since this paper focuses
on issues related to tier 2, we dedicate Section 4 to discussion and analysis pertaining to realigning of recovery scopes to recovery groups.

Organizing into Recovery Scopes
Performing fine-grained recovery in response to storage controller failures, first requires an understanding of recovery-dependencies between tasks i.e. concurrent threads in the firmware. We refer to the scope of a recovery action as a 'recovery scope'. Tasks interact with each other in complex ways. When a single task encounters an exception, more than one task may need to initiate recovery procedures in order to avoid deadlocks and return the system to a consistent state. Explicit recovery-dependencies can be specified by the programmer. However, explicit dependencies specified by the programmer may be very coarse. Likewise, some dependencies may have been overlooked due to their dynamic nature and the immense complexity of the system. Therefore one way to refine explicit dependencies is to identify implicit dependencies continuously and utilize them to refine the developer-defined recovery scopes over time. The criteria for classification of tasks into recovery scopes depends on the nature of the application and failures that are intended to be handled. In our work, we identify the following three classifications of tasks into recovery scopes.

**Resource-based**
Tasks accessing the same resources (such as device drivers or metadata) may be classified under the same recovery scope. This classification would be effective to deal with resource-based failures. For example, consider a 'queue full condition' that occurs in storage controllers. This error occurs when an adapter refuses to accept more work due to a queue full condition. Under these circumstances, the error and the subsequent recovery action would probably affect only the tasks attempting to write to the faulty adapter. One method to identify resource-based recovery dependencies is to observe the pattern of lock acquisitions. The intuition here is that, tasks that access the same resource are likely to acquire common locks. Lock acquisitions patterns can potentially be used to further refine resource-based

---

**Figure 3. Recovery-Conscious Framework**

<table>
<thead>
<tr>
<th>TIER 1: Fine grained recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery Strategy</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TIER 2: Recovery Scopes ↔ Recovery Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability Constraints</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TIER 3: Recovery Conscious Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
dependencies at runtime by utilizing the temporal aspect of dependencies apart from the spatial aspect.

<table>
<thead>
<tr>
<th>Thread T1:</th>
<th>Thread T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LockWrite</strong> (MDataLock);</td>
<td><strong>LockRead</strong> (MDataLock);</td>
</tr>
<tr>
<td><em>Set Metadata location.</em></td>
<td><em>Copy location to local variable.</em></td>
</tr>
<tr>
<td><strong>UnlockWrite(MDataLock);</strong></td>
<td><strong>UnlockRead(MDataLock);</strong></td>
</tr>
<tr>
<td><strong>...</strong></td>
<td><strong>...</strong></td>
</tr>
<tr>
<td><strong>PANIC</strong></td>
<td><strong>PANIC</strong></td>
</tr>
</tbody>
</table>

**Figure 4. Resource-based Recovery Scopes**

Figure 4 shows a situation where locking patterns could help identify recovery dependencies between two threads where one thread populates a metadata location which is consumed by the other thread. Since both threads acquire the same lock \(MDataLock\) before reading or writing into the metadata location structure, this resource dependency can be identified through the common lock. Resource-based dependencies can be identified in two ways - static code analysis or through analysis of traces collected from actual workload execution. An alternative method would be to dynamically discover these dependencies during runtime. However, the disadvantage of a dynamic approach is that the dependencies (lock acquisitions) manifest only after the thread has been dispatched. Assigning recovery scopes after dispatch would be meaningless, unless the tasks can be immediately suspended and again enqueued in the appropriate recovery group queue. However, this would result in a performance penalty due to the high amount of context switching. We therefore recommend a static assignment of tasks into recovery scopes. However, logging lock acquisitions at runtime may still be essential in order to keep track of resource ownership and perform clean-up of resources in the event of a failure. The disadvantage of a static approach compared to a dynamic approach is that it is unable to use the temporal aspect of dependencies and is likely to overlook certain dependencies that are determined dynamically. However, a wrong classification of tasks into recovery scopes will not affect the consistency of results, but only the recovery efficiency and performance during failure recovery. We show in our experimental section that the framework can tolerate some inaccuracy.

**Component-based**

Even in the absence of well-defined operational boundaries between functional components certain failures may require resetting state or performing recovery actions for tasks belonging to a particular functional component. We describe this scenario using an example depicted in Figure~\ref{fig:framework}. In this example, tasks belonging to the cache component, require a temporary data structure called the control block in order to complete successfully. In a failure situation similar to an `assert` programming construct, an error is encountered when the system runs out of control blocks. One recovery strategy in this situation is to search the list of
control blocks to identify instances that have not been freed up correctly. Another strategy is to retry the operation at a later time, in order to work around concurrency issues. However, it is very likely that other tasks belonging to the cache component are likely to encounter the same error if executed before the issue is resolved. Moreover, modifying the data structures or checking them for consistency may require further dispatch of tasks belonging to the cache component until recovery completes. In such scenarios, a functional component based classification of tasks may be effective in identifying recovery dependencies. With component-based recovery scopes, all tasks belonging to the same functional component are classified under the same recovery scope.

