Enhancing Storage System Availability on Multi-Core Architectures with Recovery-Conscious Scheduling

Sangeetha Seshadri* Lawrence Chiu† Cornel Constantinescu†
Subashini Balachandran† Clem Dickey† Ling Liu* Paul Muench†

*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, pmuench}@us.ibm.com

Abstract

In this paper we develop a recovery conscious framework for multi-core architectures and a suite of techniques for improving the resiliency and recovery efficiency of highly concurrent embedded storage software systems. Our techniques aim at providing continuous availability and performance during recovery while minimizing the time to recovery and the need for rearchitecting the system (legacy code). 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 and dividing the overall system resources into recovery-oriented resource pools, and (2) the development of recovery-conscious scheduling, which enforces some serializability of failure-dependent tasks in order to reduce the ripple effect of software failure and improve the availability of the system. We present three alternative recovery-conscious scheduling algorithms; each represents one way to trade-off between recovery time and system performance. We have implemented and evaluated these recovery-conscious scheduling algorithms on a real industry-standard storage system. Our experimental evaluation results show that the proposed recovery conscious scheduling algorithms are non-intrusive and can significantly improve (throughput by 16.3% and response time by 22.9%) the performance of the system during failure recovery.

1 Introduction

Enterprise storage systems are the foundations of most data centers today and extremely high availability is expected as a basic requirement from these systems. With rapid and exponential growth of digital information and the increasing popularity of multi-core architectures, the demand for large scale storage systems of extremely high availability (moving close to 7 nines) continues to grow. On the other hand, embedded storage software systems (controllers) are becoming much more complex and difficult to test especially given concurrent development and quality assurance processes.

With software failures and bugs becoming an accepted fact, focusing on recovery and reducing time to recovery has become essential in many modern storage systems today. In current system architectures, even with redundant controllers, most microcode failures trigger system-wide recovery [9, 10] 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 using the legacy architecture.

In order to reduce the recovery time and more importantly scale the recovery process with system growth, it is essential to perform recovery at a fine-grained level: recovering only failed components and allowing the rest of the system to function uninterrupted. However, due to fuzzy component interfaces, complex dependencies and involved operational semantics of the system, implementing such fine-grained recovery is challenging. Therefore, firstly we must develop a mechanism to perform fine-grained recovery taking into consideration interactions between components and recovery semantics. Secondly, since localized recovery spans multiple dependent threads in reality, we must bound this localized recovery process in time and resource consumption in order to ensure that resources are available for other normally operating tasks even during recovery. Finally, since we are dealing with a large legacy architecture (> 2M loc), to ensure feasibility in terms of development time and cost we should minimize changes to the architecture.

In this paper we develop a recovery conscious framework for multi-core architectures and a suite of techniques for improving the resiliency and recovery efficiency of highly concurrent embedded storage software systems. Our techniques aim at providing

continuous availability and good performance even during a recovery process by bounding the time to recovery while minimizing the need for rearchitecting the system.

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 and dividing the overall system resources into recovery-oriented resource pools; and (2) 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. We present three alternative recovery-conscious scheduling algorithms, each representing one way to trade-off between recovery time and system performance.

We have implemented and evaluated these recovery-conscious scheduling algorithms on a real industry-standard storage system. Our experimental evaluation results show that the proposed recovery conscious scheduling algorithms are non-intrusive, involve minimal new code and can significantly improve performance during failure recovery thereby enhancing availability.

2 Problem Definition

In this section we motivate this research and illustrate the problem we address by considering the storage controllers of some representative storage system architecture. 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. Although this section discusses specifically about embedded software failures in a storage controller, we believe that most of the concepts may be applicable to other highly concurrent system software too.

2.1 Motivation and Technical Challenges

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

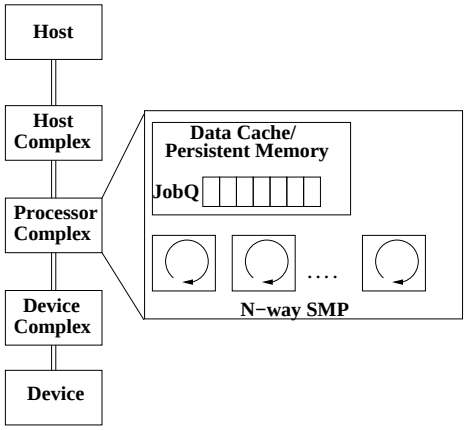

Figure 1: Storage Subsystem Architecture

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. Some storage systems may have more than one job queue (e.g. multiple priority queues).

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**'. 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 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 interactions among tasks and execution paths are determined dynamically.

Many software systems, especially legacy systems, do not satisfy the conditions outlined as essential for micro-rebootable software [3]. For instance, even though the storage software may be reasonably modular, component boundaries, if they exist, are very loosely defined. In addition, the scenario where components are stateful and interact with other components through globally shared structures (data-structures, metadata), often leads to components modifying each other's state irreversibly. Moreover, resources such as hardware and software locks, devices and metadata are shared across components. Under these circumstances, the scope of a recovery action is not limited to a single component.

The discussion above highlights some 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. We will describe our solution to the first two problems – how to implement localized recovery and how to discover efficient recovery scopes in Section 3. We will dedicate Section 4 to address the third problem: how do we bound the recovery process and ensure system availability even during localized recovery?

2.2 Taxonomy of Failures

Studies classify software faults as both permanent and transient. Gray [6] classifies software faults into *Bohrbugs* and *Heisenbugs*. Bohrbugs are essentially deterministic bugs that may be caused due to permanent design failures. Such bugs are usually easily identified during the testing phases and are weeded out early in the software life cycle. On the other hand, 'heisenbugs' which are transient or intermittent faults that occur only under certain conditions are not easily identifiable and may not even be reproducible. Such faults are often due to reasons such as the system entering an unexpected state, insufficient exception handling, boundary conditions, timing/concurrency issues or due to other external factors. Many studies have shown that most software failures occurring in production systems are due to transient faults that disappear when the system is restarted [6, 3, 15].

Our work is targeted at dealing with such transient failures in a storage software system and in particular the embedded storage controller's microcode. Below, we provide a classification of transient failures which we intend to deal with through localized recovery.

In complex systems, often code paths are dynamic and input parameters are determined at runtime. As a result many faults are not caught at compile time. On pure functions, faults may be classified as:

- **Domain errors:** are caused by bad input arguments, such as a divide by zero error or when each individual input is correct, but the combination is wrong (e.g. negative number raised to a non-integral power in a real arithmetic system).
- **Range errors:** are caused when input arguments are correct, but the result cannot be computed (such as a result which would cause an overflow).

