Performance Availability for Networks of Workstations

by

Remzi H. Arpaci-Dusseau

B.S.C.E (University of Michigan, Ann Arbor) 1993
M.S. (Univeristy of Califiornia, Berkeley) 1996

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor David A. Patterson, Chair
Professor David E. Culler
Professor Joseph M. Hellerstein
Doctor James Gray
Professor Marti Hearst

1999

The dissertation of Remzi H. Arpaci-Dusseau is approved:

Chair

Fall 1999

Performance Availability
for Networks of Workstations

Abstract

Performance Availability
for Networks of Workstations

by

Remzi H. Arpaci-Dusseau

Software systems for large-scale distributed and parallel machines are difficult to build. When run in dynamic, production environments, not only must such systems perform correctly, but they must also operate with high performance. Much of the previous work in distributed computing has addressed the design of large-scale systems that function correctly, in spite of correctness faults of individual components [18, 49, 82, 86]. However, there has been little development of techniques to tolerate *performance faults* – unexpected performance fluctuations from the components that comprise the system. Due to this shortcoming, many systems are overly sensitive to performance variations, in that global performance is high if and only if all system components perform exactly as expected.

In this dissertation, we address this deficiency by formalizing the concept of *performance availability*. Our hypothesis is that modern software systems must provide mechanisms to enable performance availability. Without such mechanisms, global system performance is likely to be unpredictable and substantially less than ideal. By furnishing application writers with the proper tools to cope with common performance faults, robust system performance can be achieved.

To test our hypothesis, we present the design and implementation of *River*. River provides a generic data-flow environment as a substrate for the construction of performance-robust applications. Two novel, distributed algorithms form the heart of performance availability in River: a *distributed queue* allows producers to flexibly move data to variable rate consumers, thus avoiding consumer-side performance faults, and *graduated declustering* carefully allocates producer bandwidth across consumers, similarly avoiding producer-side performance faults. In tandem, with no centralized components or global information, these two constructs can be used to implement performance-robust applications.

We demonstrate the utility and efficiency of the River environment and its primitives through a series of simulation and implementation experiments. First, we rigorously explore the performance properties of both the distributed queue and graduated declustering, establishing that they perform as desired under a broad range of performance faults. Then, we apply the mechanisms to the construction of several data-intensive query-processing applications, transforming them into programs that are robust to disk performance faults.

Professor David A. Patterson
Dissertation Committee Chair

Contents

Chapter 1

Introduction

The construction of robust, large-scale, production computer systems presents a formidable challenge to systems architects. Although often designed and tested in the well-controlled settings of a laboratory, such systems must perform correctly and with high performance in real-world, production environments. The differences between the two surroundings are numerous: the former is familiar, often running well-understood test-suites, and managed by local gurus with a zeal for pinpointing particular problems; the latter is much more dynamic, filled with unexpected variations in workload and underlying hardware, with system managers who often are unaware of the specifics of system operation.

The problem of attaining peak production performance is exacerbated by the methods used to build modern systems. Such systems are often constructed from a patchwork collection of commodity hardware and software components, from low-level devices such as the main microprocessor and disk drives up to the operating system, file system, and other support software. An extreme example of such a system is a Networks of Workstations (NOW), an entirely commodity system built by connecting a group of ordinary off-the-shelf workstations or PCs with a high-speed, switch-based network [5]. Worse yet, each of these constituent elements is experiencing rapid increases in its individual complexity, as recent processors consist of millions of transistors [122], and current operating systems contain millions of lines of code [73].

The result of the proliferation of complexity is a difficulty of design along two separate architectural axes: correctness and performance. A robust design must explicitly account for how the correctness of the system will be affected when one of the underlying components fails unexpectedly. It should also take into account how global performance characteristics will be altered under component performance variations.

Fortunately, much of the previous work in the field of distributed computing has addressed the design of large-scale systems that can tolerate such *correctness faults* in individual components [18, 24, 49, 61, 82, 86, 107, 114]. The practical notion behind such work is that distributed and parallel systems consist of both hardware and software components that will periodically fail; a system that works continuously on top of such unreliable components must be designed to operate in spite of such failures. A good example is a RAID disk subsystem, which can tolerate the failure of a particular disk and continue correct operation [52].

Missing from this large body of work is the notion of how the system as a whole will function when one or more components does not perform as expected. Analogously, we term the unexpected low performance of an entity a *performance fault*. Again, the RAID system presents a good example; though we understand how various RAID configurations will perform when one or more disks cease to operate, little is known about their global performance characteristics when one or more disks deliver data correctly but more slowly than expected.

To capture this idea succinctly, we introduce the concept of *performance availability*. Just as it is unrealistic to build a parallel system that functions correctly only when all underlying components are functioning correctly, we believe that it is also unrealistic to build a parallel system that performs well only when all constituent components are delivering peak or near-peak performance.

Thus, the thesis of this dissertation is that modern parallel and distributed systems must provide the proper primitives to support performance availability. Use of such mechanisms enables applications to deliver good global performance in the face of localized component performance failures. As we shall see, systems without explicit mechanisms to cope with performance faults often will run at the rate of the slowest component in the system, losing much or all performance advantage gained through the use of multiple machines.

1.1 Motivation: A Case Study

To better motivate the problem of performance faults, we perform a simple experiment with NOW-Sort [9], a high-performance parallel external sorting implementation for clusters. In developing the sorting application, many months were spent tuning the program to the various components of the system: extracting peak bandwidth from the disk drives, optimizing the in-memory sort so as to take near-zero CPU time, and choosing a communication pattern that scales to 100 machines. The final sorting program was fast and efficient: NOW-Sort broke two world records, and held those records for two years [60].

Unfortunately, while able to occasionally obtain peak performance from the system, NOW-Sort usually did not. When running at scale, the performance of the application was in fact quite unreliable [10]. Some investigation revealed the many causes of NOW-Sort's performance problems: one time it was an overly-full disk, another time an over-burdened CPU, and once even an over-taxed memory system. These problems, which are quite common in production cluster environments, occurred on just a handful of the machines. Unfortunately, NOW-Sort by its inherent design always runs at the rate of the slowest machine in the cluster; it was a fast but fragile sort, and could not adapt to changes in underlying component performance.

To demonstrate here the deleterious effects of performance faults on NOW-Sort, we perform the following experiment. In the experiment, the sort executes on 8 machines, and in each run, we induce a performance fault on the sort on only one of those machines. The results from these experiments are shown in Figure 1.1.

As we can see from the graph, each of the performance faults, induced on a single machine, has a serious and detrimental global performance effect. If a single file on a single machine has poor layout, placing data on the inner tracks of the disk versus the outer, overall performance drops by a factor of 1.5. When a single disk is a hot spot, and has a competing data stream writing to it throughout the run, performance drops by a factor of 3. An extra CPU load on any one of the machines decrease performance proportionally to the amount of CPU they steal: stealing 25% of the

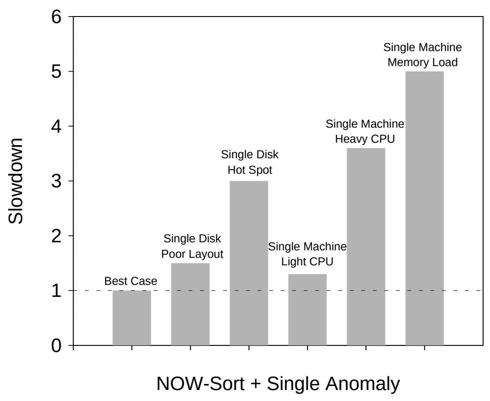


Figure 1.1: **NOW-Sort Performance Under Performance Faults.** *The graph depicts the best-case performance of NOW-Sort, versus the performance under slight disk, CPU, and memory performance faults. Each performance fault is induced on just a single machine, and in each case the effect on overall performance is stark. All performance metrics are normalized to the ideal performance of NOW-Sort, shown in the graph as the first bar labeled 'Best Case'. The first two faults are disk faults. The second bar, labeled 'Poor Layout', shows the performance of NOW-Sort when data is placed on the inner tracks of the disk instead of the outer tracks. The third bar, labeled 'Hot-Spot', shows overall performance when a load-imbalance via an extra write load is placed on a single disk throughout the run. The next two performance faults are induced on the CPU. The 'Light CPU' bar reflects performance when 25% of the CPU is taken away from NOW-Sort, and the 'Heavy CPU' bar when 75% is stolen. Finally, the last bar in the graph, 'Memory Load', shows the performance of NOW-Sort when one machine has an extra memory load placed upon it, and begins to page. All performance results are relative to the 8-node NOW-Sort, which reads and writes data at a near-peak disk rate of 40 MB/s across all machines throughout the run, depending on the phase of the application.*

CPU lowers performance by a factor of roughly 1.3, and utilizing 75% of just a single CPU reduces performance by a factor of 4. Finally, when a single machine is given an extra memory load, causing it to thrash due to a shortage of physical memory, performance drops by a factor of 5.

1.2 Problem Statement

We believe it is difficult and likely not cost-effective to construct a modern system that did not suffer from performance faults. As system size and complexity increase, carefully controlling a large-scale, general-purpose system becomes nearly impossible. Therefore, we approach the problem in a different manner, by assuming the presence of such performance faults, and providing a substrate that can operate well in spite of them. By altering our base philosophy, we hope to enable the construction of applications that not only perform well in well-controlled experimental settings, but also excel in more dynamic, production environments.

Thus, our refined problem is to provide the necessary and sufficient primitives to enable the transformation of a given set of applications into programs that are robust to performance faults. We argue that software systems for dynamic environments such as clusters *must* provide such mechanisms, or suffer the performance-fate of NOW-Sort.

To better understand the core issues surrounding performance availability, we focus on data-intensive cluster applications. These types of programs comprise an important application class, as they form the backbone of high-performance file services, database engines, and Internet services [6, 26, 35, 56]. Performance availability is also more germane in this context, as these applications place stress on all components of the system, including the processor, main memory, network components, and disks.

Given our broad agenda of converting standard data-intensive applications into their robustly performing counterparts, we are left with the more specific question of deriving mechanisms that are necessary and sufficient to support such transformations. Further, we must also demonstrate the efficacy of our primitives.

One assumption that underlies our work is that the applications we are dealing with are parallel in nature. Rigidly-designed parallel applications, in fact, are particularly sensitive to performance faults, as they are wont to run at the rate of the slowest node. Thus, the philosophy underlying our approach is as follows: to avoid dependency on the run-time of any one particular component when performing a global task.

Let us then view a parallel application as consisting of a set of many-producer to many-consumer transfers. Producers generate data, and consumers obtain data and execute some set of operations upon the data. To avoid performance faults in such transfers, we must provide producers with a method to move data to faster consumers and avoid slower, performance-faulty consumers. Analogously, we must provide consumers with a mechanism to obtain data from faster producers, and avoid slower, performance-faulty producers.

1.3 River

As a substrate for robust application development, we design and implement *River*, a data-flow programming environment for I/O-intensive cluster applications. The goal of River is to provide maximal performance to I/O-intensive applications in the common-case. Two distributed data-transfer mechanisms form the heart of performance availability in River: a *distributed queue* balances work across consumers of the system and *graduated declustering* dynamically adjusts the load generated by producers. In tandem, these constructs can be used to fashion applications that tolerate a range of common and otherwise detrimental performance faults.

The first significant component of River is a high-performance distributed queue implementation. River uses distributed queues to let data flow between operators at autonomously adaptive rates: at any given time, each producer places data into the distributed queue as fast as it can, and each consumer takes data from the distributed queue as fast as it can. By interposing distributed queues between operators in a data flow, load is naturally balanced across consumers running at different rates. An advantage of this simplicity is the lack of global coordination required: consumers can change their rate autonomously over time, without communicating with other clients. The result is full-bandwidth, balanced consumption: all available bandwidth is naturally utilized at all times, and all consumers, including those suffering from performance faults, complete near-simultaneously.

The second important aspect of River is a flexible, redundant disk layout and access mechanism called graduated declustering. A generalization of a mechanism proposed for early parallel database systems [64], graduated declustering allows the task of data *production* to be shared among replicated producers in a flexible fashion. Graduated declustering mirrors large sequential collections on the disks of different producers. During data flow, a producer multiplexes its I/O bandwidth across all the data sets it is currently handling, to ensure that it produces its share of the global bandwidth available for each collection. The result is full-bandwidth, balanced production: all available bandwidth is utilized at all times, and all producers, including those suffering from performance faults, complete near-simultaneously.

1.4 Evaluation Methodology

Given our two data-transfer primitives, we first demonstrate that they are efficient in their specific tasks. We do so via the combined use of both simulation and implementation. We utilize simulation to understand the idealized properties of the mechanisms, and to show that both algorithms behave as desired. We then measure the behavior of each in a prototype implementation, demonstrating that both the distributed queue and graduated declustering translate well into an actual cluster environment.

To show that both mechanisms are necessary and sufficient, we design and implement a set of performance-robust applications. Though this will not serve as a formal proof that the distributed queue and graduated declustering are complete, it is a demonstration via artifact that a small set of important applications can be transformed into their performance-robust counterparts.

1.5 Contributions

We now summarize the main contributions of this dissertation. They are:

- The introduction and definition of *performance faults*, *performance-fault tolerance*, and *performance availability*. Though researchers have long understood the value of building applications and services that are robust to correctness faults, the same type of effort must be applied to ensuring programs behave well under performance faults.

- The design of two particular performance-robust primitives: the distributed queue and graduated declustering. Each of these primitives is designed to solve a particular aspect of the performance availability problem; distributed queues are used to avoid consumers suffering from performance faults, whereas graduated declustering is utilized to circumvent the ill-effects of performance-faulty producers. In tandem, the primitives can be used to tolerate a wide range of performance faults. At the heart of both mechanisms are novel distributed algorithms, requiring no centralized control.

- A thorough evaluation of both the distributed queue and graduated declustering, via simulation and implementation. Given our two mechanisms, we demonstrate that the algorithms behave as desired, successfully avoiding performance faults in consumers or producers. We also show that the implementations are lightweight, adding little overhead to basic messaging costs.

- The construction of performance-available data-intensive database primitives. We demonstrate that the performance-robust mechanisms are effective in forging programs that can tolerate performance faults.

1.6 Outline

The outline of the rest of this dissertation is as follows. Chapter 2 defines our basic terminology of performance faults, performance-fault tolerance, and performance availability, and develops a simple model of how the ideal system should behave under performance faults. Then, in Chapter 3, we cover related work, building a case for the need for performance availability by documenting the existence of performance faults in many previous efforts. Chapters 4, 5, 6, and 7 present the design of the River system, as well as the two core performance-robust mechanisms, the distributed queue and graduated declustering. We then describe a simulation environment and other experimental details in Chapter 8. Chapters 9, 10, and 11 present a thorough analysis of both the distributed queue and graduated declustering, via simulation and implementation results. We then apply the core mechanisms to transform a set of database query-processing primitives into their robust counterparts in Chapter 12, and we conclude and discuss future directions in Chapter 13.

Chapter 2

Modeling Performance Under Performance Faults

Before describing our system design, implementation, and performance characteristics, we now formalize the concepts of performance faults, performance-fault tolerance, and performance availability. We utilize definitions from the field of reliable systems design to guide our discussion, drawing heavily from [24, 61, 107, 114].

We also develop a simple model of the behavior of an ideal system under performance faults. In later chapters, by comparing experimental or simulation results to the behavior of an ideal system, we are able gauge how well our actual algorithms and implementations are functioning. Of course, in practice, real or even simulated systems do not always meet this ideal; the goal of the implementations described in this dissertation will be to approximate the ideal system, taking real-world factors such as complexity of design into account.

2.1 Definitions

2.1.1 Performance Faults

In the field of fault tolerance, a component is considered *faulty* "once its behavior is no longer consistent with its specification [107]." Most work subdivides faults into the following two different representative classes. Byzantine failures are described by Lamport as follows: "The component can exhibit arbitrary and malicious behavior, perhaps involving collusion with other faulty components [77]." These represent the most general class of failure. Schneider describes a weaker failure mode known as Fail-stop failures: "In response to a failure, the component changes to a state that permits other components to detect a failure has occurred and then stops [107]."

By analogy, we define a performance fault as follows. A component is said to suffer from a *performance fault* when its level of performance is less than its specified performance level [1]. As with

[1] Note that an increase in performance could also be considered a performance fault. However, because we have never observed such behavior, we do not deal with that case explicitly.

correctness faults, performance faults come in different forms, as described below. We sometimes refer to a performance fault as a *perturbation* to the component.

One departure from the field of reliable systems is the difference between the generally discrete nature of correctness faults, *e.g.* working or not working, and the continuous nature of performance faults. Because we are concerned with the continuous domain, we must have a way in which to distinguish a severe performance defect from a mild one.

Assume that a device is expected to deliver performance at some level P_{peak} while unperturbed over a given time interval of interest. When the device is suffering from a performance fault, the fault uses some amount of the available resources. We term the rate of performance of the fault P_{fault}, where $P_{fault} < P_{peak}$. Thus, for a given resource R that is undergoing a performance fault, we can view the fault as an entity that utilizes some given portion of the resource, akin to an application that utilizes the resource. When undergoing a fault, the rate of performance of the device drops to $P_{peak} - P_{fault}$.

To characterize the strength of a performance fault, we define the *fault utilization* as:

$$U_{fault} = \frac{P_{fault}}{P_{peak}}. \tag{2.1.1}$$

The value of the fault utilization ranges from 0 to 1. Note that a fail-stop correctness fault is a special case of a performance fault, where $U_{fault} = 1$ ($P_{fault} = P_{peak}$).

Devices can exhibit performance faults for different lengths of time; we label the time characteristics of a performance fault the *fault duration*. Performance variations may be relatively long term; for example, a file whose blocks are placed poorly on disk may deliver much less performance than expected, but do so consistently. Variations may also be intermittent; a daemon may need to flush data to disk periodically, and in those moments reduce effective disk bandwidth for other applications. Within this dissertation, we will focus on the former case; variations that are dynamic but likely long-lived. We believe those are the most common types of performance faults for clusters, as we will document in Chapter 3.

Anomalous performance has been largely ignored by most of the research literature; however, a few researchers have observed it in limited domains. The unexpected run-time of an application due to processor performance intricacies has been labeled a *performance anomaly* by Kushman [76], and minor, periodic performance fluctuations due to hardware oddities have been classified as a form of *interference* by Gray [43].

2.1.2 Performance-Fault Tolerance

Schneider states that "a system consisting of a set of distinct components is t *fault tolerant* if it satisfies its specification provided that no more that t of those components become faulty during some interval of interest [107]."

Similarly, we say that a system consisting of a set of distinct components is t *performance-fault tolerant* if it delivers peak performance provided that no more than t of those components suffers from a performance fault during some interval of interest.

2.1.3 Performance Availability

Siewiorek defines *availability* as follows: "The availability of a system as a function of time, $A(t)$, is the probability that the system is operational at the instant of time, t [114]." Gray and Reuter's definition is not probabilistic, but does include a performance requirement: "System availability: The fraction of the offered load that is processed with acceptable response times [61]."

The difficulty with the former definition is that we are left to define "operational". Does an operational system have certain performance requirements? Gray's definition addresses this concern directly by including an "acceptable response time". However, the definition is still too coarse-grained for our purposes.

Thus, we define the *performance availability* of a system as follows: the performance availability of a system as a function of time, $P_A(t)$, is the fraction of peak performance that the system delivers at time t. Thus, in order to compute the performance availability of a system, this definition requires us to understand its peak performance potential, *e.g.*, how an ideal system would perform.

2.2 Behavior Under Faults: A Model

Therefore, we next develop a simple piecewise-linear model of how overall performance of an *ideal* system should be affected under a given set of performance faults, as we increase the number of components on which the fault occurs. Throughout this dissertation, we will use this model to explicitly evaluate the performance-fault tolerance and performance availability of the system. For simplicity of notation, we assume that the same performance fault occurs on each faulty component; this restriction could be relaxed if desired. By comparing real system performance to that of this idealized model, we gain insight into our system's performance characteristics.

Assume that an application A that we are observing on the system normally runs at a performance level P_{app} on a given resource R during some period of interest. For example, if A were an I/O-intensive application, P_{app} might be the rate that A reads data from a single disk during a read phase. As above, P_{peak} is the peak rate of that resource, and P_{fault} is the performance level of a performance fault. An application uses $\frac{P_{app}}{P_{peak}}$ fraction of the resource when unperturbed; we call this the application utilization of the resource, U_{app}.

In general, we will induce an increasing number of performance faults into a system, and monitor the performance level of the system under those faults. We plot the results on a graph we term the *performance availability spectrum*. Along the x-axis of the graph, we increase the number of components that are experiencing the given performance fault. The y-axis plots overall system performance, as a ratio of peak performance under non-perturbed "perfect" conditions. Thus, if performance of the application for the period of interest is denoted $P(f)$, where f is the number of faults in the system, the graph plots $\frac{P(f)}{P(0)}$, while increasing f along the x-axis. Figure 2.1 depicts the spectrum.

The first point of interest that we will derive is labeled x_{loss} in Figure 2.1. At this point, the perturbed resource R is fully utilized across all components in the system. Up to that point, the perfect system will be able to move work elsewhere, and should not suffer any performance loss. It can thus be said that the system is x_{loss} performance-fault tolerant, as seen in the graph as a flat line at 1 up to the point x_{loss}.

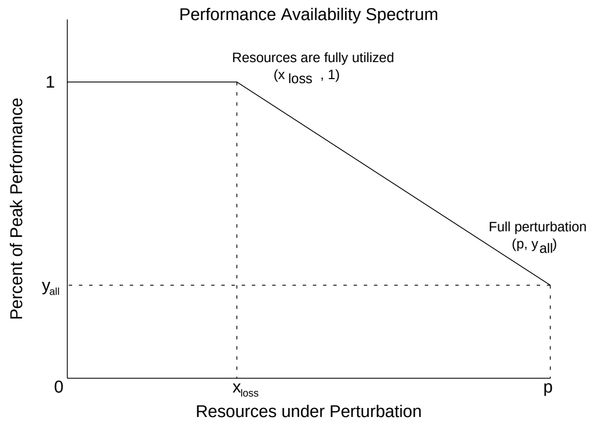


Figure 2.1: **Performance Availability Spectrum.** *Overall performance under an increasing number of performance faults is shown. Along the x-axis, the number of components experiencing the performance fault is increased up to the maximum number of components in the system, p. Overall system performance, as a fraction of ideal performance in a setting free of performance faults, is plotted on the y-axis.*

We now solve for x_{loss}. At this point, we know that 100% of resource R is utilized across all p components; thus, the sum of the application's resource usage and the perturbation's resource usage across all p components should be equal to p:

$$(p \cdot \frac{P_{app}}{P_{peak}}) + (x_{loss} \cdot \frac{P_{fault}}{P_{peak}}) \;=\; p. \tag{2.2.1}$$

Solving for x_{loss}, we arrive at:

$$x_{loss} \;=\; \frac{P_{peak} - P_{app}}{P_{fault}} \cdot p. \tag{2.2.2}$$

Rewritten in terms of utilization, we obtain:

$$x_{loss} \;=\; \frac{1 - U_{app}}{U_{fault}} \cdot p. \tag{2.2.3}$$

We now present a simple example to make this more concrete. Imagine that the CPU is the resource of interest. Assume that a parallel application runs on 16 CPUs in a cluster, and that each process of the application utilizes 75% of the CPU in an unperturbed system; thus, $\frac{P_{app}}{P_{peak}}$ is 0.75. Assume that the performance fault we are interested in has a fault utilization $\frac{P_{fault}}{P_{peak}}$ of 0.5; therefore, the fault utilizes 50% of the CPU. By substituting these values into the equation, we arrive at $x_{loss} \;=\; 8$. Thus, when more than half of the CPUs are perturbed, we expect performance to become less than 100% of peak. At this point, the application uses 75% of each of 16 CPUs, or 12 full CPUs' worth ($16 \cdot 0.75 = 12$). The performance fault, when occurring on 8 CPUs, uses a total of 4 CPUs worth of resources ($8 \cdot 0.5 = 4$). Thus, total CPU utilization under 8 performance faults is $(16 \cdot 0.75) + (8 \cdot 0.5) = 16$; in tandem, the application and the faults are utilizing the complete set of 16 CPUs. Any additional performance fault will result in application slowdown.

We can also make a few general observations. First of all, x_{loss} degenerates to 0 when $U_{app} = \frac{P_{app}}{P_{peak}}$ is equal to 1. This observation matches intuition – when the application uses 100% of the resource in the unperturbed case, any additional perturbation to the system will lead to a loss of overall performance. Without excess resources, a system is said to be 0 performance-fault tolerant.

Second, in the other extreme, if $P_{app} + P_{fault} < P_{peak}$ ($U_{app} + U_{fault} < 1$), x_{loss} is greater than p. In that case, the sum of the application resource utilization and performance-fault resource utilization does not match the total amount of resources available, even with all components under perturbation. Therefore no slowdown is experienced by the ideal system. Such a system is said to be fully performance-fault tolerant.

The other point of interest in Figure 2.1 is the y value when all p components in the collection are experiencing the performance fault. We call the y-axis value y_{all}, to denote the performance level of the ideal system when all p components are suffering from performance faults.

We now derive y_{all}. At this point, all p components are under perturbation. Thus, the amount of performance that each node can deliver under perturbation is $P_{peak} - P_{fault}$. From this, we can calculate y_{all} directly by observing that application slowdown, when resources are over-taxed, is the performance that can be delivered divided by the performance needed by the application:

$$y_{all} = \frac{P_{peak} - P_{fault}}{P_{app}}. \tag{2.2.4}$$

This can also be written in terms of utilization as follows:

$$y_{all} = \frac{1 - U_{fault}}{U_{app}}. \tag{2.2.5}$$

For example, if our application utilizes 75% of the CPU and the performance-fault utilizes 50%, when all nodes are perturbed, total ideal system performance will be $\frac{1-0.75}{0.50} = \frac{2}{3}$.

In general, as application needs increase, the value of y_{all} decreases. Similarly, as the fault utilization increases, y_{all} decreases.

Finally, now that we have the two points of interest, we can derive the slope of the line that connects them, and thus express expected ideal system behavior in closed form:

$$y = \begin{cases} 1 & 0 \leq x \leq x_{loss} \\ (\frac{y_{all}-1}{p-x_{loss}}) \cdot x + [1 - (\frac{y_{all}-1}{p-x_{loss}}) \cdot x_{loss}] & x_{loss} \leq x \leq p \end{cases} \tag{2.2.6}$$

From the model, we see how to judge whether a given system is performance-fault tolerant under a given set of performance faults. To determine the performance availability ($P_A(t)$) of a real system, all one needs to have is knowledge of the frequency and duration of performance faults, and combine that with the performance availability spectrum.

2.3 Examples

We now present three examples of potential system behavior under performance faults in Figure 2.2, to better illustrate what actual performance availability spectrums might look like. The performance of all of the mock systems is plotted against ideal system performance for the sake of comparison.

The first example in Figure 2.2 is a system that cannot tolerate any performance faults to the component under perturbation. Therefore, if all components are operating as expected, performance is excellent, but if just a single component suffers from a performance fault, global performance immediately drops to the level of that single perturbed component. Subsequent performance faults do not worsen the already bad situation. Many real systems exhibit this type of behavior, as they are designed to function only in highly-controlled though perhaps unrealistic environments.

In the second example, we observe a system that is t performance-fault tolerant. Thus, performance is ideal and at 100% with t or fewer performance faults in the system. However, when more than t faults take place, system performance drops sharply to the lowest level. This style of system has a fixed amount of adaptability, but its performance does not degrade gracefully.

Finally, the third example shows a system that can tolerate x_{loss} performance faults, and degrades gracefully with s or fewer faults occurring. We call this system x_{loss} performance-fault tolerant and a system that *degrades gracefully* under s performance-faults. Even though the system behaves ideally until there are s performance faults present, the system is still said to be only x_{loss} tolerant, because 100% of peak performance is not delivered after x_{loss} faults take place. System performance drops off sharply with more than s performance faults. The system that we develop will hopefully fall into this category.

2.4 Summary

In this chapter, we have developed a formalization of the concepts of performance faults, performance-fault tolerance, and performance availability. To better understand the performance properties of systems under performance faults, we have developed a piecewise-linear performance model of the ideal system. By plotting system performance against the ideal model, we can judge the overall performance-fault tolerance and performance availability that the system under test provides.

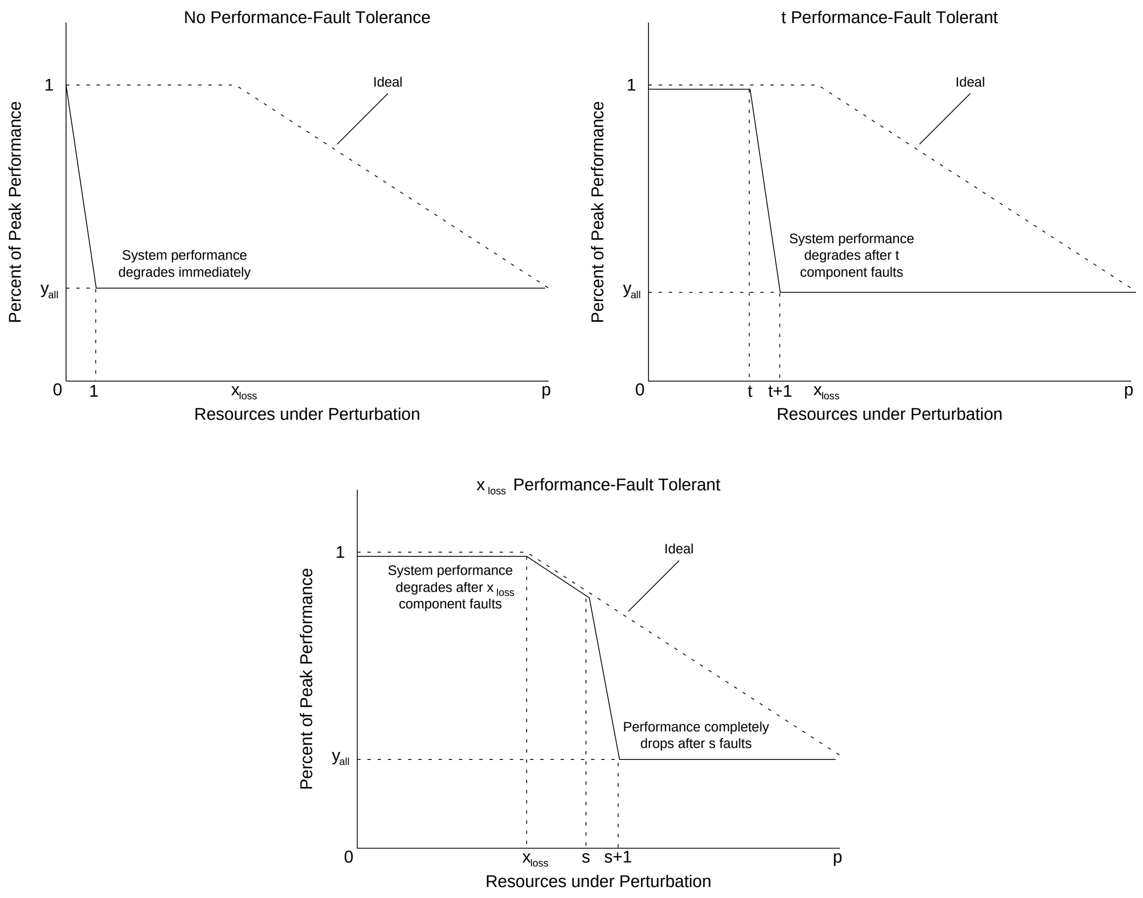


Figure 2.2: **System Performance: Examples.** *Three examples of system performance under performance faults are presented. In the first, we depict a system that cannot tolerate any performance faults. The second shows a system that is t performance-fault tolerant, and the third presents a system that is x_{loss} performance-fault tolerant and ideal under s performance faults.*

Chapter 3

The Case for Performance Availability

In this chapter, we build the case for performance-available systems by examining related work. First, we demonstrate that long-lived performance variations are common in modern platforms; surprisingly, these unexpected fluctuations in performance occur even in well-controlled research environments. Then, we inspect previous parallel I/O systems, databases, and programming environments, and discuss how well these systems adapt to underlying component performance variations.

3.1 Performance Faults

When utilizing a large collection of processors or disks, ease of programming often leads to simple, static uses of parallelism. For example, parallelism in disk arrays often comes in the form of *striping* [71], where disk blocks are distributed in a fixed, round-robin manner across the disks of the system, based on a simple "address modulo disk-count" calculation. In parallel databases, similar static techniques such as range-partitioning and hash-partitioning are often utilized [23, 42, 55], distributing keys to machines based on static functions of key value.