- **WriteToCache**
  - `sets` CB-0
- **startSCSICmd()**
- **processRead()**
  - `sets` CB-1
- **getCachedTrack()**
  - `sets` CB-2
- **getTempResource()**
  - ... PANIC(error_code)
  - ... CB → Clean-Up Block
  - RB → Resource Block

**Figure 5. Component-based Recovery Scopes**

**Request-based**

To deal with errors that require aborting or recursively recovering a user-request, it may be beneficial to classify tasks on the basis of user requests or workflows; for instance, consider a situation where a read/write request fails due to an invalid address specification. In this situation we may choose a recovery strategy of performing necessary clean-up actions and then aborting the request. Then the scope of recovery is all tasks across all components that are a part of this user request.

Depending upon the nature of failures that fine-grained recovery is expected to handle, one class or a valid combination of the above classifications may be used to define recovery-scopes. The top-tier of the framework identifies recovery scopes based on such explicitly specified (as in the case of component or request based grouping) and implicitly discovered (as in the case of resource based grouping) recovery dependencies. Besides identifying recovery dependencies, the top-tier also manages the association of developer-specified recovery strategies (such as aborting, retrying or ignoring the error) and recovery actions to the point of failure. We refer
the readers to [2] for examples and further discussion on how the framework dynamically associates recovery actions with failure points in the code.

**Recovery-conscious scheduling**

The key idea of recovery-conscious scheduling (RCS) is to ensure bounded recovery time and provide fault resiliency by optimal allocation of resources to recovery groups. The bottom-tier of the framework implements a `recovery-conscious' scheduling policy that enforces some serialization of recovery-dependent tasks in order to reduce the ripple effect of failure and ensure resource availability during a localized recovery process.

In our first prototype, we have developed three RCS algorithms to implement different methods of mapping recovery groups to resources: Static, partially dynamic, and dynamic. Each mapping technique representing different trade-offs between availability, and system performance.

Static scheduling of recovery groups determines the mapping of recovery groups to processors at compile time and is effective in situations where task level dependencies with respect to recoverability are well understood and the workloads are stable. With this scheme, processors only dispatch work from the recovery groups mapped to them. Dynamic scheduling of recovery groups to resource pools represents another end of the spectrum. This scheme works effectively, even in the presence of frequently changing workloads. With dynamic RCS, all processors are mapped to all recovery groups. However, a recoverability constraint is specified for each group. A recoverability constraint prescribes the maximum number of concurrently executing tasks permissible for that group. In order to achieve acceptable utilization, the constraint is selectively violated when no task satisfying the constraint is found while resources are idle. Between the two ends of the spectrum are the partially dynamic scheduling algorithms, which utilize partially static scheduling for those recovery groups whose resource demand is stable and well understood and apply dynamic scheduling to the rest of the recovery groups.

**Why recovery-consciousness?**

In spite of implementing fine-grained recovery and identifying recovery dependencies between tasks, without careful design, it is possible that more dependent tasks are dispatched before a recovery process can complete. This would result in an expansion of the recovery scope or an inconsistent system state. Also a dangerous situation may arise where it is possible that many or all of the threads that are concurrently executing are dependent, especially since tasks often arrive in batches. Then the recovery process could consume all system resources, essentially, stalling the entire system.

In order to overcome these problems, RCS utilizes two aspects - proactive and reactive. The proactive aspect comes into play during normal operation and aims at minimizing the number of dependent tasks executing concurrently by dispatching tasks from across different recovery
groups while closely adhering to recoverability constraints. The reactive aspect comes into play during failure recovery. At this point, utilizing knowledge of recovery dependencies afforded by recovery groups, RCS suspends dispatch from recovery groups whose tasks are currently undergoing recovery.

We measure the effectiveness of recovery-conscious scheduling against traditional performance-oriented scheduling (POS). Performance oriented scheduling, either with a single-centralized queue or multiple load-balanced queues, has no notion of recovery-dependencies in its criteria for resource allocation.

**Recovery-Conscious Mapping**

The number of recovery groups in the system and the constraints on these recovery groups are critical factors in determining both system recovery time and fault resiliency. A large number of recovery groups allows the system to exercise more fine-grained control during task dispatch and allows the system to find more useful work to perform with the limited resources available during failure recovery. At the same time, depending upon the distribution of the tasks between recovery groups and system size, the decision on the number of groups can impact performance both during normal operation and failure recovery. In general, beyond a system-specific threshold, the scheduling overhead may outweigh the benefit of decreased lock contention.

In order to effectively enforce recoverability constraints, we must map scopes appropriately to groups. A naive method would be to choose as many groups as recovery scopes. However, this may result in a performance penalty as the scheduler polls a large number of recovery groups (some of which may be empty) for work. At the same time, the benefit in recovery time may not be significant either. In order to address this question, we need an understanding of the interactions between various factors such as system size, scheduling strategy, failure and recovery rates and workload characteristics.

**Impact of Recovery Groups on System Resilience**

The number of outstanding tasks belonging to a single recovery group and hence the degree of serialization has a direct bearing on the time-to-recovery of the system. For example, in the worst case where all tasks running at the time of failure belong to the same recovery group, massive system-wide recovery will have to be initiated. Intuitively, the recovery time increases with increasing size of the system and with decreasing number of recovery groups.

