A Recovery Conscious Framework for Fault Resilient Storage Systems

Sangeetha Seshadri*, Ling Liu*, Lawrence Chiu†, Cornel Constantinescu† and Subashini Balachandran†
*Georgia Institute of Technology
801 Atlantic Drive GA-30332
{sangeeta,lingliu}@cc.gatech.edu
†IBM Almaden Research Center
650 Harry Road CA-95120
{lchiu, cornel, sbalach}@us.ibm.com

Abstract

In this paper we present 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 groups to model recovery dependencies between tasks. At the middle tier we develop highly effective mappings of dependent tasks to processor resources through careful tuning of recovery efficiency sensitive parameters. At the bottom tier, we advocate the use of recovery-conscious scheduling by careful serialization of dependent tasks, which provides high recovery efficiency without sacrificing system performance. We develop a formal model to guide the understanding and the development of techniques for effectively mapping fine-grained tasks to system resources, aiming at reducing the ripple effect of software failures while sustaining high performance even during system recovery. Our techniques have been implemented on a real industry-standard storage system. Experimental results show that our techniques are effective, non-intrusive and can significantly boost system resilience while delivering high performance.

Keywords: *Storage, Software, Fault resilience, Performance, Availability.*
Category: *Storage, Database, Transactional Systems*

1. 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. On the other hand, with software failures and bugs becoming an accepted fact, fast and efficient recovery has become more important than ever in many modern storage systems. In current system architectures, even with redundant controllers, most microcode failures trigger system-wide recovery [6, 7] causing the system to lose availability for at least a few seconds, and then wait for higher layers to redrive the operation. This unavailability is visible to customers as service outage and will only increase as the platform continues to grow under the legacy architecture.

With the growing popularity of multi-core architectures legacy system are pushed to adapt to rapidly advancing hardware and increasing system size. Under these architectures an effective way to reduce the recovery time of system failures is to perform fine-grained recovery that only recovers failed tasks. 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 and tasks and resource availability for normal operation even during failure recovery. Finally, since we are dealing with a large legacy architecture ($>$ 2M lines of code) we must ensure feasibility of techniques in terms of development time and cost and minimize changes to the architecture.

With these observations in mind, in this paper 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 groups based on both programmer specification at a coarser granularity and system-defined recovery scopes at a finer granularity; (2) a recovery conscious mapping of system resources such as number of cores to dependent tasks through careful identification of recovery-sensitive parameters such as number of scheduler queues and recoverability constraints; and (3) the development of recovery-conscious scheduling, which enforces some serializability of failure-dependent tasks in order to reduce the ripple effect of software failures and improve the availability of the system. Each tier of the framework progressively improves the system resilience and recovery efficiency by embedding "recovery-consciousness" into various operating layers of the system.

Concretely, in this paper we develop a formal model to guide the understanding and the development of techniques for effectively mapping fine-grained tasks to system resources, aiming at improving system resilience while sustaining high performance both during normal operation and during system recovery. We explore the effects of serialization of dependent tasks on the trade-off between recovery time and system performance

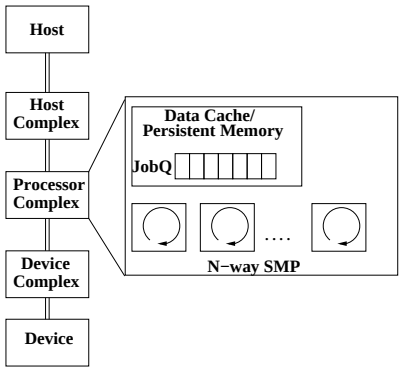

Figure 1. Storage Subsystem Architecture

with varying system size and complexity. Our techniques for fine-grained recovery have been implemented in a real-world enterprise class storage system. Based on our analysis and experimental results, we present guidelines for choosing effective recovery strategies for highly scalable, high-performance systems that can be retrofitted into existing systems with minimal architectural design changes. Experimental results show that our techniques are non-intrusive and can significantly boost system resilience while delivering high performance.

2. Motivation and Technical Challenges

In this section we motivate this research and illustrate the problem we address by considering the storage controllers of some representative storage system. We focus on system recoverability from software failures. 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 if not handled appropriately. We believe that most of the concepts and problems pertaining to software failures in a storage controller are also applicable to other highly concurrent system software.

Figure 1 gives a conceptual representation of a storage subsystem. This is a single storage subsystem node consisting of hosts, devices, a processor complex and the interconnects. In practice, storage systems may be composed of one or more such nodes in order to avoid single-points-of-failure. The processor complex provides the management functionalities for the storage subsystem. The system memory available within the processor complex serves as program memory and may also serve as the data cache. The memory is accessible to all the processors within the complex and holds the job queues through which functional components dispatch work. As shown in Figure 1, 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 ($\sim \mu$secs). We refer to each of these threads as a '**task**'. Tasks 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 as well as through message passing.