With actions based on system state there are additional complexities. For example, a configuration issue that appeared early in the installation process may have been fixed by trying various combinations of actions that were not correctly undone. As a result the system finds itself in an unknown state which manifests as a failure after some period of normal operation. Such errors are very difficult to trace, and although transient may continue to appear every so often. We classify such system state based errors as:

- **State error:** where the input arguments are wrong for the current state of the object.
- **Internal logic error:** where the system has unexpectedly entered an incorrect or unknown state. Such an error often triggers further state

errors.

Each of the above error types can lead to transient failures. Some of the transient failures can be fixed through appropriate recovery actions which may range from dropping the current request to retrying the operation or performing a set of actions that take the system to a known consistent state. For example, some of such transient faults that occur in storage controller code are:

- Unsolicited response from adapter: An adapter (a hardware component not controlled by our microcode) sends a response to a message which we did not send - or do not remember sending. This is an example of a state error.
- Incorrect Linear Redundancy Code (LRC): A control block has the wrong LRC check bytes, for instance, due to an undetected memory error; an example of an internal logic error.
- Queue full: An adapter refuses to accept more work due to a queue full condition; an example of both an internal logic error and state error.

In addition, there are other error scenarios such as violation of a storage system or application service level agreements. The 'time-out' conditions are also very common in large scale embedded storage systems. While the legacy system grows along multiple dimensions, the growth is not proportional along all dimensions. As a result hard-coded constant time-out values distributed in the code base often create unexpected artificial violations.

2.3 Recovery Models

Intuitively we can see that localized recovery may be possible for many of the failure scenarios outlined above, and thus system-wide software reboots can be avoided. Sometimes even for situations of resolving deadlock or livelock, it may be sufficient if a minimal subset of tasks or components of the system undergo restarts (e.g., deadlock resolution in transactional databases [7]). Of course there are scenarios, such as severe memory corruption, where the only high-confidence way of repairing the fault is to perform system-wide clean-up.

In production environments, techniques for fault-tolerance, i.e., coping with the existence and manifestation of software faults can be classified into two primary categories with respect to the fault repairing methods: (1) those that provide *fault treatment*, such as restarts of the software, rebooting of the system and utilizing process pair redundancy; and (2) those that provide *error recovery*, such as checkpointing and log-based recovery. Alternatively, one can categorize the recovery models based on the granularity of the recovery scopes. All the above-mentioned techniques could be applied to any recovery scope. In our context, we consider the following three types of recovery scopes:

- **System level:** Performing fault treatment at this level has proven to be an effective high-confidence way of recovering the system from transient faults [2], but has a high cost in terms of recovery time and the resulting system down-time. On the other hand performing error recovery at the system level through checkpointing and recovery can be prohibitively expensive for systems with very high volumes of workload and complex semantics.
- **Component level:** Both fault treatment and error recovery are more scalable and cost effective at this granularity. For fault treatment, the main challenge is identifying these 'component boundaries' especially in systems that do not have well defined interfaces. Again, the difficult hurdle to performing checkpoint/log-based error recovery at this level is understanding the semantics of operations.
- **Task level:** At this very fine-grained level, the issue of operational semantics still remains. However, performing fault treatment at this level is efficient both in terms of cost and system availability.

The main advantage of performing error recovery or fault-treatment at the task-level as compared to the component-level, is that it allows us to accommodate cross-component interactions and define 'recovery boundaries' in place of 'component boundaries'. Our goal is to handle most of the failures and exceptions through task-level (localized) recovery, and avoid resorting to system-wide recovery unless it is absolutely necessary.

3 Task-level Recovery Framework

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

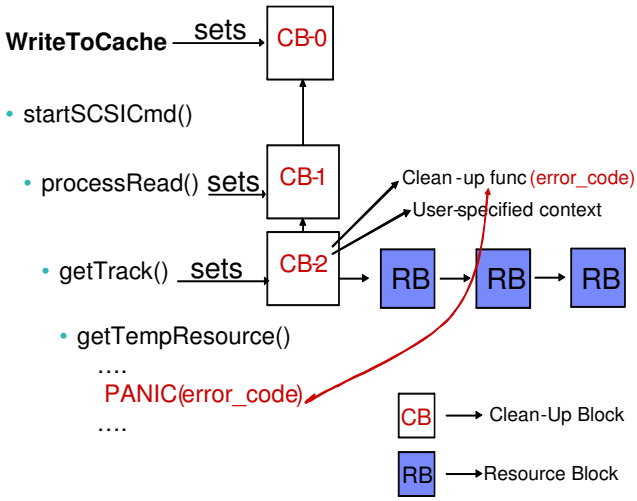

Figure 2: Framework for Task Level Recovery

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 hardware or software locks, devices or data sets that are in the possession of the task.

Bearing these design principles in mind, we develop a two-tier framework for performing task-level recovery through a set of non-intrusive recovery strategies. The top tier 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 bottom tier provides the recovery-conscious scheduling algorithms that balance the performance and the recovery efficiency.

In this framework, 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.

In our implementation, the recovery-conscious scheduler alone was implemented in approximately 1000 lines of code. 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. Note that, the clean-up block does not involve any logging or complex book-keeping and is intended to be light-weight. A more efficient handling of clean-up blocks would involve classifying common error/failure situations and then addressing the handling of the errors in a hierarchical fashion. For example, recoveries may be nested and we could re-throw an error and recover with the next higher clean-up block defined in the stack. This would involve design effort towards the classification of error codes into classes and sub-classes and identification of common error handling situations. Finally, if we are unable to address an error using our framework, existing error handling mechanisms would be used as default. The point of recovery in the stack may be determined by factors such as access to data structures and possibilities of recovery strategies such as retrying, termination or ignoring the error.

Figure 2 shows a schematic diagram of the recovery framework using the call stack of a single task. 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. On the other hand, nesting of contexts results in the nesting of the corresponding clean-up blocks and the framework keeps track of necessary clean-up blocks.

The clean-up blocks are gathered and carried along during task execution but are not invoked unless a failure occurs. Resource information can also be gathered passively. Such a framework allows a choice of recovery strategy based on task requirements and requires minimal rearchitecting of the system.

Example : We describe the selection of recovery strategy and design of clean-up blocks using an example from our storage controller implementation. Consider the error described in Figure 2 which depicts relevant portions of the call stack. The failure situation described in this example is similar to the commonly used 'assert' programming construct. The error is encountered when a task has run out of a temporary cache data structure known as a 'control block' which is not expected to occur normally and hence results in a 'panic'.