The problem with static uses of parallelism is that they make stringent *performance assumptions* of the underlying components of the system. The higher level system assumes that each underlying component will deliver performance in some pre-defined and consistent manner. However, if just one processor or disk does not perform as expected over a sustained period of time, the performance of the entire system can be severely affected. Performance faults break the performance assumptions of higher-level systems, and thus can cause detrimental global performance effects.

Unfortunately, due to the increasing complexity of modern processors, disks, and other devices, consistent performance is increasingly unlikely, especially from a large set of components. The basic question we will attempt to answer in this section is: can modern system components be expected to perform consistently? Do performance faults occur in practice? If so, how often do they occur, and what is their duration? Although we are mostly concerned with disk performance properties, we will also cover other examples of performance variations, in an attempt to understand which subsystems performance faults occur within.

3.1.1 Processors

We begin our examination with an inspection of the performance characteristics of modern processors. The processor industry has kept pace with Gordon Moore's prediction of transistor count doubling roughly every 18 months, leading to amazing improvements in performance over the past twenty years.

The cost of this innovation has been a severe increase in device complexity. More sophisticated speculation and prediction mechanisms have found their way onto processor cores. Not surprisingly, complexity, while sometimes providing higher *peak* performance, is often the enemy of *consistent* performance.

UltraSPARC processor: The Sun UltraSPARC-I processor is a second-generation RISC processor from Sun Microsystems [122]. In his masters thesis [76], Kushman discusses *performance anomalies* that occur inside of the UltraSPARC-I processor. By his terminology, a performance anomaly is an unexpected run-time behavior of a program. Three types of performance anomalies are discussed:

- discontinuity: a small increase/decrease in work leads to a disproportionately large change in run time.
- nonmonotonocity: an increase in work decreases run time, or vice-versa.
- nondeterminism: executed twice, a program exhibits different run times.

Kushman finds that the implementation of the next-field predictors, fetching logic, grouping logic, and branch-prediction logic of the UltraSPARC-I all can lead to performance anomalies. Simple code snippets are shown to exhibit non-deterministic performance. The manner in which instructions align in cache blocks is one of the main causes of such strange performance properties. However, in many cases, the addition of as little as a single instruction can reduce the run-time by a factor of three, and remove the nondeterminism entirely.

What this work demonstrates is that processor performance anomalies are likely to become more common in modern processors. Only with a highly sophisticated and detailed understanding of low-level architectural features could a programmer or compiler possibly avoid such anomalies. In fact, the exact cause of two of the four anomalies remains unknown, though methods to avoid them were found.

HP PA-RISC processor: The PA-RISC is the Hewlett-Packard RISC processor [109], introduced in 1985. In their work on replicated fault-tolerance, Bressoud and Schneider find that the PA-RISC exhibited non-determinism in its TLB replacement policy:

> "The TLB replacement policy on our HP 9000/720 processors was non-deterministic. An identical series of location-references and TLB-insert operations at the processors running the primary and backup virtual machines could lead to different TLB contents."
> [25], page 6, paragraph 2.

The reason for the nondeterminism is not given, nor does it appear to be known, as it surprised numerous HP engineers. Obviously, different TLB contents will affect application performance.

The authors also found that the HP cache mechanism would map out certain "bad" lines of the cache to improve the yield of the processor [108]. Thus, the same application running on seemingly identical processors could produce significantly different performance characteristics, based on which exact lines of the cache were active.

SPARC-10 processor: The predecessor to the UltraSPARC-I processor from Sun was the Viking series of processors [63]. In [7], the cache size of each a set of Viking processors is measured via micro-benchmark, as suggested by [104]. The results were surprising:

> "The Single SS-51 is our base case. The graphs reveal that the first level cache is only 4K and is direct-mapped."

The specifications suggest a level-one data cache of size 16 KB, with 4-way set associativity. However, some of the chips produced by TI had this performance flaw. Others, produced at different times, did not, once again leading to the case where seemingly identical processors were not actually identical. The result of these differences is that application workload performance across seemingly identical processors was different by up to 40%.

Memory subsystem: In their paper on scalar-vector memory interference, Raghavan and Hayes show that small perturbations to a vector reference stream can severely reduce memory system efficiency [99]. Their results, derived analytically, show that performance drops by roughly 50% even with small perturbations to the memory system, which are common in vector machines. No solutions are proposed.

OS Interactions: Sometimes unexpected performance arises not due to the direct interaction between application and hardware, but because of the behavior of an external software agent. In particular, the literature has shown that operating system virtual-memory mapping decisions can have a severe performance impact on applications, reducing performance by up to 50% [31]. Virtually all machines today use physical addresses in the cache tag. Unless the cache is small enough so that the page offset is not used in the cache tag, then the allocation of pages in memory will affect the cache-miss rate. Since page placements are not identical across runs in modern operating systems, performance isn't either. As noted in [16], direct-mapped caches are particularly sensitive to this type of anomaly.

3.1.2 Disk Drives

We now turn our attention to performance heterogeneity in the context of I/O. Disks are a rich source of performance fluctuations; as with processors, complexity of individual drives has been increasing steadily over time, resulting in a richer set of performance characteristics.

Vesta: Vesta is a parallel file system from IBM designed for the IBM SP series of clustered systems [36]. More details on Vesta are given below. Here, we instead draw on the experience the authors had when running experiments on their prototype hardware:

> "The results shown are the best measurements we obtained, typically on an unloaded system. The number of measurements done for each data point ranged from 2-3 up to about 20, with higher numbers being used mainly in the case of large access sizes that were expected to drive the hardware to its limits. In many cases there was only a small (less than 10%) variance among the different measurements, but in some cases the variance was significant. In these cases there was typically a cluster of measurements that gave near-peak results, while the other measurements were spread relatively widely down to as low as 15-20% of peak performance. The reason for such low performance was interference from other jobs and system activity beyond our control. A characteristic of the SP-1 was that AIX daemons were run unsynchronized on the multiple different nodes of the computer; hence, if the AIX scheduler for one of the servers involved in a performance run decided to run a daemon process during the run, the performance of that experiment was adversely affected, often dramatically. Detailed analysis of such phenomena, and indeed of system performance under load conditions, depends on the specific characteristics of the interfering workload. Such analysis is beyond the scope of this article". [36], page 250, paragraph 2.

NOW-Sort: In our own experience with parallel external sorting in a network of workstations (NOW-Sort), we found that the cluster environment was surprisingly performance heterogeneous [9, 10]. As noted in that text:

> "The performance of NOW-Sort is quite sensitive to various disturbances and requires a dedicated system to achieve 'peak' results. In this section, we discuss how we solved a set of particular run-time performance problems where a foreign agent (such as a competing process, inadequate memory, or a full disk) on one or more machines slowed down the entire application." [10], page 8, paragraph 1.

The net effect of the different performance faults varied from slight drops due to a slightly full disk to performance that was off by an order of magnitude when a node began to page heavily. Our approach to extract peak performance for a single application was to find performance-faulty nodes, and either remove the machine from the cluster entirely, or remove the cause of the performance fault from the machine. Although this eventually yielded excellent results for the NOW-Sort application, it required a large amount of human intervention, which is clearly undesirable in a more realistic, production environment.

We also came across another situation where two identical disks (Seagate Hawks, 5400-RPM) performed noticeably differently under a simple bandwidth experiment. Although most disks of this brand deliver 5.5 MB/s on sequential reads from the outer tracks, we observed that one such disk only delivered 5.0 MB/s. Our hypothesis is that SCSI bad-block re-mappings, which are transparent to both the user and the file system, were the culprit. Upon inspection, the lesser-performing disk was revealed to have three times the number of block faults than other, normally-performing devices.

San Diego Sorting: In related work on parallel external sorting, Rivera and Chien also encounter some disk performance irregularities [102]. Their environment is a cluster of 64 PC-class machines, each with a single 10K RPM Seagate Cheetah attached. However, the final sort performance they report is on a cluster of only 60 machines. The reason is as follows:

> “Each of the 64 machines in the cluster was tested; this revealed that four of them had about 30% slower I/O performance. Therefore, we excluded them from our subsequent experiments.” [102], page 7, last paragraph.

The approach here is identical to that of NOW-Sort: find the “bad” nodes by hand, and remove them from contention. No explanation of the faulty behavior is given.

Illinois Panda: The parallel I/O group at Illinois has also recognized the volatility of cluster environments, especially in the realm of I/O, and have also been addressing it in the context of the Panda parallel I/O library. The authors note that:

> “...unlike the NASA Ames SP2 that we used earlier, the Cornell I/O nodes ran at different speeds.” [127]

The basic issue they came across was a static difference in disks: “fat” nodes had faster disks than “thin” nodes. The performance difference between the two disks was roughly a factor of two. Further, there were originally some older disks in the cluster, which, not surprisingly, were also slower.

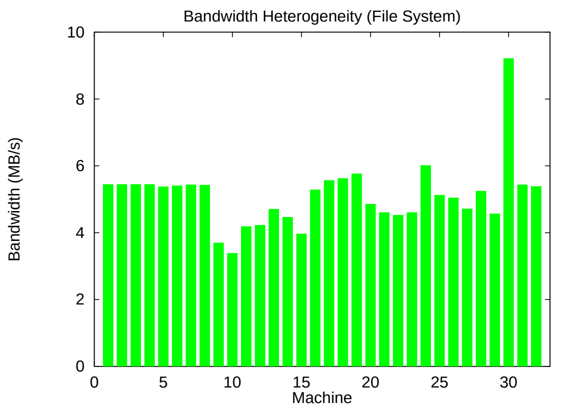
Tertiary Disk: The Tertiary Disk (TD) group at Berkeley performed a study of disk behavior on a 400-disk farm over a 6-month period [119], with the goal of uncovering what kinds of faults occur in practice in large-scale systems. The basic application running on the cluster was the serving of art images to web clients from the San Francisco Museum of Art as described in [118]. They found that:

> “The largest source of errors in our system are SCSI timeouts and parity problems. SCSI timeouts and parity errors make up 49% of all errors; when network errors are removed, this figure rises to 87% of all error instances.” [119], page 7, paragraph 3.

In examining the data further, it can be ascertained that a timeout or parity error will occur roughly 2 times per day on average. These errors often lead to SCSI bus resets, which will noticeably effect the performance of the disks on the degraded SCSI chain. Note that the strength of this particular performance fault is difficult to gauge.

Tiger Video Server: The Tiger Video Server is a project from Microsoft Research designed to deliver video streams from a cluster of PCs to a large number of clients via a fast network [22]. In dealing with a large number of disks, the authors encountered certain odd behaviors. In particular, they noticed that the disks that they were dealing with would go off-line at random intervals for short periods of time. After further investigation, they found that the disks were performing internal “thermal recalibrations”. However, because of the relatively controlled environment and uniform workload, the solution the authors arrive at is to read-ahead on a given video stream; by paying a slight cost in buffering, these particular performance variations by the disks can be masked.

Multi-zone disks: Though the previous discussions focus on performance heterogeneity *across* devices, there is also heterogeneity present *within* a single disk. As is documented in [88], modern disks have multiple zones, each with different performance characteristics. The presence of multiple zones arises from the circular geometry of the disk, which allows manufacturers to place more data in outer tracks. Because disks spin at a constant rate, the outer tracks will deliver data at a higher rate, nearly



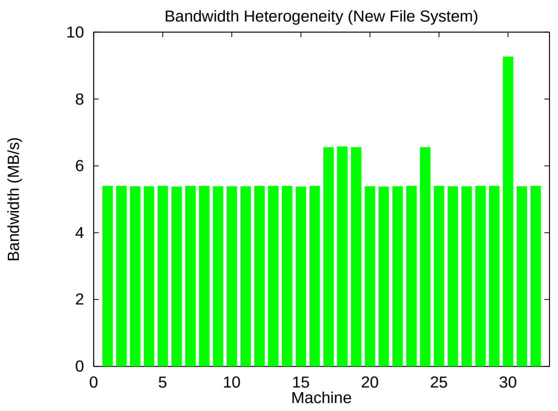


Figure 3.1: **Solaris File System Performance: A Comparison.** *The graphs plot the results of two simple experiments. In both, a 100 MB file is created on disk, and then read back sequentially. Unbuffered reads are utilized, guaranteeing no reads come from the file cache. The results plot the average of 30 runs; there was little variation across trials. The only difference in the two experiments is that in the right-most graph, a new file system was constructed before running the trials. With a new file system, the file is allocated in large, sequential chunks, and therefore performance is consistent. When run upon "aged" file systems as in the left-most graph, block allocation is randomized (due to the free list), and performance suffers accordingly. All but disk 30 are 5400-RPM Seagate Hawk disks, whereas disk 30 is a 7200-RPM Seagate Barracuda. The slightly higher bandwidth delivered from disks 17-19 and 24 is due to the placement of the Hawks on a fast-wide rather than fast-narrow SCSI bus.*

a factor of two on modern drives. Unless the disks are nearly empty and the file system provides an interface to control data layout, different disks will have different layouts and thus different performance. Such pristine conditions only occur in highly-controlled benchmark settings, and are highly unlikely to arise in practice. The performance difference across tracks is roughly a factor of two.

Solaris File System Measurements: To demonstrate how modern file systems can contribute to the problem, we perform two simple experiments. Figure 3.1 plots their results. In the first, we simply allocate a large file to the local disks of 32 machines, and then read the file back sequentially. The average bandwidth attained is plotted for each machine. The second graph shows the performance of the same benchmark; however, this time, we built a new file system over the disk before creating the file. Each data point represents the average of 30 trials; the variation in performance was less than 1%.

The performance differences in the two diagrams is striking. In the left-most diagram, the performance of the disks is quite variable, ranging from 3 MB/s (disk 10) up to almost 10 MB/s (disk 30). Though all but one of the disks are identical (5400-RPM Seagate Hawks), there is a great discrepancy in delivered sequential performance.

However, in the right-most graph, almost all disks deliver identical performance; the only exceptions are disks 17-19, 24, and 30. Further investigations reveals the cause for this slight heterogeneity: disks 17-19 and 24 are Seagate Hawks situated on fast-wide SCSI busses, and therefore are capable of delivering slightly more bandwidth than the Hawks on fast-narrow busses. Disk 30 is a different disk entirely, a 7200-RPM Seagate Barracuda.

These performance differences can be attributed to the effects of file system aging [115]. With repeated use, blocks on a free list tend to get ordered randomly; when allocating a large sequential file, blocks are not laid out sequentially; not surprisingly, performance suffers. Even with identical hardware, the behavior of certain software systems such as the file system can induce performance heterogeneity.

3.1.3 Workload

Though we will not document this in detail, external workload fluctuations often result in a non-uniform and dynamic execution environment. As discussed by A. Arpaci-Dusseau in [8], the modern cluster environment will likely contain a mix of both parallel and sequential jobs. Unless physically separated, the likelihood that parallel applications will run on identically loaded machines is quite small.

In disk systems, similar dynamic workload fluctuations occur; they are termed hot-spots, and arise when a particular data item on a particular disk is more frequently accessed than other data items. Global operations such as static disk striping suffer severe performance losses in such cases, typically running at the rate of the hot disk.

3.1.4 System Evolution

Component-based systems such as clusters have a natural advantage over other more fixed, rigid systems: over time, it is easy to add new components to deal with excess demand. Termed *incremental scalability* by Brewer [26], this allows new, often less expensive parts to be plugged into the system, further improving the cost/performance advantages of clusters over more statically assembled systems such as SMPs. However, this natural advantage has an “unfortunate” side effect: newer parts are not only likely to be less expensive, but they are also likely to be *faster*. Thus, some form of performance heterogeneity naturally occurs.

Even without cluster expansion, this phenomenon will arise due to the replacement of failed components. If a failed part is replaced with a newer model, the same heterogeneity arises, as Moore’s Law suggests that newer implies faster.

3.2 Previous Parallel Systems

In this section, we examine a cross section of work in the areas of parallel file and storage systems, parallel databases, and parallel programming environments. Most previous systems pay little attention to the kind of run-time adaptation that is necessary to deal with large-scale dynamic environments.

3.2.1 Parallel I/O

We begin our exploration with work in parallel I/O, which can be divided into two distinct realms: storage servers and file systems. Parallel storage servers export a block-level interface to clients, and are therefore built at a much lower level of abstraction. Parallel file systems export a file or file-like interface to clients, in something akin to the traditional UNIX notion of a file system [101].

Parallel Storage Servers

RAID: Redundant arrays of inexpensive disks (RAIDs) are a popular way to organize collections of multiple disks [52, 69, 95]. The idea is quite simple: aggregate a set of less-expensive disks together behind a block-level interface. Commonly, some amount of this storage is used to circumvent failures via a variety of redundancy mechanisms; see [32] for an excellent survey.

Striping is commonly used to extract the full aggregate bandwidth from multiple disks. Striping spreads blocks across disks in a fixed, round-robin pattern, based on the logical address of the block. Simple striping breaks down when any one or more of the disks in the collection runs at a slower rate than expected. The performance of simple striping can thus be classified as *fragile*: every entity must perform as expected for global performance to match expectations. Too many performance-assumptions are made of each disk.

Petal: Petal is a distributed system that also exports a block-level interface [80]. Assembled from a group of workstations or PCs, each with multiple disks attached, Petal presents this collection to clients as a highly available *virtual disk* on which to place data. The main objective of Petal is to provide easily administrable, high-performance storage via a scalable, switch-based network.

Petal is one of the few I/O systems to provide some run-time adaptation. Because Petal mirrors data to two or more disks, a set of reads from a given client can be directed to multiple locations, based on load information. Petal currently uses a simple dynamic algorithm:

> "Each client keeps track of the number of requests it has pending at each server and always sends read requests to the server with the shorter queue length." [80], page 5, paragraph 1.

It is currently unclear what the performance characteristics of this algorithm are. Further, Petal provides no load-balancing for writes, and global operations, such as striping, still suffer the same fate as they would in traditional storage systems.

Chained Declustering: Chained declustering is a technique that performs better than a naive mirrored system when there is a failure present in the system [64]. In typical mirrored systems, replication is naive, as blocks of a file or its replica are kept contiguous on a single disk. When a failure occurs, the disks that contain the mirrors for the data sets on the failed disk become overloaded. Chained declustering avoids this problem by spreading the replica blocks over many disks, and thus balances load under read-intensive workloads. However, there are no provisions for handling disks that perform at different rates.

NASD: A recent trend in storage systems moves disks away from a single attachment to a "server" machine and into the network. Known as Network-Attached Storage Devices (NASD) [53], this affords a greater level of flexibility in assembling large-scale storage systems. With this flexibility comes a potential danger: it is highly likely that such systems will have components with differing performance capabilities. Therefore, simple applications of parallelism such as striping are likely to be even less effective in these environments.

Parallel File Systems

We now turn our attention to the large body of work in parallel file systems. Most systems have focussed on extracting high performance from a set of uniform disks, including PPFS [65], Bridge [46], Panda [110], Galley [89], Vesta [36], Swift [28], CFS [90], SFS [84], the SIO specification [15], and SPIFFI [50]. Some common features include scatter-gather transfers, asynchronous interfaces, layout control, prefetching, and caching support at the client or server or both. Most of these parallel file systems stripe data naively across the set disks in the I/O subsystem, which can have undesirable performance properties.

Shared File Pointers: One interesting feature provided by some of these systems is the notion of a *shared file pointer*, as found in CFS [90] and SPIFFI [50]. With a shared file pointer, multiple processes on different machines can access a file concurrently in a consistent manner, as if sharing a local file pointer. Shared file pointers have some excellent performance properties. For example, when a group of processes is reading from a data collection, faster processes will read more data, providing coarse-grained load balancing for the application. However, shared-file pointers only provide these properties for a sequentially-read file, and provide no support for load-balancing on writes to disk.

Collective I/O: More advanced parallel file systems have specified higher-level interfaces to data via *collective I/O* [75]; a similar concept is expressed with two-phase I/O [33]. In the original paper, Kotz found that many scientific codes show tremendous improvement by aggregating I/O requests and then shipping them to the underlying I/O system; the I/O nodes can then schedule the requests, and often noticeably increase delivered bandwidth. However, because requests are made by and returned to specific consumers, load is not balanced across those consumers dynamically. Thus, though these types of systems provide more flexibility in the interface, they do not solve the problems we believe are common in today's clustered systems.

Vesta: Vesta is a production parallel file system from IBM [36]. One interesting aspect of Vesta is its new abstraction of a file. Instead of just supporting the typical UNIX abstraction of a linear sequence of bytes, Vesta promotes a two-dimensional format, where each file can be viewed as a collection of cells, and each cell is a collection of blocks. Upon opening the file, different access modes can be set, including row-major, column-major, and a block-oriented mode, all of which derive from the desire to mesh with High-Performance Fortran [62]. Though this data layout scheme does provide more flexibility to the user, the system has no method to deal with dynamic performance variations.

Panda: Of all the systems discussed, Panda [127, 110] is the only one that deals explicitly with performance heterogeneity. However, its solutions are limited. First, it only deals with heterogeneity on disk writes; reads are left unbalanced if the previous write has not perfectly balanced the load across disks, or if the access pattern changes. Further, its approach uses an *a priori* static measurement of disk performance to calculate how to layout data across disks. Thus, if performance during the write changes, their system will not properly react until the next round of measurement.

3.2.2 Parallel Databases

Large-scale I/O operations are common not only in parallel file systems but in parallel database systems as well. There are a number of parallel databases found in the literature, including Gamma [41], Volcano [56], the parallel-load prototype from the Digital Rdb project [14], and Bubba [35]. Many of these systems are based on similar *data-flow* techniques, where parallel queries are described as a directed graph that connect different sequential data *operators*.

Gamma: Gamma is a parallel database system developed at Wisconsin [41]. The initial prototype was developed for a *shared-nothing* cluster: 20 VAX 11/750 processors, each with 2MB of main memory, connected via a 10MB/s token-ring network. Eight of those machines had identical 160MB hard drives attached [40].

There are four basic partitioning techniques provided to distribute data among processors: round-robin, hash, range with a user-specified key value, and range assuming a uniform distribution. Communication among processors is performed via a *split table*, which takes tuples from the sending processor and distributes them to receiving processors in one of the aforementioned distribution styles.

All data distribution techniques in Gamma make strong performance assumptions; with any of the partitioning techniques, the total time to completion is determined by the slowest consumer. Further, the network that connects the machines is a shared medium, in this case, a token-ring network. Thus, because network bandwidth is a concern, data cannot be easily moved through the cluster for remote consumption.

Volcano: Another prominent parallel database system in the literature is Volcano [55, 56, 128]. Volcano uses a construct called the *exchange operator* to move data among processors, which is quite similar to the Gamma split table. As was the case with Gamma, Volcano makes use of solely non-robust distribution techniques such as hash-partitioning, range-partitioning, round-robin, and replication.

The major difference between the Volcano and Gamma models of parallelism is that Gamma uses a *demand-driven* approach, where sinks pull data from sources with request messages. Conversely, Volcano uses a *data-driven* approach, where data is eagerly sent to consumers before the consumers explicitly request the data. In message-passing libraries, the same issues arise in the form of *pull-based* messages layers versus *push-based* ones [68]. The push-based or data-driven approaches imply the need for flow control, so as to avoid the case where producers over-run slow consumers.

Although conceptually similar to Gamma and other parallel database systems, Volcano was intended primarily for use on a shared-memory machine. In particular, early prototypes ran on a 12-processor Sequent Symmetry. Although removing some potential bottlenecks, such as the high cost of messaging, this adds other performance concerns, including potential interactions with the caching and coherence architecture. Later work discusses execution on a hybrid cluster of shared-memory machines [57].

Digital Rdb: In work on a parallel-load prototype for the Digital Rdb project, Barclay et. al. describe another data-flow execution environment. Connections between N producers and M consumers are known as *data-flow rivers*, as they connect $N \times M$ streams of data. As stated therein:

> "River partitioning is based on a *split-table*. All the streams of a river have the same split table. As the name suggests, when a record is inserted into a river, the river program uses the split table to pick a destination stream for the record. The river program first extracts field values from the record. Then it compares these values to values in the split table to pick a destination stream. The split-table can be a range-partitioning, a hash partitioning, a round-robin, or even a replication (in which input records are sent to all sink operators)." [14], page 2, paragraph 7.

Once again, these static techniques do not provide performance availability, and will run at the rate of the slowest "sink". As the authors themselves state:

> "If different nodes have different speeds and different amounts of memory, then it is no longer straight-forward to distribute the work evenly among the nodes." [14], page 7, paragraph 1.

Parallel DB2: One example of a system that takes advantage of unordered processing to provide some form of run-time adaptation is the IBM DB2 for SMPs [81]. In this system, shared data pools are accessed by multiple threads, with faster threads acquiring more work. Lindsey refers to this access style as "the straw model", because each thread "slurps" on its data straw at a potentially different rate. Implementing such a system is quite natural on an SMP; a simple lock-protected queue will suffice, modulo performance concerns. In this dissertation, we will argue that this same type of data distribution can be performed on a cluster, due to the relatively high bandwidth of the interconnect.

ParSets: The notion of applying operations on a data set in parallel has been explored with ParSets [44]. In this object-oriented database system, an application could essentially write a function and direct the system to apply it to all objects in a collection. However, rather than move the data to the computation, this system moved the computation to the storage; data layout determines where computation occurs and is static after the placement decision has been made, and thus the system is not robust to performance variations.

NCR TeraData: Current commercial systems, such as the NCR TeraData machine, exclusively use hashing to partition work and achieve parallelism. A good hash function has the effect of dividing the work equally among processors, providing consistent performance and achieving good scaling properties. However, as Jim Gray recently said of the TeraData system, "The performance is bad, but it never gets worse" [59]. Consistency and scalability are the goals of the system, perhaps at the cost of getting the best use of the underlying hardware.

3.2.3 Parallel Programming Environments

There have been many parallel programming environments that have exploited the benefits of run-time adaptation. Some examples include Cilk [20], Lazy Threads [54], and Multipol [30]. All of these systems dynamically balance load across consumers in order to facilitate the programming of highly-irregular, fine-grained parallel applications.

Cilk: Cilk [100] is a parallel programming environment designed for parallel machines. Parallelism is attained by spawning extremely lightweight threads, allowing users to express arbitrarily complex parallel control constructs. Load-balancing is achieved in Cilk via *work stealing*: when a processor has no work to do, it examines another processor's work queue, picked uniformly at random, and steals work from there, if any is available. The authors have proven that the work-stealing scheduler achieves space, time, and communication bounds that are all within a constant factor of optimal.

Multipol: Multipol provides run-time support for irregular applications via distributed data structures, with a focus on hiding communication latency via asynchrony [30]. Load balancing is provided via a distributed task queue [126], but the user can tailor load-balancing as he or she desires to suit the needs of the application.

Linda: Linda provides a shared, globally-addressable, tuple-space to parallel programs [29, 51]. Applications can perform atomic actions on tuple-space, inserting tuples, and then querying the space to find records with certain attributes. Because of the generality of this model, high performance in distributed environments is difficult to achieve [11].

There have been a large number of parallel and distributed programming environments for parallel machines. Most provide some form of coarse- or fine-grained load balancing in order to deal with irregular or dynamically created tasks. However, as we will see in subsequent chapters, good load balancing does not necessarily imply good tolerance of performance faults. Many algorithms that would achieve good load balance have too much "performance trust" implicit within them. Also, these environments are designed for compute-centric applications, and therefore have little or no explicit support for I/O.

3.3 Summary

There is no such thing as a performance-homogeneous cluster. The combination of complex hardware and software systems is highly likely to lead to serious and often unexpected performance variations under typical circumstances. Although some examples of performance heterogeneity are more static in nature, and perhaps can be dealt with by fiat, others are dynamic, hard to avoid, and can lead to erratic global system performance in the common case. System design must be infused with this knowledge from the beginning, and should provide mechanisms that allow the system to cope with component fluctuations.

Table 3.1 presents a summary of the performance faults documented within this chapter. Of particular interest are the rightmost two columns, which list the duration and utilization of the faults. In general, most of the faults last for at least the length of an entire program run, and the utilization of the faults range from mild at 0.09 to quite extreme at 0.80, with most faults in the 0.50 to 0.67 range.

Many of the cases of performance variations we have documented come from research papers in well-controlled laboratory settings, often running just a single application on homogeneous hardware. We can only imagine that the variations present in real-world environments, with multi-user workloads and heterogeneous machines, would be noticeably worse.

In examining previous parallel file systems, databases, and programming environments, we have seen that I/O systems mostly exploit static notions of parallelism, via striping, hash-partitioning, and range-partitioning of data over multiple disks. All of these uses of parallelism rely too heavily

System	Device	Explanation	D	U
UltraSPARC-I [76]	CPU Logic	Alignment of instructions in instruction cache	Run	0.67
HP PA-RISC [25, 108]	Cache/TLB	TLB replacement policy and bad cache lines	Forever	0.50*
SPARC Viking [7]	Cache	Poor implementation lead to lower associativity	Forever	0.44
Vector Memory [99]	Memory	Sequential programs use memory bandwidth	Periodic	0.50
Page Placement [31]	Cache	OS page placement affects cache mapping	Run	0.50
Vesta [36]	Disk/CPU	Variable/poor performance due to interference	Run	0.80
NOW-Sort [10]	Disk/CPU	Foreign agents steal machine resources	Run	0.80
	Disk	SCSI bad-block remapping	Forever	0.09
San Diego Sort [102]	Disk	Unknown disk problem	Forever	0.29
Illinois Panda [127]	Disk	Statically different disks	Forever	0.50
Tertiary Disk [119]	Disk	SCSI timeouts and parity errors lower performance	Periodic	0.50*
Tiger Server [22]	Disk	Thermal recalibrations	Periodic	0.33*
Multi-zone Disks [88]	Disk	Higher densities on outer tracks of modern disks	Forever	0.50
Solaris File System**	Disk	Layout of files degenerates due to fragmentation	Run	0.44

Table 3.1: **Summary of Performance Faults.** *The table presents a summary of the performance faults documented in this section. The first column gives a name to each fault, in accordance with the subsections of this section. The second column lists the affected component of the system, such as the CPU logic, cache, TLB, memory system, or disk. The third column, labeled 'Explanation', describes the particular performance fault. The final two columns, 'D', 'U', give the duration and utilization of the faults, respectively. A duration of length 'Run' means that that fault will likely affect the length of an entire program run, 'Periodic' implies an intermittent behavior, and 'Forever' implies that the fault is permanent. The utilizations are just the fraction of the resource utilized by the fault. Finally, '*' means that the utilization is an estimate, and '**' means that the Solaris File System data was introduced in this document.*

on the performance consistency of each underlying component to achieve good global performance. More dynamic uses of parallelism are found in parallel programming environments, as many of these systems have had to deal with programs with irregular or dynamically generated parallelism. However, these do not directly solve the problems that performance faults cause, and are not tailored for an I/O-intensive environment.

Chapter 4

River Design and Implementation: An Overview

The next four chapters cover the design and implementation of *River*, an adaptive cluster programming environment for data-intensive cluster applications. River provides a standard data-flow application environment, with influence from Volcano [56] and Gamma [41], which eases the construction of a broad class of applications. Moreover, River provides adaptive mechanisms that allow large-scale applications to cope with dynamic performance faults, thus facilitating performance availability.

Two distributed algorithms form the core of performance availability in River. The first is a *distributed queue*, which allows consumers of a given data set to consume data at variable rates and thus tolerate consumer-side performance faults. The second is known as *graduated declustering*, which utilizes replication to tolerate producer-side performance faults. By using the two mechanisms in tandem, applications can tolerate a range of performance faults, seen in Chapter 12.

Both distributed queues and graduated declustering are examples of feedback-driven distributed algorithms. Feedback allows an algorithm to gauge the performance of underlying components at run-time, instead of making *a priori* assumptions of their performance characteristics. By actively monitoring communication channels with remote entities, each agent in the system can make better data movement decisions, hopefully avoiding performance-faulty machines.

In this chapter, we begin with a description of the abstract problem that we are attempting to solve: how to move data from a set of P producers to C consumers in a performance-robust manner. We then present a set of design principles, which will guide us during the process of system construction. We proceed by detailing our particular software environment, and then explain the specifics of the River programming environment. Subsequent chapters cover the design and implementation of both the distributed queue and graduated declustering, and a discussion follows.