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. Since the system state may either be lost, or cannot be trusted to be consistent, some higher layer must now redrive operations after the system has performed basic consistency checks of non-volatile metadata and data. While the system reinitializes and waits for the operations to be redriven by a host, access to the system is lost contributing to the downtime. This recovery process is widely recognized as a barrier to achieving high(er) availability. Moreover, 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.

The necessity to embark on system-wide 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. Also, an individual developer may only be aware of the clean-up routines for the limited scope being handled by them. This knowledge is insufficient to recover the entire system from failures, 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:

- How do we implement fine-grained recovery in a highly concurrent storage system?
- Can we identify recovery dependencies across tasks and construct efficient recovery scopes?
- How do we ensure availability of the system during a recovery process? 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 application code.

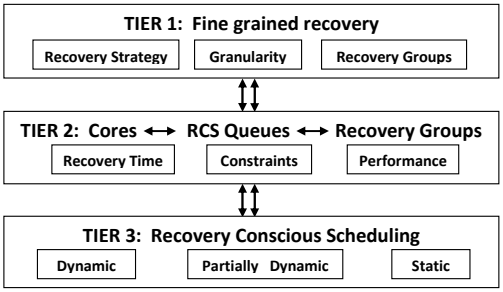

Figure 2. Recovery-Conscious Framework

3. 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. By fault-resilience, we mean that our framework and techniques provide the ability to reduce system recovery time, enhance availability and sustain good performance even during failure recovery. We achieve this objective by embedding recovery conscious techniques into various operating layers to infuse fault resilience and recovery efficiency in the system and its run-time environment.

The unique characteristics of our framework is its three tier architecture and a suite of techniques that can easily be retrofitted even into legacy systems. Figure 2 provides a sketch of our framework. The top tier analyzes the recovery dependencies among concurrent tasks in embedded storage components, identifies the recovery scopes and organizes tasks into recovery groups. The middle tier configures the system resources such as the processors queues through recovery-conscious mapping of recovery groups to system resources. The bottom tier works with a given number of cores and a given number of resource queues and tries to bound recovery time and provide fault resiliency through careful scheduling of resource queues assigned to each recovery group. We show that each tier progressively improves the recovery efficiency of the overall system by embedding recovery consciousness into its various operating layers.

3.1 Fine Grained Group Formation

Transactional recovery in relational DBMSs is a success story of fine-grained error recovery, where the set of operations, their corresponding recovery actions and their recovery scopes are well-defined in the context of database transactions. However, this is not the case in many legacy storage systems. For example, consider the embedded storage controller in which tasks executed by the system are involved in more complex operational semantics, such as dynamic execution paths and complex interactions with other tasks. Under these circumstances, in order to implement task-level recovery, we have to deal with both the semantics of recovery and the identification of recovery scopes.

Recovery from a software failure involves choosing an appropriate strategy to treat/recover from the failure. The choice of recovery strategy depends on the nature of the task, the context of the failure, and the type of failure. For example, within a single system, the recovery strategy could range from continuing the operations (ignoring the error), retrying the operation (fault treatment using environmental diversity) or propagating the fault to a higher layer. In general, with every failure context and type, we could associate a recovery action. In addition, to ensure that the system will return to a consistent state, we must also avoid deadlock or resource hold-up situations by relinquishing resources such as locks, devices or data sets that are in the possession of the task.

Briefly, we refer to the context of a failure as a **recovery point** and provide mechanisms for developers to define **clean-up blocks** which are recovery procedures and re-drive strategies. A clean-up block is associated with a recovery point and encapsulates failure codes, the associated recovery actions, and resource information. The specification of the actual recovery actions in each of the clean-up blocks is left to the developers due to their task-specific semantics. As the task moves through its execution path, it passes through multiple recovery points and accumulates clean-up blocks. When the task leaves a context, the clean-up actions associated with the context go out of scope. The clean-up blocks are gathered and carried along during task execution but are not invoked unless a failure occurs.

In order to characterize the dependencies between tasks we define the concept of **Recovery groups**. A recovery group is the unit of a localized recovery operation, i.e., the set of tasks that undergo recovery concurrently. When recovery procedures are initiated for one task within a recovery group, all tasks within the group that are executing concurrently with the failed task also need to undergo recovery in order to restore the system to a consistent state. We refer to the scope of this localized recovery operation as the recovery scope.

The top tier of our framework provides the capabilities of defining the recovery scope at task level through a careful combination of both the programmer's specification at much coarser granularity and the system-determined recovery scope at finer-granularity. The key issues in the tier one design is how to adequately identify the recovery scopes or boundaries, and how to concretely determine what are the set of tasks that need to undergo recovery upon a failure?