Based on the definition of recovery groups, we assume that when a task $t$ belonging to the $k^{th}$ recovery scope fails, all tasks belonging to the scope that are executing concurrently with the failed task $t$ need to undergo recovery.

Let $\lambda_k$ represent the failure rate and $\mu_k$ represent the repair rate for failures in the $k^{th}$ recovery scope. The number of processors or cores in the system is represented by variable $m$ and let
\( \alpha_k(i) \) represent that probability that \( i \) outstanding tasks belonging to the \( k^{th} \) recovery scope are executing concurrently at the time of failure.

We assume that the recovery process executes serially even for concurrently executing threads in order to restore the system to a consistent state. As a result, the time to complete system recovery is a product of the number of recovering processes and the individual task recovery time. Then the mean time to complete system recovery is given by:

\[
\mu = \alpha_k(1) \frac{1}{\mu_k} + \alpha_k(2) \frac{2}{\mu_k} + \alpha_k(3) \frac{3}{\mu_k} + \ldots + \alpha_k(m) \frac{m}{\mu_k}
\]

Let \( \gamma_k \) represent the probability that a task belongs to recovery scope \( k \). Then using the Poisson approximation for the binomial probability mass function, the probability that there are \( i \) outstanding tasks belonging to the \( k^{th} \) recovery scope is given by:

\[
\alpha_k(i) = b(i; m, \gamma_k) = \gamma_k^m \frac{e^{-\gamma_k m} \times (\gamma_k m)^i}{i!}
\]

With performance-oriented scheduling (POS), there is no notion of bounding the recovery process. Interdependent tasks belonging to the same recovery scope can potentially be executing on all processors. As a result up to \( m \) dependent tasks may be executing concurrently at the time of failure. Under these circumstances the system mean-time-to-recovery (MTTR) for POS given that the failure occurred in the \( k^{th} \) recovery group denoted by MTTR\(_{POS}\)\( |k \) is:

\[
\text{MTTR}_{POS} | k = \sum_{i=1}^{m} \frac{e^{-\gamma_k m} \times (\gamma_k m)^i}{i!} \times \frac{i}{\mu_k}
\]

On the other hand, RCS enforces constraints on recovery groups there by ensuring some degree of serialization of dependent tasks. Let us assume that the constraint on the maximum number of concurrent tasks of the recovery group containing the \( k^{th} \) recovery scope is given by \( c_k \). Then the system mean-time-to-recovery (MTTR) for RCS given that the failure occurred in the \( k^{th} \) recovery group denoted by MTTR\(_{RCS}\)\( |k \) is:

\[
\text{MTTR}_{RCS} | k = \sum_{i=1}^{m} \frac{e^{-\gamma_k m} \times (\gamma_k m)^i}{i!} \times \frac{i}{\mu_k}
\]

However, with dynamic RCS, a more flexible mapping of resources to recovery groups is employed in order to reduce resource idling and improve utilization. Under this scheme in the event that there are spare idle resources even after all tasks have been dispatched according to recoverability constraints, keeping in mind the high-performance requirements of the system, the constraints are selectively violated. Let the number of active recovery groups in the system be denoted by \( R \). Let \( c_k \) be the constraint specified on the maximum number of concurrent tasks for the group containing the \( k^{th} \) recovery scope. Without loss of generality we assume that
there are idle resources only when $\sum_{i=1}^{n} c_i < m$. For the sake of simplicity let us assume that the available spare resources $m - \sum_{i=1}^{n} c_i$ is allocated evenly amongst all groups. Then in the worst case violation of a constraint $c_k$, denoted as $c_k$, is given by:

$$R_k = c_k + \left| \frac{m - \sum_{i=1}^{n} c_i}{R} \right|$$

Thus, the system recovery time with dynamic RCS is obtained by replacing the constraint $c_k$ by $R_k$ in the expression for system recovery time for RCS ($MTTR_{RCS|k}$).

Clearly, the system availability under POS is affected by the failure rate $\lambda_k$, the repair rate $\mu_k$ for failures in the $k^{th}$ recovery scope, the number $m$ of processors or cores in the system, and the probability $\gamma_k$ that a task belongs to recovery scope $k$. In contrast, with RCS, availability is also influenced by additional parameters such as the number $R$ of active recovery groups in the system and the constraint $c_k$ on the maximum number of concurrent tasks of the group containing the $k^{th}$ recovery scope.

**Impact of RCS Queues on System Performance**

In this section we present analysis that shows the impact of recovery groups on the system performance and based on these results we describe criteria for the selection of number of recovery groups for efficient scheduling. Each recovery group is mapped to a single scheduler queue and the serialization constraint imposed on the group applies to all scopes that are mapped to the group.

While evaluating system performance, we must take into consideration both the good-path (i.e. normal operation) and bad-path (during failure recovery) performance. Good path performance is primarily impacted by the efficiency of the scheduler. On the other hand, bad-path performance will be impacted by the extent of failure and recovery (i.e. the degree of serialization) and the availability of resources for normal operation during local recovery.