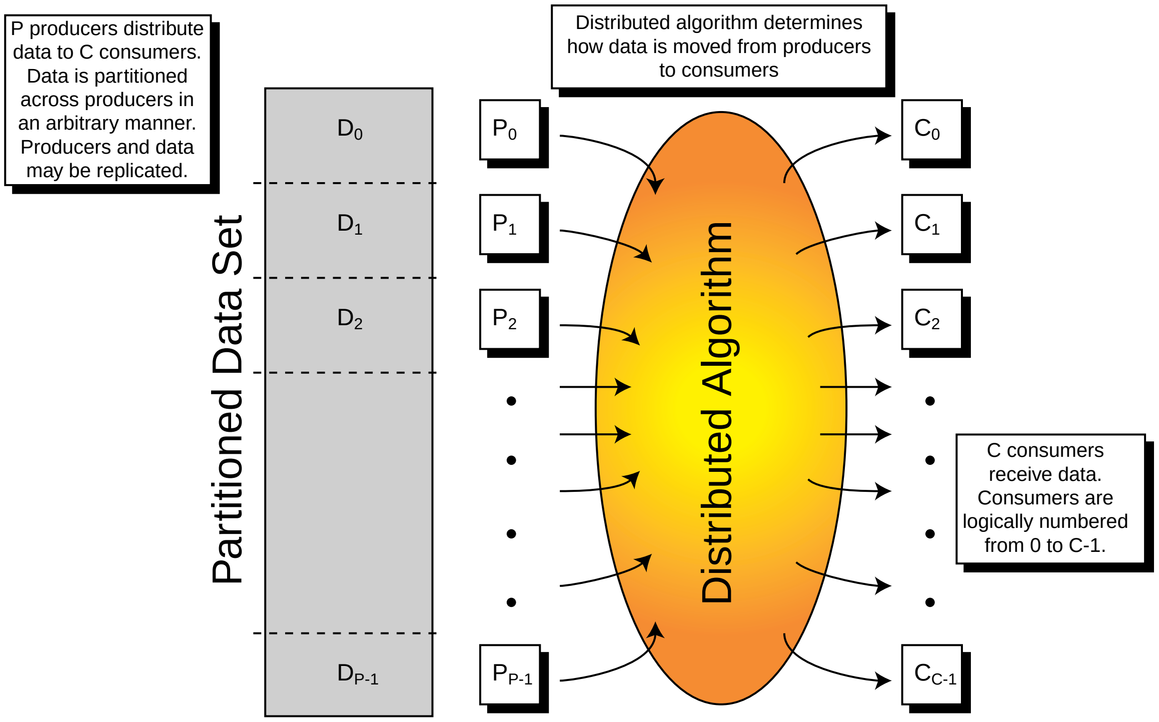

P producers distribute data to C consumers. Data is partitioned across producers in an arbitrary manner. Producers and data may be replicated.
Distributed algorithm determines how data is moved from producers to consumers
Partitioned Data Set
D0
D1
D2
DP-1
P0
P1
P2
PP-1
Distributed Algorithm
C0
C1
C2
CC-1
C consumers receive data. Consumers are logically numbered from 0 to C-1.

Figure 4.1: **The Abstract Problem.** *In this figure, we present the abstract data transfer problem. The data set D is partitioned among P producers. We wish to transfer the data set to the set of consumers, C. The particular distributed algorithm we employ will depend on both the constraints of the application as well as the desire to tolerate performance faults at the producers, consumers, or both. Both producers (data sources) and consumers (data sinks) may be replicated.*

4.1 The Abstract Problem

In this section, we reduce the general problem of providing support for data-intensive cluster applications to a simpler, higher-level abstraction. By doing so, we hope to gain insight on the salient aspects of the performance-availability problem.

In subsequent subsections, we will map the problem from the abstract into two concrete constructs: distributed queues and graduated declustering. Each of these mechanisms is a specific solution to a portion of the general data-movement problem presented below. In later chapters, we will demonstrate the utility of these constructs by using them to improve the performance availability of applications.

Figure 4.1 depicts our generalized problem. In this figure, we can see that there are a set of P producers, with a data set D partitioned among them, and C consumers, who wish to access that data in parallel. The exact semantics of which data items can be delivered to which consumers are application specific. For example, the consumers may desire a particular partitioning of the data among themselves, or they may have no preference whatsoever for any particular data item.

Further, either producers or consumers may be replicated. For example, a data set D might be replicated exactly once; in that case, each partition, D_i, would be represented by two producers, P_i and P_i'. Thus, the data set D_i would be available from two locations. A logical consumer C_i could also be replicated. In that scenario, data that received by C_i would also be received by its replicas.

We define $T_{transfer}$, the data transfer time, as the time to successfully complete the movement of D from the set of producers P to the set of consumers C. We are interested in minimizing $T_{transfer}$ under perturbations to producers, to consumers, or to both. Thus, our refined goal is to design mechanisms to facilitate performance-available data transfers between producers and consumers, subject to application-specific data-movement constraints.

Two common performance perturbation scenarios are likely to arise. In the first, one or more consumers of the given data set suffers from a performance fault, and runs at a slower rate than the others. We term this the *consumer problem.* The analogue to the consumer problem occurs when one or more producers of the data is perturbed; we call this the *producer problem*.

Our constraints in providing performance-availability mechanisms for these two problems are as follows. Where we can send a particular data item *to* affects how effectively we can deal with the consumer problem. Thus, if a particular data item must go to a particular consumer, and that consumer is perturbed, the data transfer is not likely to be performance robust. Similarly, where we can retrieve a particular data item *from* affects how well we can deal with the producer problem. If the data item is only available from a particular producer, and that producer is perturbed, then performance robustness is again difficult to attain.

Note that in a specific data transfer, the producers or the consumers, but rarely both, will be the bottleneck for the transfer. Thus, we are interested in perturbations to the bottleneck resource; perturbations to other components of the system will likely go unnoticed.

In summary, we have presented the general data transfer problem of moving a data set D from a set of producers P to a set of consumers C. We wish to build mechanisms to make such data transfers performance-robust. Two specific problems are likely to occur. The consumer problem arises when consumers are the bottleneck and suffer from performance faults. Similarly, the producer problem takes place when producers are the bottleneck and suffer from performance perturbations.

4.2 Design Principles

With the abstract problem defined, we now lay out three design principles that we use in making specific implementation decisions about our system. During the design stage of a system, having such principles is crucial: without them, it is difficult to separate the necessary implementation details from those that are artifacts.

Principle 1: Leave performance-robustness to the application. Our first principle will be to leave the addition of performance-robustness to the application programmer, who presumably best understands their application, and how to add robustness to it. Thus, instead of attempting to automate this process, we will provide what we believe are the necessary tools for performance availability, and the programmer will choose to apply them where appropriate.

This "application-specific" argument meshes well with recent work in the field of operating systems, where many researchers have simultaneously realized the folly of attempting to construct a truly general-purpose system [17, 48, 111]. Instead, many have chosen to provide basic, powerful primitives that do not dictate specific usage policies, and let applications tailor the system to their needs.

Principle 2: Take advantage of application flexibility. As a corollary to the first principle, this second principle dictates that one of the best ways in which to achieve performance availability is to take advantage of application flexibility. For example, some applications have relaxed ordering constraints on how they process data; by exploiting such semantics, performance-robustness can often be more easily attained, as we will see in later chapters.

This principle has the most impact on system interfaces, in particular data-access and data-movement interfaces. Thus, we desire interfaces that do not impose arbitrary constraints upon applications.

Principle 3: Focus on demands of data-centric applications. Finally, the third principle states that we should focus on the specific needs of data-centric applications in cluster environments. Solving the general performance-availability problem is difficult; by restricting the scope of the problem to I/O-intensive applications, we hope to uncover deeper insights into our realm of interest.

Some of the benefits of this focus are as follows. Our main techniques can work well for only coarse-grained data movement, which is likely to be found in a data-centric environment, instead of supporting arbitrarily fine-grained computations. Further, disks are often the sources, sinks, or both for computations, and thus the main performance bottleneck. Thus, we can often concentrate on making applications tolerant only to disk performance faults, thus reducing our problem to one that is more tractable.

As Lampson wrote, “Make it fast, rather than general or powerful.” [78], page 4, paragraph 14. We take this advice to heart.

4.3 Assumptions

We now outline some of the base assumptions that we make in building the River system. Along with our design principles, these assumptions serve to limit the scope of our problem, as well as to help us hone in on our particular contributions.

Performance faults are the focus, not all types of faults. In this dissertation, we focus strictly on the performance aspects of building software systems. Thus, we do not investigate the construction of a system that is both tolerant of performance faults as well as correctness faults. In future work, it would be of great value to marry the results of our work with previous research on distributed system design, and thus build a system robust to both performance and correctness faults.

The scale of our system is on the order of 100 machines. Because we are interested in high-performance clusters, the question of scale arises. In this work, we concentrate on a design for clusters of roughly 100 machines; not surprisingly, this matches the size of our cluster of workstations. Though we believe many of our techniques would be successful on larger clusters of machines, we do not show that to be the case.

The network that connects cluster is a high performance, switch-based network. We also assume the presence of a high-performance switch that connects of the machines of the cluster together. In particular, we assume that the aggregate bandwidth of the switch should at least roughly equal the total bandwidth available from all disks of the system. If the network were not capable of meeting our demands, it would become the bottleneck in almost all data transfers, and therefore much of our work would not be feasible. Fortunately, modern networks are capable of handling such high demands, assuming that they are functioning properly.

4.4 Software Environment

Before describing the details of the implementation, we present background information about the software systems that we utilize to build the River system. The River prototype currently runs on a cluster of Sun Ultra1 workstations connected together by the Myrinet local-area network [21]. More details on the exact hardware configuration will be presented later, in Chapter 8.2.

Three separate software systems are employed. The first is the single-workstation operating system, Solaris 2.6, a modern multi-threaded version of UNIX, which presents users with the familiar UNIX programming environment [74, 101]. Salient aspects include the UNIX file system (UFS), buffer management subsystem, and threads libraries. We also make heavy use of the dynamic library support, as well as the `proc` file system.

The second and perhaps most important piece of software is the communication layer. All communication is performed with Active Messages (AM), a second generation communication layer designed for distributed computing [85]. With heavy influence from a long lineage of supercomputer and MPP implementations, AM exposes most of the raw performance of Myrinet to the cluster while integrating smoothly with more modern features such as threads, blocking on communication events, and multiple independent endpoints.

Finally, a centralized name service is sometimes utilized as a global bulletin board of non-persistent information, which is useful in constructing distributed services [4]. However, the performance characteristics of the name server are poor, so we only use it infrequently, in particular to boot-strap communication connections.

4.5 Basic River Components

This section describes the River programming environment, including both the data model and programming model. River presents users with a highly flexible, component-oriented application environment, which allows applications to be written in a natural, data-flow style. The programming model is heavily influenced by parallel database systems such as Gamma [42] and Volcano [56].

In constructing both the data model and programming model, we attempt to adhere to our design goals to leave performance availability to the application, to take advantage of application flexibility, and to focus on data-intensive applications.

In this section, we first present the River data model: how data is stored and accessed on disk. We continue by explaining the components of the River programming model, including details of how a typical River program is constructed.

4.5.1 The Data Model

Single Disk Collections

On a single disk, data is represented as a group of on-disk records known as a *collection*. Each record has a set of named fields, which can be of various types, though all records of a given collection have the same fields. This catalog of information is kept as meta-data by the system.

Data can be accessed on disk as an *unordered collection*. Unordered collections provide no ordering constraints between records of the collection. Thus, an application reading such a collection may receive records in an arbitrary order, subject to optimizations by the system. Because some applications can make use of an unordered collection of data, we do not wish to impose arbitrary ordering constraints, in line with our second design principle. Other systems, such as traditional UNIX file systems, provide only a single file abstraction: a randomly-accessible, ordered array of bytes [101].

When ordering is desired by the application, data can be accessed as a *stream*, similar in nature to a Fortran or VMS sequential file [12, 70]. A stream is an ordered set of records. When an application writes a collection to disk as a stream, the write order is preserved; applications accessing the collection directly will receive the records in that order.

The current implementation uses the underlying Solaris 2.6 UNIX file system (UFS) to implement record collections. To read from disk, we use either `read()` with `directio()` enabled, or the `mmap()` interface, both of which deliver data at near the raw disk rate for sequential accesses. With `directio()` enabled, data read from disk bypasses the buffer cache, whereas simple use of the `read()` interface without `directio()` leads to double-buffering inside of the file system, which is undesirable for most of our applications. Writes to disk use the `write()` system call, with or without `directio()` enabled.

When implemented on top of UFS, layout information is not available, and therefore some optimizations that would be possible with unordered collections are not currently implemented in the disk manager. In current UNIX environments, a disk manager would have to be built upon the raw disk interface to exploit the full range of disk layout and scheduling optimizations.

Parallel Collections

Most of the applications in our system wish to access data spread across multiple disks and machines. To facilitate this, we provide the abstraction of a *parallel collection*, which allows the grouping of a set of single-disk collections into a single logical entity. The parallel collection facility only tracks cross-disk meta-data, such as the names and physical locations of each single-disk collection that form the parallel collection, and any desired ordering between the single-disk collections.

The way that applications interpret the grouping of single-disk collections is left entirely up to them. Thus, if there are no ordering constraints across collections, none need be enforced. If applications wish to impose a strict ordering across collections, that option is also available to them.

In the current River implementation, the parallel collection meta-data is stored in an NFS-backed file, as suggested in [79, 84]. Because NFS provides no consistency guarantees under concurrent access, all parallel meta-data operations are serialized through a single process of the application.

These operations are rare, because they only occur when a file is being opened or created, and therefore are not a performance bottleneck.

4.5.2 The Programming Model

River provides a generic data-flow environment for applications. The overall programming model draws heavily on work in the literature on parallel databases, including systems such as Gamma [42] and Volcano [56]. This general model has been shown to be useful for both database and scientific workloads [41, 128].

Applications are constructed in a component-like fashion into a set of one or more *modules*. Each module has a logical thread of control associated with it, and must have at least one input or output channel, sometimes having more of each. A simple example is a filter module, which gets a record from a single input channel, applies a function to the record, and if the function returns *true*, puts the data on a single output channel.

Modules are connected both within a machine and across machine boundaries by *queues*. A queue connects one or more producers to one or more consumers and provides rate-matching between modules. All queues in the system are throttled via flow control and thus producers do not overrun slow consumers. By dynamically sending more data to faster consumers, queues are an important construct for performance availability.

To begin execution of an application, a master program constructs a *flow*. A flow connects the desired set of modules, from one or more sources to one or more sinks. Any time a single module is connected to another, a queue must be placed between them. When the flow is instantiated by the master program, the computation begins, and continues until the data has been processed. Upon termination, control is returned to the master program.

River Modules

As mentioned above, a module is the basic unit of programming in River. Modules operate on records, calling `Get()` to obtain records from one or more input channels, and then calling `Put()` to place them onto one or more output channels. For convenience, we refer to a set of records that is moving through the system as a *message*. Logically, each module is provided a thread of control. Thus, a one-input, one-output module performs a simple loop: `Get()` to obtain records from an upstream channel, operate on those records, and then `Put()` to pass the records downstream, as illustrated in Figure 4.2.

More complex modules may have more than one input or output; in that case, they must specify the input/output number as an argument to `Get()` or `Put()`. Non-blocking versions of these interfaces are also available, as is the ability to perform a `Select()`: this operation waits until one of a specified set of channels is ready, and then returns control to the user.

In the current implementation, modules are written as C++ classes. Each module is given its own thread of control; this design has both benefits and drawbacks. The main advantage of this approach is that applications naturally overlap computation with data movement; thus, the user is freed from the burden of carefully managing I/O. However, thread switches can be costly. To amortize this cost, modules should pass data among themselves in relatively large chunks. In our experience, this has not complicated modules in any noticeable fashion; thus, we felt that the inclusion of complex

```
// module loop: get records + process
while ((msg = Get() != NULL) {
    // operate on given message
    rc = Operate(msg);

    // conditionally pass message downstream
    if (rc) Put(msg);
}
// indicate completion
return NULL;
```

Figure 4.2: **Module API.** *This code snippet represents a simple River module. The module* `Get()`*s messages from upstream, performs some operation on them by calling a user-defined* `Operate()`*, and then (conditionally)* `Put()`*s messages downstream.*

buffer management necessary to accommodate smaller chunks was not worth the implementation effort.

Queues

Queues connect multiple producers to multiple consumers, both within a single machine and across the cluster. We term the connection within a machine a *local* queue, and one across machines a *distributed* queue. During flow construction, queues are placed between modules and messages are transmitted from producers to consumers. Modules that are placed on either side of local or distributed queues are oblivious to the type of queue with which they interact.

Messages in River may move arbitrarily through the system, depending on run-time performance characteristics and the constraints of the flow. Dynamic load balancing is achieved by routing messages to faster consumers through queues that have more than one consumer.

To improve performance robustness, ordering is relaxed across queues. In a multi-producer queue, a consumer may receive an arbitrary interleaving of messages from the producers. The only ordering guarantee provided in a queue is on a point-to-point basis; thus, if a producer places message A into queue Q before message B, and if the same consumer receives both messages, it receives A before it receives B. This ordering is necessary, for example, to retain the ordering of a disk-resident stream. By attaching a single consumer to the single producer of a stream, the ordered property of the stream can be properly maintained.

Local queues are implemented as shared-memory constructs, which can be accessed by multiple threads. More interesting is the design and implementation of distributed queues, which is described in detail in the next chapter.

Flow Construction

To execute a program in the River environment, one or more modules must be connected together to form a *flow*. A flow is a graph from one or more data sources to one or more sinks, with as many intermediate stages as dictated by the given program.

```
// simple copy program
Flow f;
Module *m1, *m2;
// instantiate module instances
m1 = f.Place("UFSRead", "file=in.1");
m2 = f.Place("UFSWrite", "file=out.1");
// attach read module to write
f.Attach(m1, m2);
// execute flow
f.Go();
```

Figure 4.3: **Flow API.** *A simple reader to writer flow is shown. The `UFSRead` module reads in collection `''in.1''`; its output goes to the input of the `UFSWrite` module, which writes it to disk under the name `''out.1''`.*

There are three phases involved in instantiating a flow: construction, operation, and teardown. During construction, a *master program* specifies the global graph, describing where and how data will flow, including which modules to use and their specific interconnection. When the construction phase is complete, the master program instantiates the flow. In the operation phase, threads are created across machines as necessary, and control is passed to each of the modules. The flow of data begins at the data sources, and flows through the system as specified by the graph, until completion.

Flow construction can be performed programmatically with a simple flow API, or graphically as described below. The flow construction API is quite simple: to add a node to a graph, the `Place()` routine is called, specifying the name of the module and any arguments it might take. For example, to read an on-disk collection, the programmer might specify the UFSRead module, with an argument of the filename, as shown in Figure 4.3.

`Place()` returns a reference to the module, which is then used to attach modules together via a simple `Attach()` interface, the interface used to specify graph edges. In the figure, a simple copy flow is formed: both a Read and Write module are placed in the flow, and then attached together. Attaching two modules together places a queue between them. Modules can have more than one input or output; in this case, the user must specify extra arguments to the `Attach()` routine, to specify which input to connect to which output.

Finally, to instantiate the flow, a `Go()` interface is provided, which starts the threads, performs the necessary attachments, and waits for their completion. An asynchronous version of `Go()` is also available.

The flow description up to this point has been restricted to single-machine flow specification, for the sake of simplicity. To construct *parallel flows* across multiple machines, the programmer need only specify which nodes to place the various modules upon; local and distributed queues are inserted where appropriate, and when the program is run, it is spawned across the nodes of the system using a simple remote execution module, internal to the system. The user can add extra arguments to the `Attach()` routine to specify details about remote connections between producers and consumers. For example, whether to use a single m-to-n distributed queue, n 1-to-1 distinct queues, or an $m \times n$ fully-connected graph, can all be easily specified.

In the current implementation, numerous languages can be used to program flows. A C++ interface is available, but we have found it overly cumbersome to re-compile codes for each simple change to a flow. Therefore, we provide both Tcl and Perl interfaces, allowing for the rapid assembly of flows in a scripting language.

Finally, we have built a graphical user interface (GUI) for specifying data-flow graphs, similar in spirit to Tioga [117]. The GUI allows programmers to select modules from a module library and draw the data-flow graph as desired. The user can then execute the program, or generate the flow construction code for later re-use. The GUI also allows variables to be added to the program, thus enabling the user to easily construct generic programs. In the example of the simple copy, the user might choose to have the input and output collection names as variables, and then generate a general-purpose copy program. In general, we have found the GUI easier to use than the programmatic interface, and less error-prone.

4.5.3 Summary

We have presented the generic River data-flow environment, in which applications can be constructed by connecting various computational modules together in arbitrary flows. However, no mention has yet been made of how to improve an application's performance availability. In the next two chapters, we discuss the two key elements for doing so: distributed queues and graduated declustering.

Chapter 5

The Distributed Queue

The first of two important distributed algorithms that we will study in this dissertation is the design of a logically-shared, physically-distributed queue. By implementing such a *distributed queue* (DQ), we seek to provide a simple yet efficient mechanism to distribute data across a set of consumers, with the primary goal of supporting performance availability. Thus, as consumer consumption rates change due to performance faults, we wish to ensure that producers quickly recognize and react to such faults.

In this chapter, we present the design and implementation of the distributed queue. We describe the high-level design goals, basic interface, algorithms to implement the interface efficiently, and finally, a discussion of limitations and related work. Note that in the River environment, the programmer of a module will not use this interface directly, but instead would use the River module interface, which in turn would call the distributed queue internal interface as appropriate.

5.1 Desired Behavior

We first discuss the desired behavior of the distributed queue. As shown in Figure 5.1, we wish to arrive at a design that provides a constraint-free transfer of data between an arbitrary number of producers and consumers.

In the figure, we can see that the distributed queue is placed between P producers and C consumers, and has the following behavior. Data placed into the queue by one of the producers will be sent to exactly one of the consumers. The ideal queue will deliver the data to consumers at rates proportional to their rates of consumption. Thus, if over a fixed time interval, C_0 consumes at rate R_0, and C_1 at rate R_1, the ratio of data received by C_0 as compared to C_1 should be $\frac{R_0}{R_1}$.

Of course, the rates of consumption at the consumers may change dramatically over time, subject to performance faults. Therefore, we would also like our design to quickly adapt to such changes.

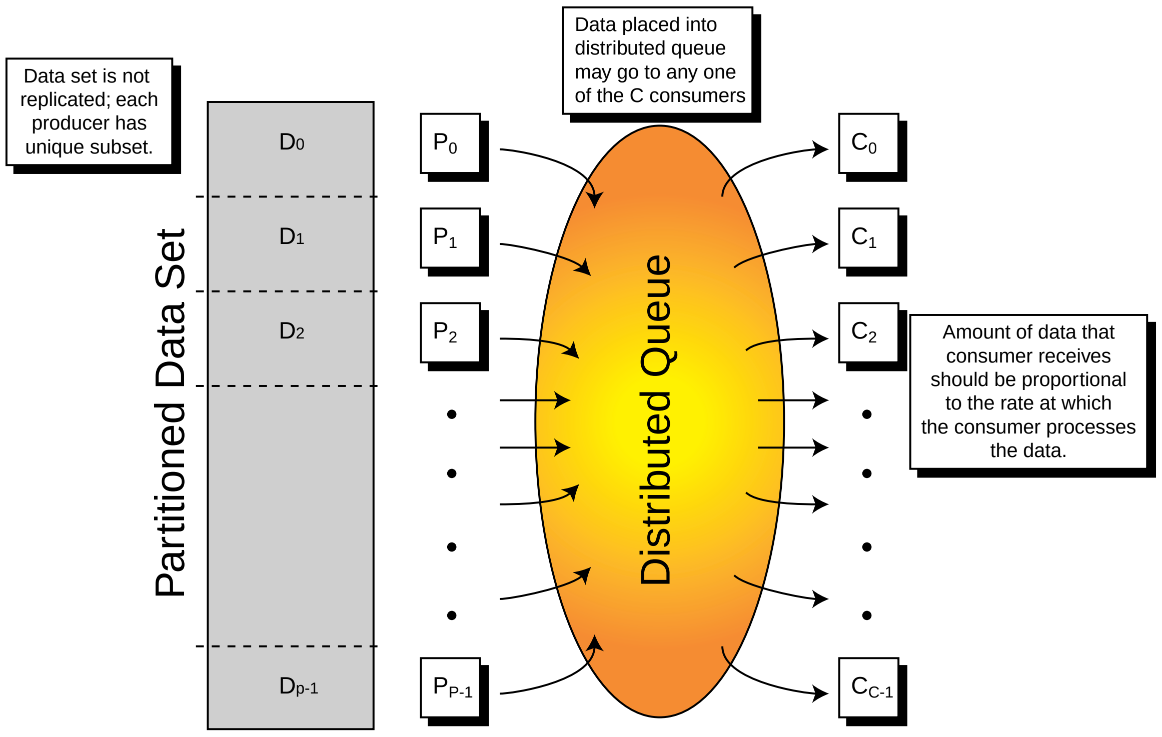

Data set is not replicated; each producer has unique subset.
Data placed into distributed queue may go to any one of the C consumers
Amount of data that consumer receives should be proportional to the rate at which the consumer processes the data.
Partitioned Data Set
D0
D1
D2
Dp-1
P0
P1
P2
PP-1
Distributed Queue
C0
C1
C2
CC-1

Figure 5.1: **The Distributed Queue.** *We present the abstract view of a distributed queue. P data producers, each with a portion of the data set D, distribute data to C consumers. There are no constraints on data movement – any consumer can receive data from any producer. Further, there is no use of replication, at either the producer or consumer. The desired behavior is that faster consumers should receive proportionally more of the data, and that changes in rates of consumption should be noticed immediately.*

```
// begin joining the queue
DQ(String QueueName, int UniqueID,
   int MemberCount, bool AmIAProducer);
// wait for all members to join
void Wait();

// leave the queue
~DQ();

// blocking put interface
void Put(char *Buffer, int Size,
         uint Arg1, uint Arg2, uint Arg3, uint Arg4);
void PutSync();

// blocking get interface (returns NULL on END-OF-STREAM)
char *Get(int &Size, uint &Arg1, uint &Arg2,
          uint &Arg3, uint &Arg4);
```

Figure 5.2: **DQ Interface.** *This figure shows the basic library interface to the distributed queue. Note that this is an internal interface, not available directly from the River programming environment. The join protocol includes the object constructor and `Wait()` routines; the leave protocol consists of only the object destructor; and the `Get()` and `Put()` routines constitute the data transfer interface. Note that the data buffer and size are what we are primarily interested in, but that the four arguments are added as a convenience.*

5.2 Interface

There are two important components to the distributed queue interface. The first specifies how members join or leave the queue. The second is the data transfer interface. Both components of the interface are shown in Figure 5.2. We discuss each component in turn.

5.2.1 Join

To join a particular distributed queue, a client follows a two-phase protocol. First, the client begins the process by creating a DQ object, and specifies the following arguments: `QueueName`, `UniqueID`, `MemberCount`, and `AmIAProducer`. Both the `QueueName` and `UniqueID` together form a unique identifier for this particular distributed queue; specifying a different name or ID number will join a different distributed queue. The `MemberCount` argument informs the system of how many total members to expect; total member count is the sum of producers and consumers. Finally, the client informs the system whether it is a producer or not. After instantiating the DQ object, the client must complete the protocol by calling the `Wait()` routine. This blocks until all of the members have joined the queue before allowing any data transfer to take place.

5.2.2 Data Transfer

Once the join phase is complete, producers can begin sending data, and consumers can begin receiving data. To place data into the queue, a producer calls `Put()`, with appropriate arguments. This routine returns when the buffer and arguments have been accepted by the system; at that point, the buffer can be re-used [1]. On the consumer side, consumers call `Get()` to retrieve data from the queue. The function returns a buffer and arguments that one of the producers has placed into the system, or returns NULL if there is no more data present and all producers have left the queue.

On the producer side, if a producer wishes to guarantee that all of its data has been received by a consumer, `PutSync()` can be called, which blocks until that is the case.

The only data ordering that is guaranteed by the distributed queue is on a point-to-point basis: if a particular producer p_i places data block B_1 into the distributed queue before B_2, and a consumer c_i receives both data items, c_i will receive B_1 before B_2.

5.2.3 Leave

To leave the queue, each producer deletes its DQ object. When this occurs, the system sends a notification to each of the consumers that this particular producer has exited the system. When all of the producers have exited the system, a consumer that attempts to `Get()` data from the queue will receive a end-of-queue notification (NULL), and thus know that they too should leave the queue, and do so by deleting the DQ object.

5.3 Implementation

We now turn to the implementation of the DQ interface. First, we again discuss the construction of the join protocol, the data transfer implementation, and the leave implementation.

5.3.1 Join

When joining a queue, each client, whether producer or consumer, creates an Active Message endpoint, and then registers two pieces of information with a centralized name service: the endpoint name, so that other members of the queue will know how to communicate with this client, and whether this client is a producer or a consumer.

When the `Wait()` routine is called, each client contacts the name service to look up the list of clients who have joined so far, until all members have joined. Once all have joined, each client walks the list of queue members, and binds each remote name into its communication endpoint; binding is required by Active Messages to enable communication. When the `Wait()` routine returns, the clients are ready to communicate with one another.

[1] Even though the buffer can be re-used, this does not imply that a consumer has received the data when `Put()` returns; the data could be in the network, for example.

```
// wait for some credits to become available
while (TotalCredits == MaxOutstanding) {
    Comm_WaitForMessage();
}
TotalCredits++;

// find random consumer that meets threshold criterion
do {
    c = Random() % TotalConsumers;
} while (Credits[c] == Threshold);

// send data
Credits[c]++;
Comm_Send(c, Data);
```

Figure 5.3: **DQ Algorithm: Producer-side Put Internals.** *The basic producer side of the DQ* `Put()` *routine is shown. When a producer has data to send, it first waits for some credits to become available. Then, it randomly picks a consumer that has some per-destination flow control available, and sends the data to that consumer.*

5.3.2 Data Transfer

The most important aspect of the distributed queue implementation is the data transfer protocol. There are many implementation possibilities here: should producers push data at consumers, or should consumers pull data from producers? How much information should be exchanged about the relative rates of execution of consumers? The main concern in the construction of the data transfer protocol is to make no performance assumptions – producers must not rely on *a priori* performance characteristics of consumers.

The basic algorithm is show in Figure 5.3. There are two key ideas which are combined to arrive at the algorithm: randomness, when picking among equivalent consumers, and feedback, via flow control. The latter of these is the critical element to attain the desired behavior under consumer perturbation.

When a producer has data to send to a consumer, it calls the `Put()` routine. Internally, the library goes through the following three steps. First, the producer checks to see how many total outstanding messages it has in the network. If there are "too many" outstanding, as determined by a threshold value, the producer waits until at least one has returned. Once there are credits available, the producer has to pick a particular consumer to which it will send the given data. It does so by picking a random consumer, and then checking to see if there are already too many outstanding messages to that consumer. If there are "too many" to that consumer, the producer picks another consumer and repeats the check. If not, the producer proceeds, and performs the third step of the algorithm, which is to send that data to the chosen consumer.

The reason for two levels of credit is as follows. When there are no credits available, the desired behavior of the producer is to wait for a message to return; the value `TotalCredits` provides a simple check for this condition. Thus, though the first counter is redundant, it simplifies the implementation.

On the receive side, the implementation is straight-forward. When a consumer calls `Get()`, it waits for a message to arrive by using the event mechanism available from the Active Message layer. When that message arrives, the DQ library wakes up and extracts the message from the network by polling. The message is packaged and returned to the consumer. The DQ library also sends a reply to the producer, indicating that it has received the message. When the producer receives this reply, it updates the flow control counters mentioned above.

5.3.3 Leave

Finally, when leaving the queue, a producer deletes its DQ object. The deletion has the side effect of sending a message to each consumer informing them that this producer has left the queue. When all producers have left the queue, each consumer knows that no more data will be sent to them. Thus, after they have consumed any remaining data, the `Get()` call will return NULL, indicating an end-of-stream. The consumer should then delete its DQ object, which removes the DQ information from the name server, completing the session.

5.4 Discussion

5.4.1 Why does it work?

The basic DQ algorithm relies heavily on flow control to achieve the desired behavior. The flow control works at two different levels. First, there is a counter of the total amount of credits available represented by the variable `TotalCredits` in Figure 5.3. The number of credits is equivalent to the total number of messages that can be outstanding at any given time from this producer to all other consumers. The counter is incremented on message sends, and decremented on reply receipt, and serves to ensure that there are only a finite number of total messages in the network at once, and provides an easy check as to when the producer should block and wait for replies.