3.2 Mapping Recovery Groups to System Resources

Multi-core processors are delivering huge system-level benefits to embedded applications. An important goal for providing fine-grained recovery (task or component level) is to improve recoverability and make efficient use of resources on the multi-core/SMP architectures. This ensures that resources are available for normal system operation in spite of some localized recovery being underway and that the recovery process is bounded both in time and in resource consumption especially since recovery takes orders of magnitude longer (ranging from milliseconds to seconds) compared to normal operation (μ secs).

The middle tier of the framework is dedicated to the development of highly effective mapping of dependent tasks to

both resource queues and processor resources in order to ensure system availability through reduced recovery time while meeting the performance requirements. Concretely, we identify four recovery-efficiency sensitive parameters: the number of resource queues (i.e. scheduler queues in the current context), number of recovery groups, the type of mapping between recovery groups and resource queues and recoverability constraints. A **recoverability constraint** is specified for each group and prescribes the maximum number of concurrently executing tasks permissible for that group.

Our framework maps recovery groups to processor resources using the scheduler queues, (henceforth referred to as '**RCS queues**') as intermediate data structures. The key challenges in this tier are determining configurations for recovery-efficiency sensitive parameters. A number of factors such as system size, distribution of tasks between recovery groups, task recovery time and variation of service time with the number of scheduler queues are critical in determining the right configurations for recovery-sensitive parameters.

3.3. Recovery-Conscious Scheduling (RCS)

Once the number of resource queues and the number of recovery groups under a given number of cores (processor resources) are determined at the middle tier, the system enters the bottom tier where the decision will be made regarding how to schedule the resource consumption of a recovery group to achieve higher recovery efficiency while maintaining good system performance. Without careful design, it is possible that more dependent tasks are dispatched before a recovery process can complete, resulting 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.

The concept of **resource pools** is used as a method to partition the overall set of resources into a smaller independent units of resources called resource pools. A resource pool is a unit of resource allocation amongst RCS queues and although we restrict ourselves to processing resources in this work, it can be extended to any system resource such as metadata or data replicas. Processors in each resource pool dispatch tasks from the RCS queues assigned to them.

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

Static scheduling of resource pools to recovery groups determines the resource allocation at compile time and thus is independent of the run-time situations and is only effective in situations where task level dependencies with respect to recoverability are well understood and the workloads are stable. Dynamic scheduling of recovery groups to resource pools represents another end of the spectrum by promoting dynamic assignment of resources to recovery groups and works effectively in the presence of frequently changing workloads. Dynamic RCS algorithms is more effective in utilizing resources, but is more costly in terms of scheduling management. Between the two ends of the spectrum are the partially dynamic scheduling algorithms, which utilize partially static scheduling for those recovery groups whose resource consumption demand is stable and well understood and apply dynamic scheduling to the rest of the recovery groups.

Compared to the performance oriented scheduling (POS), which uses a single centralized queue from which tasks are dispatched, under RCS, with "recovery-consciousness" infused into the scheduler, the scheduler enforces some serialization of dependent tasks through recovery groups, thereby controlling the extent of a localized recovery operation within the boundary of a recovery group.

Due to the space constraints, in this paper we focus on the design issues and challenges associated with the middle tier and develop optimal configuration regarding the number of RCS queues and the number of recovery groups for a given number of processing cores. Dynamic RCS algorithms are used as they are are best suited for changing or uncharacterized workload. Tasks are organized into recovery groups with recoverability constraints specified for each group. Under dynamic RCS, all resource pools are mapped to all RCS queues. The scheduler cycles through all RCS queues giving preference to groups that are still within their recoverability bounds. If no such group is found, then tasks are dispatched while trying to minimize resource consumption by any individual recovery group.

4. Recovery-Conscious Mapping

The goal of recovery conscious system configuration involves the analysis and the decision on how to set the recoverability constraints, the number of queues and the number of recovery groups a storage system should have under a specific recovery efficiency and system availability objective. The main idea is to improve system availability by bounding the resource consumption of recovering processes and thereby ensuring the availability of resources to normal operation even during a localized recovery process.

4.1. 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 group fails, all tasks

belonging to the group that are executing concurrently with the failed task t need to undergo recovery.