In this particular situation, ignoring the error is not a possible recovery strategy since the task would be unable to complete until a control block is available. One possible strategy is to search the list of

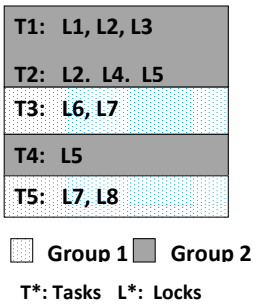

Figure 3: Implicit recovery scopes

control blocks to identify any instances that are not currently in use, but have not been freed correctly (for example, due to incorrect flags). If any such instances exist, they could be made available to the stalled task. An alternative strategy would be to retry the operation beginning at the 'WriteToCache' routine at a later time in order to work around concurrency issues. Retrying the operation may involve rolling back the resource and state setup along this call path to their original state. Resource blocks are used to carry the information required to successfully execute this strategy. Finally, in the case of less critical tasks, aborting the task may also be an option. Alternatively, consider a situation where an error is encountered due to a component releasing access to a track to which it did not have access in the first place. The error was caused due to a mismatch in the number of active users as perceived by the component. In this case, a possible recovery strategy would be to correctly set the count for the number of active users and proceed with the execution, effectively ignoring the error.

Note that, it is important we ensure that the interfaces with the recovery code and the recovery code itself are reliable. The most important issue 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?

3.1 Identifying fine-grained recovery scopes

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. In order to identify the necessary and yet sufficient scope of a recovery action, we need to characterize dependencies between tasks.

Dependencies between tasks may be explicit as in the case of functional dependencies or implicit such as those arising from shared access to stateful structures (e.g., data structures, metadata) or devices. For example, tasks belonging to the same user request may be identified as having the same recovery scope. Likewise, identical tasks belonging to the same functional component may also be marked with the same recovery scope. Explicit dependencies can be specified by the programmer.

However, explicit dependencies specified by the programmer may be very coarse. For example, an 'adapter queue full' error should only affect tasks attempting to write to that adapter and should not initiate recovery across the component. 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. For example, one approach to identifying implicit dependencies at runtime is by observing patterns of hardware and software lock acquisitions. We can group the tasks that share locks into the same recovery scope, since sharing locks typically implies that they have shared access to a stateful global object. Figure 3 illustrates this approach through an example. It shows five tasks and their respective lock acquisition patterns. Tasks T1, T2 and T4 are accessing overlapping sets of locks during their execution and thus are grouped into one recovery scope. Similarly, tasks T3 and T5 are grouped into another recovery scope. Clearly, this approach further refines the developer-specified recovery scope at task level into smaller recovery scopes based on runtime characterization of the dependencies with respect to lock acquisition.

Due to the space limit of this paper, we will omit the detailed development of recovery scope refinement mechanisms. In the remaining part of this paper, we assume that tasks are organized into disjoint recovery scopes refined based on implicit dependencies identified dynamically at runtime. In addition to recovery scopes, we argue that recovery-conscious resource management can be critical for improving system availability during localized recovery.

3.2 Ensuring resource availability

Multi-core processors are delivering huge system-level benefits to embedded applications. Both SMP-based and multi-core systems are very popular in this segment. With the number of processing cores increasing continuously, we argue that the storage software needs to scale both in terms of performance and recoverability to take advantage of the system resources.

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 architectures. This ensures that re-

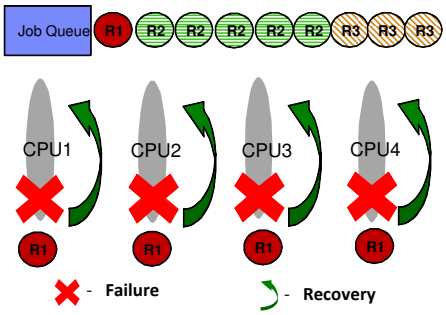

Figure 4: Current Scheduler

sources 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. 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. This problem is aggravated by the fact that recovery takes orders of magnitude longer (ranging from milliseconds to seconds) compared to normal operation ($\sim \mu$ secs). 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.

Ideally we would like to "fence" the failed and recovering tasks until the recovery is complete. In order to do so we must control the number of dependent tasks that are scheduled concurrently, both during normal operation and during recovery. In the next section we discuss how to design a recovery conscious scheduler that can control how many dependent tasks are dispatched concurrently and what measures should be taken in the event of a failure.

4 Recovery-Conscious Scheduling

The goal of recovery-conscious scheduling (RCS) is to ensure system availability even during localized recovery. By recovery-consciousness, we mean that the scheduler must assure availability of resources for normal operation even during a localized recovery process. One way to achieve this objective is to intelligently isolate the recovery process by bounding the amount of resources that will be consumed by the recovering tasks.

Figure 4 shows a performance-oriented scheduling algorithm that does not take recovery dependencies into consideration while scheduling tasks. The diagram shows a 4-way SMP system where each processor independently schedules tasks from the same job queue. This scheduling algorithm aims at maximizing the throughput and minimizing the response time of user requests, which are internally translated by the system into numerous tasks of three types `R1, R2, R3`. The ovals represent tasks and the same shading scheme is used to denote tasks that are dependent in terms of recoverability. As shown in Figure 4, when all CPU resources are utilized for concurrently executing the tasks that have failure/recovery dependencies, then failure and subsequent recovery can consume all the resources of the system, stalling other tasks that could have proceeded with normal operation. Moreover, continuing to dispatch additional dependent tasks before the localized recovery process can be completed only further aggravates the problem of unavailability.

4.1 Recovery Groups and Resource Pools

In order to deal with the problem illustrated in Figure 4, we infuse "recovery consciousness" into the scheduler. Our recovery-conscious scheduler will enforce some serialization of dependent tasks thereby controlling the extent of a localized recovery operation that may occur at any time. To formally describe recovery conscious scheduling, we first define two important concepts: **recovery groups** and **resource pools**.

Recovery Groups: A recovery group is defined as the unit of a localized recovery operation i.e., the set of tasks that will undergo recovery concurrently. When clean-up procedures are initiated for any task within a recovery group, all other tasks belonging to the same recovery group that are executing concurrently will also initiate appropriate clean-up procedures in order to maintain the system in a consistent state. Note that recovery groups are determined based on explicitly specified dependencies that are further refined by the system based on observations of implicit dependencies. By definition, every task belongs to a single recovery group. Thus tasks in the system can be partitioned into multiple disjoint recovery groups.

Resource Pools: The concept of resource pools is used as a method to partition the overall processing resources into smaller independent resource units, called **resource pools**. Although we restrict resource pools in this paper to processors, the concept can be extended to any pool of identical resources such as replicas of metadata or data. Recovery conscious scheduling maps resource pools to recovery groups, thereby confining a recovery operation to the resources available within the resource pool assigned to it.