Second, there is the per-consumer array `Credits`, which is incremented upon message send, and decremented upon reply receipt as well. This array is the method in which the algorithm builds "performance trust" – if a consumer replies frequently, its counter in the array will more often be low or zero, and therefore that consumer will receive more data. If the consumer is sent data, but does not consume the data, the array will eventually indicate this to the producer, which will then choose another target. Thus, the array serves as the producer's window into the progress of remote consumers.

Note that when the consumers are the bottleneck in the data transfer, the producers will have most of their credits outstanding, and therefore the algorithm degenerates to a pull-based, feedback-driven system, with each message reply essentially pulling the next piece of data to the replying consumer. When the consumers are not the bottleneck, the producers have plenty of credits, and the algorithm degenerates to a completely randomized choice of consumers. In that scenario, we believe that the random selection of a consumer is more stable than a deterministic choice, and less likely to lead to unexpected performance problems, in accordance with advice from Brewer and Kuszmaul [27].

5.4.2 Push-based or pull-based?

There is also the question of whether to utilize a push-based or pull-based algorithm, which we believe is a false dichotomy. The algorithm described above is push-based on the surface, as producers pick a consumer and push data to them. However, when the consumers are the bottleneck, each reply that is sent from a consumer to a producer implicitly pulls the next piece of data to that consumer. Thus, it becomes much more like a pull-based algorithm when consumers are the bottleneck.

5.4.3 Isn't it load-balancing?

It would seem that the functionality that the distributed queue is providing is only load balancing, which has been studied extensively in the literature [2, 20, 67, 126]. However, there are many effective load-balancing algorithms that are not performance-available; they make performance assumptions of one form or the other, and thus do not meet our demands.

For example, a centralized scheme could use a single machine as a rendezvous point, perhaps best matching producers with consumers for each data item. The performance assumption that this algorithm makes is that the centralized match-maker will not suffer from performance faults. If it does, the performance of the entire system suffers.

More advanced schemes have been proposed in [2, 67]. The basic algorithms therein utilize a probe-then-send model. In this scheme, n consumers are chosen uniformly at random, and then queried as to their current queue length. When all of the replies filter back, the producer picks the consumer with the least data in its queue, and sends the data to it. The performance assumption that this family of algorithms makes is that the probes will return in a timely manner. However, if any one of the probed consumers exhibits performance deficiencies, the result is that the producer spends too long of a time waiting for a response to its query.

5.4.4 What are the limitations?

One problem with the distributed queue algorithm is that due to interactions with the underlying message layer, it does not work well with "large" messages. A large message is one that must be segmented into smaller units in order to be transmitted to a remote host, due to constraints dictated by the underlying message layer. If the large message must be sent to a single consumer, the algorithm described above will not work properly – in sending a long message to a single host, the producer will use all of its flow control credits for that host, and if that consumer is perturbed, the producer's performance will become proportional to the rate of the slow consumer throughout the entire transfer. We will discuss how to work around this problem when it occurs in later chapters.

Another problem with this implementation is that although the data transfer protocol is performance robust, as we show experimentally in the next chapters, the join and leave protocols are not. Therefore, if a single member of the queue is slow during the join or leave phases, it will have a severe global performance impact. Though this could be remedied with a more sophisticated set of protocols, we found that this dramatically impacted code complexity, while solving what is only a second-order performance problem. As most time is spent in data transfer, Amdahl's Law [3] and Occam's Razor [121] combined to suggest the retention of the basic join/leave protocols.

Finally, the distributed queue does not contain any notion of locality. Thus, there is no provision for applications that would like to bias a particular producer to deliver its data to a particular consumer. In the future, a preference-aware distributed queue might be of some value.

Chapter 6

Graduated Declustering

The second core distributed algorithm that we will study is called *graduated declustering* (GD). Whereas distributed queues can be used to circumvent the consumer problem, they are not of use when the producer of a given data set is perturbed, and thus becomes the bottleneck for the data transfer. Thus, graduated declustering can be used to cope with the producer problem.

In this circumstance, if there is no alternate source for the perturbed portion of the data set, there is little one can do – the data transfer will execute at the rate of the slow producer. However, if the data set has been replicated to one or more alternate sites, the question arises as to how to best make use of those replicas.

In this chapter, we will explore a particular distributed algorithm for utilizing mirrored data sets to solve the producer problem. By carefully scheduling bandwidth usage from the perturbed producers among consumers of the given data set, consumers all progress at the same rate, and thus no excessive harm arises from producer-side performance faults.

Graduated declustering is particularly useful in the case when an application is reading a large data set in parallel. In that instance, disks are the producers of the data, and the parallel application is the consumer. This scenario arises in many data-intensive environments, including both data-mining and decision-support systems.

6.1 Desired Behavior

We first discuss the desired behavior of graduated declustering. As shown in Figure 6.1, we are designing a construct that is slightly more complex than the distributed queue.

The basic goal is to tolerate producer-side performance faults. The manner in which we accomplish this is as follows. First, we assume that each D_i subset of the data set D is replicated at least once. If there are N copies of each subset, we will refer to that as N-way graduated declustering. Further, we assume that the $N \cdot P$ producers are in most cases physically co-located across P entities; for example, $N \cdot P$ producers are spread across a cluster of P machines. Thus, we are not assuming any extra machine resources other than the capacity for data set replication.

We now describe how N-way graduated declustering behaves. Assume that there are P machines on which producers are running. Thus, on machine M_i, there are N producers, P_i,

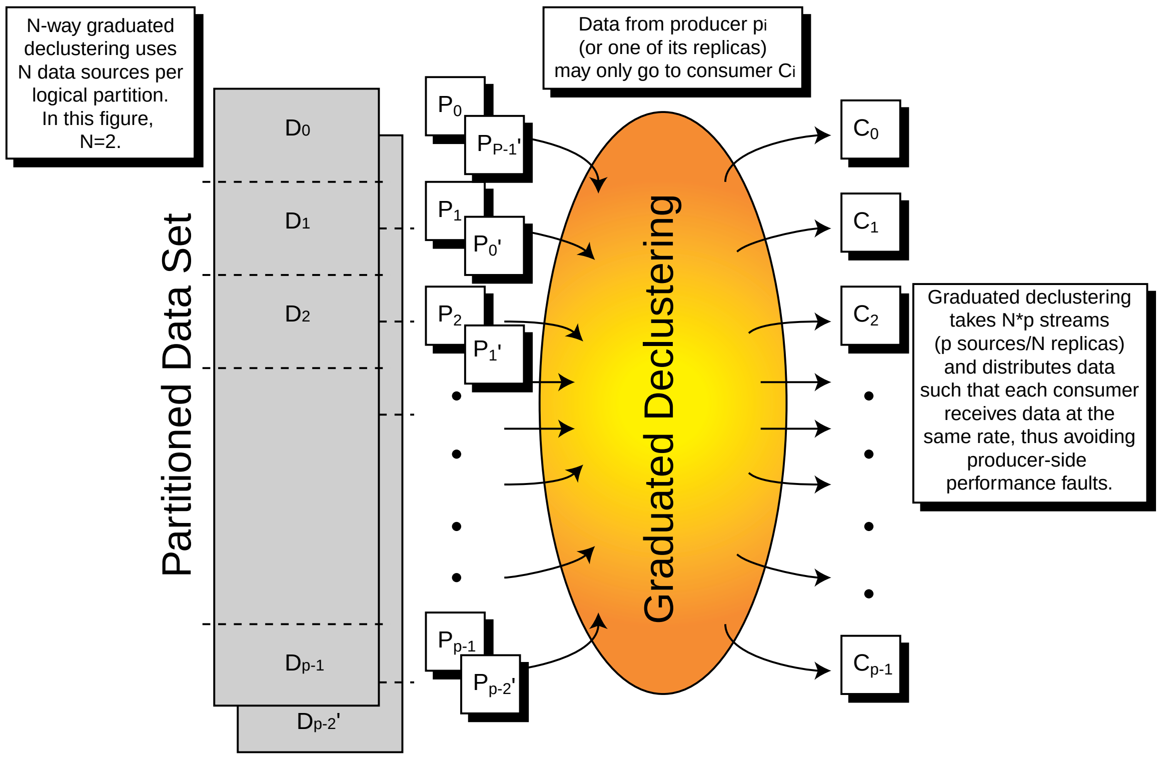

N-way graduated declustering uses N data sources per logical partition. In this figure, N=2.
Partitioned Data Set
D0
D1
D2
Dp-1
Dp-2'
P0
PP-1'
P1
P0'
P2
P1'
Pp-1
Pp-2'
Data from producer pi (or one of its replicas) may only go to consumer Ci
Graduated Declustering
C0
C1
C2
Cp-1
Graduated declustering takes N*p streams (p sources/N replicas) and distributes data such that each consumer receives data at the same rate, thus avoiding producer-side performance faults.

Figure 6.1: **Graduated Declustering.** *The abstract view of graduated declustering is shown. The data set D is replicated on a per-partition basis. In this algorithm, there is a specific data movement constraint – data generated by producer i or by one of its replicas must move to consumer i. The corollary to this states that the number of consumers must equal the number of producers. The basic algorithm functions by adjusting the rates of the producers such that each consumer receives a global proportional share of the available bandwidth.*

$P_{(i+P-1)\%P}$, ..., $P_{(i+P-N)\%P}$, all but the first of which are replicas. Assume that on P remote machines that there are P consumers, C_0 through C_{P-1}, and that each consumer C_i consumes only its portion of the data set produced by P_i and its replicas. Note that this is quite a bit different than the distributed queue, where data from any producer can be sent to any consumer, and where the number of producers and consumers is not necessarily equivalent.

Assume that each producer i produces data at a rate of R_{P_i}. All producers on the same machine share the same resource, and therefore the sum of the R_{P_i}s on a particular machine equal the rate of that bottleneck resource, R_{B_i}. Thus, the total bandwidth available across all machines is:

$$B_{max} = \sum_{i=0}^{i=P-1} R_{B_i}. \tag{6.1.1}$$

However, there will be some number of perturbations added to the system, which will take away some of that bandwidth. Each perturbation or performance fault on resource i uses R_{F_i} bandwidth. Assume that there are F faults present in the system, F_0, F_1, ..., F_{F-1}, where $0 \leq F \leq P$ Thus, the total bandwidth available to the consumers is:

$$B_{avail} = \sum_{i=0}^{i=P-1} R_{B_i} - \sum_{i=0}^{i=F-1} R_{F_i}. \tag{6.1.2}$$

The goal of graduated declustering is to take the available bandwidth and divide it equally among the P consumers, such that $R_{C_0} = R_{C_1} = ... = R_{C_{P-1}} = \frac{B_{avail}}{P}$. If this is accomplished, then all the consumers will proceed at the same rate, and all finish the data transfer at the same instant, and thus the performance faults in the system will have been tolerated as best as they could have been.

This global redistribution is accomplished via local producer bandwidth adjustments. Thus, the total bandwidth from a given resource R_i is $R_{B_i} - R_{F_i}$, and this must be divided among all of the producers that share that resource, that is P_i and all replica producers that share R_i. The one piece of flexibility we have is how we apportion the bandwidth from the producers.

To give a concrete idea of how this should work, we present an example in Figures 6.2 and 6.3. In the first figure, we present a 4-disk, 4-consumer setting, with each disk slated to deliver data at R MB/s. Without graduated declustering, each consumer would read data from the disk where the data is, with the obvious producer problem occurring if any one of the disks runs at a rate less than expected. In the example, the performance fault uses $R/2$ MB/s of the bandwidth of one of the disks, and therefore the consumer reading the perturbed partition completes its scan in twice the time of the others.

In the second figure, we see how graduated declustering solves the problem: by reallocating the bandwidths from the mirrored partitions, each consumer receives its fair share of the global available bandwidth, $\frac{7}{8} \cdot R$ MB/s. Of course, the challenge presented to the implementor is how to arrive at this global reallocation of bandwidth quickly and in a distributed manner.

6.2 Interface

The basic library interface to graduated declustering is presented in Figure 6.4. Not unexpectedly, it is nearly identical to that of the distributed queue, as both of them are specific examples of a general producer-consumer data transfer.

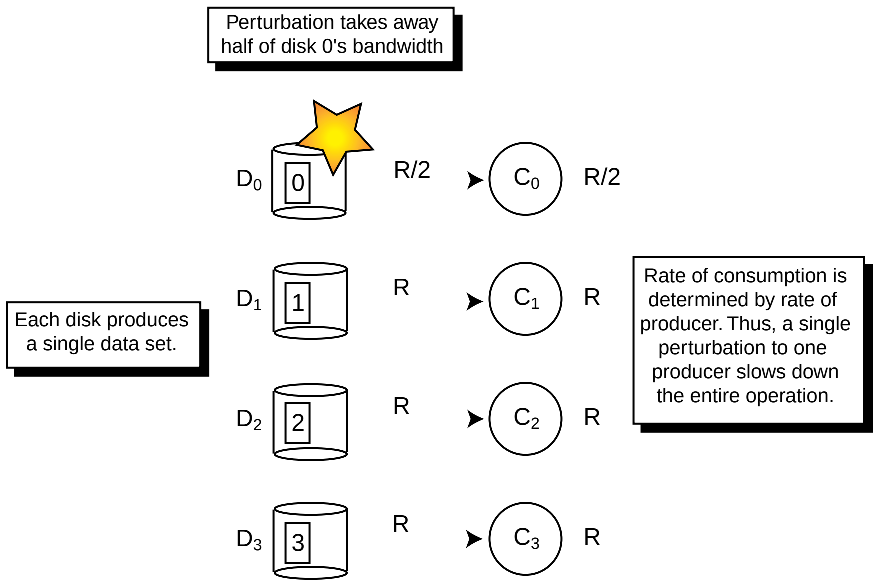

Perturbation takes away half of disk 0's bandwidth
D0
0
R/2
C0
R/2
Each disk produces a single data set.
D1
1
R
C1
R
Rate of consumption is determined by rate of producer. Thus, a single perturbation to one producer slows down the entire operation.
D2
2
R
C2
R
D3
3
R
C3
R

Figure 6.2: **No Graduated Declustering: An Example.** *This diagram depicts a parallel read from disk without graduated declustering. Unperturbed disks normally deliver R MB/s of bandwidth, and the one perturbed disk delivers half of that, R/2. Disk 0 is perturbed, and thus only half of its bandwidth is available to the application. Without graduated declustering, client 0 does not receive as much bandwidth as clients 1, 2, and 3.*

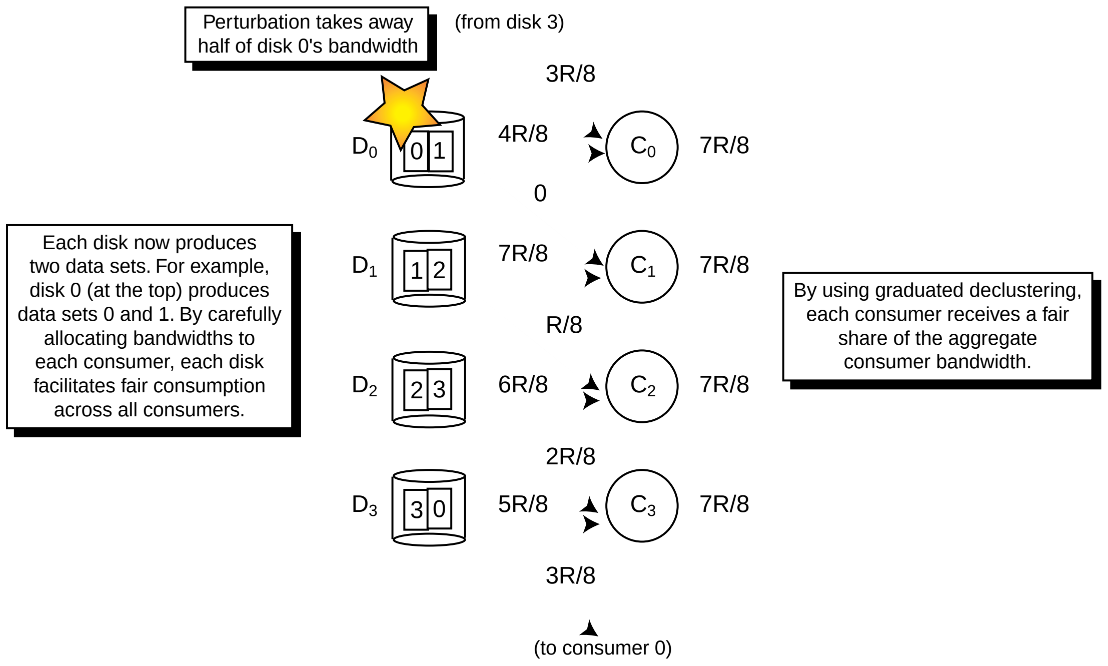

Perturbation takes away half of disk 0's bandwidth
(from disk 3)
3R/8
D_0
0 1
4R/8
C_0
7R/8
0
Each disk now produces two data sets. For example, disk 0 (at the top) produces data sets 0 and 1. By carefully allocating bandwidths to each consumer, each disk facilitates fair consumption across all consumers.
D_1
1 2
7R/8
C_1
7R/8
By using graduated declustering, each consumer receives a fair share of the aggregate consumer bandwidth.
R/8
D_2
2 3
6R/8
C_2
7R/8
2R/8
D_3
3 0
5R/8
C_3
7R/8
3R/8
(to consumer 0)

Figure 6.3: **With Graduated Declustering: An Example.** *This diagram shows how graduated declustering alleviates the problem of producer-side perturbations. Though disk 0 is perturbed, other disks compensate their bandwidth allocations such that all consumers receive a fair-share of available aggregate bandwidth.*

```
// join the group
GD(String GDName, int UniqueID, int MemberCount,
   int ProducerArray[], int Replicas);

// wait (MUST be called before sending any data)
void Wait();

// specify which data-set to consume
void GetInit(int LogicalProducer);

// leave the queue
~GD();

// blocking put interface
Request_t *GetRequest(int LogicalProducer);
void Put(Request_t *R, char *Buffer);
void PutSync();

// blocking get interface (returns NULL on END-OF-STREAM)
char *Get(int &Size);
```

Figure 6.4: **GD Interface.** *This figure shows the basic library interface to graduated declustering. The join and leave protocol are nearly identical to the DQ interface, except that each producer that joins must also specify which portion of the data set it is producing via the ProducerArray argument. The data transfer interfaces are nearly identical to that of the DQ, except that the producer must first pull a request from the consumer via the `GetRequest()` interface, and then respond to it via the slightly modified `Put()` interface.*

6.2.1 Join

The join protocol is quite similar to the DQ join protocol. Here we discuss the differences. The major departure is that each producer must specify exactly which partition of the data set that they are providing. Also, `GetInit()` must be called by each consumer to specify which portion of the data set that they are consuming. The `Wait()` routine is called in order to wait for all members to join, before any data transfer occurs.

6.2.2 Data Transfer

The data transfer interface is similar to that of the distributed queue. Producers call the `GetRequest()` routine to get requests from consumers, and then respond to those requests with the `Put()` routine. Consumers call `Get()` to obtain data. The `GetRequest()` interface blocks when there are no requests available, and the `Put()` interface will block when no more data can be accepted; similarly, `Get()` will block when there is no data ready for the consumer.

6.2.3 Leave

Finally, the leave interface is identical to that of the DQ, and is accessed via the object destructor. When producers are finished producing, they delete the object, which informs the consumers of their exit. More information on this is available in the previous chapter.

6.3 Implementation

In this section, we describe how graduated declustering is implemented. The main challenge lies in deriving a distributed algorithm for the data transfer, including details such as how producer bandwidths are adjusted, how consumers pick the producer from which to get each data item, and so forth.

6.3.1 Join

The GD join implementation is again quite similar to the DQ join implementation. All members register information with a centralized name service, and then wait for all members to join. Once all have joined, data transfer may begin.

6.3.2 Data Transfer

The most complex component of the graduated declustering algorithm is the data transfer protocol. It involves adaptation on both the consumer and producer side. Figure 6.5 shows the basic data transfer algorithms.

Consumer: Let us start with the consumer-side of the algorithm, for that is more straight-forward. In N-way graduated declustering, a consumer C_i of a particular partition of the data set, D_i, has N choices of where to request a particular data item from. To jump-start the process, the consumer sends out requests for a fixed number of blocks in the `GetInit()` routine, distributing them in a

Consumer:

```
// set-up: prefetch some requests to get things going
GetInit(int LogicalProducer) {
    PrefetchRequests(LogicalProducer, HowMany);
}

// client sends next request to replica that responded last
char *Get(int &Size) {
    M = WaitForResponse(); // data from producer
    // send next request to the producer that just responded
    SendNextRequest(M->which, Progress);
    // return last block to consumer
    return M->data;
}
```

Producer:

```
// producer threads -- application writer creates these
Request_t *R;
while ((R = gd->ProducerGet(DataSetNum)) != NULL) {
    char *Buffer = Read(R); // get data for request
    gd->Put(R, Buffer);     // return data to consumer
}

// scheduler thread -- internal to library
while (NotDone) {
    p = CalculateNext(); // decide how to bias scheduling
    ProcessRequest(p);   // send data from chosen producer
}
```

Figure 6.5: **GD Algorithm.** *This figure shows how the consumer-side and producer-side of the GD library behave. The producer side consists of a single scheduler thread, internal to the library, and then one thread per data set produced, which are created external to the library. Each external thread repeatedly calls `ProducerGet()` to get a work description, performs the specified work, and then hands the data to the library via the `Put()` call. The scheduler thread does the interesting work. Based on the progress information that is sent along with each request, the scheduler decides how to bias its processing of requests. Thus, if a particular consumer is lagging, it will get a higher fraction of service. Finally, on the consumer side, we utilize an adaptive algorithm internal to the `Get()` routine. First, a number of requests are prefetched by the `GetInit()` routine, which gives the producers some work. Second, in the steady state, as each response comes in, the next request is sent to the producer that responded. Thus, the more responsive a producer is, the more requests it receives from a consumer.*

round-robin fashion among the possible producers. Thus, if we are sending out 10 requests, and there are 2 producers, request 0 would go to producer 0, request 1 to producer 1, request 2 to producer 0, request 3 to producer 1, and so forth.

The only adaptation on the consumer side occurs when replies come back from producers. When a reply for a particular data block returns from a producer, the consumer requests the next data item from that very producer. Thus, if one of the producers is much more responsive than another, the consumer-side of the algorithm will adapt to that responsiveness by requesting more data from the more active producer.

One more aspect of the consumer-side algorithm is quite important. Along with each request, the consumer sends extra information to the producer: the rate at which it is progressing through the data set. This small amount of extra knowledge will be crucial to the producer, in determining which consumers' requests should be served. The progress metric that we use is the total number of bytes that have been received by each consumer.

Producer: The producer-side of the algorithm is more complex. In Figure 6.5, the producer is represented by two components. The first is a set of producer threads, one per replica, which each receive requests from consumers, perform the work that is needed, and then send the data back to the consumers. The client of the library must create these threads. Note that in N-way graduated declustering, each machine will receive requests from N different consumers.

The second component is the scheduler thread. This thread is crucial to the proper operation of graduated declustering. It examines the rate of progress of each consumer, and then biases the scheduling of requests so as to "catch-up" the lagging consumer. One important variable that needs to be determined is how exactly to bias the queues. If the producer-side reacts too quickly, unstable fluctuations may arise. If it reacts too slowly, it may never achieve the proper redistribution of bandwidth. We will explore some of the possibilities in the next chapter.

Our current implementation schedules requests based on the exact progress of each consumer; the consumer that is lagging is given preference over all other consumers. Thus, if a producer is serving requests from two consumers, C_1 and C_2, and C_1 has only received 100 blocks of data while C_2 has received 200 blocks, C_2 will not get any blocks from the producer until C_1 has caught up to C_2.

6.3.3 Leave

Finally, the leave implementation again closely matches the distributed queue leave implementation; more information on it is available in the previous chapter.

6.4 Discussion

6.4.1 Why does it work?

The distributed algorithm that implements graduated declustering is based on the following intuition: if a producer is able to balance the rate of progress of the consumers to which it delivers data to, and all producers strive for this localized balance, a global balance among all consumers will be achieved. The key to success is the producer-side scheduler, which is directly in charge of the biasing that must occur.

6.4.2 When are perturbations too severe?

In N-way graduated declustering, data for a particular consumer is only available from N data sources. N is often small, for example, 2 or 3, due to the high costs of replication. Not surprisingly, limited redundancy can lead to the case where the perfect fair redistribution of global bandwidth is not attainable.

Recall that the desire is for each of the consumers to receive a fair share of available bandwidth, such that $R_{C_0} = R_{C_1} = ... = R_{C_{P-1}} = \frac{B_{avail}}{P}$. However, imagine that the producers for some partition of data set are not able in sum to deliver the average bandwidth, because of a severe perturbation. In that case, this global goal is not attainable.

There is not much one can do to cope with this limitation. As is the case with redundancy to deal with actual failures, the more redundancy that is available to handle performance faults, the better.

6.4.3 What are other scheduler metrics?

In the current implementation, the consumer "piggy-backs" information in consumer requests in order to help the producer decide which consumer should receive what proportion of service. The metric in use is currently the total number of bytes that each consumer has received, summed over all of its data sources, over the lifetime of the run.

Clearly, other metrics are possible. The important aspect of the metric is that it should give some notion of global progress towards the final goal. Average bandwidth also fits this definition, when each partition of the data-set is roughly identical in size. In the future, this metric could be exposed to applications instead of kept internal to the library to allow for application-specific scheduling.

It should be noted that if there is no metric available, because, for example, the amount of data each producer will produce is unknown and likely to vary widely, the algorithm in its current form will not work. Fortunately, we are mostly interested in reading data from disks, where the amount of data to be read is always known.

6.4.4 Does each consumer have to read only its partition?

Finally, one might notice that the requirement that each consumer must read only its partition is overly restrictive. In some cases, one could imagine the utility of a general data transfer that combines the flexibility of the DQ by sending more data to faster consumers with the GD replication at the producer side.

However, we chose to implement a less general transfer for two reasons. First, it is common in many real applications that need to scan through large data sets sequentially. Second, it is much less complex to implement than the more general producer to consumer transfer. These are both in alignment with our design principles.

Chapter 7

River Design and Implementation: Discussion

In the previous three chapters, we have presented the design and implementation of the River system. First, we presented the generalized data transfer problem, three principles to guide our design, the specifics of the River programming environment, and then a concentrated discussion of both distributed queues and graduated declustering.

In closing the design component of this dissertation, in this chapter we describe the keys to performance availability, and then discuss the functionality we desire from underlying communication libraries, operating systems, and file systems.

7.1 Keys to Performance Availability

The key to performance availability in the River environment is to avoid performance assumptions in the design of distributed data transfer operations. Performance assumptions lead to the design of fragile transfers: a single misbehaving component can adversely affect the performance of the entire operation.

However, in assumption-free systems, we incur extra design complexity. Instead of making performance assumptions, the system must constantly gauge the performance of underlying components. In designing such feedback-driven systems, we must be careful to keep them as lightweight as possible; otherwise, the cost of adaptation will outweigh its benefits.

Our distributed queue algorithm uses *implicit* feedback [8]: no explicit information probes are sent from producers to consumers. Instead, the producers track the responsiveness of consumers; those consumers that respond more frequently to data requests are sent a larger fraction of subsequent data requests.

Feedback in graduated declustering is more complex, and makes use of explicit information exchanges. Consumers, in asking for data from replicated producers, continually update producers with information on consumer-side progress. The producers, in turn, make use of that information to balance the rate of progress of all of the consumers in a distributed fashion. Careful control of this feedback loop determines the effectiveness of the algorithm.

7.2 Requirements

We now discuss requirements from underlying layers, including the operating system, file system, and communication layer.

7.2.1 Operating System

The operating system that the system is built upon is Solaris 2.6.1, a modern multi-threaded UNIX [74]. We require one primary feature from it: good support for threads, which is present in Solaris. Other modern operating systems, including Windows NT, HP-UX 10.0, and SGI Irix also provide adequate thread support.

The importance of the thread system can not be understated. Threads facilitate the overlap of communication, disk I/O, and computation. By assigning each module its own thread, we free the application programmer from the concerns of explicit management of I/O. Because applications often interact with I/O devices, including disks and the network, it is critical that the thread-support is provided by the kernel, and not just a user-level package. Without kernel support, I/O operations will block and suspend the entire process, and thus lose the benefits of overlap.

Virtual memory, though perhaps not as crucial as true thread support, also simplifies application programming a great deal. Without it, buffer management becomes quite difficult, and must be carefully controlled by the user. That said, the buffer management interface currently available in Solaris is quite primitive. The only method to control application buffering, and thus avoid costly paging, is to pin down memory pages, a brute-force and costly operation. A better interface to applications, perhaps informing them of how much memory they have via upcalls [34], would be preferred.

The Solaris environment also offered some other features that were useful, although not crucial to the implementation of River. Support for dynamic loading enabled the addition of scripting capabilities. Also, the `proc` file system, and specifically the `pstack` program, which prints the stack of an active program, was quite helpful in debugging in a distributed environment.

7.2.2 File System

Because many of our applications deal with disks, the behavior of the file system can be quite important. The River system would like significant control over the disk, much of which is currently not available in modern UNIX file systems. For example, data layout is completely hidden from the application and system. Sometimes this information would be quite useful, in particular when scheduling multiple data streams onto the same disk.

Buffer management, as performed by the file system, also can lead to some difficulties. Straight-forward usage of the `read()` interface will result in file-system caching; for sequential reads of large files, the result of this is the double-buffering problem [116]: half of memory is wasted by buffering the file unnecessarily.

Fortunately, Solaris does provide alternatives to the `read()` interface. First, the `mmap()` and auxiliary `madvise()` interfaces allow advice to be passed to the operating system, which sometimes can avoid unnecessary buffering. Also, a `directio()` interface is available, which allows applications to bypass the buffer cache entirely. Though this is an improvement, there is no easy way for multiple applications to share buffers. A richer interface, in which the application could inform the underlying file system of its intended access patterns, would be useful [96].

7.2.3 Communication Layer

Finally, we discuss the functionality of the communication layer. The communication layer we use throughout this dissertation is a second-generation communication layer for clusters, Active Messages (AM) [85].

AM provides reliable, unordered delivery of messages. Reliability is required for all of the applications we have experience with to date, so placing this feature inside of the communication layer is reasonable from our perspective. Ordering sometimes is required by applications, and therefore we must explicitly construct that feature on top of AM.

Another crucial feature of Active Messages is the support for blocking on communication events. For example, when waiting for a message to arrive, a thread can go to sleep, freeing valuable CPU resources. Other fast message layers [94, 123, 124] do not support blocking on communication events and thus require polling the network interface to receive messages; boundless polling consumes many CPU cycles and is not appropriate for building an I/O infrastructure such as River.

The main point of contention is with the flow control provided by the library. Flow control limits the number of outstanding messages an application can have, which can have quite important interactions with our distributed algorithms, as we will see in the next chapter. For example, the distributed queue requires a certain number of outstanding messages to be able to move data efficiently; if the right number of outstanding messages is not provided by the message layer, the distributed queue will not function properly.

Of secondary concern is the request/response protocol that lies at the heart of AM. In some circumstances, this is quite natural, but in others, it makes programming difficult. It would be useful to provide an interface that extracted the next data item from the network and handed it directly to the thread that requested it, instead of having control pass through an out-of-context handler as is standard in all Active Message implementations [124].

Chapter 8

Experimental Environment

Before presenting results across a broad set of experiments, we describe both the simulation and prototype experimental environments. We then present a method for generating performance faults in a controlled manner into our experiments, allowing us to explore how the system copes with such faults in a controlled, scientific manner.

8.1 Simulation Environment

Along with experimental results from a prototype implementation, we employ a set of simulations to demonstrate some of the properties of our distributed algorithms. The simulator which we have constructed provides only low-level primitives to work with: *queues*, which consume data at user-specified rates, and *sources*, which generate data at user-specified rates. Simulations of both the distributed queue and graduated declustering can be readily constructed from these simple abstractions.

The simulator core consists of an event-based simulator, which takes as input actions to be executed at specified virtual times, and processes them in virtual-time order. The event subsystem is written in C for the sake of efficiency, whereas the rest of the simulation system, including queue and source abstractions, callbacks, and other glue code, is written in Tcl [92]. Tcl allows for great flexibility in assembling arbitrarily complex simulations, and in combination with the Tk Toolkit [93], allows for visualization and animation of simulated scenarios.