Let λ_k represent the failure rate and μ_k represent the repair rate for failures in the k^{th} recovery group. 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 group 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:

$$\mu = \alpha_k(1) \times \frac{1}{\mu_k} + \alpha_k(2) \times \frac{2}{\mu_k} + \ldots + \alpha_k(m) \times \frac{m}{\mu_k}$$

Assume that there are R active (i.e. with tasks) recovery groups and let γ_k represent the probability that a task belongs to recovery group 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 group is given by:

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

With performance-oriented scheduling (POS), there is no notion of bounding the recovery process. Interdependent tasks belonging to the same recovery group 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:

$$MTTR_{POS}|_k = \sum_{i=1}^{m} \frac{e^{-\gamma_k m}(\gamma_k m)^i}{i!} \times \frac{i}{\mu_k}$$

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 k^{th} recovery group 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:

$$MTTR_{RCS}|_k = \sum_{i=1}^{c_k} \frac{e^{-\gamma_k c_k}(\gamma_k c_k)^i}{i!} \times \frac{i}{\mu_k}$$

However, with dynamic RCS, a more flexible mapping of resource pools 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. Recall that the number of active recovery groups in the system is denoted by R. Let c_k be the constraint specified on the maximum number of concurrent tasks for the k^{th} recovery group. Without loss of generality we assume that there are idle resources only when $\sum_{i=1}^{R} c_i \leq m$. For the sake of simplicity let us assume that the available spare resources $m - \sum_{i=1}^{R} c_i$ is allocated evenly amongst all groups. Then in the worst case violation of a constraint c_k, denoted as $\overline{c_k}$ is given by:

$$\overline{c_k} = c_k + \left[\frac{(m - \sum_{i=1}^{R} c_i) - (m - \sum_{i=1}^{R} c_i) \mod R}{R} \right] + 1$$

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

Clearly, the system availability under POS is affected by the failure rate λ_k, the repair rate μ_k for failures in the k^{th} recovery group, the number m of processors or cores in the system, and the probability γ_k that a task belongs to recovery group 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 kth recovery group. Our results show that the right choice of constraints for a recovery group given the system size, task recovery time and the distribution of tasks between recovery groups can improve recovery time by an order of magnitude compared to POS which fails to take any of these factors into consideration (see Section 5.3).

While the number of RCS queues do not affect the system time to recovery, the number of RCS queues and degree of serialization do impact the system performance. In general, performance oriented scheduling operates over a single centralized queue. Due to lock contention issues, centralized queue schedulers do not scale beyond a few processors [2]. On the other hand recovery conscious scheduling utilizes multiple queues to which the recovery groups are mapped and from which all processors choose tasks for dispatching. While operating over multiple distributed queues for scheduling reduces lock contention, there by improving scheduler performance, keeping track of recoverability constraints increases scheduler overhead potentially causing an increase in service time.

4.2. Impact of RCS Queues on System Performance

In this section we present analysis that shows the impact of RCS queues on the system performance and based on these results we describe criteria for the selection of RCS queue parameters for efficient scheduling. For the purpose of this analysis we assume that each recovery group is mapped on to a single RCS queue and the serialization constraints enforced on that queue are applied to all recovery groups mapped to it.

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

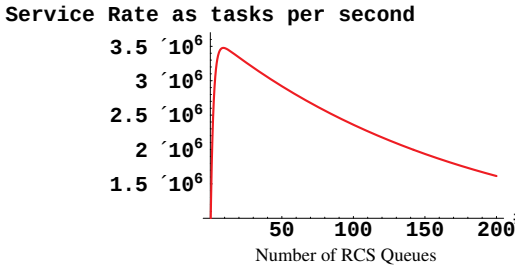

Figure 3. Variation of Service Rate

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 3 shows 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 [18].

Consider a system where tasks arrive as a Poisson process with rate λ_a and service times for all cores are independent, identically distributed random variables. Let the mean service rate for performance oriented scheduling be denoted by μ_{pos} and for recovery conscious scheduling be denoted by μ_{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:

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

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

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

For POS, the value ρ, 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 ρ by ρ_{pos} and ρ_{rcs} respectively in the expressions for $E[N]$ and p_0.

In each case, based on Little's formula [17] the average response time for performance -oriented scheduling ($E_{POS}[R]$) and RCS ($E_{RCS}[R]$) is given by:

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

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:

$$E_{POS}[T] = \mu_{pos}U_0^{pos} \text{ and } E_{RCS}[T] = \mu_{rcs}U_0^{rcs}$$

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 (number of tasks) of the recovery process.

As described in the Section 4.1, 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 group encounters a failure causing in all executing tasks belonging to the k^{th} recovery group to under go recovery. Let f_k^{pos} and f_k^{rcs} denote the extent of the failure-recovery. Let, $\overline{m}_{pos}$ and $\overline{m}_{rcs}$ denote the expected number of cores available for normal operation during failure recovery. Then, as explained in Section 4.1

$$\overline{m}_{pos} = m - f_k^{pos} = m - \sum_{i=0}^{m}\frac{e^{-\gamma_k m}(\gamma_k m)^i}{i!} \times i$$

$$\overline{m}_{rcs} = m - f_k^{rcs} = m - \overline{c}_k$$

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 $\overline{m}_{pos}$ and $\overline{m}_{rcs}$ respectively.