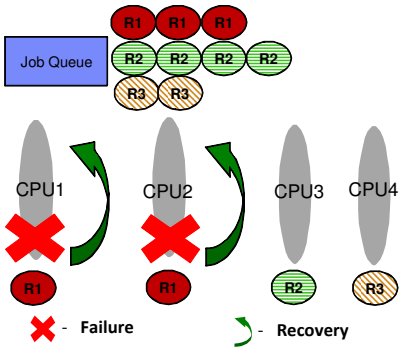

Figure 5: Recovery Oriented Scheduling

4.2 Mapping of Resource Pools to Recovery-Groups

The recovery-conscious scheduling (RCS) algorithms implement the mapping between recovery groups and resource pools. There are different ways that one can map recovery groups to resource pools. The choice of decision depends on the type of trade-offs one would like to make between recovery time and system availability and performance. Static scheduling of resource pools to recovery groups is one end of the spectrum and is only effective in situations where task level dependencies with respect to recoverability are well understood and the workloads of the system is stable. Dynamic scheduling of recovery groups to resource pools represents another end of the spectrum and may better adapt to the changing workload and more effectively utilize resources, but it is more costly in terms of scheduling management. Between the two ends of the spectrum are the partially dynamic scheduling algorithms.

Figure 5 depicts a recovery-conscious scheduler for the same set up as the one used for the performance-oriented scheduler, where tasks are organized into recovery groups – R1 (shaded as fill), R2 (horizontal lines) and R3 (downward diagonal). The processing resources (four CPUs in this example) are organized into three resource pools such that recovery group R1 is mapped to a pool consisting of two processors and recovery groups R2 and R3 are each mapped to a pool consisting of one processor. In case of a failure within group R1, the recovering tasks are now restricted to two of the available four processors so that the other two processors remain available for normal operation. Additionally, the scheduler suspends further dispatching of tasks belonging to group R1 until the localized recovery process completes. This example highlights two aspects of a recovery-conscious scheduler: **proactive** and **reactive**.

Proactive RCS comes into play during normal operation and enhances availability by enforcing some

```
while true do
  repeat
    repeat
      ScanDispatch(HighPriorityQueue)
    until HighPriorityLoopCount
    ScanDispatch(MediumPriorityQueue)
  until MediumPriorityLoopCount
  ScanDispatch(LowPriorityQueue)
end while
```

Figure 6: Qos-based scheduling

```
while true do
  repeat
    repeat
      ScanDispatch(HighPriorityQueue for ρ1)
    until HighPriorityLoopCount
    ScanDispatch(MediumPriorityQueue for ρ1)
  until MediumPriorityLoopCount
  ScanDispatch(LowPriorityQueue for ρ1)
end while
```

Figure 7: Recovery conscious scheduling

degree of serialization of dependent tasks. The goal of proactive scheduling is to reduce the impact of a failure by trying to bound the number of outstanding tasks per recovery group. Then in the event of a failure within any recovery group, the number of tasks belonging to that recovery group that are currently executing and need to undergo recovery are also controlled. By limiting the extent of a recovery process, proactive scheduling can help the system recover sooner, and at the same time, it controls the amount of resources dedicated to the recovery process. Proactive RCS thereby ensures resource availability to normal operation even during a localized recovery process.

The reactive aspect of recovery conscious scheduling takes over *after a failure has occurred*. When localized recovery is in progress, reactive RCS suspends the dispatch of tasks belonging to the group undergoing recovery until the recovery completes. This ensures quick completion of recovery by preventing transitive expansion of the recovery scope and avoiding deadlocks.

4.3 System Considerations

The deployment of recovery conscious scheduling in practice requires the design and implementation of the scheduler to meet the stringent performance requirements of the storage system, sustaining the desired high throughput and low response time. Put differently, recovery-conscious scheduling should offer comparable efficiency in throughput and latency as those provided by performance oriented scheduling.

We outline below some factors that must be taken into consideration while comparing recovery conscious scheduling with performance oriented

scheduling in a multi-core/SMP environment.

Note that our scheduling algorithms are concerned with partitioning resources between tasks belonging to different "components" of the same system which adds a second orthogonal level to the scheduling problem. We continue to respect the QoS or priority considerations specified by the designer at the level of user requests. For example, Figure 6 shows an existing QoS based scheduler using high, medium and low priority queues. Figure 7 shows how recovery-conscious scheduling used by a pool ρ_1 dispatches jobs based on both priority and recovery-consciousness (by picking jobs only from the recovery groups assigned to it).

We use *good-path* and *bad-path* performance as the two main metrics for comparison of the recovery-conscious schedulers with performance oriented schedulers. By 'good-path' performance we mean the performance of the system during normal operation. We use the term 'bad-path' performance to refer to the performance of the system under localized failure/recovery.

Both good path and bad path performance can be measured using end-to-end performance metrics such as throughput and response time. In addition, we can also measure the performance of a scheduler from system-level factors, including cpu utilization, number of tasks dispatched over time, queue lengths, the overall utilization of other resources such as memory, and the ability to meet service level agreements and QoS specifications.

5 Classification of RCS Algorithms

We classify recovery conscious scheduling (RCS) algorithms based on the method in which resource pools are distributed across recovery groups. As discussed in the previous section, we categorize recovery-conscious scheduling algorithms into three classes: static, partially dynamic, and fully dynamic. This classification represents varying degrees of trade-offs between fault isolation and performance, ranging from static mappings which emphasize recoverability over performance, to different ways of balancing between recoverability and performance, to a completely dynamic mapping of resources to recovery groups, which maximizes the utilization of resources while trying to meet recovery constraints.

In order to provide a better understanding of the design philosophy of our recovery-conscious scheduling, we devise a *running example scenario* that is used to illustrate the design of all three classes of RCS algorithms. This running example has five resource pools: $\rho_1, \rho_2, \rho_3, \rho_4$ and ρ_5 and four recovery groups: $\gamma_1, \gamma_2, \gamma_3$ and γ_4. We use σ_i to denote the *recoverability constraint* for the recovery group

Recovery Groups	γ_1	γ_2	γ_3	γ_4
% of Workload	40%	20%	20%	20%
Recoverability constraints (σ_i)	2	1	1	1

Table 1: Recovery constraints

Recovery Groups	γ_1	γ_2	γ_3	γ_4
Resource Pools	ρ_1, ρ_2	ρ_3	ρ_4	ρ_5

Table 2: Static mapping

γ_i. Constraint σ_i specifies the upper limit on the amount of resources (processors in this case) that can be dedicated to the recovery group γ_i ($1 \leq i \leq 4$ in our running example). Since we are concerned with processing resources in this paper, it also indicates the number of tasks belonging to a recovery group that can be dispatched concurrently. The recoverability constraint σ_i is determined based on both the recovery group workload i.e., the number of tasks dispatched, and the observed task-level recovery time. Although recoverability constraints are specified from the availability standpoint, they must take performance requirements into consideration in order to be acceptable. Recoverability constraints are primarily used for proactive RCS.