We employ the simulator to explore basic algorithmic behavior under a wide range of system parameters, most of which would not be feasible to measure in the prototype implementation. The combination of both simulation and implementation leads to a better understanding and separation of algorithmic properties from implementation details.

An example of a simulation of the distributed queue is shown in Figure 8.1. There are only two constructs: producer data sources, which produce data blocks at a fixed, user-specified rate, and consumer queues, which consume data at a fixed, user-specified rate. Each producer may send data to any of the consumers, using the distributed queue algorithm specified in the previous chapter. After a block is produced, it takes *latency* microseconds to reach the desired consumer queue, at which point it is placed in the consumer queue. Perturbations are modeled by altering the rate of consumption

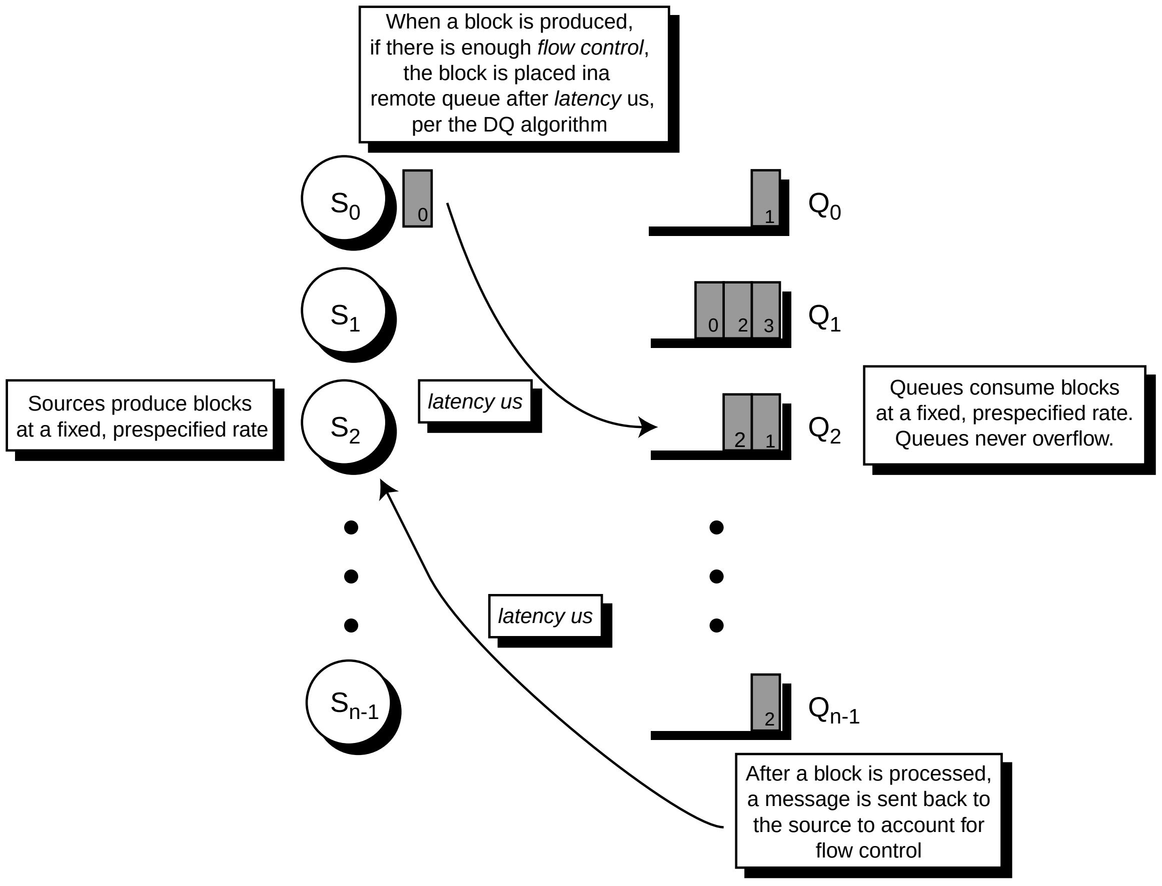

When a block is produced, if there is enough *flow control*, the block is placed ina remote queue after *latency* us, per the DQ algorithm
S_0
0
1
Q_0
S_1
0 2 3
Q_1
Sources produce blocks at a fixed, prespecified rate
S_2
latency us
2 1
Q_2
Queues consume blocks at a fixed, prespecified rate. Queues never overflow.
latency us
S_{n-1}
2
Q_{n-1}
After a block is processed, a message is sent back to the source to account for flow control

Figure 8.1: **DQ Simulation Overview.** *The distributed queue simulation is presented. Data sources on the left-hand side generate data blocks at a fixed, pre-specified rate. When a block is generated, the distributed queue algorithm chooses which consumer to send the block too, if there are any flow-control credits available, or otherwise waits for a response to previously sent messages. Sending a block to a consumer is simulated by the block arriving in the selected consumer queue 'latency' microseconds after it is sent. Once in the consumer queue, the blocks are processed in FIFO order at the rate of the consumer queue. When the block is processed, a message is sent back to the producer indicating completion, and thus increasing the number of credits by one. The simulation continues until each producer has produced a pre-specified, fixed number of blocks.*

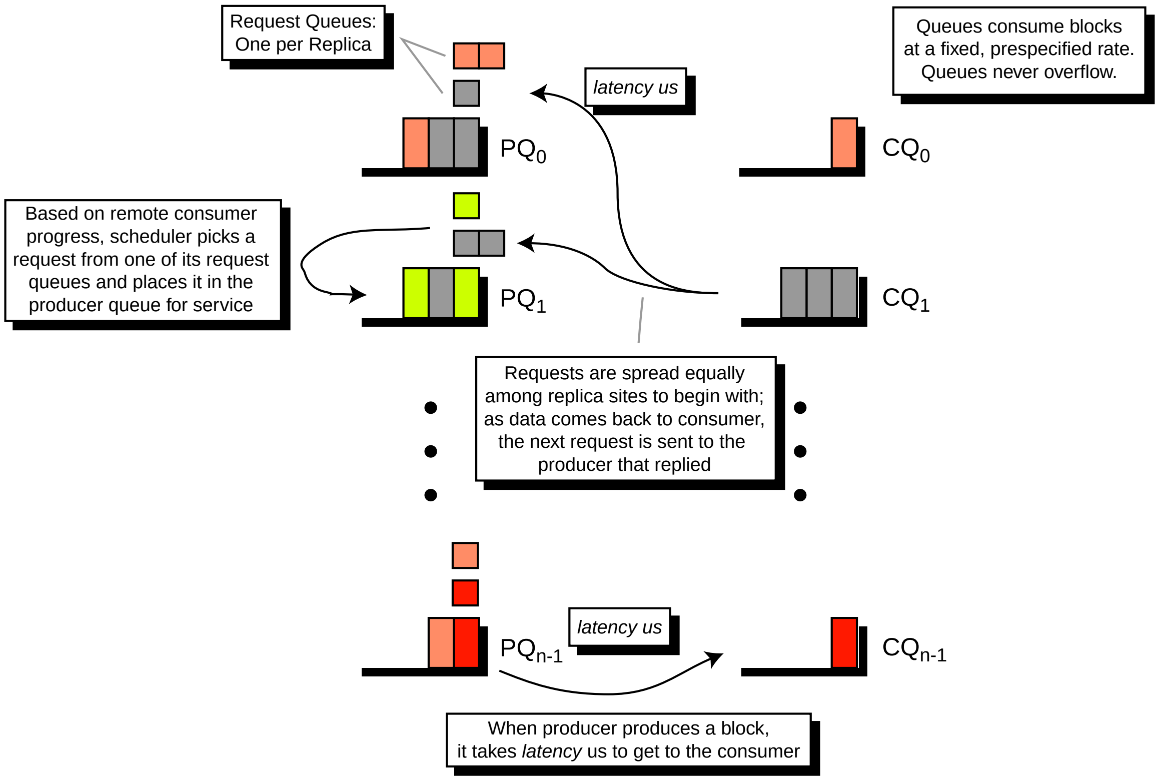

Request Queues: One per Replica
latency us
Queues consume blocks at a fixed, prespecified rate. Queues never overflow.
PQ0
CQ0
Based on remote consumer progress, scheduler picks a request from one of its request queues and places it in the producer queue for service
PQ1
CQ1
Requests are spread equally among replica sites to begin with; as data comes back to consumer, the next request is sent to the producer that replied
latency us
PQn-1
CQn-1
When producer produces a block, it takes *latency* us to get to the consumer

Figure 8.2: **GD Simulation Overview.** *The graduated declustering simulation is presented. The simulation begins when each of the consumers send requests to the data replicas for various data items. When a threshold number of requests have been received at the producer-side, a scheduler is activated, which decides which request queue to service, based on its notion of remote consumer progress (each consumer request includes the latest information on consumer progress). Serviced blocks come out of the producer queue, and then are sent back to the requesting consumer, which places the block in the consumer queue after 'latency' microseconds. When the block is finished processing, a request for the next block is sent to the producer that produced the current block. The simulation continues until each consumer has received a pre-specified, fixed number of blocks.*

of one or more consumer queues. Note that no network contention is modeled; we believe this is a reasonable simplification, as we are trying to understand the algorithmic properties of the distributed queue.

The graduated declustering simulation is depicted in Figure 8.2, and is slightly more complex than the distributed queue simulation. Each producer has *replica* number of request queues, one for each stream that it serves. A consumer that desires data from a particular producer sends a small request to the proper request queue. The producer decides which consumer request to schedule using a user-specified scheduling discipline; in most simulations, this will be the standard graduated declustering algorithm as described in previous chapters. At the beginning of the simulation, each consumer sends requests for data blocks to each of the producers that can serve those requests; as replies filter back to the consumer, the consumer issues subsequent requests to those producers who have replied.

8.2 Experimental Environment

We now describe the hardware environment of the prototype implementation. The River prototype currently runs on a cluster of Ultra1 workstations connected together by the Myrinet local-area network [21]. Each workstation consists of the following hardware:

- A 167 MHz UltraSPARC I processor [122], with separate on-chip instruction and data caches, each 16 KB in size, and an off-chip 512 KB level-two cache. The on-chip instruction cache is 2-way set associative, whereas both data caches are direct-mapped. A read from the L1 data cache takes 18 nanoseconds (3 cycles), and a read from the L2 cache roughly 54 nanoseconds (9 cycles), as measured by the Saavedra memory micro-benchmark [104].
- Two Seagate Hawk 2.1 GB 5400-RPM disks, attached on a fast-narrow SCSI bus to the main Sun I/O bus, the S-bus. One is commonly used for the OS and swap space, whereas the other is used by our system for data. These disks can deliver roughly 5.45 MB/s of bandwidth from the outer tracks, and 3.18 MB/s from the inner tracks[1].
- 128 MB of main memory, with 324 ns (54 cycle) read time. Roughly 20 MB of memory is used by the operating system.
- A single Myrinet card, also on the S-bus. Connecting the system together, these cards are capable of moving data into and out of the workstation at roughly 40 MB/s.

The entire system is connected together via a series of Myrinet switches. A single 8-port switch is capable of transferring 640 MB/s of data under no port contention. The topology of the network is approximately a 3-ary fat tree; more details can be found in [37].

[1]Note that there is some slight variation across machines; some disks have been replaced with newer disks, and three of the Ultra1 workstations are “Creators”, which have a fast-wide, not fast-narrow SCSI bus. This variation will be noted where appropriate.

8.3 Perturbations

We now discuss how we will generate performance faults into our experiments. In general, our methodology will be to slow the performance of one or more hardware resources, and to measure the overall effect on elapsed run-time. We will then compare results with that of an unperturbed system, as well as plotting ideal and static system behavior.

As suggested by Noble et. al. in [91], we apply techniques from the area of control systems to understand how our system reacts to perturbations [103]. The basic idea is to generate perturbations to the system under test with *reference waveforms*. The system will be subjected to some form of performance perturbation as dictated by the given waveform, and the reaction of the system will be monitored to understand how quickly the adaptation to the input occurs. Sample waveforms are shown in Figure 8.3.

We will use these waveforms to model perturbations in the system to a particular resource, whether it be CPU, disk, or the like, primarily focusing on disk performance faults. An increase in a waveform graph should be understood as an increase in fault utilization, and a decrease in the graph is a corresponding decrease in utilization. We mainly utilize the step-up and step-down waveforms, to judge the asymptotic behavior of the system under a change in resource availability.

8.4 Summary

In this chapter, we have described the simulation framework, the hardware environment for the prototype implementation, and the method that we will apply to induce performance faults into our experiments. We now proceed to the main technical portion of this dissertation, an experimental study via simulation and implementation of the distributed queue and graduated declustering.

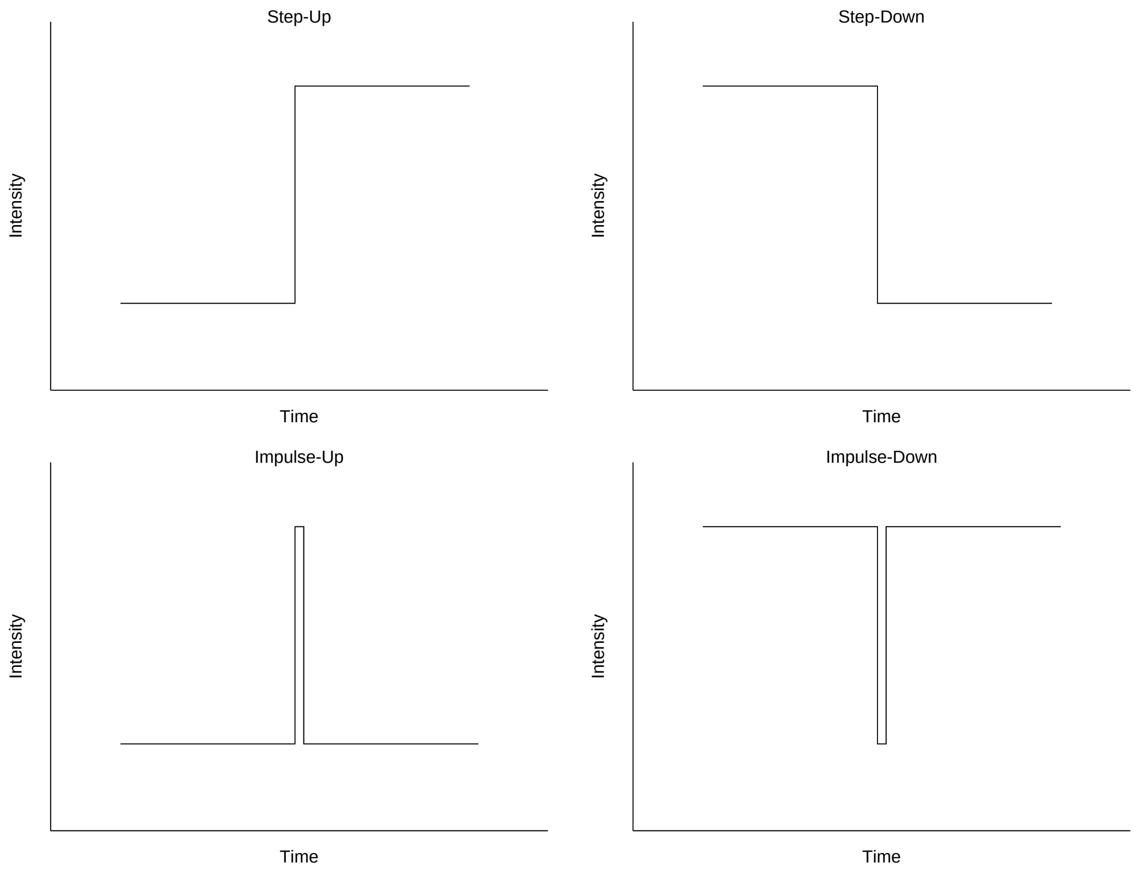


Figure 8.3: **Reference Waveforms.** *Four reference waveforms are shown. The first two, step-up and step-down, are used to gauge how quickly the system reacts when a particular resource in the system either becomes perturbed (step-up) or becomes unperturbed (step-down). The next two waveforms are impulse-up and impulse-down. By varying the width of the pulse, these waveforms can be used to determine how long a perturbation must be in place before the system takes note of it.*

Chapter 9

Experiments with the Distributed Queue

The next three chapters present experimental results of the two core distributed algorithms of River, the distributed queue and graduated declustering. In this chapter, we begin our exploration with micro-benchmarks of the distributed queue. Recall that distributed queues can be used to cope with consumer-side performance faults, by sending data from producers to faster consumers. First, we study the absolute performance of distributed queues under no perturbation, allowing us to focus on and understand basic algorithmic trade-offs. Then, we proceed with a series of experiments where performance faults are induced, which are designed to uncover performance characteristics of distributed queues when the system is under duress.

9.1 Absolute Performance

First, we examine the performance of the distributed queue algorithm in the ideal case, where all components are functioning at expected performance levels. Crucial to the acceptance of a mechanism such as the distributed queue is its performance in this scenario; if it is too heavyweight or does not scale when the system is behaving properly, it is unlikely users will go to the effort of using it to avoid performance faults.

The experiments within this section vary a large number of parameters, including the scale of the system, network latency, message size, and the rates of the producers and consumers. In all cases, our goal is to discover how sensitive the distributed queue algorithm is to these variables, and properly configure the number of outstanding message credits for maximal performance.

9.1.1 Scale

The most basic parameter that we vary is the size of the system, *i.e.*, the number of producers and consumers, commonly referred to as *s*cale. Ideally, the distributed queue algorithm will perform well in both small and large cluster environments.

Figure 9.1 shows the results from a set of simulations of the distributed queue, varying the scale of the cluster, the rate of the both producers and consumers, and the number of outstanding messages each producer is allowed. The scale of the cluster is plotted along the x-axis, and the percent of ideal performance is plotted along the y-axis. Ideal performance is the case where the consumers are kept busy 100% of the time when no consumers are perturbed; because bandwidth is our performance metric, we are not concerned with response time.

From the graphs, we can ascertain a number of basic properties of the distributed queue. The most prominent result is the number of outstanding messages that are needed. As one can see, with only 1 total outstanding message allowed per producer, performance is good with only 1 producer and 1 consumer in the system (near 100%), but then drops dramatically, reaching only 58% of ideal at 32 producers/32 consumers.

Increasing the number of outstanding messages allowed per producer substantially improves performance. As seen in the figure, just 3 or 4 messages per producer attains 90 to 95% of ideal, and roughly 10 messages outstanding per producer keeps consumers busy nearly 100% of the time, even at large cluster sizes.

A model of the performance of the distributed queue in this situation is difficult to develop, due to its stochastic nature, but can be approximated as follows. Given a certain number of outstanding messages, the question we are trying to answer is: how often will each consumer be busy on average? For the sake of simplicity, assume the network latency is zero, and that the processing time per message is constant.

We develop a model as follows: assume we have p balls, and n bins. In the first round, we will place all p balls into the n bins, uniformly at random. Then, in each subsequent round, we will remove a *single* ball from any bin that contains one or more balls. All of those balls will then be re-distributed uniformly at random over all n bins. Repeat *ad infinitum*. The obvious analogy to our simulation is that each bin is a consumer queue, and each ball a message. Taking a ball out of a bin is akin to a consumer processing a message.

We can solve the problem directly by enumerating the possible states of the bins, deriving the probabilities of transitions from $state_i$ to $state_j$ for all pairs of states i, j, and then solving for the resulting probabilities of residence in a given state. We can then deduce the average number of consumers that are left idle over time, which gives us the fraction of peak performance that we would achieve. However, as the number of states grows, solving the problem analytically becomes quite involved; thus, a simple simulation of balls and bins is a good solution in practice.

As seen in Figure 9.1, this model, plotted as a line in the figure, matches the data points from the simulation quite closely, for the case where each producer is allotted a single outstanding message. Thus, we have confidence that our simulation is behaving as expected. However, when the number of outstanding messages is increased, the model is no longer a perfect match. In this situation, our balls in bins example does not keep track of which producer sent which message, which is increasingly important. Solving the exact problem directly would thus become even more complicated, and therefore we do not pursue it further.

We also investigate the performance of the distributed queue in the prototype implementation. As seen in Figure 9.2, the performance of the distributed queue is quite good under scale, and with enough outstanding messages, extracts full or near-full bandwidth from the system. The lone exception is when consumer bandwidth is set 15 MB/s; in that case, performance drops at scale, which we discuss further below.

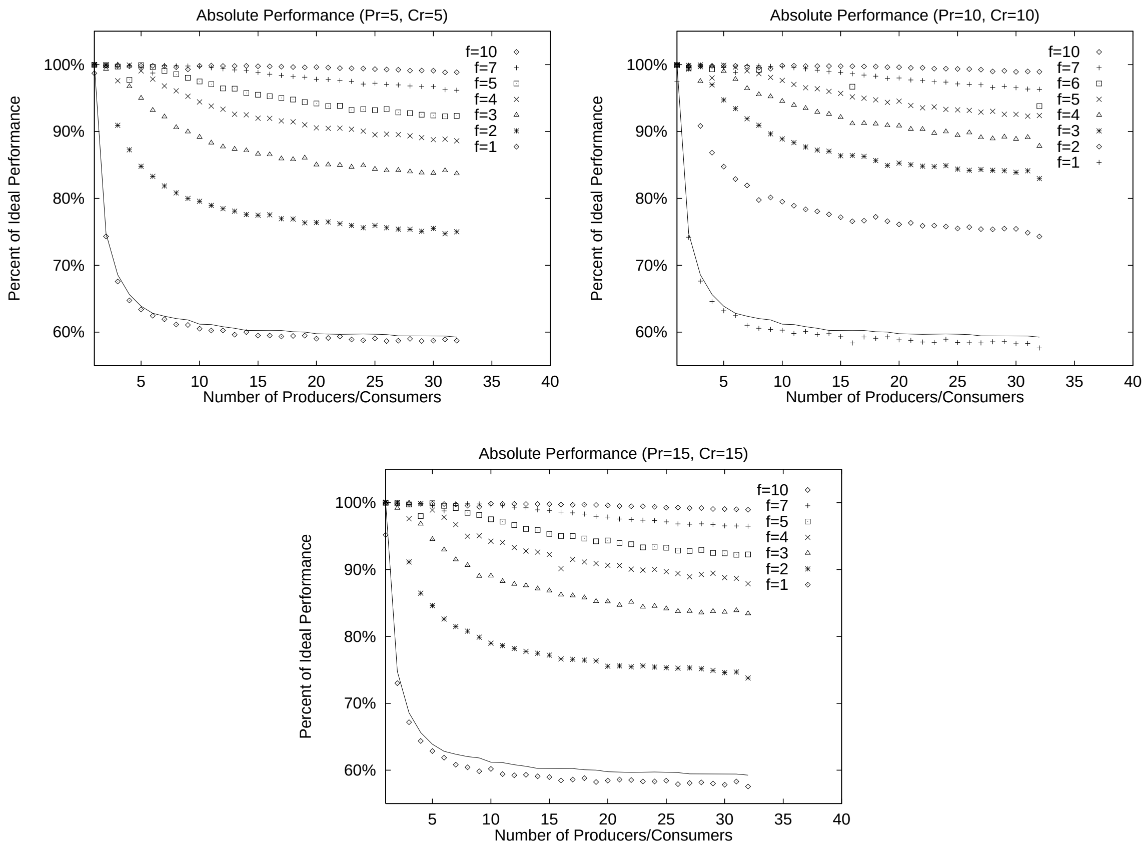


Figure 9.1: **DQ Absolute Performance: Scale (Simulation).** *Simulations of the performance of the distributed queue under scaling are shown. Each graph varies the rate of the producer and consumer, shown at the top of each graph, in MB/s. For example, in the upper-left graph, the producers' rate (Pr) is 5 MB/s, as is the each consumers' rate. The number of producers and consumers is varied together along the x-axis; therefore, at x=32, there are 32 producers and 32 consumers in the system, for a total of 64 entities. Each set of points in each graph represents the particular number of outstanding messages allowed per producer (f=x); multiplying this by the number of producers in the system gives the total number of potential outstanding messages in the system. As one call see, having roughly 10 messages outstanding per producer achieves full throughput even at large cluster sizes. A model, as developed in the text, is plotted as a line in the figures, and matches the data quite closely. For these experiments, all messages are 8KB blocks, and the network latency is fixed at 10 microseconds.*

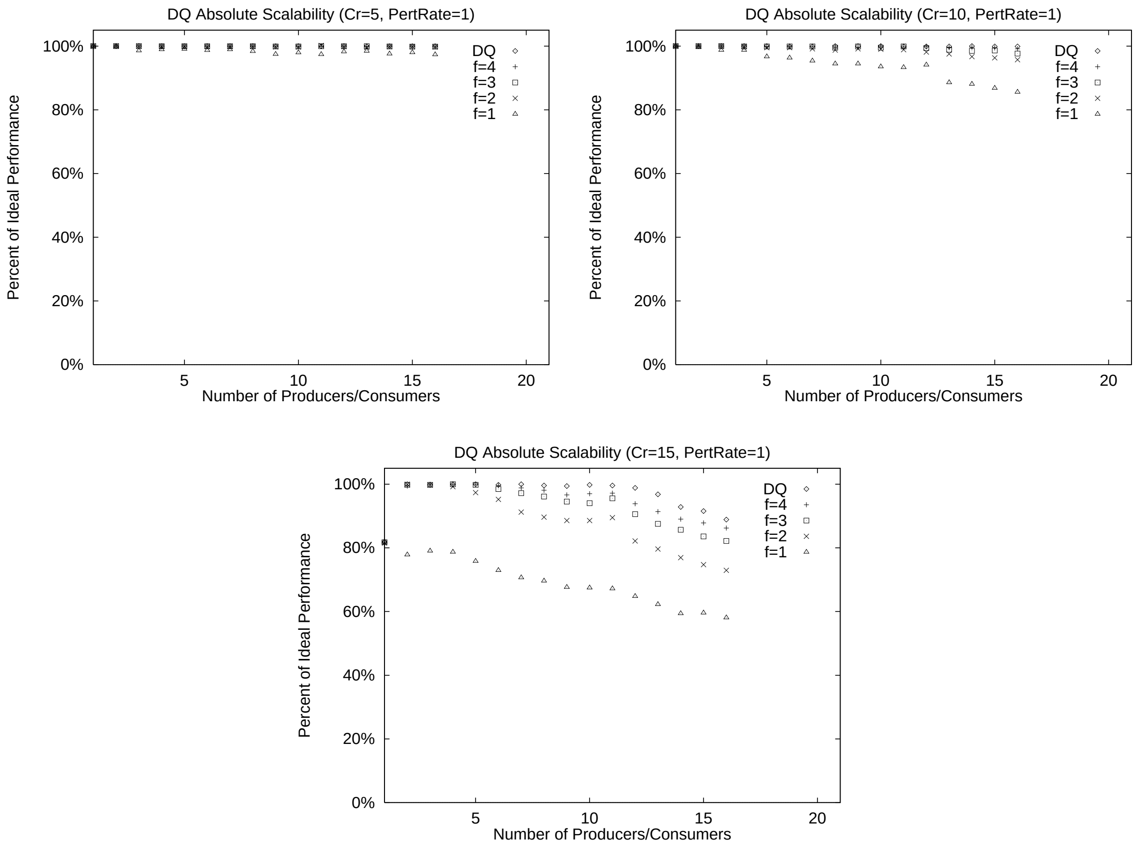


Figure 9.2: **DQ Absolute Performance: Scale (Prototype).** *Results from experiments with the prototype implementation of the the distributed queue under scaling are shown. Each graph varies the consumption rate of the consumers, from 5 MB/s up to 15 MB/s, while producer rates are unchecked. Because of experimental difficulties, the producer rate is not constrained as in the simulations. In each graph, the number of producers and consumers is co-varied along the x-axis; therefore, at x=16, there are 16 producers and 16 consumers in the system, for a total of 32 entities. Each set of points plots a particular number of outstanding messages allowed per producer (f=x); the data points demarcated with 'DQ' allots one credit per consumer (and thus matches the x-axis value). Performance, as a percentage of ideal, non-perturbed performance, is shown along the y-axis. Performance is as expected except when the consumer rate is set to 15 MB/s; at that point, network contention comes into play; see Figure 9.3 for details.*

One difference between the simulations and implementation is the time at which a message receipt is acknowledged by the consumer. In the idealized distributed queue algorithm, a message response, indicating the processing of a message is complete, is sent to the producer only after the message has been processed. However, this cannot be efficiently realized on top of Active Messages due to the request-response communication model therein, which dictates that a message reply must occur immediately after the message is polled from the network interface into memory. Therefore, the implementation sends a response *before* placing the message in the consumer queue for processing. The result is that in the implementation, a message acknowledgment is not as strong of an indicator of how the remote consumer is progressing; however, it is still a reasonably good steady-state indicator that the consumer is active and accepting messages. Thus, fewer outstanding messages are needed in order to keep performance high, because producers do not need to tolerate the latency imposed by consumer consumption.

One other point of interest occurs in the third graph of Figure 9.2, when consumer bandwidth is set to 15 MB/s, for all sets of data points. In that set of experiments, at 12 producers sending to 12 consumers and higher, performance suddenly dips below expectations; even with one outstanding message per consumer, performance is only 88% of ideal.

After some instrumentation, we find that the reason for the performance drop-off is unfairness in the network. Figure 9.3 plots the average bandwidth that each producer achieves while it is active, for consumer rates of 5, 10, and 15 MB/s. In the first two graphs, each producer gets a fair share of consumer bandwidth, as seen in the figure. However, for the case where the consumer rate is 15 MB/s, some producers receive a noticeably higher fraction of the bandwidth; producer 1 receives 17 MB/s, 2 MB/s higher than expected, while producer 16 receives about 13 MB/s, 2 MB/s less than expected. The performance drop-off is commensurate to this, as performance is dictated by the slow node.

What we realize from the figure is that the distributed queue algorithm presumes fairness in the network. If the network is not fair, some consumers will receive more packets from some of the producers, who will thus finish more quickly. Other producers, who receive less of the aggregate consumer bandwidth, will finish later than expected. The problem could be solved in software, by having each consumer monitor the rate of progress of each producer, and adjust to consumption accordingly, but, due to increased implementation complexity as well as a low likelihood of occurrence, we leave this enhancement to future work.

We also investigate the resource costs of the basic implementation, by determining the number of CPU cycles spent in transferring data from producers to consumers. Figure 9.4 plots the percent of the CPU utilized by both a producer and a consumer over the lifetime of a run.

From the graphs, we can make two basic observations. First, the consumer utilizes more of the CPU than the producer, roughly 55% to 45%. Second, we also notice that a good deal of time is spent in the kernel – 21.5% for the producer and 19.6% for the consumer. All of the time spent in the kernel is attributed to Active Messages, and therefore we do not explore it any further.

We can also calculate cycles/byte ratio from the figures. During the experiment, the distributed queue sustains a throughput of 26 MB/s. Therefore, the producer cost of the distributed queue is roughly 2.78 cycles/byte, and the consumer cost is 3.39 cycles/byte, all relative to the 167 MHz UltraSPARC-I processor.