Based on the above analysis we have established that system performance with both POS and RCS is dependent on the number of RCS queues during good-path operation and on the number of recovery groups as well as the number of RCS queues during bad-path.

5. Experimental Results

We have implemented our recovery-conscious scheduling algorithms on a real industry-standard storage system. Our implementation involved no changes to the functional architecture of a legacy application of nearly 2 million lines of code. In this section we present both analytical and experimental results on the impact of several recovery-sensitive parameters such as the number of recovery groups, RCS queues and recoverability constraints on performance and fault resilience. Our results provide valuable insights into the impact and effectiveness of our proposed framework, techniques to tune these critical parameters and show us that through suitable choice of parameters, significant improvements in fault resilience can be

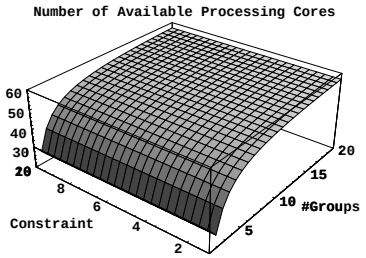

Figure 5. POS: Available Resources in Bad-Path

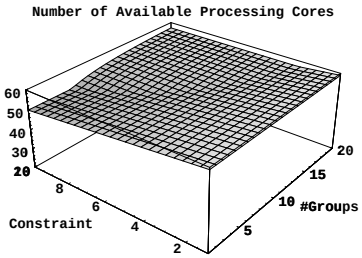

Figure 6. RCS: Available Resources in Bad-Path

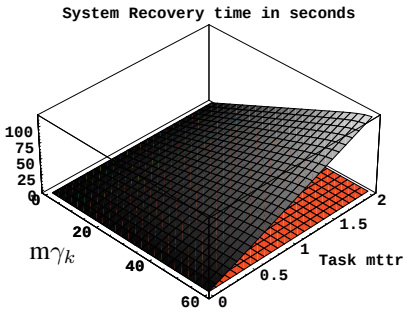

Figure 7. System MTTR

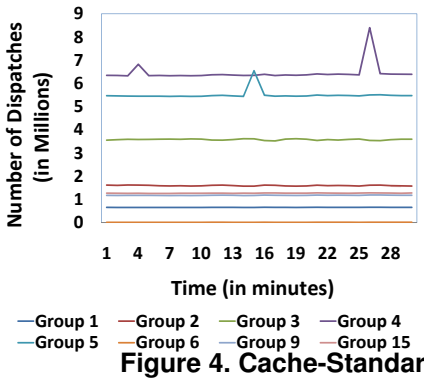

Figure 4. Cache-Standard

achieved while continuing to deliver good performance even during failure recovery.

5.1 Experimental Setup

Our algorithms were 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 Section 2. The system has a number of interacting components which dispatch a large number of short running tasks. For the experiments in this paper we identified 16 recovery groups based on explicit recovery dependency specifications.

5.2 Workload

We use the z/OS Cache-Standard workload to evaluate our algorithms. The z/OS Cache-standard workload 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 was CPU-bound. Figure 4 shows the number of tasks dispatched per-recovery group under the workload over 30 minutes. We use this workload to measure throughput and response times in our prototype experiments. Group 4 has the highest task workload (6.5M tasks/min) followed by group 5 (5M/min). Eight of the groups which have nearly negligible workload are not visible in the graph. Task arrival rate λ_a was set at $3 \times 10^6 tasks/second$. The variation of core/processor service rate with the number of RCS queues was modeled as the hypoexponential distribution shown in Figure 3 and the services rates for all cores were assumed to be identical.

5.3 Effect on System Recovery Time

Figure 5 and Figure 6 show the number of cores that remain available in a 60 core system during failure recovery as the number of recovery groups and constraints (for RCS) vary under POS and RCS respectively. For the purpose of this analysis, it is assumed that the tasks are equally distributed across recovery groups, although this assumption can be relaxed without any modifications to the model and do not affect our conclusions.

Under both approaches, as the number of recovery groups increase, the expected number of outstanding tasks belonging to the failed recovery group at the time of failure decreases. However, for a system with only a small number of recovery groups, with POS, a large number of cores may become unavailable at the time of recovery in the average case. In the worst case, all resources may become unavailable if all outstanding jobs belong to the failing recovery group.

On the other hand, with RCS, more resources remain available even during bad-path operation compared to POS. Figure 6 shows that as the constraints on the maximum concurrent tasks per recovery group become larger, i.e., more lenient, the expected number of available resources during failure recovery decreases, but still remains higher than POS even for small number of recovery groups. As in the case of POS, as the number of recovery groups increase, the expected number of available resources increases.