**Variation of service rate with RCS queues**: We model the variation of service rate with the number of queues as a hypoexponential distribution with 2 phases where the first phase describes the scenario where the service rate increases with the number of queues due to reduced lock contention. The second phase models the scenario where the increase in the number of queues causes the service rate to drop due to the additional scheduling overhead. Figure 6 shows an example of this model for variation of service rate with the number of queues.
In order to study the impact of recovery-consciousness on the performance of the system, we model both POS and RCS with varying system size and during good-path and bad-path operation. In order to model utilization, response time and throughput we adopt the models for M/M/m queuing systems [3].

Consider a system where tasks arrive as a Poisson process with rate $\lambda_a$, and service times for all cores are independent, identically distributed random variables. Let the mean service rate as a function of the number of scheduler queues (groups in the case of RCS), for performance oriented scheduling be denoted by $\mu_{pos}$ and for recovery conscious scheduling be denoted by $\mu_{rcs}$. We assume that the service times include the time required to dequeue tasks from the job queue(s) and iterate through queues (for RCS). Let $m$ denote the total number of cores in the system.

**Good-path Performance**: During good-path operation, all system resources are available and storage controller performance is limited only be scheduler efficiency. Accordingly, the average number of jobs, $N$, in the system is given by:

$$E[N] = m\rho + \rho \frac{(mp)^m}{m!} \frac{p_0}{(1-\rho)^2}$$

where $p_0$, the steady state probability that there are no jobs in the system is given by:

$$p_0 = \left[ \sum_{k=0}^{\infty} \frac{(mp)^k}{k!} + \frac{(mp)^m}{m!} \frac{1}{(1-\rho)} \right]^{-1}$$

For POS, the value $\rho$, the traffic intensity, is given by, $\rho_{pos} = \frac{\lambda_a}{m\mu_{pos}}$ and that for RCS is given by $\rho_{rcs} = \frac{\lambda_a}{m\mu_{rcs}}$. $E_{POS}[N]$ and $E_{RCS}[N]$ are obtained by substituting $\rho$ by $\rho_{pos}$ and $\rho_{rcs}$ respectively in the expressions for $E[N]$ and $p_0$. In each case, based on Little's formula [4] the

![Figure 6. Variation of Service Rate](image-url)
average response time for performance-oriented scheduling \( (E_{POS[R]} \) and RCS \( (E_{RCS[R]} \) is given by:

\[
E_{POS[R]} = \frac{E_{POS[N]}}{\lambda_a} \quad \text{and} \quad E_{RCS[R]} = \frac{E_{RCS[N]}}{\lambda_a}
\]

Assuming that our system utilizes a non-preemptive model where individual tasks complete execution within the service time allocated to them on system cores, the system throughput \( T \) can be modeled as follows:

\[
E_{POS[T]} = \mu_{POS} U_0^{POS} \quad \text{and} \quad E_{RCS[T]} = \mu_{RCS} U_0^{RCS}
\]

where \( U_0 \) the utilization of the system is given by \( U_0 = 1 - p_0 \) and the values for utilization with POS \( (U_0^{POS}) \) and RCS \( (U_0^{RCS}) \) are obtained by substituting appropriate values for \( p_0 \).

**Bad-path Performance:**

In order to model system performance during bad-path operation we assume that the amount of system resources consumed by the recovery process is proportional to the extent (i.e. the number of outstanding tasks undergoing recovery) of the recovery process.

As described previously, with POS, the extent of the recovery process is unbounded and can potentially span all the available cores in the system. As with the analysis of system availability, assume that a task \( t \) belonging to the \( k^{th} \) recovery scope encounters a failure causing in all executing tasks belonging to the \( k^{th} \) recovery group to undergo recovery. Let \( f_{k}^{POS} \) and \( f_{k}^{RCS} \) denote the extent of the failure-recovery for POS and RCS respectively. Let, \( \bar{m}_{POS} \) and \( \bar{m}_{RCS} \) denote the expected number of cores available for normal operation during failure recovery. Then, as explained in the case of the impact on availability,

\[
\bar{m}_{POS} = m - f_{k}^{POS} = m - \sum_{l=0}^{m} e^{-\gamma_{k}m} \left(\gamma_{k}m\right)^l \times l
\]

\[
\bar{m}_{RCS} = m - f_{k}^{RCS} = m - c_{k}
\]

Then the expected response time and throughput during bad-path: \( E'_{POS[R]} \), \( E'_{POS[T]} \) and \( E'_{RCS[R]} \), \( E'_{RCS[T]} \) for POS and RCS respectively can be computed by substituting \( m \) in the original expressions with \( \bar{m}_{POS} \) and \( \bar{m}_{RCS} \) respectively.
Experimental Results

In this section we present results from experiments conducted using both simulations and a prototype. The experimental results illustrate the complex dynamics between the various factors that affect performance and recovery efficiency. Our results provide valuable insights into the implications and trade-offs associated with an implementation of fine-grained recovery and the effectiveness of our proposed framework.

Implementation and Experimental Setup

Our framework is implemented on a high-capacity, high-performance and highly reliable enterprise storage system built on proprietary server technology due to which some of the setup and architecture details presented in this paper have been desensitized. The system is a storage facility that consists of a storage unit with two redundant 8-way server processor complexes (controllers), memory for I/O caching, persistent memory (Non-Volatile Storage) for write caching, multiple FCP, FICON or ESCON adapters connected by a redundant high bandwidth (2 GB) interconnect, fiber channel disk drives, and management consoles. The system is designed to optimize both response time and throughput. The embedded storage controller software is similar to the model presented in this paper. The system has a number of interacting components which dispatch a large number of short running tasks. For the prototype experiments in this paper we identify 16 recovery scopes based on component-based explicit recovery dependency specifications.