We now break down the user-time in more detail. Figures 9.5 and 9.6 show the producer-side and consumer-side breakdown, respectively, of user-time spent in the distributed queue. To

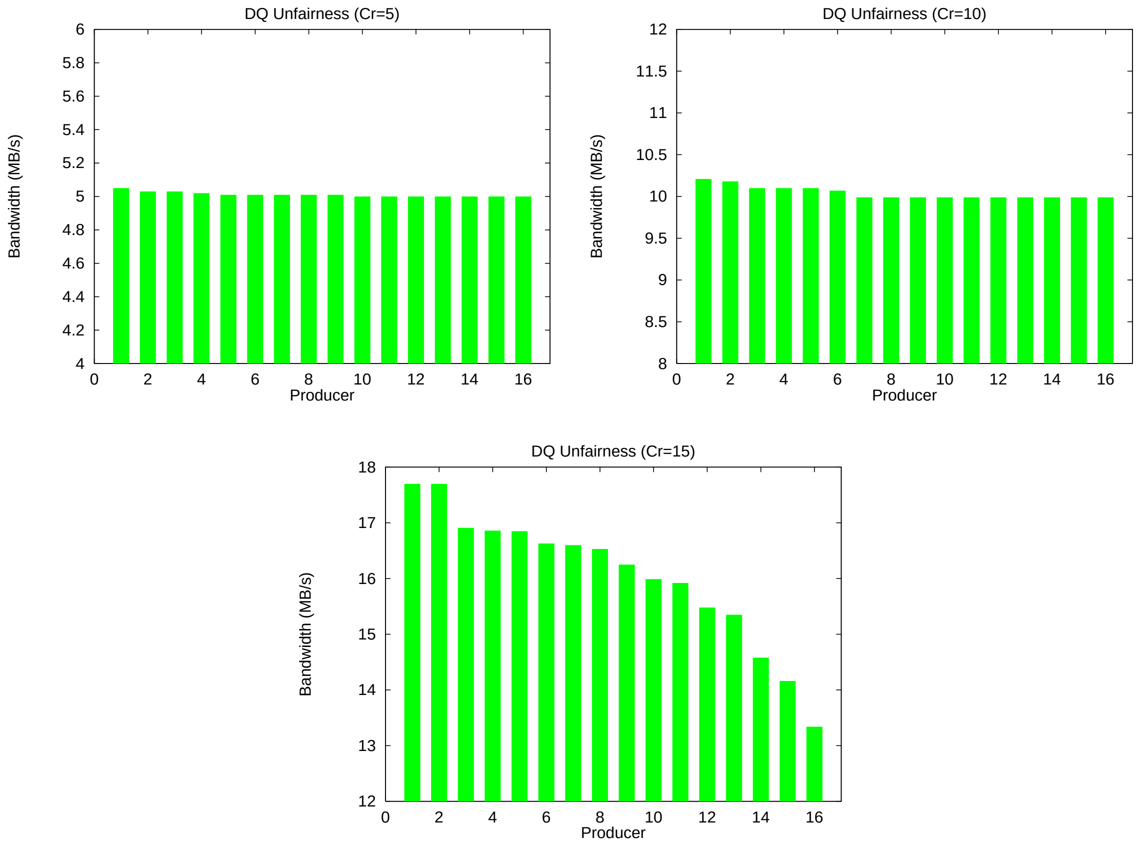


Figure 9.3: **DQ Producer Fairness (Prototype).** *The graphs plot the average producer bandwidth over the lifetime of an experiment. In the experiment, 16 producers send data to 16 consumers. The consumers receive data at 5 MB/s, 10 MB/s, and 15 MB/s, which is varied across the three graphs. In the first two graphs, each producer receives a fair share of the bandwidth, and thus they all are able to finish at the same time. However, when each producer sends data at 15 MB/s, contention in the network begins to appear, leading to an unfair biasing towards some of the producers.*

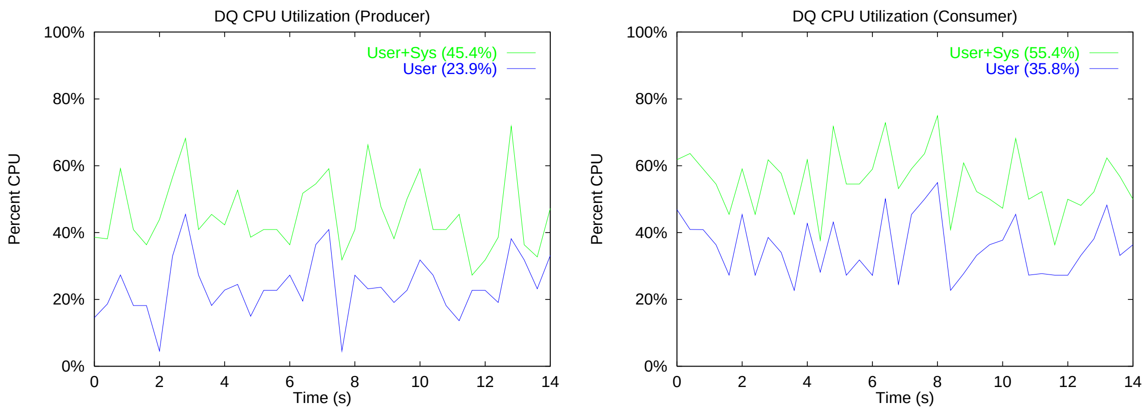


Figure 9.4: **DQ CPU Utilization (Prototype).** *The figure plots CPU utilization during a run of a micro-benchmark of the distributed queue, with producers and consumers unthrottled (running at full rate). On the left, the CPU utilization of the producer is shown, and the consumer utilization on the right. Both graphs plot two lines: the bottom line is user CPU time, and the upper line the sum of user and system time. Average utilization is shown in the key. Average throughput per producer/consumer for this experiment is 26 MB/s.*

```
Copy                                 7280 / 11310 (64.37%)
    memcpy                 -- 7280 / 11310 (64.37%)

AM Routines                          2763 / 11310 (24.43%)
    SetAndWait/Poll/Other -- 2442 / 11310 (21.59%)
    Transfer              --  321 / 11310 ( 2.84%)

DQ Library                           1190 / 11310 (10.52%)
    Modulo Arithmetic     --  282 / 11310 ( 2.49%)
    Random Number         --  154 / 11310 ( 1.36%)
    Translation           --  203 / 11310 ( 1.79%)
    Transfer              --  406 / 11310 ( 3.59%)
    Wait                  --   87 / 11310 ( 0.77%)
    Other                 --   58 / 11310 ( 0.51%)

Unaccounted                            77 / 11310 ( 0.68%)
```

Figure 9.5: **DQ Producer CPU Utilization (Prototype).** *The figure shows the user-time CPU breakdown of where time is spent on the producer-side of the distributed queue. Memory copy is dominant, accounting for 64% of user-CPU time, and raw AM costs another 24%. Inside the distributed queue library, the random number generation, and the modulo arithmetic required to reduce it to the proper set, account for a total of 3.85% of user-time. Data is gathered via PC sampling.*

```
Copy                                 7995 / 13150 (60.80%)
    memcpy               -- 7995 / 13150 (64.37%)

AM Routines                          2220 / 13150 (16.89%)
    SetAndWait/Poll/Other -- 1972 / 13150 (15.00%)
    Transfer             --  248 / 13150 ( 1.89%)

Memory Management                    1457 / 13150 (11.08%)
    Malloc/New           --  644 / 13150 ( 4.90%)
    Free/Delete          --  813 / 13150 ( 6.18%)

Locking                               456 / 13150 ( 3.47%)

DQ Library                            888 / 13150 ( 6.75%)
    Transfer             --  493 / 13150 ( 3.75%)
    Translation          --   21 / 13150 ( 0.16%)
    Other                --  374 / 13150 ( 2.84%)

Unaccounted                           134 / 13150 ( 1.02%)
```

Figure 9.6: **DQ Consumer CPU Utilization (Prototype).** *The figure shows the user-time CPU breakdown of where time is spent on the consumer-side of the distributed queue. Memory copy is again dominant, at almost 61% of user-time, and raw AM calls account for only almost 17%. Memory management, which involves allocating (and eventually deallocating) a buffer per message receipt, is noticeable at 11%. The actual time spent in the receive side of the distributed queue library is small, only 6.75%. Locking, which only occurs inside the C and AM library (not in the distributed queue), accounts for 3.5%. Data is gathered via PC sampling.*

collect the data, we utilize PC sampling, a well-known technique to gather information about where an application is spending its time [58].

Roughly two-thirds of user-time is spent in the copy routine, internal to the Active Messages library, which copies data on outgoing sends. Raw Active Messages calls account for most of the rest of the time spent. Little time is spent in the distributed queue library, which picks a consumer to send data to and does so immediately, though the time spent in random number generation and subsequent modulo are noticeable.

Thus, the performance of our prototype implementation is quite satisfactory, scaling well to 32 total machines, and not using an excessive amount of CPU cycles, roughly 3 cycles/byte.

9.1.2 Latency

We now vary the latency of the network, in order to understand its effects on the distributed queue. In the simulations, latency is modeled as the time for the data to travel from producer to the consumer queue, and matches the definition found in the LogP model [38]. Outside of the explorations in this set of experiments, we assume a latency of 10 microseconds, in accordance with networks of modern networks [39].

Figure 9.7 plots the performance of the distributed queue as a fraction of ideal performance versus the latency of the network, for three different cluster sizes of 4, 16, and 32 total nodes; each set of points refers to a different number of outstanding messages per producer. Each line in the figure is a plot of a model of expected performance, which is developed below. As one can see, the model predicts performance as a function of latency quite accurately.

The results reveal that, not surprisingly, higher latencies demand an increase in the number of outstanding messages. By increasing the amount of flow control, each producer is able to hide the latency of the network, even in networks of unrealistically high delays of 10s of milliseconds.

We now develop a model to explain the behavior of the distributed queue under increased latency. By assuming that the number of outstanding messages is "high enough", we avoid the stochastic difficulties of the model developed in the previous section, and can generate useful models of system behavior.

For the following development, we assume that the consumer is the bottleneck in the transfer, and that each producer sends data infinitely fast [1]. Each of P producers is allowed n total outstanding messages.

We assume the vantage point of a single consumer. The consumer processes each message it receives in time T_B. Because each of P producers is allowed n outstanding messages per consumer, at startup, the consumer will have $P \cdot n$ messages to process in its queue. The time it takes the consumer to process those messages is:

$$T_{process} = P \cdot n \cdot T_B. \tag{9.1.1}$$

For the consumer to remain busy 100% of the time, it must receive the next message to process from any producer *before* it is finished with this set of messages, and continue to receive messages at that rate. The next message will be sent from the producer whose message is processed first, and when the consumer finishes processing that first message, it sends an acknowledgment to the producer, which causes the producer to send another message to the consumer.

[1] We can remove this assumption without loss of generality; however, it simplifies the discussion.

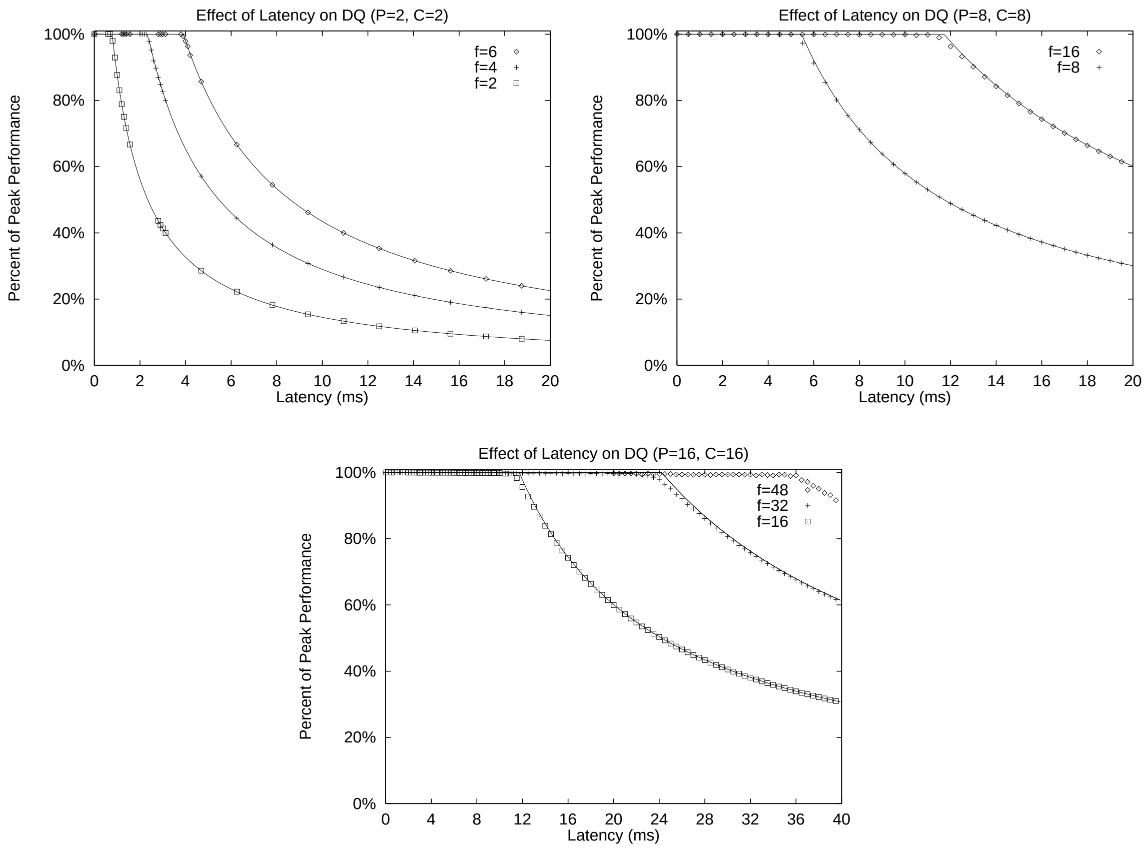


Figure 9.7: **DQ Latency Effects (Simulation).** *The graphs plot the performance of the distributed queue as a function of increasing latency in the cluster; note that latency is in milliseconds, not microseconds. Each graph represents a different cluster scale, as noted in the titles (P=number of producers in simulation, C=number of consumers). The different data points represent different numbers of flow control credits given to each producer (f=x); divide that number by the number of consumers to get the number of credits per consumer allowed each producer. The lines plotted are generated from models, as developed in the text, and match the simulations quite closely. In general, higher latencies can be overcome by increasing the number of credits available. For most reasonable latencies, producers do not need more than one outstanding message per consumer.*

The time to receive that next message can thus be derived. It is the sum of the time to process the first block, plus the latency of the network to send the acknowledgment from the consumer to the producer, plus the latency to send a message to the consumer:

$$T_{next} = T_B + 2 \cdot T_L, \tag{9.1.2}$$

where T_L is the latency of the network.

Thus, for the consumer to remain 100% busy, T_{next} must be less than $T_{process}$. If this is the case, the consumer will remain busy in steady-state. Substituting in Equations 9.1.1 and 9.1.2, and solving for n, the number of messages outstanding per producer, we arrive at:

$$n \geq \frac{2 \cdot T_L + T_B}{P \cdot T_B}. \tag{9.1.3}$$

We can also rearrange this to derive the "break-even point"; the highest value of latency that will achieve 100% of ideal performance:

$$T_L = \frac{(P \cdot n - 1) \cdot T_B}{2}. \tag{9.1.4}$$

Finally, in the regime where performance is less than 100%, the percent of time the consumer will be busy, known as its *duty cycle*, is given by the following, which corresponds directly to the percent of ideal performance achieved:

$$Percent_{Busy} = \frac{P \cdot n \cdot T_B}{2 \cdot T_L + T_B}. \tag{9.1.5}$$

Thus, the model of performance as a function of latency, T_L, can be written closed form as follows:

$$Performance(T_L) = \begin{cases} 100\% & T_L \leq \frac{(P \cdot n - 1) \cdot T_B}{2} \\ \frac{P \cdot n \cdot T_B}{2 \cdot T_L + T_B} & T_L > \frac{(P \cdot n - 1) \cdot T_B}{2} \end{cases} \tag{9.1.6}$$

The models are plotted in Figure 9.7 as lines, and match the data obtained from the simulations quite closely.

We end our discussion of latency with a few generalizations. First, for most reasonable cluster sizes of 10 or more machines, and large messages of 1 KB or more, latency in modern networks is insignificant. That said, in environments with larger latencies, the number of outstanding messages allotted per producer by the distributed queue per producer must be tuned properly, otherwise performance will suffer proportionally. If the latency has a fixed cost, this can be perhaps be configured once, before the system comes on-line. In more dynamic environments, an adaptive algorithm, similar to the dynamic TCP congestion control algorithm, could be employed [66].

9.1.3 Message Size

We next investigate the results of varying the size of each message, which in turn alters the message-processing cost, T_B. T_B is calculated directly by dividing the size of the message by the rate of the consumer. Thus, an increase or decrease in message size will affect a proportionate change

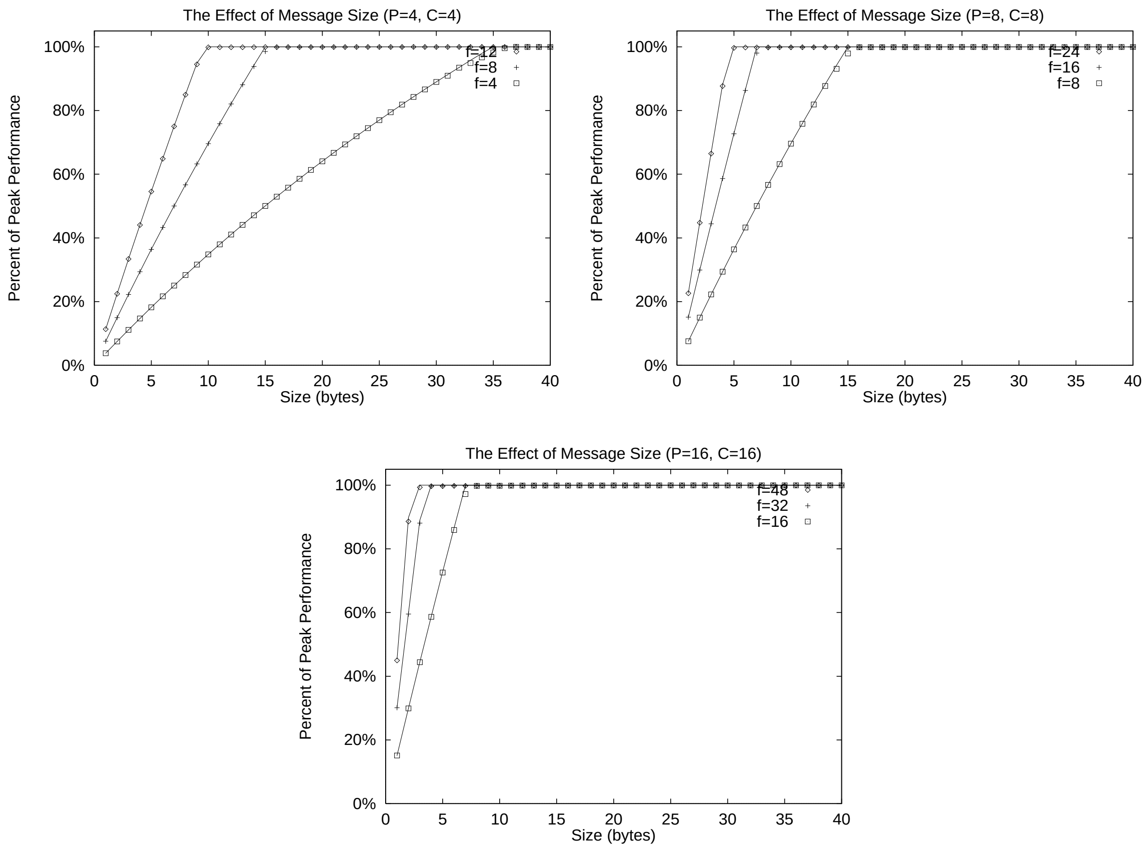


Figure 9.8: **DQ Message Size Effects (Simulation).** *The size of messages (and thus the processing time per message) is varied. The percent of peak (ideal) performance is shown on the y-axis. Each of three graphs varies the number of producers and consumers in the cluster; in the title, (P=x, C=x) implies that there are x producers and x consumers in the simulation. Each set of points in a particular graph corresponds to the total amount of flow control allowed per producer (f=x); divide x by the number of consumers to get the per-consumer flow control amount. For these experiments, producer and consumer production rates are fixed at 5 MB/s, and the network latency is 10 microseconds. In general, only for very small message sizes is more than one outstanding message per consumer required.*

in processing time. A slightly more sophisticated model, that included a fixed overhead per message, could similarly be developed.

Figure 9.8 plots the results of simulations that vary the message size, the scale of the cluster, and the number of outstanding messages allowed per producer. The question we are attempting to answer is: how many outstanding messages are necessary in order to keep the remote consumers busy? In general, with smaller cluster sizes, or smaller messages, more outstanding messages will be required, as is seen in the graphs. The reason small cluster sizes need to allow more outstanding messages per producer is straight-forward: with few producers in the system, it is quite difficult to keep the consumers busy, as the stochastic effects of the randomized distributed queue algorithm take effect. However, for reasonable-sized clusters of 8 producers/consumers or more, or reasonable-sized messages of 32 bytes or larger, only 1 outstanding message per consumer is required to attain peak bandwidth.

The models shown in the graphs are the same as developed in the section on latency, rearranged in order to plot performance as a function of message size. Once again, the models match the simulated results quite closely.

Thus, the simulations tell us that the distributed queue implementation is free to use message sizes of almost any reasonable size. Currently, the choice is 8 KB; thus, if data is placed into the queue that is smaller than that size, it will be buffered, and when enough data is gathered, it will be sent. Though smaller sizes would be effective, because the latency of remote processing and network transit time can be hidden with just a few outstanding messages, they exact a different cost: processing overhead, which is not modeled in the simulations. Thus, larger messages should be utilized where possible.

9.2 Performance Under Perturbation

Having understood the basic performance properties of the distributed queue, we now turn to a study of its behavior under perturbation. We induce controlled performance faults into the system by altering the consumption rates of consumers, and monitor how the distributed queue algorithm is able to cope with them.

9.2.1 Perturbation Spectrum

We begin our experiments by examining overall performance under a broad spectrum of performance variations. We first examine simulation results, shown in Figures 9.9 and 9.10. In general, within the experiments, we increase the number of consumers that are undergoing performance faults along the x-axis, and monitor overall system performance in response.

In Figure 9.9, the rate of consumption is varied across each graph, from 5 MB/s up to 40 MB/s, and the the number of perturbed consumers increases along the x-axis, from zero to the full cluster size. All other parameters are held constant. The data is plotted as a set of points, and the model of ideal system performance, as developed in Chapter 2, is plotted as a line. In all cases, the performance of the distributed queue is quite good, closely matching ideal. In general, as the rate of consumption per consumer is increased, while holding the rate of production per producer constant, the amount of slack in the system increases. Accordingly, the number of perturbations to consumers that can be tolerated increases.

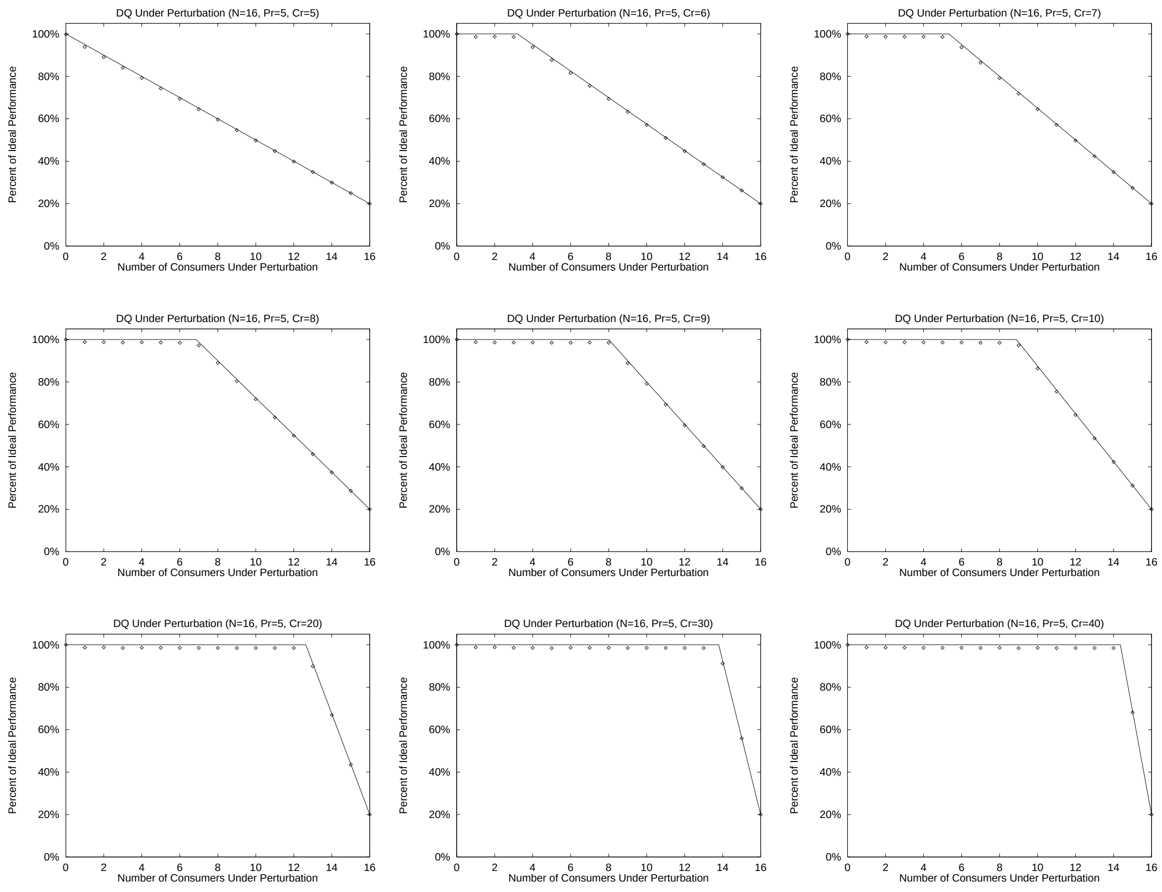


Figure 9.9: **DQ Perturbation Spectrum: Consumption Rate (Simulation).** *The simulated performance of the distributed queue is shown under a range of perturbation experiments. In each graph, the scale is set to 16 producers and 16 consumers, the latency is 10 microseconds, the rate of the perturbed consumer is 1 MB/s, and each producer is allowed 16 outstanding messages. Across graphs, the rate of each producer is 5 MB/s, and the consumer's rate is varied, from 5 MB/s to 40 MB/s. The data points plot the percent of peak performance achieved by the DQ algorithm under a range of perturbations, with the number of consumers perturbed increasing along the x-axis, and the lines plot the model of ideal performance developed in earlier chapters. In general, performance is quite good, always quite close to ideal. As the rate of the consumer increases (from left to right, and top to bottom, in the graphs), the number of perturbations that can be tolerated without any performance loss increases, as there is more slack available in the system.*

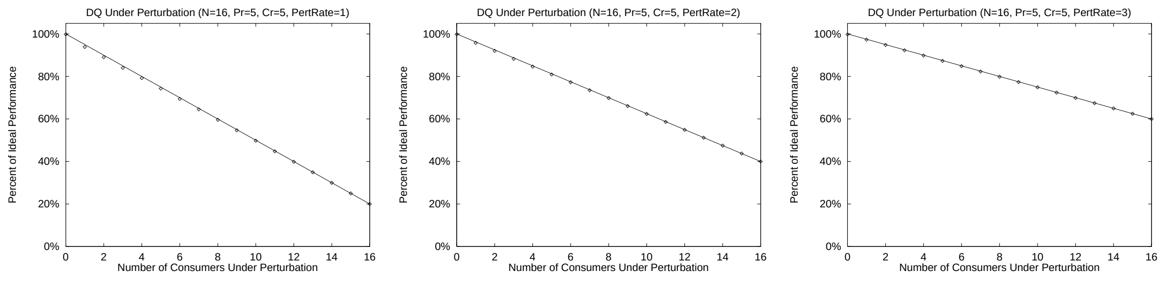


Figure 9.10: **DQ Perturbation Spectrum: Perturbation Rate (Simulation).** *In this set of graphs, simulated performance of the distributed queue is plotted, this time varying the rate of the perturbed consumer. Again, scale is set to 16 producers and 16 consumers, latency to 10 microseconds, flow control to 16, and the rate of producers and consumers is fixed at 5 MB/s. In the graphs, the perturbations reduce consumption to 1 MB/s (PertRate = 1 MB/s), 2 MB/s, and 3 MB/s. The milder the perturbation, the less impact it has on overall performance.*

Figure 9.10 varies the rate of the perturbed consumer, while holding all else constant. With milder perturbations, system performance does not drop as severely.

Results from the prototype implementation are shown in Figure 9.11. As one can see from the graphs, performance is excellent across the range explored, and scales well even as the number of nodes is increased in the system. Though we do not cover the same range of performance perturbations as in the simulations, all results that we have gathered to date perform excellently as expected.

One crucial aspect to the implementation of the distributed queue is the behavior of the underling message layer. In particular, the message layer must not restrict the number of outstanding messages to less than the distributed queue desires; if the layer does so, the performance of the distributed queue under perturbation will not be as expected. Most message layers, including AM, restrict the number of outstanding messages to an arbitrary number, chosen by the message-layer developers. We now explore what happens when the number of outstanding messages is restricted.

9.2.2 Flow Control

In earlier sections, we have demonstrated that the number of outstanding messages is an important parameter, and must be adjusted properly in order to keep messages flowing efficiently when there are no perturbations in the system. We now explore how perturbations impact flow control.

Figure 9.12 plots the results of simulations that vary the amount of available flow control, for three different rates of consumption. Each set of points represents performance with a given amount of flow control.

The simulations reveal the importance of flow control to the algorithm in the face of perturbation. In cases where the number of flow control credits is less than or equal to the number of perturbed consumers, performance suffers. In other words, if a producer does not have enough message credits to keep at least one outstanding to each of the slow producers plus some at the non-perturbed others, performance will suffer, as the algorithm will not be able to “remember” which

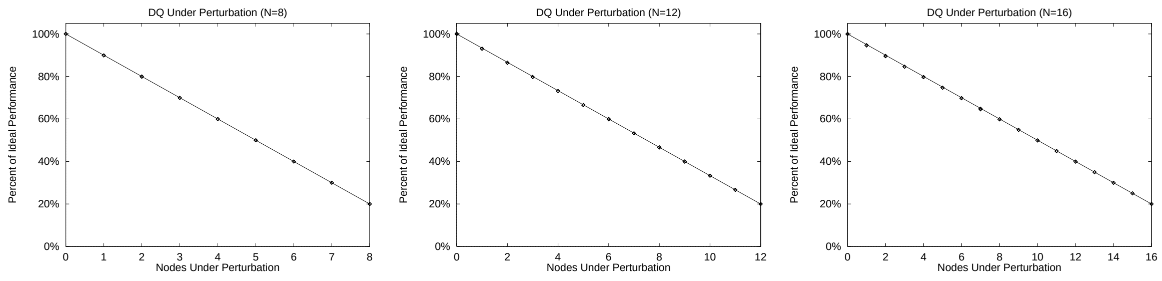


Figure 9.11: **DQ Perturbation Spectrum: Scale (Prototype).** *The performance of the distributed queue prototype implementation is shown under a range of perturbation experiments. As with the simulations, the number of perturbed consumers is increased along the x-axis, and the percent of ideal performance is shown on the y-axis. Each graph changes the scale of the system under test, from 8, to 12, and then 16; a scale of size n implies that there are n producers and n consumers on a total of 2n machines. For this set of experiments, consumption rates are set to 5 MB/s, and a perturbation reduces the rate of consumption to 1 MB/s (PertRate = 1 MB/s). From the graphs, one can see that performance across all three system sizes is quite good. Note that the right-most graph in this figure matches the simulation results in the left-most graph of Figure 9.10.*

nodes were the slow nodes.

From these simulations, we can conclude that the distributed queue algorithm should always have at least one message outstanding message per consumer; this delivers to each producer a perspective on the remote performance of all consumers in a straight-forward, simple, and effective manner. Without at least one outstanding message per consumer, there is a possibility that some number of faults could combine to deliver less than ideal performance. Thus, the underlying message layer must not restrict the number of outstanding messages to less than the number of consumers in the system. The current Active Messages implementation has a hard limit of 30 total outstanding messages[2]; thus, if there are more than 30 consumers in our system, we will not be able to successfully tolerate more than 30 performance faults to consumers.

9.2.3 Total Work

One essential ingredient to tolerating performance faults is the presence of excess parallelism. For example, if there were only n items of work to perform, and n consumers, a perfect distribution through the distributed queue would lead to 1 piece of work per consumer, and performance would be dictated by the rate of the slowest consumer. Even in more realistic settings, excessively perturbed nodes could lead to end-of-run effects that surprisingly become first-order performance factors. Therefore, we next explore the amount of parallelism necessary to tolerate performance faults.

Figure 9.13 plots the simulated performance of the distributed queue under a single performance fault, as a function of the total amount of data pushed through the queue. The data from

[2]This number arises from a hardware limitation on the Myrinet network-interface card in tandem with design decisions in AM.