Figure 7 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 gen-

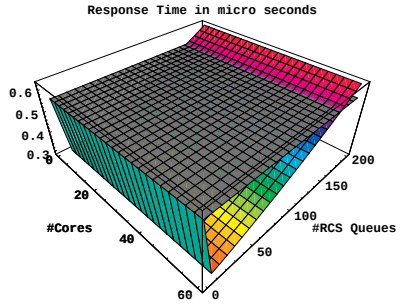

Figure 8. Response time

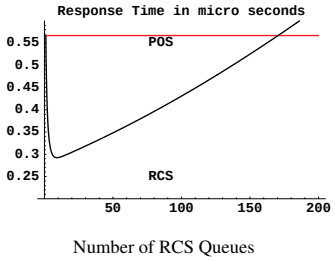

Figure 9. Response time (60 cores)

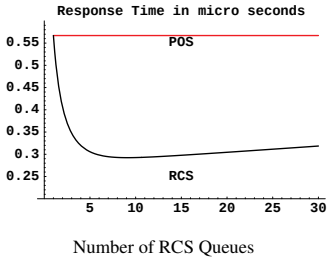

Figure 10. Response time (1-30 RCS Queues)

erated based on the model for $MTTR_{POS}$ and $MTTR_{RCS}$ described in Section 4.1. 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 γ_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.

5.4 Effect on System Response Time

In this section we study the impact of the number of RCS queues on system response time during good path operation. The response time values are based on the analysis presented in Section 4.2. Figure 8 shows the change in response time for POS and RCS with varying number of RCS queues and system size. The upper surface (gray) represents POS and the lower surface (colored) represents RCS. The constraint for RCS was set as $c_k = 10$. Based on this service rate function, Figure 8 depicts the variation in response time with increasing number of queues in the case of RCS as initially dropping and then gradually increasing to eventually exceed that of performance oriented scheduling. On the other hand, POS (upper surface) which operates over a single queue remains nearly constant even with increasing system size. Figure 9 presents a closer picture of the variation of response time with the number of queues when system size is fixed at 60 cores and Figure 10 shows the regions where RCS gains significantly over POS for the same system size.

Two key points to be noted are (1) response time is nearly independent of system size and more significantly influenced by service time and (2) the benefit of RCS to response time during normal operation primarily accrues from the effect of number of scheduler queues on service time.

5.5 Effect on System Throughput

In this section we study the impact of the number of RCS queues on system throughput during good path operation based on the model for throughput developed in Section 4.2.

Figure 11 shows two surfaces - the gray surface represents POS which operates over a single queue and thus remains unaffected by the number of RCS queues. The colored surface represents the variation of throughput with RCS. As the figure shows, in both cases throughput increases with increasing system size. With RCS, throughput initially increases benefiting from the improved service time, but with increasing number of RCS queues gradually drops to eventually become lower than that with POS.

Figures12 and 13 are two dimensional projections of Figure 11 under 60 cores and 30 RCS queues respectively, presented for better understanding of the variation. In Figure 12, throughput with POS remains constant since POS uses a single centralized queue. However, the figures show that for carefully chosen number of RCS queues, significant benefits in throughput can be achieved under the current model.

More importantly, the figures show that it is possible to sustain good throughput during normal operation with RCS, since the primary focus of recovery-consciousness is to ensure system availability *during* failure recovery. Thus, although the models for service rate variation may be different across systems, our numbers prove that in general it is possible to sustain (if not improve) performance even with the overhead of "recovery-consciousness".

5.6 Bad-Path System Performance

Figure 14 and 15 depict the system throughput and response time respectively under bad-path operation, i.e., during failure recovery with a system size of 60 cores. The numbers are based on our analytical models for bad-path performance developed in Section 4.2. The constraint for RCS was assumed to be 10, i.e. $c_k = 10$.

Figure 14 shows the variation of throughput with the number of recovery groups. We assume that each group is mapped to one RCS queue and that tasks are evenly distributed between

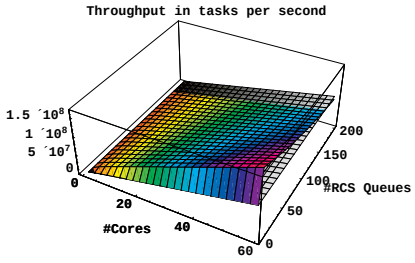

Figure 11. POS vs RCS: Variation of throughput

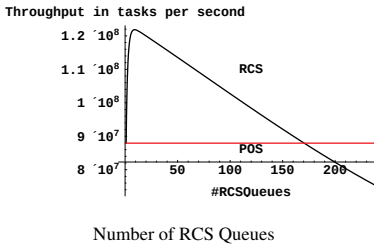

Figure 12. Throughput (60 cores)