For ease of exposition we assume that all resource pools are of equal size (1 processor each). Table 1 shows the workload distribution between the recovery groups and the recoverability constraint per group, where two processors are assigned to the recovery group γ_1 and one processor is assigned to each of the remaining three groups.

In contrast to the scenario in Table 1 where no resource pools are shared by two or more recovery groups, when more than one recovery group is mapped to a resource pool the scheduler must ensure that the dispatching scheme does not result in starvation. By avoiding starvation, it ensures that the functional interactions between the components are not disrupted. For example in our implementation we used a simple round-robin scheme for each scheduler to choose the next task from different recovery groups sharing the same resource pool. Other schemes such as those based on queue lengths or task arrival time are also appropriate as long as they avoid starvation.

5.1 Static RCS

Static recovery conscious scheduling algorithms construct static mappings between recovery groups and resource pools. The initial mapping is provided to the system based on the observations of the workload and known recoverability constraints, such as previously observed localized recovery times. The

mappings are static in the sense that they do not continuously adapt to changes in resource demands and workload distribution. Table 2 shows a mapping between the pools $\rho_1 \ldots \rho_5$ and the recovery groups $\gamma_1 \ldots \gamma_4$. This mapping assigns resource pools to recovery groups based on the workload distribution and the recoverability constraints given in Table 1. Concretely, in this mapping recovery group γ_1 is mapped to two pools ρ_1 and ρ_2. Similarly groups γ_2, γ_3 and γ_4 are each assigned a single resource pool. Each processor dispatches work only from its assigned recovery group.

This approach aims at achieving strict recovery isolation. As a result, it loses out on utilization of resources, which in turn impacts both throughput and response time. Although this is a naive approach to performing recovery-conscious scheduling it helps us in understanding issues related to the performance and recoverability trade-off. Note that all our RCS algorithms avoid starvation by using a round-robin scheme to cycle between recovery groups sharing the same resource pool. In systems where the workload is well understood and sparse in terms of resource utilization, static mappings offer a simple means of achieving serialization of recovery dependent tasks.

Implementation Considerations: There are two main data structures that are common to all RCS algorithms: (1) the mapping tables and (2) the job queues. Mapping table implementations keep track of the list of recovery groups assigned to each resource pool. They also keep track of groups that are currently undergoing recovery for the purpose of reactive scheduling. In our system we used a simple array-based implementation for mapping tables.

There are a couple of options for implementing job queues. Recall that recovery-consciousness is built on top of the QoS or priority based scheduling. We could use multiple QoS based job queues (for example, high, medium and low priority queues) for each pool or for each group. In our first prototype, we chose the latter option and implemented multiple QoS based job queues for each recovery group for a number of reasons. Firstly, this choice easily fits into the scenario where a single recovery group is assigned to multiple resource pools. Secondly, it offers greater flexibility to modify mappings at runtime. Finally, reactive scheduling (i.e., suspending dispatch of tasks belonging to a group undergoing localized recovery) can be implemented more elegantly as the resource scheduler can simply skip the job queues for the recovering group. Enqueue and dequeue operations on each queue are protected by a lock. An additional advantage of a mapping implemented using multiple independent queues is that it reduces the degree of contention for queue locks.

Recovery Groups	γ_1	γ_2	γ_3	γ_4
Resource Pools	All	All	ρ_4	ρ_5

Table 3: Partial Dynamic RCS: Alternative mapping

```
...
repeat
   workFound := false
   for γi in current mapping do
      workFound := ScanDispatch(HighQueue for γi)
      if workFound then
         break
      end if
   end for
   if !workFound then
      AcquireLease()
      for γj in alternative mapping do
         workFound := ScanDispatch(HighQueue for γj)
         if leaseExpired() OR workFound then
            break
         end if
      end for
   end if
until HighPriorityLoopCount
//Similarly for Medium and Low Priority tasks
```

Figure 8: Partial Dynamic RCS

5.2 Partial dynamic RCS

The second class of algorithms are partially dynamic and allow the recovery-conscious scheduler to react (in a constrained fashion though) to sudden spikes or bursty workload of a recovery group.

The main drawback of static RCS is that it results in poor utilization of resources due to the strictly fixed mapping. Partial dynamic RCS attempts to alleviate this problem by using a relatively more flexible mapping of resources to recovery groups, allowing groups to utilize spare resources. Partially dynamic RCS algorithms begin with a static mapping. However, when the utilization is low, the system switches to an alternative mapping that redistributes resources across recovery groups.

For example, with the static mapping of Table 2 with changing distribution of workloads, resources allocated to recovery groups γ_3 and γ_4 may be under utilized while groups γ_1 and γ_2 may be swamped with work. Under these circumstances, the system switches to an alternative mapping shown in Table 3. Now groups γ_1 and γ_2 can utilize spare resources across the system even if this may mean potentially violating their recoverability constraints specified in Table 1. Note that γ_3 and γ_4 still obey their recoverability constraints. In summary, partially dynamic mappings allows the flexibility of selectively violating the recoverability constraints when there are spare resources to be utilized, whereas static mappings strictly obey recoverability constraints.

The aim of the partial dynamic mapping is to improve utilization over static schemes by opening up spare resources to recovery groups with heavy workloads. With the above example although there

is a danger of a single recovery group (for e.g., γ_1) running concurrently across all resource pools, note that this is highly unlikely if other groups have any tasks enqueued for dispatching. There are multiple combinatorial possibilities in designing alternative mappings for partially-dynamic schemes. The choice of which components should continue to stay within their recoverability bounds is to be made by the system designer using prior information about individual component vulnerabilities to failures.

Implementation Considerations: There are two implementation considerations that are specific to the partially dynamic scheduling schemes: (1) the mechanism to switch between initial schedule and an alternative schedule, and (2) the mapping of recovery group tasks to the shared resource pools.

We use a lease expiry methodology to flexibly switch between alternative mappings. Note that the pool schedulers switch to the alternative mapping based on the resource utilization of the current pool. With the partially dynamic scheme, the alternative mappings are acquired under a lease, which upon expiry causes the scheduler to switch back to the original schedule. The lease-timer is set based on observed component workload trends (such as duration of a burst or spike) and the cost of switching. For example, since our implementation had a very low cost of switching between mappings, we set the lease-timer to a single dispatch cycle. Figure 8 shows the pseudo-code for a partial dynamic scheduling scheme using a lease expiry methodology. For the sake of simplicity we do not show the tracking method (round-robin) used to avoid starvation in the scheduler.