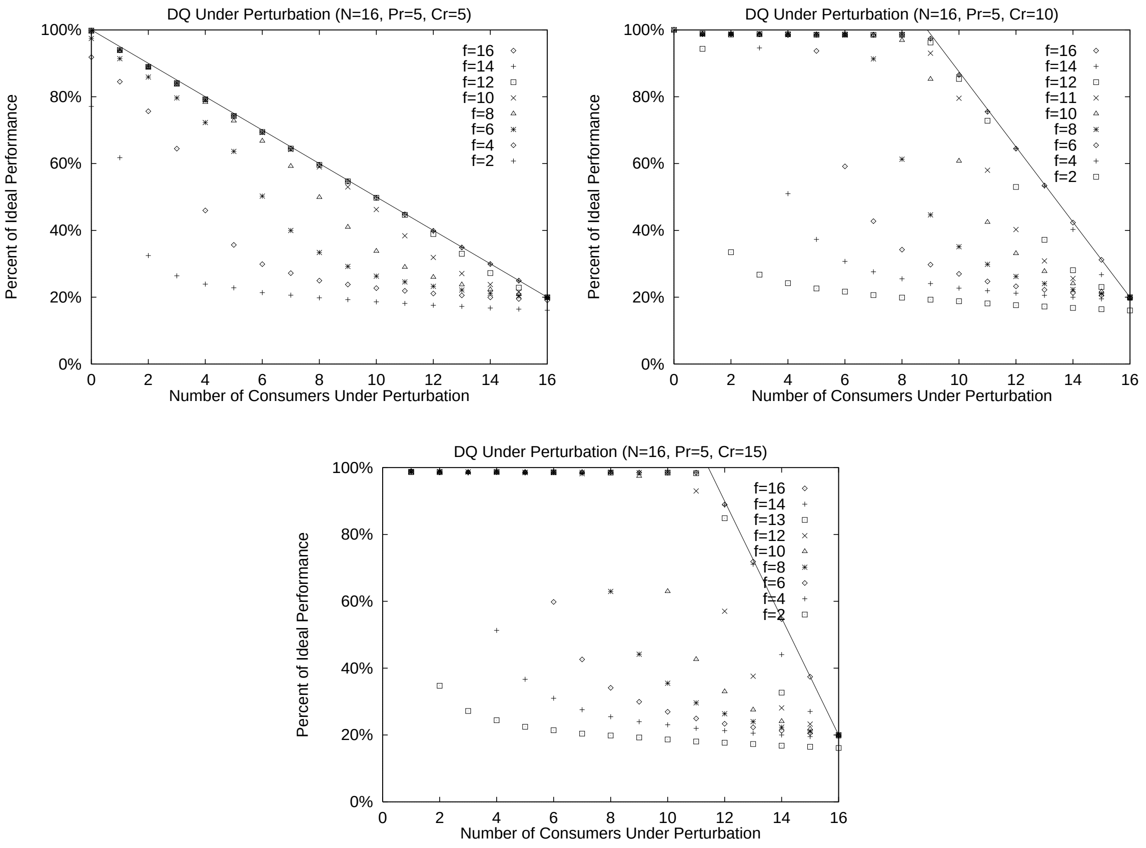


Figure 9.12: **DQ Importance of Per-Consumer Flow Control (Simulation).** *The amount of flow control is varied under a set of perturbation simulations. Each graph plots the performance under perturbation for a range of flow control credits (f=x, the total number of credits given to each producer) for a particular rate of consumption, in this case, rates of 5, 10, and 15 MB/s. Without a credit per consumer, it is not possible to tolerate the full range of perturbations. For these experiments, there are 16 producers and 16 consumers, the rate of production is 5 MB/s, network latency in 10 microseconds, and block size is set to 8 KB.*

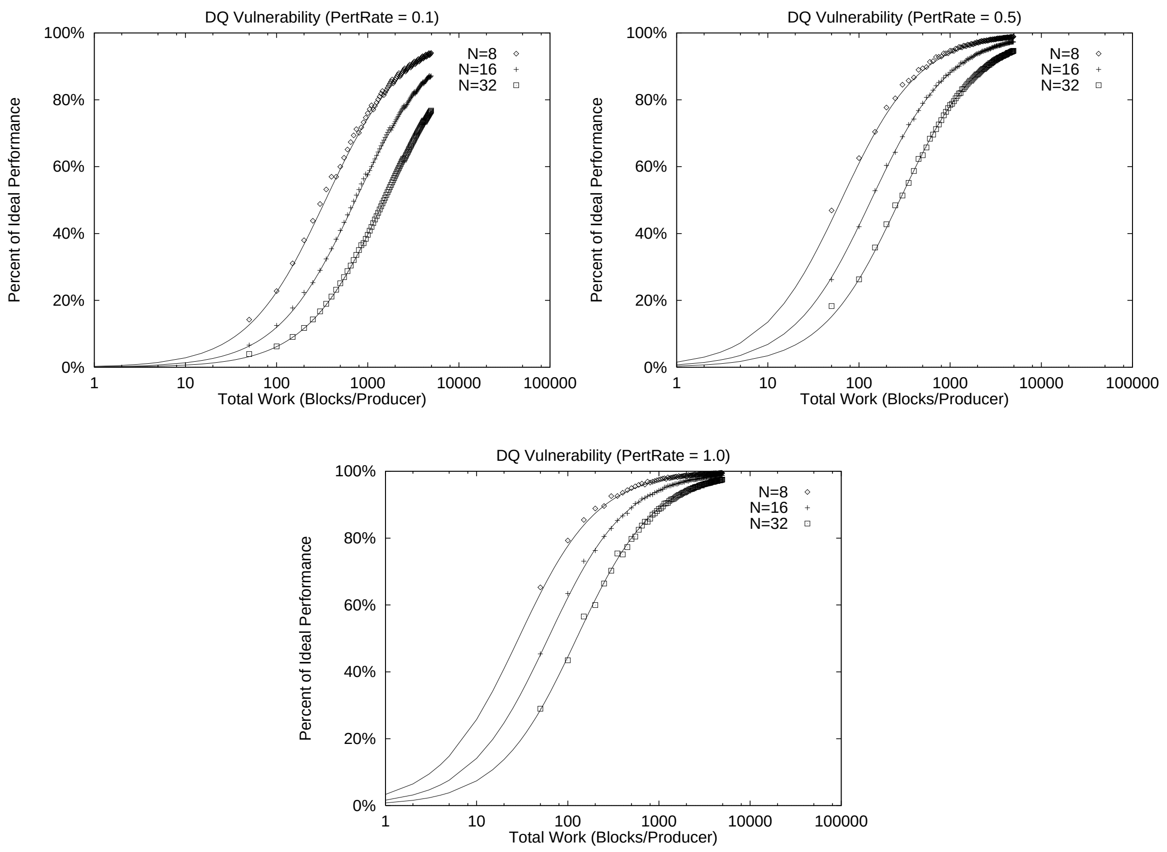


Figure 9.13: **DQ Needs Excess Parallelism (Simulation).** *The figures plot performance under a single perturbation as a function of the total amount of work that is pushed through the distributed queue. In a given figure, each set of points represents a particular size of the simulated cluster (N), and each line is based on the model as developed in the text. The consumption rate of the perturbed consumer is varied across the three graphs, from quite severe (0.1 MB/s, or 50x slower than an unperturbed consumer) to less so (1.0 MB/s). In all cases, if only a small number of blocks pass through the distributed queue, the system will be vulnerable to performance faults. For these experiments, there are 16 producers and 16 consumers, the rate of production is 5 MB/s, network latency in 10 microseconds, and block size is set to 8 KB.*

the simulations are plotted as points, and the lines represent models of expected performance, as developed below.

As we can see from the graphs, if only a small amount of work is placed into the distributed queue, it does not perform well, as the time for the perturbed consumer to process its remaining blocks after all others have finished takes a significant proportion of overall run-time. Because each producer is willing to send one message to each consumer, even the slowest consumer will receive p messages, where p is the number of producers in the system.

Thus, the expected time for a run, as determined by the slowest node in the system, can be determined as follows. The following development is for the case where only a single consumer in the system is perturbed, but could easily be generalized. The time for the entire run is as follows:

$$T_{expected} = \frac{W}{R_{total}}, \tag{9.2.1}$$

where W is the total number of bytes moving through the system, and R_{total} is the total rate, across all consumers, at which data are processed. Refining this a little further, we arrive at:

$$W = N_b \cdot S_b \tag{9.2.2}$$

$$R_{total} = \frac{(c-1) \cdot R_c + (1) \cdot R_{pert}}{c} \tag{9.2.3}$$

where N_b is the number of blocks that the system will process, S_b is the size of each block, R_c is the rate of each non-perturbed consumer, and R_{pert} is the rate of the perturbed consumer. There are c consumers in the system. Thus, in the best case, we could hope for all blocks to be processed at the average rate of all of the consumers.

Unfortunately, because of end effects, that is not the case. Because the perturbed consumer is so much slower than normal consumers, its processing time at the end of the run can noticeably affect overall performance. The extra time can be modeled as follows:

$$T_{extra} = T_{pert} - T_{norm}, \tag{9.2.4}$$

where

$$T_{pert} = \frac{p \cdot S_b}{R_{pert}} \tag{9.2.5}$$

and

$$T_{norm} = \frac{p \cdot S_b}{R_c}. \tag{9.2.6}$$

At the end of the run, each consumer will have p outstanding messages to process, one from each producer, because there are p producers in the system, and each allows only one outstanding message to each consumer. The normal unperturbed consumers will process them in T_{norm}, and the perturbed consumer in T_{pert}. The difference between the two (T_{extra}) is what can cause a serious performance problem.

Thus, the actual time for a run will be:

$$T_{overall} = T_{expected} + T_{extra}. \tag{9.2.7}$$

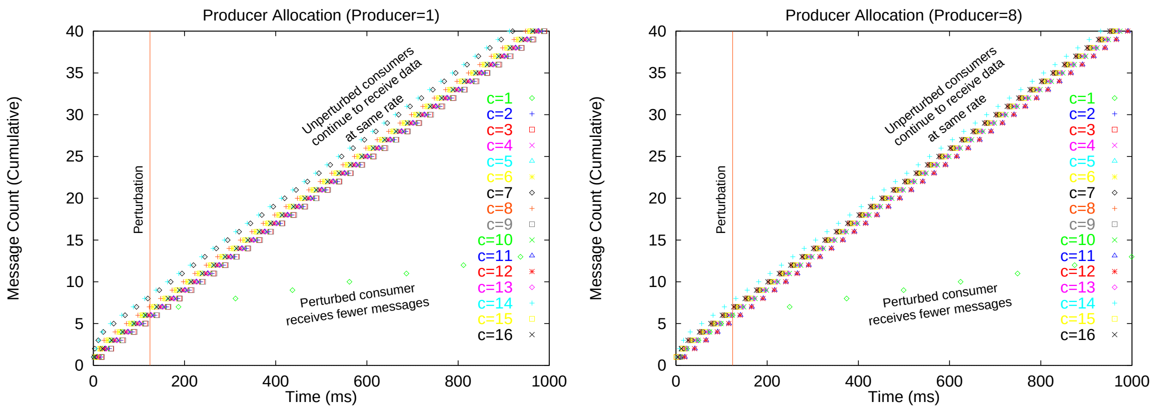


Figure 9.14: **DQ Producer Allocations Under Step-Up (Simulation).** *Simulated message counts from each producer to each consumer is plotted in the graphs. Each graph plots the cumulative number of messages sent from a particular producer to each of its two consumers over the life of the experiment. From the figure, one can see that immediately after the perturbation (the vertical line in the graph), consumer 1 (c=1) begins to receive far fewer messages over time from both producers 1 and 8 (and in fact, all producers). From the time of the perturbation (roughly 100 ms) to the end of the experiment (1000 ms), the perturbed consumer only receives 7 messages from each producer, wherease other consumers receive roughly 35. Note that the factor of five difference is directly proportional to the bandwidths of the consumers. In the experiment, there are 16 producers and 16 consumers, each who normally operate at 5 MB/s. The perturbed consumer drops to 1 MB/s. Message blocks are 8 KB, and the network latency is set to 10 microseconds.*

The ratio $\frac{T_{overall}}{T_{expected}}$ is plotted in the graphs in Figure 9.13 as the total number of block per producer is increased.

Summarizing, end effects can be noticeable, especially when there is not much work overall in the system; the problem worsens with scale, because each of p producers gives at least a single piece of work to each consumer. If the total amount of work is small, $T_{expected}$ will be small, and thus the extra time T_{extra} will become quite noticeable.

9.2.4 Time to Convergence

We close our experimentation with the distributed queue by examining the effect of a single perturbation over time, with the goal of understanding how long it takes the system to react to a perturbation. Figure 9.14 plots the number of messages sent to all consumers from the vantage point of 2 of the 16 producers, plotted over time[3].

The perturbation is a 'step-up', which reduces the bandwidth of the perturbed consumer from 5 MB/s to 1 MB/s. The vertical line in each of the graphs indicates the exact point at which the single consumer (consumer '1') was perturbed. Each graph plots the cumulative message count from

[3]We only show 2 of the 16 because behavior across producers is nearly identical, and showing all 16 makes the graphs difficult to read.

a single producer to each of the consumers. In the graphs, one can detect the immediate response of each producer to the perturbed consumer. While messages continue to be sent at the full rate to the non-perturbed consumers, as shown by the broad set of points in a straight line up and to the right, the perturbed consumer suddenly receives much less of the total message allocation. Note that the bandwidth received by each consumer can be calculated from the slope of each line.

The reason that we achieve this desirable behavior is the design of the distributed queue algorithm, which keeps exactly one outstanding message to each consumer. Perturbations are implicitly detected quite rapidly, as the perturbed node stops replying to messages at the same rate of the other nodes, which in turn does not generate new message sends from the producer at the same rate.

Chapter 10

Experiments with Graduated Declustering

This chapter presents an experimental study of graduated declustering. Recall that graduated declustering is designed to cope with producer-side performance faults via replication; by adjusting the bandwidth allocations from replicated producers to consumers, each consumer should receive its fair share of aggregate producer bandwidth. Similar to our approach in the previous chapter on the distributed queue, we begin our examination by studying the performance characteristics of the algorithm in non-perturbed scenarios. We then investigate behavior of the graduated declustering algorithm under a host of perturbation scenarios.

10.1 Absolute Performance

We begin our study of the performance properties of graduated declustering by first examining it under tightly controlled, non-perturbed scenarios. Our performance goals are straightforward: lightweight mechanisms with high performance when no performance faults are present in the system.

As was the case with the distributed queue, we seek to configure the graduated declustering algorithm properly for maximal performance. However, we do not explore the latency parameter, because we have learned in the previous chapter that it is not relevant to system performance.

10.1.1 Scale

The first parameter that we vary is the scale of the system. Figure 10.1 plots the simulated performance of graduated declustering under increasing scale, while varying the rates of producers and consumers, as well as the number of outstanding messages allowed.

As one can see from the figure, the number of message credits needed does not scale with the size of the system, as was the case with the distributed queue, because each consumer only requests data from two sources, the original copy of the data and its replica. However, the number

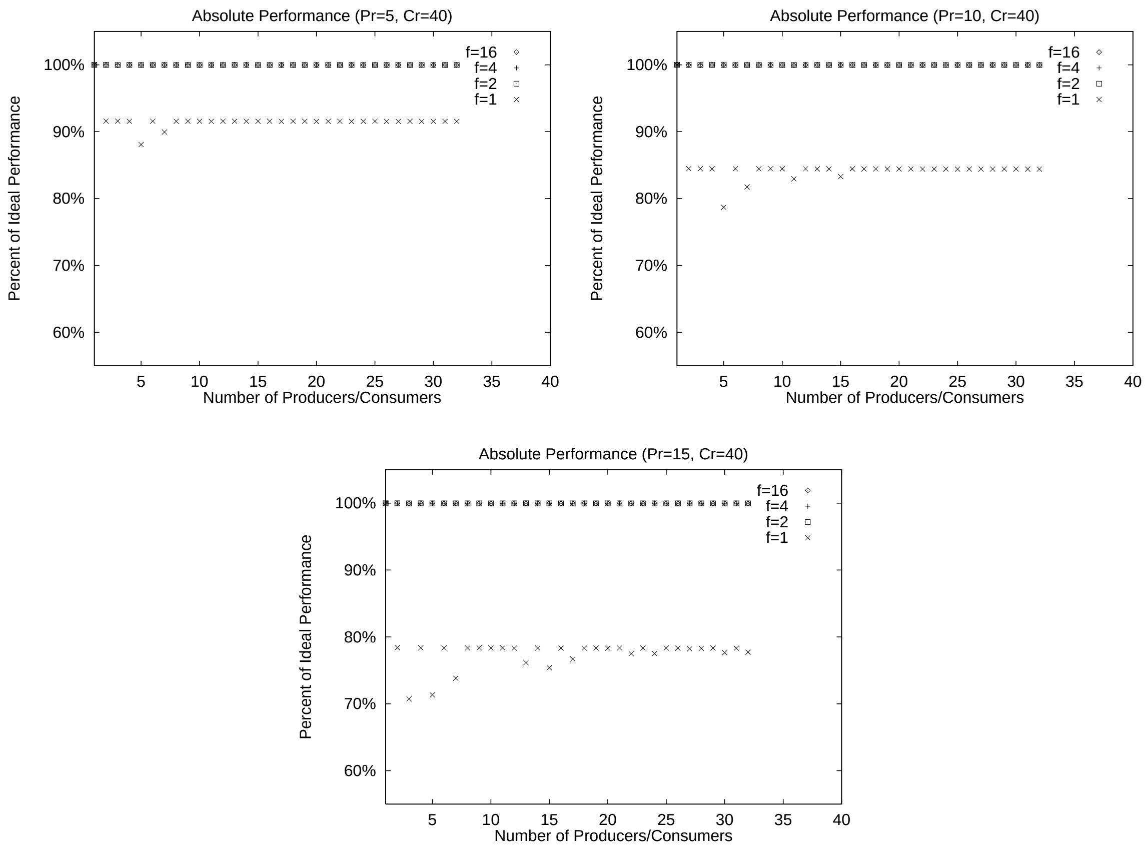


Figure 10.1: **GD Absolute Performance: Scale (Simulation).** *The simulated performance of graduated declustering is shown in the graphs. The number of producers/consumers is co-varied along the x-axis; thus, at x=32, there are 32 producers and 32 consumers in the experiment. The y-axis plots percent of ideal performance under increasing scale. Each graph alters the rate of production, from 5 MB/s up to 15 MB/s, while the rate of consumption is held constant at 40 MB/s. Each set of points in the figures plots the result of an experiment that varies the number of outstanding messages allowed per producer (f=x). For all simulations, the latency of the network is 10 microseconds, there are 2 copies of each data set, and each consumer requests 5000 8 KB blocks. As one can see, the graduated declustering algorithm scales well, and does not require an increase in flow control proportional to scale.*

does increase with the number of replicas. Thus, with just a few outstanding messages, performance is excellent, nearly 100% of ideal.

Results from the prototype implementation are shown in Figure 10.2. As we can see from the figure, the prototype implementation performs quite well, matching our expectations from the implementation.

Figure 10.3 shows that there were a few experimental runs, however, in which this was not the case. In those cases, performance was fine for a period of time, but then suffered from dramatic, system-wide two second pauses, which severely impacted overall performance. Investigation of this symptom lead to the conclusion that some of the Myrinet switches were deadlocking, which halts progress entirely until the switch detects the deadlock and recovers from it.

Fortunately, the AM library was altered to avoid this problem [1]. However, the experience is illustrative of a general point: that our system relies on the global characteristics of the network, and if and only if the network is performing globally as expected will the system be able to tolerate producer-side and consumer-side performance faults. Global performance faults in the network have a global effect, and cannot be avoided by our mechanisms. However, note that localized performance faults in the network, such as link contention, are naturally handled by our system.

We also investigate the basic resource costs of the graduated declustering implementation. Figure 10.4 plots graduated declustering CPU utilization over time of both a producer and a consumer. During this experiment, the average sustained throughput of graduated declustering is 17.5 MB/s.

From the figure, we observe that the total CPU utilization of the producer at 57.8% is much greater than the consumer utilization at 28.2%, and a great deal of producer time is spent in the kernel. Threads and context-switching are the reason for the high percent of time spent in kernel mode, as the producer side employs three separate threads: one for the message request receipt and scheduling, and one for each replica produced; in this case, there are two replicas. The consumer-side only uses a single thread, and the resulting simplicity of implementation leads to a lower CPU cost.

We can also calculate the number of cycles per byte graduated declustering expends. For the producer, the cycles/byte ratio is 5.78, and for the consumer, the ratio is 2.82. Compared to the cost of the distributed queue, where roughly 3 cycles/byte are used by both the producer and consumer, we can see that the producer-side graduated declustering implementation is relatively heavyweight, due to the use of excessive threading. Future work could address this deficiency, perhaps by developing an event-based producer library.

We again break down the user-time component of CPU utilization further. Figures 10.5 and 10.6 present the producer-side and consumer-side CPU breakdowns, respectively. From the figures, we can see that once again, memory copy costs dominate the transfer, and the next largest component is raw Active Messages calls. Both the graduated declustering library and memory management costs are noticeable however, each utilizing roughly 1/10th of user-time. Note that even the user-level component of the graduated declustering library is more heavyweight than the distributed queue library, largely due to its increased complexity, but also because the manner in which it was constructed. Because the current graduated declustering implementation is built upon higher-level messaging primitives, as described in the Chapter 4, a slight CPU cost is exacted, in the form of

[1] The library had to be changed so as not to fragment messages into smaller chunks. When fragmented, the switches would observe fragment inter-arrival time, and sometimes erroneously assume that a delayed fragment implied a deadlock, and therefore would go into deadlock recovery mode. When the implementors of AM installed a fix, by not fragmenting messages, the switches no longer had reason to time-out.

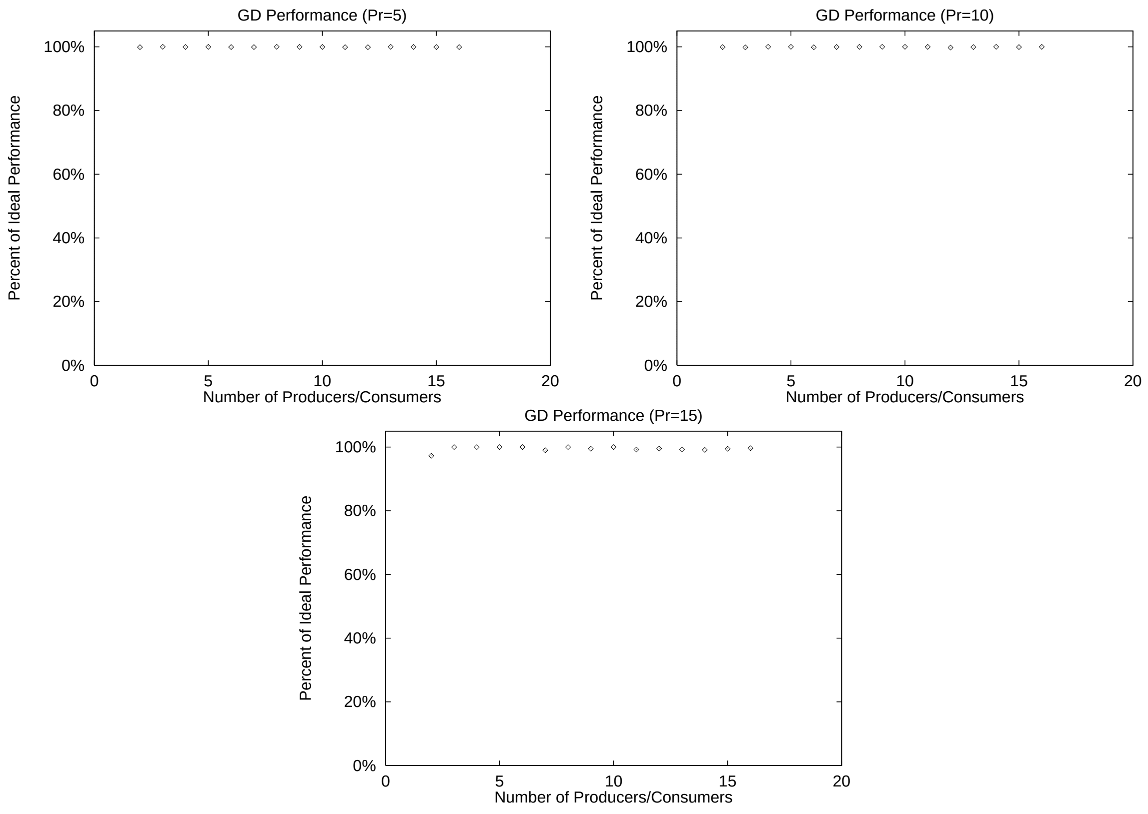


Figure 10.2: **GD Absolute Performance: Scale (Prototype).** *The figure plots the performance of graduated declustering under scale. In each graph, the number of producers and consumers is co-varied along the x-axis, and the percent of ideal performance is plotted along the y-axis. Thus, at x=16, there are 32 machines involved in the experiment. Across graphs, we vary the rate of production of each producer, from 5 MB/s to 15 MB/s. In all figure, the number of outstanding messages per producer is fixed at 16. As we can see from the figures, performance is excellent across the range.*

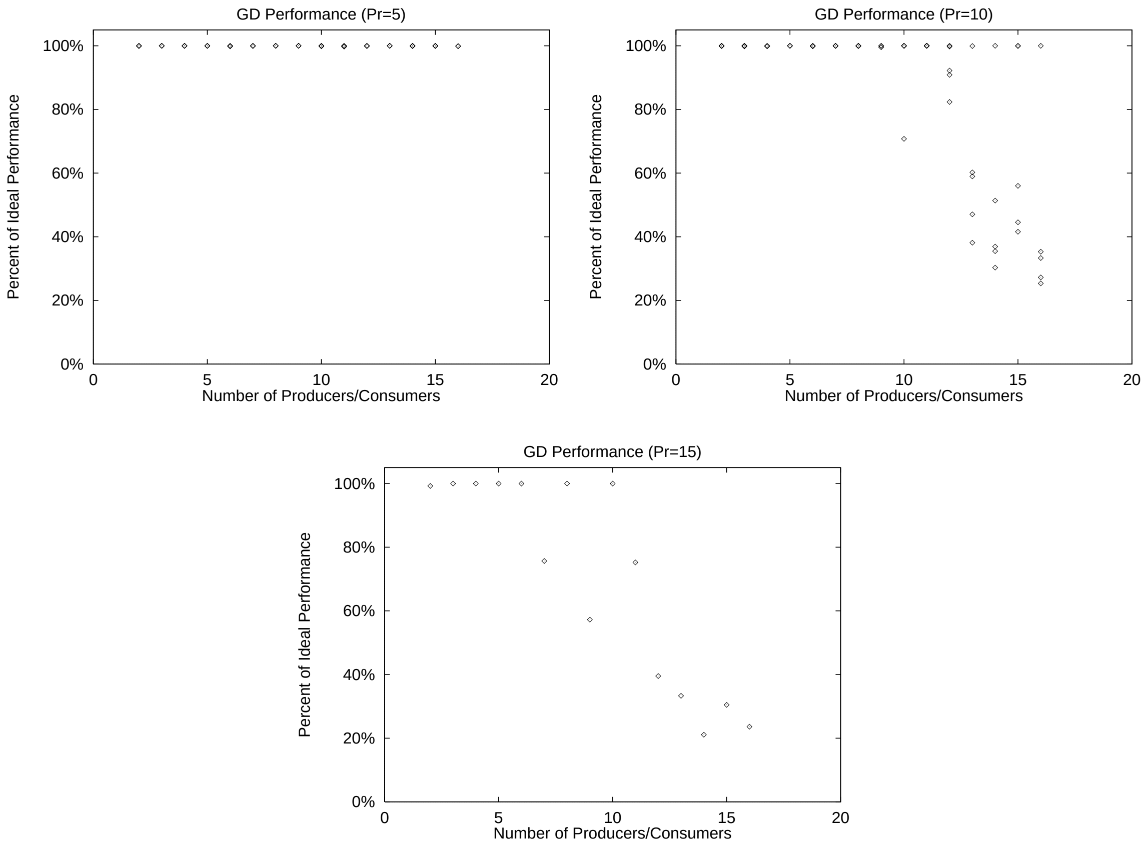


Figure 10.3: **GD Absolute Performance: Deadlock (Prototype).** *The figure plots the performance of graduated declustering under scale, in cases where the switch deadlock would occur. In each graph, the number of producers and consumers is co-varied along the x-axis, and the percent of ideal performance is plotted along the y-axis. Thus, at x=16, there are 32 machines involved in the experiment. Across graphs, we vary the rate of production of each producer, from 5 MB/s to 15 MB/s. As we can see from the figures, the deadlock only occurs at scale and higher rates, but when it does occur, system performance drops noticeably.*

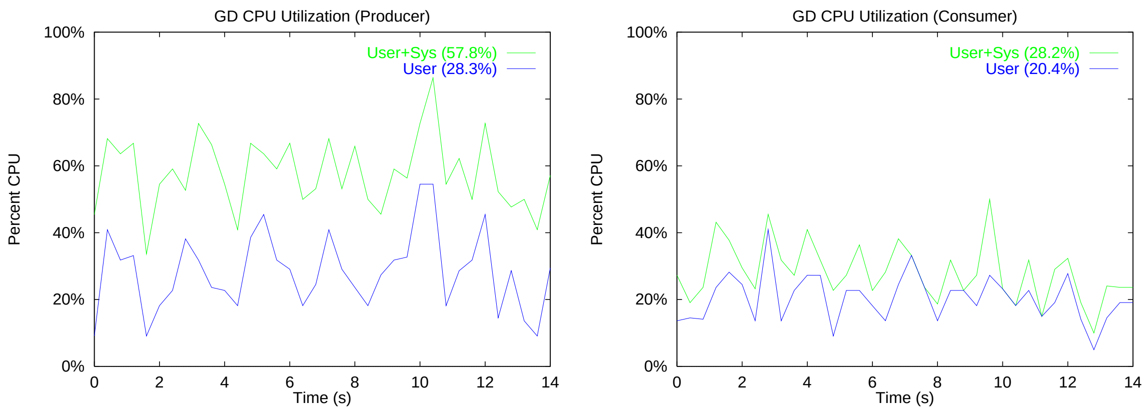


Figure 10.4: **GD CPU Utilization (Prototype).** *The figure plots CPU utilization during a run of graduated declustering. On the left, the CPU utilization of the producer is shown, and the consumer utilization on the right. Both graphs plot two lines: the bottom-most line is user CPU time, and the upper-most line the sum of user and system time. Average utilization is shown in the key. Average throughput for this particular experiment is 17.5 MB/s.*

memory management overhead.

10.1.2 Block Request Size

We now explore I/O request size. Because graduated declustering naturally multiplexes multiple logically-sequential data streams onto each disk, performance can be lost due to the disruption of each sequential stream. In our prototype, we attempt to control this loss by issuing larger block requests, which amortizes the cost of seeks between each stream, and thus approaches sequential performance. Previous and subsequent experiments will assume a large block size; we now investigate the performance of concurrent sequential operations under a range of different request sizes.

Figure 10.7 shows the performance of simple sequential read benchmark on two different disks. The leftmost graph depicts the performance of sequential reads on a 2 GB 5400-RPM Seagate Hawk disk, and the rightmost graph the same experiments on a 4 GB 7200-RPM Seagate Barracuda. Two sets of data points are plotted: the performance of a single sequential read request stream, and the performance of two concurrent sequential request streams, each plotted versus the request size.

For the single read stream, peak bandwidth is attained at request sizes of roughly 8 KB for the Hawk and 64 KB for the Barracuda. At those points, request sizes are large enough to amortize the overhead of each request and attain near-peak sequential bandwidth from the disks. A simple model that uses a fixed overhead plus transfer cost is plotted as a line of the graph, and matches the data quite well.

For the two concurrent read streams, the effect of interference is shown. Even with reasonably large request sizes, performance is lessened. However, with 1 MB blocks, the cost of seeking between the two streams reduces bandwidth by only 5%, which is acceptable.