Our framework was implemented in the storage-controller firmware micro-code. In our implementation, the recovery-conscious scheduler alone was implemented in approximately 1000 lines of code. Task-level recovery can be implemented incrementally for each failure situation that we intend to handle. Currently, our implementation specifies system-level recovery as the default action, except for cases for which task-level recovery has been implemented. A naive coding and the design effort for task level recovery would be directly proportional to the number of "panics" or failures in the code that are intended to be handled using our framework. In general, the coding effort for a single recovery action is small and is estimated to be around a few tens of lines of code (using semicolons as the definition of lines of code) per recovery action on average.

Our simulation studies allowed us to experiment with different system configurations, failure scenarios, recovery parameters and scheduling strategies and various combinations of these factors. The simulator written in C allows configuration of system specifications (such as number of processors and scheduling policy), recovery strategies (proactive/reactive, recovery scope specification) and fault injection parameters (failure rate, failure type, recovery rate). The simulator is driven by an externally provided workload trace specifying individual task descriptions, lock acquisition patterns, task arrival times and execution times. For the simulation experiments described in this paper, we utilized traces of the cache-standard workload described next. Note that, the simulator models the performance behavior of the storage controller but does not actually execute the underlying tasks.
The Z/OS Cache-standard workload [5, 6] is considered comparable to typical online transaction processing in a z/OS environment. The workload has a read/write ratio of 3, read hit ratio of 0.735, destage rate of 11.6% and a 4K average transfer size. The setup for the Cache-standard workload in the prototype implementation was CPU-bound. In addition to the Cache-standard workload, we also present prototype results with a second workload which is a disk-bound internal workload similar to Cache-standard. Consequently this set-up has low cpu utilization (~25%). We refer to this workload as Workload-2.

**Methodology**

We use our workload to measure throughput and response times in our prototype experiments and scheduler efficiency (as measured by the number of task dispatches per unit time) in the simulation experiments. For the prototype experiments we identified 16 component-based recovery scopes. Each recovery scope corresponded to a functional component such as a host adapter, device manager or cache manager.

In order to understand the impact on system performance when localized recovery is underway, we inject faults into the workload. We choose a candidate task belonging to recovery scope 5 and introduce faults at a fixed rate. The time required for recovery is specified by the recovery rate. During localized recovery, all tasks belonging to the same recovery scope that are currently executing in the system and that are dispatched during the recovery process also experience a delay for the duration of the recovery time. For example, in our implementation, a recovery time of 20 ms and a failure rate of 1 in every 10K dispatches, for tasks belonging to component 5, introduces an overhead of 5% to aggregate execution time per minute of component 5 execution on average. The recoverability constraint for dynamic RCS was set to 1. Note that in the case of dynamic RCS, the constraint would selectively be violated only if no task satisfying the constraint was found.

**Effect of Fine-Grained Recovery on System Performance**

In order to infuse recovery-consciousness into the allocation of resources, we need to keep track of recovery-dependencies. However, when recovery dependencies are tracked at a very fine granularity, the overhead of managing a large number of recovery scopes may induce a severe performance penalty. Therefore, we need to first understand the performance impact of tracking fine-grained recovery scopes. For the experiments, the tasks were redistributed between recovery scopes based on different granularity of dependency tracking and each of these scopes were mapped to a recovery group. In effect, the scope to group mapping was 1:1 only in the case of 512 recovery groups and many:1 in all other cases. Using this mapping, we study the performance impact of fine-grained recovery under different system sizes, and scheduling policies. We measure scheduler performance during normal operation and failure recovery using the number of task dispatches per unit time as a metric.
Figure 7 shows the average number of dispatches per minute during normal operation with varying number of recovery groups. The plot shows the curves for 2, 4 and 8 cores with the dynamic RCS scheduling scheme and that of performance-oriented scheduling with 8 cores. In the case of POS, the workload is uniformly distributed between the queues. Recall that each recovery group is managed using a separate scheduler queue. With RCS, in all three cases, the number of dispatches initially increase (as much as 16%, 14% and 65% in the case of 8, 4 and 2 cores respectively) and then decreases (as much as 21%, 30% and 45% in the case of 8, 4 and 2 cores respectively). The high performance peak is achieved with 64 groups in the case of 8 cores, 32 groups with 4 cores and 8 groups with 2 cores.

- This shows that the decision on the best choice of number of recovery groups depends on the system size.

Next, although the scheduler initially benefits from the increased concurrency afforded by additional scheduling queues, as the number of queues increases, due to the uneven distribution of workload between recovery groups, scheduling efficiency decreases.

- Depending upon system size, beyond a certain granularity, recovery-consciousness and keeping track of fine-grained recovery scopes may degrade system performance.

On the other hand, while performance oriented scheduling also exhibits decreasing efficiency with a large number of queues, the degradation in scheduler performance is more graceful due to the uniform distribution of workload.