Recall from the implementation considerations for the static mapping case that we chose to implement job queues on a per recovery group basis. This allowed for easy switching between the current and alternative mapping which only involves consulting a different mapping table. Task enqueue operations are unaffected by the switching between mappings.

5.3 Dynamic RCS

Dynamic recovery-conscious scheduling algorithms assign recovery groups to resource pools at runtime. In dynamic RCS, tasks are still organized into recovery groups with recoverability bounds specified for each group. However, all resource pools are mapped to all recovery groups. The schedulers cycle through all groups giving preference to groups that are still within their recoverability bounds, i.e., occupying fewer resources than specified by the bound. If no such group is found, then tasks are dispatched while trying to minimize the resource consumption by any individual recovery group.

This class of algorithms aim at maximizing utilization of resources at the cost of selectively violating the recoverability constraints. Note that all recovery-conscious algorithms are still designed to perform reactive scheduling, i.e., suspend the dispatching of tasks whose group is currently undergoing localized recovery. The aspect that differentiates the various mapping schemes is the proactive handling of tasks to improve system availability. The dynamic scheme can be thought of as trying to use load balancing among recovery groups in order to achieve both recovery isolation and good resource utilization.

Implementation Considerations: A key implementation consideration specific to dynamic RCS is the problem of keeping track of the number of outstanding tasks belonging to each recovery group. We maintain this information in a per-processor data structure that keeps track of the current job.

Recall that implementing job queues on the per recovery-group basis helps us implement dynamic mappings efficiently and flexibly. One of the critical optimizations for dynamic RCS algorithms involves understanding and mitigating the scheduling overhead imposed by the dynamic dequeuing process. In on going work we are conducting experiments with different setups to characterize this overhead. However our results in this paper show that even with the additional scheduling cost dynamic RCS schemes perform very well both under good-path and bad-path conditions.

6 Experiments

We have implemented our recovery-conscious scheduling algorithms on a real industry-standard storage system. Our implementation involved no changes to the functional architecture. Our results show that dynamic RCS can match performance oriented scheduling under good path conditions while significantly improving performance under failure recovery.

6.1 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

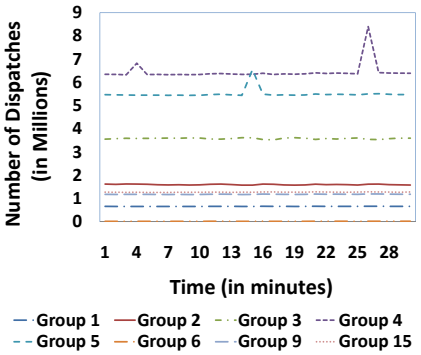

Figure 9: Cache-Standard

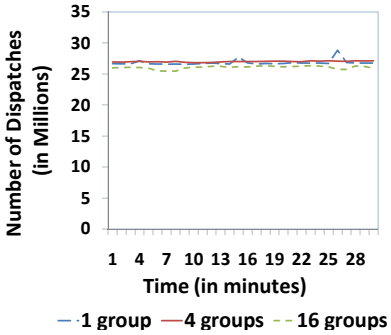

Figure 10: Efficiency vs Recovery groups

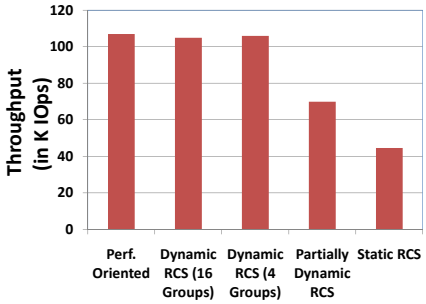

Figure 11: Good path throughput

management consoles. The system is designed to optimize both response time and throughput. The basic strategy employed to support continuous availability is the use of redundancy and highly reliable components.

The embedded storage controller software is similar to the model presented in Section 2. The software is also highly-reliable with provisions for quick recovery(under ∼6 seconds) at the system-level. The system has a number of interacting components which dispatch a large number of short running tasks. For the experiments in this paper based on programmer explicit recovery dependency specifications, we identified 16 recovery groups which roughly correspond to functional components such as cache manager, device manager, SCSI command processor etc. However, some recovery groups may perform no work in certain workloads possibly due to features being turned off. We chose a pool size of 1 CPU which resulted in 8 pools of equal size. The system already implements high, medium and low priority job queues. Our recovery-conscious scheduling implementation therefore uses three priority based queues per recovery group. For the partially dynamic case, based on the workload we have identified two candidates for strict isolation - groups 4 and 5. For the static mapping case each recovery group is mapped to resource pools proportional to its ratio of the total task workload.

6.2 Workload

We use the z/OS Cache-Standard workload [1, 14] 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 9 shows the number of tasks dispatched per-recovery group under the workload over 30 minutes. 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. We use this workload to measure throughput and response times. While measuring cpu utilization we only count time actually spent in task execution and do not include time spent acquiring queue locks, dequeuing jobs or polling for work.

6.3 Experimental Results

We compare RCS and performance oriented scheduling algorithms using good-path (i.e. normal condition) and bad-path (under failure recovery) performance.

6.3.1 Effect of additional job queues

We first performed some benchmarking experiments to understand the effect of additional job queues on the efficiency of the scheduler. Using the cache-standard workload, we measured the aggregate number of dispatches per minute with varying number of recovery groups - 16, 4 and 1 (which is identical to performance-oriented scheduling) to measure scheduler efficiency with dynamic RCS. The four and one recovery group cases were implemented by collapsing multiple groups into a single larger group. Recall that each recovery group results in three job queues for high, medium and low priority jobs. Figure 10 shows the aggregate number of tasks dispatched per minute with 1, 4 and 16 recovery groups. As the figure shows the number of dispatches are almost identical in the three cases (+/- 2%). Although more job queues imply having to cycle through more queue locks while dispatching work, increasing the number of job queues reduces contention for queue locks both when enqueuing and dequeuing tasks. For most of the experiments in this paper we choose a configuration with 16 recovery groups.