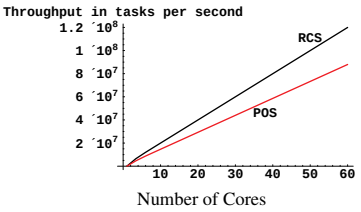

Figure 13. Throughput (30 RCS Queues)

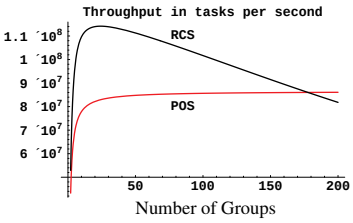

Figure 14. Bad Path Throughput (60 cores)

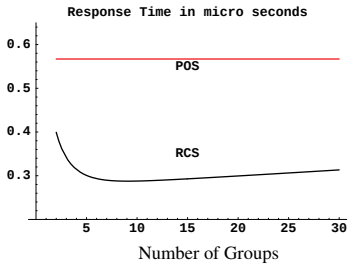

Figure 15. Response Time Bad Path - POS vs RCS

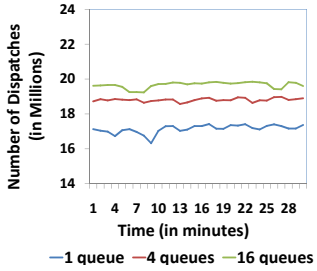

Figure 16. Impact of RCS queues

recovery groups. In a system with only one recovery group, failure of that recovery group corresponds to system wide failure. Thus with both RCS and POS, with a single recovery group, the throughput drops to zero. However, as the number of recovery groups increase, RCS performs significantly better than POS sustaining good path throughput even during failure recovery. For example, with 5 recovery groups, throughput of POS dropped by nearly 17.2% while that of RCS dropped by only 3.3% compared to good path operation.

Recall that with POS, as the number of groups increase, assuming that tasks are distributed evenly across groups, the resilience to failure improves since the probability that a large proportion of tasks belong to the failed recovery group drops. As a result, the performance of POS gradually improves as the number of recovery groups increase (provided tasks are fairly evenly distributed between groups).

Figure 15 represents the variation of response time with the number of groups during bad-path. As the figure shows, RCS retains a significant advantage over POS even during failure recovery. However, with both RCS and POS the response time during bad-path operation is very close to that during normal operation. This can be attributed to the fact that response time is dominated by the service rate factor and is nearly independent of the number of available cores. As a result even when fewer cores are available, the system is able to continue operating with nearly the same response time.

However, note that these numbers still represent only the average case. In the worst case, all cores may become unavailable with POS since no care is taken to bound the resources

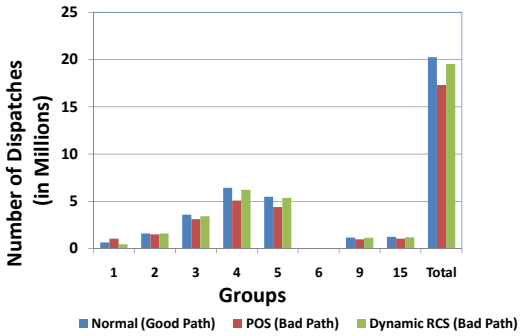

Figure 17. Storage System Task Dispatches: Bad Path

consumed by a single group. However, RCS can ensure both system resilience, i.e. reduce the time to recovery (Figure 7) and ensure good performance (Figures 14- 15).

5.7 Prototype Experiments

We have implemented the proposed recovery conscious techniques in storage controller micro-code. By conducting experiments with our prototype implementation, 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.

Concretely, our techniques were evaluated using the standard Z/OS cache standard workload. In order to understand the impact on system throughput and response time when localized recovery is underway, we inject faults into the cache standard workload. We choose a candidate task belonging to recovery group 5 and introduce faults at a fixed rate (1 for each 10000 dispatches). Recovery from each fault takes approximately 20 ms. On an average this introduces an overhead of 5% to aggregate execution time per minute of the task. During localized recovery, all tasks belonging to the same recovery group that are currently executing in the system and that are dispatched during the recovery process also experience a recovery time of 20 ms each. We measured performance (throughput and latency) averaged over 30 minutes.

Figure 16 shows the average number of task dispatches per minute over 30 minutes with varying number of RCS queues. As the figure shows, the number of dispatches increases (although modestly) with the increase in the number of RCS queues. For instance, when the number of queues increase from 1 to 16, the number of dispatches increase by nearly 13%. This experiment was used to establish the preferred number of RCS queues as 16 for further experimentation.

Figure 17 shows the average number of task dispatches per minute per recovery group and in total under POS and RCS. Based on the previous experiment, the number of queues for RCS was chosen to be 16. 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. Clearly, 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.