- The choice of number of recovery groups and mapping of tasks to recovery groups should take into consideration workload distribution between the groups and try to achieve load-balancing.

In order to emphasize the importance of right choice of number of recovery groups on performance, we next compare scheduler performance under dynamic RCS and a load-balanced performance-oriented scheduler with varying system size. We use 16 recovery groups...
for the dynamic RCS scheduler and 16 queues, with uniform workload distribution for the performance oriented scheduler. Figure 8 shows the average number of dispatches per minute in both cases with varying system size. With this hand-picked choice of number of recovery groups, we see that the system can achieve performance that is very close to a performance-oriented architecture, even while tracking recovery dependencies across varying system sizes.

![Figure 8. Average number of dispatches per minute](image1)

**Figure 8. Average number of dispatches per minute**

**Figure 9. Comparison with Bad Path performance**

**Figure 10. Bad-Path Performance: 4 queues**

**Effect of Fine-Grained Recovery on System Availability**

The benefit from tracking recovery dependencies is realized during failure recovery. Figure 9 compares scheduler performance during normal operation with that during failure recovery for a system with 8 cores. By availability, we refer to service availability and also the ability of the service to meet performance expectations during failure-recovery. We measure this using scheduler performance during failure-recovery. Failure was emulated by injecting faults into a chosen component at the rate of once in every 10K dispatches of the tasks belonging to that component.

First, the graph shows that, during failure-recovery, for low number of recovery groups, i.e. a coarse granularity of recovery tracking, the benefit from recovery consciousness is low - although still higher than the performance-oriented case. However, at the right granularity, recovery consciousness can make a significant improvement in scheduler performance. In this case, at a group size of 16, recovery-consciousness can effect a 23% improvement in scheduler performance.
Next, consider the group sizes 4 and 32 where POS almost matches the performance of RCS. Figure 10, 11 and 12 represent the number of dispatches per minute over a duration of 30 minutes. The graphs show that even at group sizes of 4 and 32 where POS matches RCS in average number of task dispatches per minute, POS results in serious fluctuations of scheduler performance. At some instances, the number of dispatches with POS drops to as low as 65% of that with RCS. Recall that, POS distributes workload equally amongst all processors without considering recovery dependencies. Therefore, during failure, many tasks dependent on the failing task may be executing concurrently. As a result, in spite of fine-grained recovery, the entire recovery process takes longer, resulting in a drop in performance due to unavailability of resources for normally operating tasks.

- We can argue that even with some inaccuracy in the selection of number of recovery groups, there is a conclusive advantage over performance oriented scheduling during failure recovery by being able to track recovery dependencies.
- Also, in spite of implementing fine-grained recovery, it is crucial to track recovery-dependencies to improve performance during failure recovery.

Figure 13 shows the variation in system recovery time by varying individual task recovery time, the number of cores, and the distribution of tasks between groups. The figure is generated based on the model for MTTR_{POS} and MTTR_{RCS} described in the previous section. The lower surface (in red) depicts the recovery time variation for RCS and the upper surface (in gray) depicts the recovery time variation under POS. The x-axis represents the variable \( m \gamma_k \) where \( m \) represents the number of cores in the system and \( \gamma_k \) represents the probability that a task belongs to the failing recovery group \( k \). Intuitively the x-axis can be thought of as the number of cores per recovery group. The y-axis represents individual task recovery time in seconds and the z-axis represents the total system recovery time in seconds. The constraint for RCS is set as \( c_k = 10 \). As the graph shows, for POS, the system recovery time increases rapidly with increasing task recovery time and \( m \gamma_k \). The extent of recovery may increase either due to increase in system size or due to a large proportion of tasks belonging to the failing recovery group. On the
other hand, with RCS, the recoverability constraint ensures that the system recovery time remains low by restricting the number of cores assigned per recovery group.

![Figure 13. System MTTR](image1)

![Figure 14. Variation with Recovery Rate](image2)

**Sensitivity to Recovery and Failure Rate**

Figure 14 shows the variation of scheduler performance for different recovery rates for tasks belonging to component 5. The failure rate was fixed at 1 in every 10K dispatches of tasks belonging to component 5. The figures tells us that the choice of number of recovery groups is nearly independent of the recovery rate, since if `x` number of recovery groups is a better choice than `y` for a certain recovery rate, it is almost true for all other recovery rates also.

![Figure 15. Variation with Failure Rate](image3)

Figure 15 shows the variation of scheduler performance with different failure rates. The recovery time for a single failed task was set to 100 ms and failure injected into tasks belonging to component 5. As with the case of recovery rate, the figure shows that the choice of number
of recovery groups is nearly independent of failure rate. Also, the figures 14 and 15 show that the system performance is far more sensitive to the recovery rate than the failure rate. For example, an 80% improvement in recovery rate improves system performance by 27% on average, as compared to a 100% improvement in failure rate effecting a 13% improvement in performance.