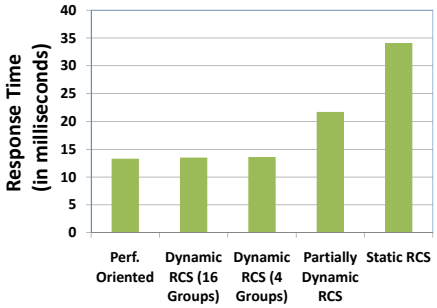

Figure 12: Good path latency

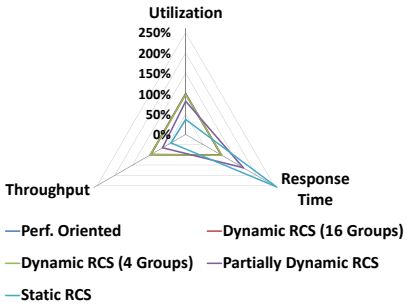

Figure 13: CPU utilization

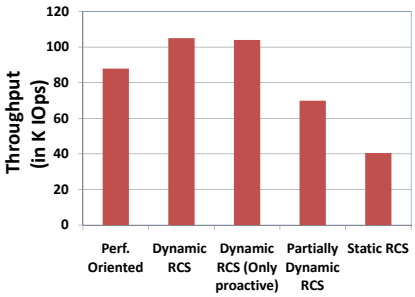

Figure 14: Bad path throughput

6.3.2 Good-path Performance

Recovery-conscious scheduling can be an acceptable solution only if it is able to meet the stringent performance requirements of a high-end system. In this experiment we compare the good-path (i.e. under normal operation) performance of our RCS algorithms with the existing performance-oriented scheduler.

Figure 11 shows the good-path throughput for the performance-oriented and recovery-conscious scheduling algorithms. The average throughput for the dynamic RCS case with 16 groups (105 KIOps) and 4 groups (106 KIOps) was close to that for the performance-oriented scheduler (107 KIOps). On the other hand, with partially dynamic RCS, the system throughput drops by nearly 34% ($\sim$ 69.9 KIOps), and with static RCS by nearly 58% ($\sim$ 44.6 KIOps) compared to performance oriented scheduling.

Figure 12 compares the response time with different RCS schemes and performance-oriented scheduling. Again, the average response-time for the dynamic RCS case with 16 recovery groups (13.5 ms) and 4 recovery groups (13.6 ms) is close to the performance-oriented case (13.3 ms). However, with the partially-dynamic RCS scheduling, the response time increases by nearly 63% (21.7 ms) and by 156% with static RCS (34.1ms).

Both the throughput and response time numbers can be explained using the next chart, Figure 13. The radar plot shows the relationship between throughput, response time and cpu-utilization for each of the cases. As the figure shows, the cpu utilization has dropped by about 19% with partially dynamic RCS and by 63% with static RCS. The reduction in cpu utilization eventually translates to reduced throughput and increased response time in a cpu-bound workload intensive environment. These numbers seem to indicate that in such an environment schemes that reduce the utilization can result in significant degradation of the overall performance. However note that the normal operating range of many customers may be only around 6-7000 IOps [22]. If that be the case, then partially-dynamic schemes can more than meet the system requirement even while ensuring some recovery isolation.

6.3.3 Bad-path Performance

Next, we compare RCS algorithms with performance-oriented scheduling under bad-path or failure conditions. 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 was emulated and recovery from each fault was set to take 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 a 30 minute run.

In the case of recovery-conscious scheduling algorithms, reactive scheduling kicks in when any group is undergoing recovery. Under those circumstances, tasks belonging to that recovery group already under execution are allowed to finish, but further dispatch from that group is suspended until recovery completes.

Figure 14 shows the average system throughput with fault injection. The average throughput using only performance oriented scheduling (87.8 KIOps) drops by nearly 16.3% when compared to dynamic RCS (105 KIOps) that also uses reactive policies. On the other hand dynamic RCS continues to deliver the same throughput as under normal conditions. Note that this is still not the worst case for performance oriented scheduling. In the worst case, all resources may be held up by the recovering tasks

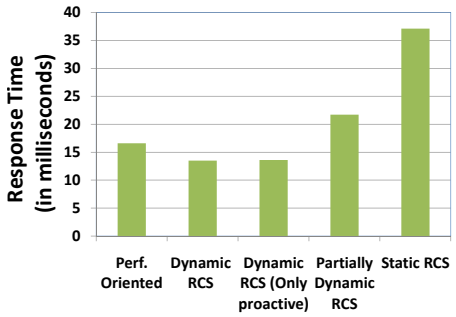

Figure 15: Bad path latency

resulting in actual service outage and the problem would only worsen with increasing localized recovery time and system size.

The figure also compares proactive and reactive policies in dynamic RCS. The results show that with only proactive scheduling we are able to sustain a throughput (104 KIOps) which is just $\sim 1\%$ less than that using both proactive and reactive policies (105 KIOps).

The graph also compares partially dynamic RCS (69.9 KIOps) and static RCS (40.4 KIOps). While these schemes are able to sustain almost the same throughput as they do under good path, overall, the performance of these schemes results in 20% and 54% drop in throughput respectively compared to performance oriented scheduling.

Figure 15 compares the latency under bad-path code with different scheduling schemes. Compared to dynamic RCS (13.5 ms), performance oriented scheduling (16.6 ms) results in a 22.9% increase in response time. At the same time, even without reactive scheduling, dynamic RCS (13.6 ms with only proactive) increases response time by only 0.7%. Again, partially dynamic RCS (21.7ms) and static RCS (37.1 ms) result in latency close to their good path performance but which is still too high when compared to dynamic RCS.

We performed experiments with other configurations of dynamic, partially dynamic and static schemes and using other workloads too. However due to space constraints we only present key findings from those experiments. In particular we used a disk-bound internal workload (and hence low cpu utilization of about $\sim$25%) to study the effect of our scheduling algorithms under a sparse workload. We used the number of task dispatches as a metric of scheduler efficiency. The fault injection mechanism was similar to the cache-standard workload, however due to the workload being sparse, we introduced an overhead of only 0.3% to the aggregate execution time of the faulty recovery group. Our results showed that dynamic RCS was able to achieve as many dispatches as performance oriented scheduling under good path operation and increase the number of dispatches by 0.7% under bad-path execution. With partial dynamic RCS dispatches dropped by 20% during good path operation and by only 3.9% during bad path operation compared to performance oriented scheduling. The same static mapping used in the cache standard workload when run in this new environment resulted in the system not coming up. While this may be due to setup issues, it is also likely that insufficient resources were available to the platform tasks during start-up. We are investigating further on a more appropriate static mapping for this environment.

6.4 Discussion

The fact that recovery-conscious scheduling requires minimal change to the software allows for it to be easily incorporated even in legacy systems.

Dynamic RCS can match good path performance of performance oriented scheduling and at the same time significantly improve performance under localized recovery. Even for the small 5% recovery overhead introduced by us, we could witness a 16.3% improvement in throughput and a 22.9% improvement in response time with dynamic RCS. Moreover, the qualitative benefits of RCS in enhancing availability and ensuring that localized recovery is scalable with system size makes it an interesting possibility as systems are moving towards more parallel architectures. Our experiments with various scheduling schemes (some not reported in this paper) have given us some insights into the overhead costs such as lock spin times imposed by RCS algorithms. In ongoing work we are continuing to characterize and investigate further optimizations to RCS schemes.