6. Discussion

We have presented analytical and experimental study on several recovery-sensitive parameters such as the number of recovery groups, RCS queues and an effective mapping of recovery groups to RCS queues, and show that these parameters not only determine the recovery efficiency and failure resiliency of the system but also impact system performance.

Our analysis and experimental results give us valuable and interesting insights into the impact of the proposed framework and the parameter tuning techniques on the overall system performance. In particular, we see that

- 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.
- Depending upon the distribution of the tasks between recovery groups and the variation of service rate, the decision on the number of queues can impact performance both during normal operation and failure recovery.
- Recovery group granularity has a direct bearing on resource availability during failure recovery and hence system performance during bad-path operation.

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 group is getting finer. The benefit from recovery groups at a given granularity is determined by the following factors: distribution of tasks between groups, task recovery time, and system size. Depending on these parameters we can predict the expected recovery time in the event of a failure. Based on the maximum recovery time, one can determine the unavailability of resources, the bad-path performance that can be tolerated, as well as constraints and granularity of groups.

Depending upon the model for variation of service rate with the number of scheduler queues, the performance impact of the number of scheduler queues varies. In general, beyond a system-specific threshold, the scheduling overhead outweighs the benefit of decreased lock contention. However, lock contention can be serious for a centralized queue. In fact, in our implementation the centralized queue lock had the highest contention accounting for nearly 30% of all lock contentions. In order to effectively enforce recoverability constraints, we must map groups appropriately to queues which are the data structures on which the constraints are actually imposed. A naive method would be to choose as many queues as recovery groups. For example, based on in-depth offline analysis we have identified 200 recovery groups where the single largest group contains 60% of the tasks. However, given the system size of 8 cores, utilizing 200 RCS queues may only hurt performance without delivering significant benefits in recoverability. Thus in order to choose an appropriate number of queues to which groups would be mapped, we must analyze the variation in system performance as described in previous sections of this paper.

By appropriate selection of recovery groups and RCS queues, we would be able to derive maximum benefit from recovery-consciousness. Since our recovery-conscious framework requires minimal changes to the software architecture it can be easily incorporated even in legacy systems. In addition, RCS can match good path performance of performance oriented scheduling and at the same time significantly improve performance under localized recovery and minimize resource unavailability. We believe that the recovery conscious framework and techniques described in this paper represent a promising direction for systems that need to adapt to rapidly growing multi-core architectures.

7. 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 [4], micro-rebooting

sub-components of the system [1], through periodic rejuvenation [8, 3] of the software, or by retrying the operation in a different environment [10]. Error processing techniques are primarily checkpointing and recovery techniques [5], application-specific techniques like exception handling [15] and recovery blocks [12] or more recent techniques like failure-oblivious computing [13].

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 [9], application-specific recovery mechanisms such as recovery blocks [12], and exception handling [15] 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 [14] 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.

Much work in the virtualization context has been focused on improving system reliability [11] 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 [16], we are concerned with the embedded storage software reliability. These techniques are at different levels of the storage system and are complementary.

8. 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 group 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 tasks 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.

References

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

[2] B. Caprita, J. Nieh, and C. Stein. Grouped distributed queues: distributed queue, proportional share multiprocessor scheduling. In *PODC*, 2006.

[3] S. Garg, A. Puliafito, M. Telek, and K. Trivedi. On the analysis of software rejuvenation policies. In *COMPASS*, 1997.

[4] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[5] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, October 1992.

[6] M. Hartung. IBM totalstorage enterprise storage server: A designer's view. *IBM Syst. J.*, 42(2):383–396, 2003.

[7] HP. HSG80 array controller software.

[8] N. Kolettis and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *FTCS*, page 381, Washington, DC, USA, 1995. IEEE Computer Society.

[9] 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*, 17(1):94–162, 1992.

[10] 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.

[11] H. Ramasamy and M. Schunter. Architecting dependable systems using virtualization. In *Workshop on Architecting Dependable Systems in conjunction with DSN*, 2007.

[12] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449. ACM Press, 1975.

[13] 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.

[14] S. Seshadri, L. Chiu, C. Constantinescu, S. Balachandran, C. Dickey, and L. Liu. Enhancing storage system availability on multi-core architectures with recovery-conscious scheduling. In *FAST (to appear)*, 2008.

[15] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh. Using rescue points to navigate software recovery. In *IEEE Symposium on Security and Privacy*, pages 273–280, 2007.

[16] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. *ACM Transactions on Storage (TOS)*, 1(2):133–170, May 2005.

[17] S. StidhamJr. A last word on $l = \lambda w$. In *Operations Research*, volume 22, pages 417–421, Montreal, Que., Canada, 1974. IEEE Computer Society Press.

[18] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.