Prototype Experiments
By conducting experiments with our prototype implementation using the Cache-standard workload, we observe that our recovery conscious architecture improved system throughput by 16.3% and response time by 22.9% during failure recovery compared to POS. The throughput with POS was observed to be 107 KIOps and 87.8 KIOps during good-path and bad-path respectively and with RCS 105 KIOps during both good-path and bad-path. Similarly, the response time with POS was observed to be 13.3 ms and 16.6 ms during good-path and bad-path respectively and with RCS 13.5 ms during both good-path and bad-path.

Figure 16 shows the average number of task dispatches per minute over 30 minutes with varying number of recovery groups under the Cache-standard workload. The figure also shows the scheduler performance for the same configuration using the simulation. As the figure shows, the number of dispatches initially increases (although modestly) with the increase in the number of groups. For instance, when the number of groups increase from 1 to 16, the number of dispatches increase by nearly 13% (9% in the simulation) and from 1 to 4, the number of dispatches increases by 10% (8% in the simulation). This experiment was used to validate the simulator and establish the preferred number of recovery groups as 16 for further experimentation with the prototype.

Figure 17 shows the average number of task dispatches per minute per recovery group and in total under POS and RCS with the Cache-standard workload. Of the 16 recovery groups, only 8
have active tasks. The figure shows the number of dispatches under normal operation (which are nearly identical for POS and RCS) and those under bad-path for RCS and POS. Under bad-path operation, the number of dispatches with POS drops by nearly 14.4% while the number of dispatches for RCS drops by only 3%, which corresponds to a 16.3% improvement in throughput and 22.9% improvement in response time of RCS over POS, in the average case. In the worst case POS may cause complete system unavailability.

Summary of Experiments
While our experiments provide some insights into the selection of parameters such as recovery groups, clearly these decisions are largely impacted by the nature of the software. Below, we present guidelines for the selection of these parameters which must be validated for the particular instance of software and system configuration. A possible procedure to perform this validation is to evaluate the impact of various parameters using simulation based studies and workload traces as shown in our paper.

The number of recovery scopes in the system is a characteristic of the software and the dependencies between tasks. Once, the granularity of recovery has been identified, and the dependency information has been specified (with explicit dependencies being specified initially and the system identifying implicit dependencies over certain duration of observation), the `recovery scopes` are identified. During runtime, when tasks are being enqueued, they are enqueued based on the recovery scope that has been identified for the task. The scheduler efficiency now depends on the number of recovery groups that need to be iterated through at runtime. This choice of recovery groups, depends on the system size (i.e. the number of cores) and the distribution of tasks between the recovery scopes. Thus, the guidelines for selection of recovery-conscious parameters can be summarized as follows:

- The ideal choice of the number of recovery groups depends on the system size. However, choosing the number of groups to be more than the number of cores can help improve performance by reducing contention for job queue locks for example.

- The choice of the number of recovery groups and the mapping of tasks to recovery groups should take into consideration workload distribution between the groups and try to achieve load-balancing and avoid idle cycling of the scheduler through empty queues looking for work. The information required to perform load-balancing can be acquired by studying the workload for distribution of tasks between recovery scopes and their arrival rates.

- Even with some inaccuracy in the selection of number of recovery groups, there is a conclusive advantage of recovery conscious scheduling over performance oriented scheduling during failure recovery by being able to track recovery dependencies. This gives the developer some flexibility in choosing the number of recovery groups.
Discussion

We see that as the number of recovery groups in the system increases, indicating a finer granularity of recovery, a natural resiliency develops in the system, thus improving availability in the average case. However, the rate at which the resource availability improves tends to decrease as the granularity of recovery dependency tracking gets finer. The benefit from recovery groups at a given granularity is determined by the following factors: distribution of tasks between groups, task recovery time, failure rate, and system size. However, the choice of the number of recovery groups mainly depends on system size, distribution of workload and nature of failure versus recovery-dependency tracked. Depending on these parameters, we can predict the expected recovery time in the event of a failure. By appropriate selection of the number of recovery groups and the recovery scope to group mapping, we can derive the maximum benefit from the recovery-conscious framework.

In this section, we would like to make some comments on the design of our recovery conscious scheduling framework.

(1) Effectiveness of the recovery-conscious scheduling.
One of the important questions is how to make the scheduler stronger by considering other factors, such as detecting and avoiding faulty adapters, to avoid the meaningless rescheduling. Our recovery conscious scheduling to some extent avoids the kind of circumstances such as continuing to send tasks to a fault adapter repeatedly. The concept of recovery groups is meant to serialize dependencies by which more faulty tasks are not dispatched while recovery is being attempted. Two types of recovery-conscious scheduling – proactive and reactive were introduced and discussed in our FAST paper [1]. The goal of proactive scheduling is to enhance availability and reduce the impact of failure by bounding the number of outstanding tasks per recovery group even during normal operation. Reactive scheduling, on the other hand, takes over after a failure has occurred. Reactive scheduling suspends the dispatch of the tasks belonging to the group undergoing recovery until localized recovery completes. If a problematic adapter results in the failure of a task belonging to a recovery group, dispatch of tasks from that recovery group will be suspending until the recovery completes. Instead, the resources belonging to that recovery group may be used for recovery, or be utilized by other failure-free recovery groups.