Our results also seem to indicate that for small localized recovery time and system sizes, proactive policies i.e. mapping resource pools to recovery groups, can deliver the advantage of recovery-consciousness. However as system size increases or localized recovery time increases, we believe that the actual benefits of reactive policies such as suspending dispatch from groups undergoing recovery may become more pronounced. In ongoing research we are experimenting with larger setups and longer localized recovery times.

Static and partial dynamic RCS schemes are limited by their poor resource utilization in workload intensive environments. Hence we do not recommend these schemes in an environment where the system is expected to run at maximum throughput. However, the tighter qualitative control that these schemes offer may make them, especially partially dynamic RCS, more desirable in less intensive environments where there is a possibility to over-

provision resources, or when the workload is very well understood. Besides in environments where it 'pays' to isolate some components of the system from the rest such mappings may be useful. We are continuing research on optimizing these algorithms and understanding properties that would prescribe the use of such static or partially dynamic schemes.

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 [6, 24], micro-rebooting sub-components of the system [2], through periodic rejuvenation [13, 5] of the software, or by retrying the operation in a different environment [17]. Error processing techniques are primarily checkpointing and recovery techniques [7], application-specific techniques like exception handling [21] and recovery blocks [19] or more recent techniques like failure-oblivious computing [20].

In general our recovery conscious approaches are complementary to the above techniques. However we are faced with several unique challenges in the context of embedded storage software. First, the software being legacy code rules out re-architecting the system. Second, the tight coupling between components makes both micro-reboots and periodic rejuvenation tricky. Rx [17] 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.

Failure-oblivious computing [20] introduces a novel method to handle failures - by ignoring them and returning possibly arbitrary values. This technique may be applicable to systems like search engines where a few missing results may go unnoticed, but is not an option in storage controllers.

The idea of localized recovery has been exercised by many. Transactional recovery using checkpointing/logging methods is a classic topic in DBMSs [16] and is a successful implementation of fine-grained recovery. In fact application-specific recovery mechanisms such as recovery blocks [19], and exception handling [21] are used in almost every software system. However, very few have made an effort on understanding the implications of localized recovery on system availability and performance in a multi-core environment where interacting tasks are executing concurrently. Likewise, the idea of recovery-conscious scheduling is to raise the awareness about localized recovery in the resource scheduling algorithms to ensure that the benefits of localized recovery actually percolate to the level of system availability and performance visible to the user. 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 [25, 11, 12, 8, 4]. Also 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 by identifying fine-grained recovery scopes. Compared to earlier work on improving storage system availability at the RAID level [23], we are concerned with the embedded storage software reliability. These techniques are at different levels of the storage system and are complementary.

8 Conclusion and Future Work

In this paper we presented a recovery conscious framework for multi-core architectures and techniques for improving the resiliency and recovery efficiency of highly concurrent embedded storage software. Our main contributions include a task-level recovery model and the development of recovery-conscious scheduling, a non-intrusive technique to reduce the ripple effect of software failure and improve the availability of the system. We presented a suite of RCS algorithms and quantitatively evaluated them against performance oriented scheduling. Our evaluation showed that dynamic RCS can significantly improve performance under failure recovery while matching performance oriented scheduling during normal operation.

In order to adopt RCS for large software systems, a significant challenge is to identify efficient recovery scopes. In ongoing work we are working on developing more generic guidelines that would assist in identifying fine-grained recovery scopes. Even with pluggable mechanisms like RCS it is necessary to emphasize that high-availability should still be a design concern and not an after-thought. We hope our framework would encourage developers to incorporate additional error handling and anticipate more error scenarios and that our scheduling schemes would aid in scaling efficient error handling with system size.

9 Acknowledgments

The authors would like to express their gratitude to David Whitworth, Andrew Lin, Juan Ruiz (JJ),

Brian Hatfield, Chiahong Chen and Joseph Hyde for helping us perform experimental evaluations and interpret the data. We would also like to thank K.K.Rao, David Chambliss, Brian Henderson and many others in the Storage Systems group at IBM Almaden Research Center who provided us with the resources to perform our experiments and provided valuable feedback. We thank our anonymous reviewers, our shepherd Dr. Mary Baker and Prof. Karsten Schwan for the useful comments and feedback that have helped us improve the paper.

This work is partially supported by an IBM PhD scholarship and an IBM Storage Systems internship for the first author, and the NSF CISE IIS and CSR grants, an IBM SUR grant, as well as an IBM faculty award for the authors from Georgia Tech.

References

[1] Ibm z/architecture principles of operation. *SA22-7832, IBM Corporation*, 2001.

[2] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: a soft-state system case study. *Perform. Eval.*, 56(1-4):213–248, 2004.

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

[4] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *SPAA*, pages 105–115, New York, NY, USA, 2007. ACM Press.

[5] S. Garg, A. Puliafito, M. Telek, and K. Trivedi. On the analysis of software rejuvenation policies. In *Proceedings of the 12th Annual Conference on Computer Assurance (COMPASS'97)*, 1997.

[6] 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.

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

[8] A. Gulati, A. Merchant, and P. J. Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. In *SIGMETRICS*, pages 13–24, New York, NY, USA, 2007. ACM Press.

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

[10] HP. HSG80 array controller software.

[11] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Symposium on Operating Systems Principles*, pages 117–130, 2001.

[12] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage*, 1(4):457–480, 2005.

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

[14] L. LaFrese. Ibm totalstorage enterprise storage server model 800 new features in lic level 2.3.0 (performance white paper). *ESS Performance Evaluation, IBM Corporation*, 2003.

[15] I. Lee and R. K. Iyer. Software dependability in the tandem guardian system. *IEEE Trans. Softw. Eng.*, 21(5):455–467, 1995.

[16] 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 Trans. Database Syst.*, 17(1):94–162, 1992.

[17] 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.

[18] H. Ramasamy and M. Schunter. Architecting dependable systems using virtualization. In *Workshop on Architecting Dependable Systems in conjunction with 2007 International Conference on Dependable Systems and Networks (DSN-2007)*, 2007.

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

[20] 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*, Berkeley, CA, USA, 2004. USENIX Association.

[21] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh. Using rescue points to navigate software recovery. In *SP*, pages 273–280, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Sirius. Sirius enterprise systems group disk drive storage hardware. *Solicitation EPS050059-A4*.

[23] 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.

[24] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. *FTCS*, pages 2–9, 1991.

[25] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. *Trans. Storage*, 2(3):283–308, 2006.