(2) Programmer Assistance in Specifying Recovery Handler
One of the requirements of our recovery-conscious framework is the need for programmer to specify the recovery handler. There are a number of reasons for this requirement. First, we acknowledge that writing error-recovery code is a complex task. As a first step, our framework provides guidance for identifying dependencies and also ensures that recovery handlers are non-intrusive and have minimal impact on good-path execution. We are working on the
techniques to identify and incorporate micro-feedback based learning mechanisms into the identification of failure scenarios and recovery strategies.

Second, our analysis of the software shows that due to the complexity of the system, not all failures can be recovered using fine-grained recovery. The recovery strategies are often determined by the semantics of the failure and the nature of the tasks that encountered the failure. For example, a straightforward recovery strategy for a failure during a back-ground task that is not critical can be to simply ignore the failure, while the strategy may be different for a critical task that must complete on time. Such task specific semantic based recovery handler is best defined by the developers of these tasks. Due to the inherent complexity of the system and semantics involved, fully automated approach to determining the recovery strategies, without programmer assistance, is difficult and less effective. In the event that the developer specifies an ineffective recovery strategy such that the problem caused by the failure could not be resolved, our framework recommends setting a recovery threshold which specifies the number of times that the micro-recovery should be attempted before falling back to system-level recovery. If the failure is not prevented by the micro recovery mechanism and the failure threshold has been reached, then system-level recovery will be performed.

Finally, we would like to point out that, in the case of identifying recovery dependencies, our framework combines the programmer assistance with system initiated learning through recovery logs. While the programmer is given the facility to provide explicit dependencies based on experience, the system uses the programmer’s specification of dependencies as a starting point and continues to refine these dependencies throughout the life cycle of the system. In other words, the recover conscious framework does not rely on the completeness or the correctness of developer’s specified dependencies. One of our ongoing work is to develop a recovery-log based architecture that utilizes the information provided by lock accesses as a guideline to understanding system state changes. Based on such interactions between concurrent tasks, the architecture dynamically identifies dependencies at runtime and alerts the developer to events like dirty reads of shared state. This log-based architecture will relieve the developer from the burden of tracking resources such as shared buffers and locks.

Related Work

Our work is largely inspired by previous work in the area of software fault tolerance and storage system availability. Techniques for software fault tolerance can be classified into fault treatment and error processing. Fault treatment aims at avoiding the activation of faults through environmental diversity, for example by rebooting the entire system [7], micro-rebooting sub-components of the system [1], through periodic rejuvenation [8, 9] of the software, or by retrying the operation in a different environment [10]. Error processing techniques are primarily checkpointing and recovery techniques [11], application-specific techniques like exception handling [12] and recovery blocks [13] or more recent techniques like failure-oblivious computing [14].
In general our recovery conscious approaches are complementary to the above techniques. However, the need to minimize re-architecting the system and the tight coupling between components makes both micro-reboots and periodic rejuvenation tricky. Rx [10] demonstrates an interesting approach to recovery by retrying operations in a modified environment but it requires checkpointing of the system state in order to allow ‘rollbacks’. However given the high volume of requests (tasks) experienced by the embedded storage controller and their complex operational semantics, such a solution may not be feasible in this setup. While the idea of localized recovery such as transactional recovery in DBMSs [15], application-specific recovery mechanisms such as recovery blocks [13], and exception handling [12] have been used before, the implications of localized recovery on system availability and performance in a multi-core environment where interacting tasks are executing concurrently is not very well understood.

In earlier work [2] we presented three alternative recovery-conscious scheduling algorithms each representing one way to trade-off between recovery time and system performance. The recovery-conscious scheduling algorithms help bound the recovery process in time and resource consumption assuming that effective configurations for recovery-sensitive parameters have been identified. Likewise, although vast amounts of prior work have been dedicated to resource scheduling, to the best of our knowledge, such work has mainly focused on performance [16, 17].

Much work in the virtualization context has been focused on improving system reliability [18] by isolating VMs from failures at other VMs. In contrast, our development focuses more on improving system availability by distributing resources within an embedded storage software system. Compared to earlier work on improving storage system availability at the RAID level [19], we are concerned with the embedded storage software reliability. These techniques are at different levels of the storage system and are complementary.

Conclusion

We have presented a recovery-conscious framework for improving system dependability in terms of fault resiliency and recovery efficiency of highly concurrent embedded storage software systems. Our framework consists of a three-tier architecture and a suite of recovery conscious techniques. In addition to overview of the recovery scope formation tier, the recovery conscious system configuration tier and the recovery conscious scheduling tier, we focused on developing highly effective mappings of dependent tasks to processor resources through careful tuning of recovery efficiency sensitive parameters. We presented a formal model to guide the understanding and the development of techniques for effectively mapping fine-grained recovery scopes to recovery groups to system resources, aiming at reducing the ripple effect of software failures while sustaining high performance even during system recovery. Our proposed recovery conscious framework and techniques have been implemented on an enterprise storage controller. Through our analysis and experimentation we have shown that through careful tuning of the system configuration and parameters that affect recovery
efficiency, it is possible to improve the system resiliency while sustaining good performance even during failure recovery.

Acknowledgement. This work is partially funded by grants from NSF CISE and an IBM SUR grant. We would like to thank the team members in the enterprise storage system architecture group at IBM Almaden for many fruitful discussions.

References