Thus, graduated declustering should use large block requests to the disks in order to amor-

```
Copy                                3193 / 7155 (44.63%)
    memcpy              -- 3193 / 7155 (44.63%)

AM Routines                         2083 / 7155 (29.11%)
    Transfer            -- 1830 / 7155 (25.58%)
    SetAndWait/Poll/Other --  253 / 7155 ( 3.54%)

GD Library                           775 / 7155 (10.83%)
    Transfer            --  334 / 7155 ( 4.67%)
    Scheduling          --   94 / 7155 ( 1.31%)
    Memory Management   --   87 / 7155 ( 1.22%)
    Translation         --   81 / 7155 ( 1.13%)
    Wait                --   80 / 7155 ( 1.12%)
    Other               --   99 / 7155 ( 1.38%)

Memory Management                    626 / 7155 ( 8.75%)
    Malloc/New          --  325 / 7155 ( 4.54%)
    Free/Delete         --  301 / 7155 ( 4.21%)

Locking                              312 / 7155 ( 4.36%)

Unaccounted                          166 / 7155 ( 2.32%)
```

Figure 10.5: **GD Producer CPU Utilization (Prototype).** *The figure shows the user-time CPU breakdown of where time is spent in the producer-side of graduated declustering. Most time is spent in either the copy code or the raw AM routines (73.74% total). The GD library routines account for roughly 10% of total user time. Memory management take almost 9%, because message data is placed in a dynamically allocated buffer before handing it to the user. Finally, locking, which must occur between threads for safe access to shared data structures, takes over 4% of user-time. Data is gathered via PC sampling.*

```
Memory Copy                            4892 / 9249 (52.89%)
    memcpy                -- 4892 / 9249 (52.89%)

AM Routines                            2326 / 9249 (25.15%)
    Transfer              --  511 / 9249 ( 5.52%)
    SetAndWait/Poll/Other -- 1915 / 9249 (20.70%)

Memory Management                       845 / 9249 ( 9.14%)
    Malloc/New            --  337 / 9249 ( 3.64%)
    Free/Delete           --  508 / 9249 ( 5.49%)

GD Library                              809 / 9249 ( 8.75%)
    Transfer              --  581 / 9249 ( 6.28%)
    Translation           --  109 / 9249 ( 1.18%)
    Wait                  --   71 / 9249 ( 0.77%)
    Other                 --   80 / 9249 ( 0.86%)

Locking                                 181 / 9249 ( 1.96%)

Unaccounted                             164 / 9249 ( 1.77%)
```

Figure 10.6: **GD Consumer CPU Utilization (Prototype).** *The figure shows the user-time CPU breakdown of where time is spent in the consumer-side of graduated declustering. The copy routine takes almost 53% of the user CPU, and raw AM routines another 25% of user time. Memory management costs are noticeable at almost 10% of user-time, and the GD Library accounts for another 9% or so. Locking costs are small at 2%. Data is gathered via PC sampling.*

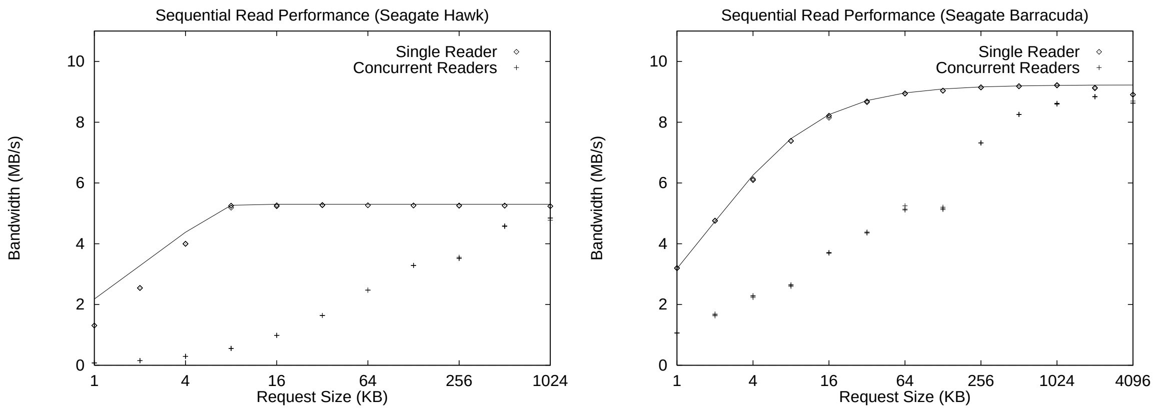


Figure 10.7: **GD Effect of Block Request Size (Prototype).** *The graphs reveal the results of two experiments run upon two disks. The leftmost graph presents results for a 2 GB 5400-RPM Seagate Hawk disk, whereas the rightmost graph is for a 4 GB 7200-RPM Seagate Barracuda. The 'Single Reader' data in each graph plots the achieved bandwidth versus the request size of a program that reads a 100 MB file from the disk sequentially. The 'Concurrent Reader' data shows the performance of the same benchmark, but this time with two copies running concurrently over different files. The benchmark uses the directio() feature in Solaris to avoid the buffer cache and read data directly from disk into user-space buffers. The models plotted for the 'Single Reader' lines use a simple fixed overhead plus cost-per-byte model; for the Hawk disk, the model is not perfect, due to the fact that the disk is on a fast-narrow SCSI chain, which limits peak bandwidth.*

tize the cost of multiple streams. If smaller block sizes are used, performance will suffer noticeably, and a factor of two can be easily lost.

10.2 Performance Under Perturbation

We now turn to performance of graduated declustering in the presence of performance faults. Because of the nature of the graduated declustering algorithm, its reaction to faults is much more complex than that of the distributed queue. More specifically, to correct a performance fault to a single producer, all other producers must compensate their bandwidth allocations in some manner, even though producers do not directly communicate with one another. The rate of this information propagation determines how quickly and effectively the graduated declustering algorithm adapts.

10.2.1 Perturbation Spectrum

The simulated perturbation spectrum of graduated declustering is shown in Figures 10.8 and 10.9. The first set of graphs vary the rate of each producer, and the second set vary the strength of the perturbation. In all graphs, the ideal model developed in Chapter 2 is shown as a line for the sake of comparison.

In each graph in the two figures, two sets of performance points are plotted. The upper set of points is labeled ‘Best’, and the lower ‘Worst’. These two extremes represent different perturbation layouts, and represent upper and lower bounds on performance under perturbation, respectively. In the ‘Best’ case, the perturbations are spread out among the producers so that there is minimal adjacency between them. In the ‘Worst’ case, when n perturbations occur, they occur on producers $1...n$, which are all adjacent. Thus, in the best case, no performance faults will occur on a replica until all primary data sets are perturbed, and in the worst case, in a system with N-way graduated declustering, the first N performance faults will always affect the producers of a given data set.

As one can see in Figure 10.8, under the ‘Best’ allocation, graduated declustering performs significantly better. In the ‘Worst’ allocation, performance drops significantly when just two of the producers are perturbed, to significantly less than ideal. The number of replicas of each data set is the important factor here. When two adjacent producers are perturbed, and there are only two copies of each data set in the system, the consumer that is reading the perturbed data set will not be able to obtain the requisite bandwidth from the producers. Note that the effects are particularly noticeable because of the high utilization of the performance fault of 0.8 used in the experiment.

However, in the ‘Best’ allocation, no adjacent producers are perturbed until more than half of producers are perturbed. Thus, we would expect a performance fall-off at 9 of 16 producers, which is exactly what we see in the graphs.

In Figure 10.9, one can see the effect of lessening the producer-side perturbation. In these figures, performance under the ‘Best’ and ‘Worst’ allocations does not differ nearly as widely as in the previous figure, as would be expected.

We next investigate the performance of the prototype implementation. Figure 10.10 plots the performance of the prototype under perturbation across three different system sizes, utilizing the ‘Best’ layout of perturbations. In all cases, performance is excellent, dropping off only when more than half of the producers are perturbed. Though we do not run the full range of experiments that

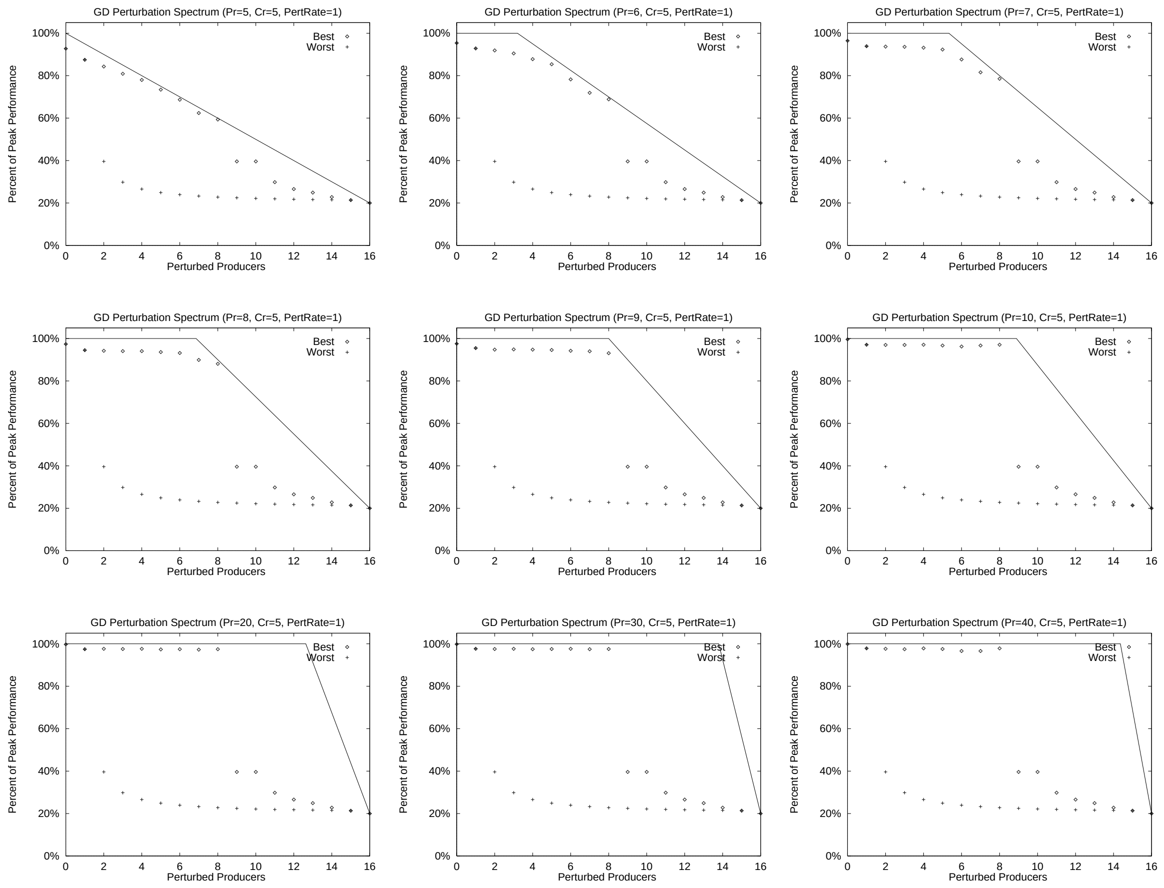


Figure 10.8: **GD Perturbation Spectrum: Consumption Rate (Simulation).** *The simulated performance of graduated declustering is plotted under perturbation. The number of producers that are perturbed is varied along the x-axis, and the percent of peak performance is plotted on the y-axis. Each graph varies the rate of production of each producer, while the rate of consumption is held constant at 5 MB/s. Two sets of points are plotted on each graph: 'Best' represents the best possible layout of perturbations across producers, and 'Worst' the worst. The line plotted is the model of ideal performance as developed in Chapter 2. In the simulation, there are 16 producers and 16 consumers, each data-set is replicated once, the latency of the network is 10 microseconds, the rate of a perturbed producer is 1 MB/s, and 5000 8 KB blocks are requested by each consumer. Performance of the 'Worst' allocation drops immediately under just two perturbations, as the consumer with its data sets on those two producers is unduly affected. The 'Best' allocation performs much better, sustaining nearly ideal performance until more than half the producers are perturbed.*

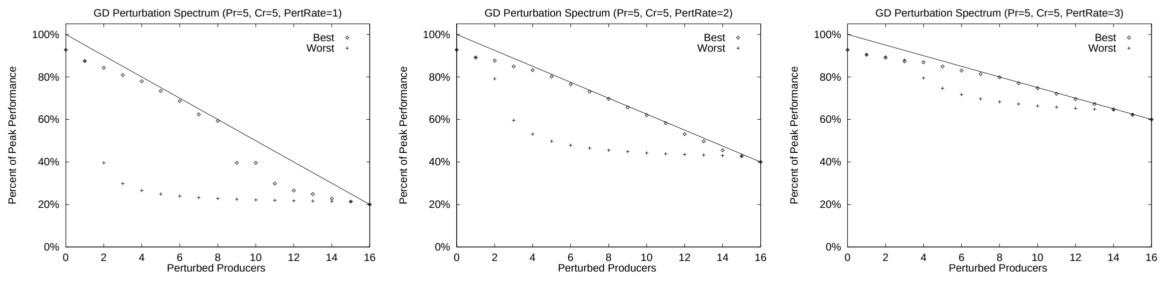


Figure 10.9: **GD Perturbation Spectrum: Perturbation Rate (Simulation).** *The simulated performance of graduated declustering is plotted under a variety of perturbations. The number of perturbed producers again increases along the x-axis, while percent of ideal performance is plotted along the y-axis. Each graph varies the strength of perturbation, from stronger on the top left (perturbed producers produce at 1 MB/s) to weaker on the bottom right (3 MB/s). The rate of consumption and of unperturbed producers is 5 MB/s. Performance under 'Best' and 'Worst' allocations are plotted, as is an ideal performance line. In the simulation, there are 16 producers and 16 consumers, each data-set is replicated once, the latency of the network is 10 microseconds, and 5000 8 KB blocks are requested by each consumer. As one can see, smaller perturbations have smaller overall impact on performance, and decrease the discrepancy between 'Best' and 'Worst' allocations.*

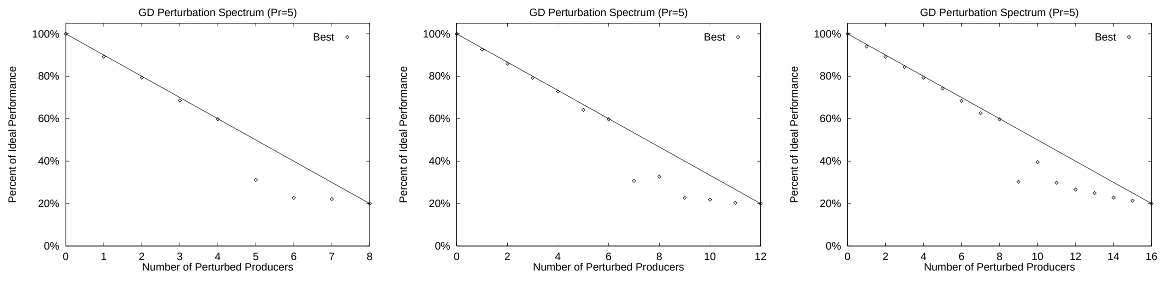


Figure 10.10: **GD Perturbation Spectrum: Scale (Prototype).** *The performance of graduated declustering prototype implementation is shown under a range of perturbation experiments. Within a graph, the number of perturbations is increased along the x-axis, from 0 up to the size of the cluster under study. The y-axis plots percent of ideal performance. Across graphs, the scale of the system is varied, from 8 (8 producers and 8 consumers on 16 machines) up to 16 (32 machines). The rate of unperturbed production is 5 MB/s, whereas the rate of a perturbed producer is 1 MB/s. The line in the graphs is a model of ideal performance, as developed in Chapter 2. In all experiments, the 'Best' layout of perturbations is used. As one can see from the figures, performance of the prototype is excellent, as simulations would predict.*

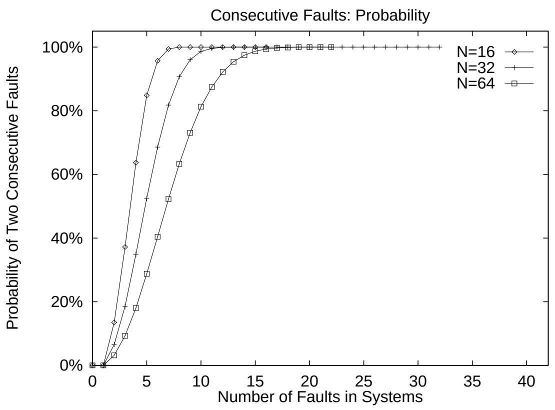


Figure 10.11: **GD Probability That Faults Will Be Consecutive.** *The graph plots the probability that any two faults will occur consecutively in systems of sizes 16, 32, and 64 machines. The number of faults is increased along the x-axis. Thus, as more faults enter the system, the more likely it is that two will occur next to each other. The data is generated by a simulation that places balls in bins and checks the consecutive criteria. For each data point, 100,000 trials were generated.*

we had in the simulations, all comparable results we have obtained to date match simulation results quite closely.

We should note that in a real system, neither the 'Best' or 'Worst' layout of performance faults is likely to occur, but rather something in between. The average case can be calculated by assuming a random distribution of performance faults across machines, and determining how often consecutive faults are likely to occur. For example, with 2-way graduated declustering, on a system with N disks, if there are two faults, the probability that those two faults will not occur on consecutive disks is $1 - \frac{2}{N-1} = \frac{n-3}{n-1}$, assuming $N \geq 4$. Thus, the larger the system, the better the odds are that the 'Worst' case will not occur. For three faults, the probability that any two will not occur on consecutive disks is more complicated: $(\frac{n-3}{n-1})((\frac{n-3}{n-1})(\frac{n-6}{n-2}) + (\frac{2}{n-1})(\frac{n-5}{n-2}))$, assuming $N \geq 6$.

From this equation, one can see the difficulty of developing the general closed-form model of average case behavior. However, we can employ a simulation to show the behavior in certain cases of interest. Figure 10.11 plots the results, increasing the number of faults along the x-axis, and plotting the probability that any two faults will occur on consecutive nodes on the y-axis.

From the graph, we can see that for a system of size 16, if 0 or 1 faults occur, there is obviously a 0% chance that two faults will be consecutive. However, the probability quickly ramps up and approaches 100% with just 6 faults in the system. For larger system sizes of 32 and 64, we can see that the odds of avoiding consecutive faults improve; with 32 nodes, roughly 13 faults must occur for a 100% certainty, and with 64 nodes, 22 faults must occur. However, if we look at the half-way point, we see that with 3, 5, and 7 faults, on systems of size 16, 32, and 64, respectively, we obtain a 50% chance of avoiding consecutive faults. Thus, every time our system size doubles, we only improve our chances of avoiding consecutive faults by a linear amount.

10.2.2 Scale Under Perturbation

Our next set of experiments explores perturbed performance under scale. Both the number of producers and consumers in the system are varied, as well as the number of replicated data sets. Figures 10.12 and 10.13 plot the performance of graduated declustering under increasing scale and replication, with the first set of graphs presenting the best possible perturbation layout, and the second set the worst.

As one can see from the figures, the number of replicas in the system plays an important role in performance under perturbation. In a system with R copies of a data set, where $R = 1$ implies that each data set is available at a single location, R perturbations can lead to worst case performance, depending on the location of the perturbations. Thus, increasing R improves the tolerance of the system.

The corollary to this is that in systems with more replicas, performance in the case with just a few replicas is slightly lessened, due to the higher overhead of managing an increasing number of data streams. Thus, though worst-case drop-offs are better avoided with more replicas, average-case performance may be worse.

The other costs of increasing the number of replicas are more global in nature. First, when does one create the replicas? If this replicas are created on every write to the system, all write bandwidth is reduced by a factor of R, where R is the number of replicas. Thus, an off-line strategy may be preferred, where oft-read data sets are replicated, and other temporary data sets are left uncopied.

Second, the storage cost of replicas must be taken into account. If all data sets are replicated, storage space is reduced by a factor of R. The same solution as proposed above must also take this factor into account.

In summary, replication is costly if performed naively. Therefore, an off-line strategy is suggested, though not explored here. However, just a single replica can potentially help to tolerate a large number of performance faults – up to $n/2$, where n is the size of the cluster. In extremely large clusters, where perturbations are more likely to be the common case, a single replica of heavily-used data sets may not be enough to tolerate common-case performance faults, as two consecutive faults will drastically reduce performance, and therefore extra replicas are recommended.

In the future, utilizing partial replicas could provide a lower-cost alternative, analogous to the use of partial redundancy in the AFRAID system [106]. In such a system, one could take advantage of partial collection replication, by reading what one could from the partial replica, in order to avoid some performance faults; however, this would limit the flexibility of the consumer in that only some data could be obtained from both producers, and would also increase the complexity of the implementation.

10.2.3 Total Work

As with the distributed queue, we now explore the total amount of work needed in order to tolerate a single perturbation. Without enough excess parallelism, perturbations are quite difficult to tolerate. Figure 10.14 plots the simulated performance of graduated declustering under a single producer-side perturbation, while varying the total amount of data read.

Similar to the distributed queue, the performance of graduated declustering under perturbation decreases when the total amount of data that passes through the construct decreases to too small

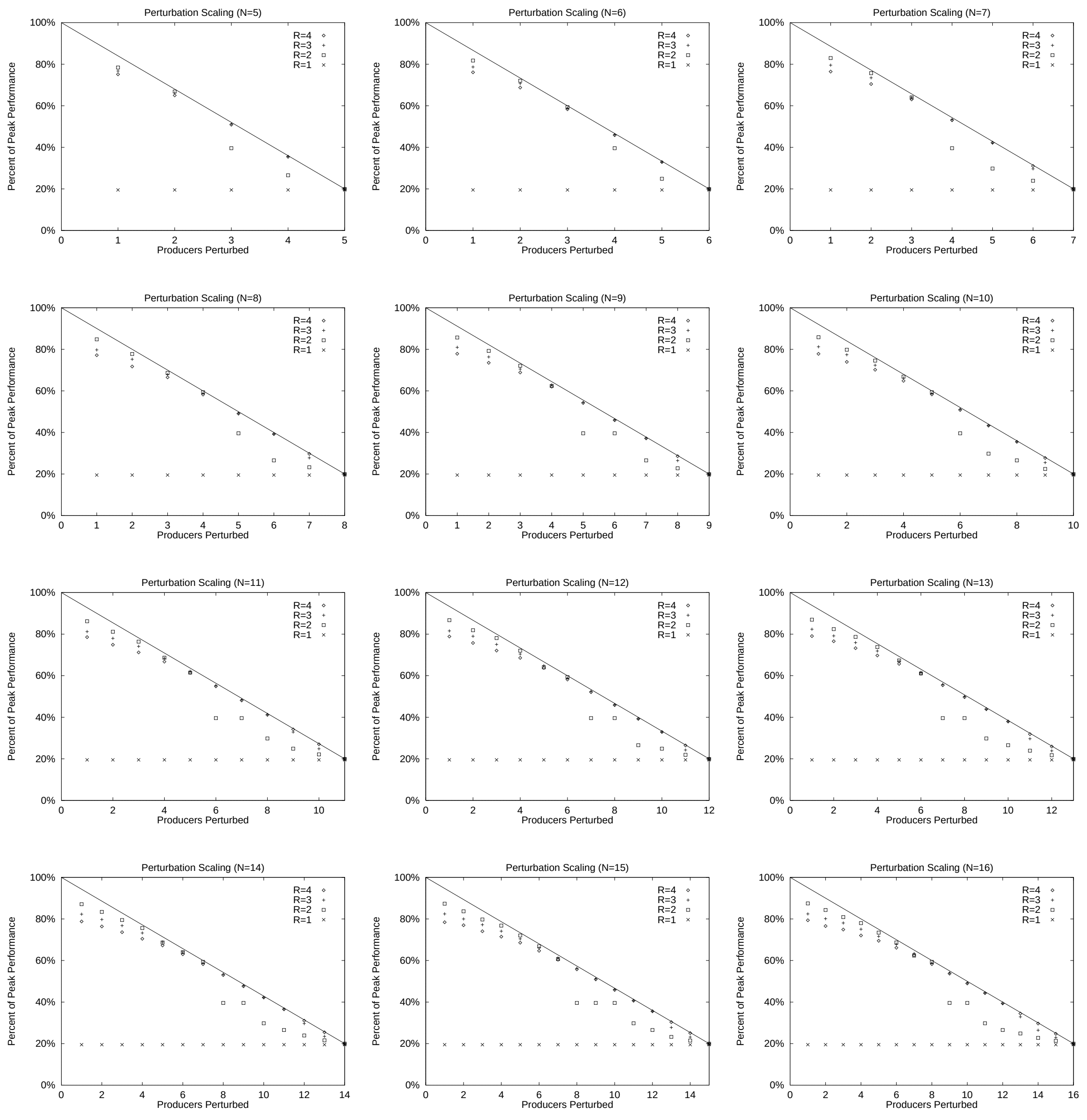


Figure 10.12: **GD Perturbations under Varying Scale: Best Layout (Simulation).** *The simulated performance of graduated declustering is shown under the best possible perturbation layout. The x-axis of each graph increases the number of perturbations in the system, whereas the y-axis plots the percent of ideal performance. Each set of points plots a different number of data sets; 'R=1' implies there is no replication present, 'R=2' implies there is a single replica, and so forth. Each graph varies the scale of the system under perturbation. Unperturbed producers and consumers produce at 5 MB/s, whereas perturbed producers produce at 1 MB/s. The network latency is 10 microseconds, and each consumer requests 5000 8 KB blocks. Without any replication, performance suffers under just a single perturbation. With one or more replicas, performance is much better. Note that with more replicas, performance is worse in the case of fewer perturbations.*

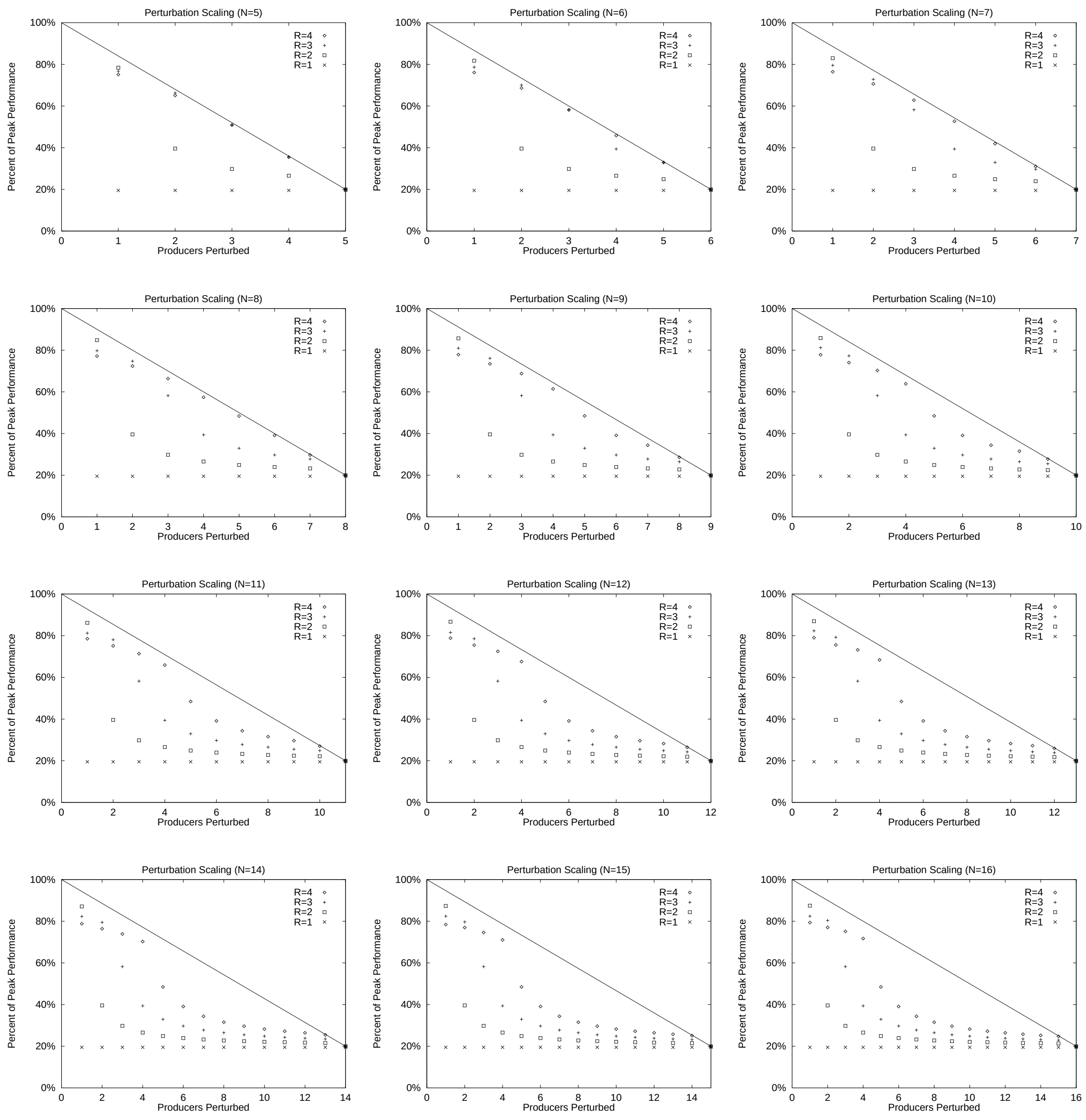


Figure 10.13: **GD Perturbations under Varying Scale: Worst Layout (Simulation).** *The simulated performance of graduated declustering is shown, with the worst possible allocation of perturbations (all consecutive). The x-axis of each graph increases the number of perturbations in the system, whereas the y-axis plots the percent of ideal performance. Each set of points plots a different number of data sets; 'R=1' implies there is no replication present, 'R=2' implies there is a single replica, and so forth. Each graph varies the scale of the system under perturbation. Unperturbed producers and consumers produce at 5 MB/s, whereas perturbed producers produce at 1 MB/s. The network latency is 10 microseconds, and each consumer requests 5000 8 KB blocks.*

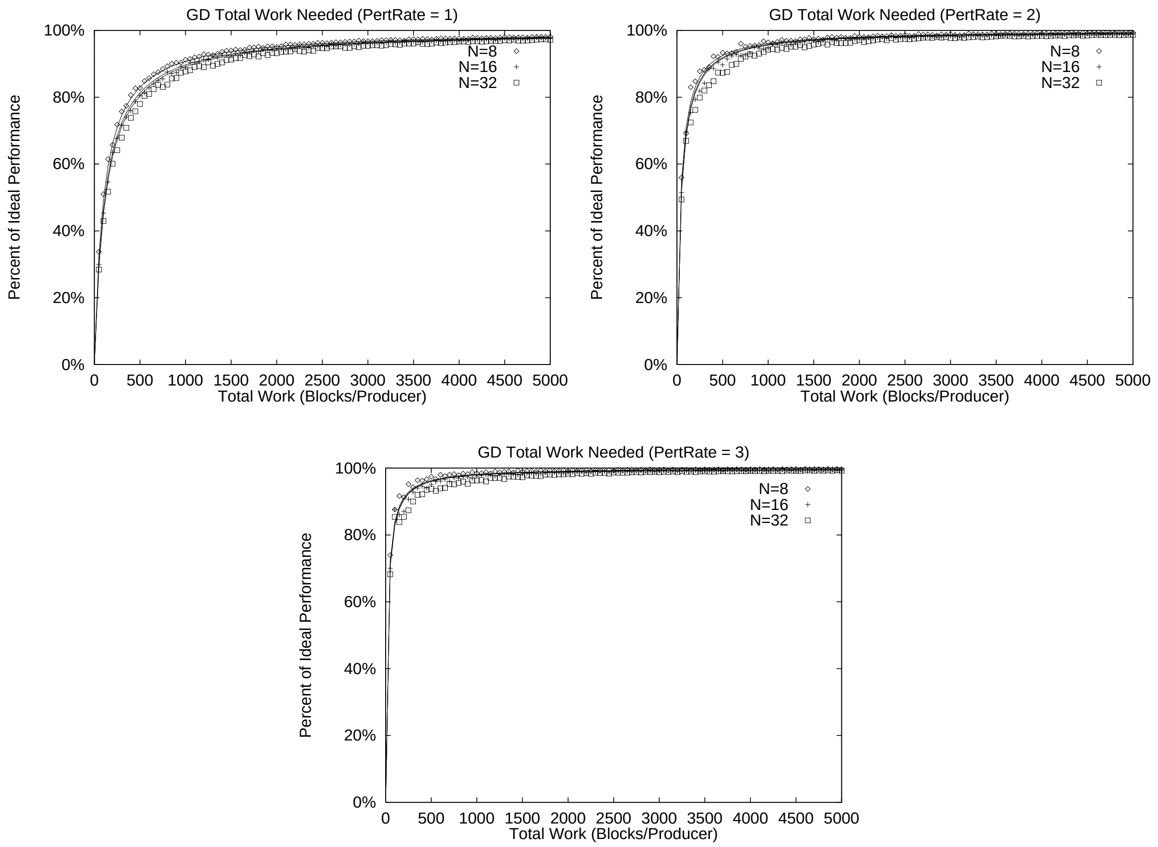


Figure 10.14: **GD The Need for Parallelism (Simulation).** *Simulated performance as a percent of ideal of graduated declustering is plotted while varying the total amount of data read along the x-axis. Each graph varies the rate of the perturbed producer, from 1 MB/s to 3 MB/s; all other producers and consumers run at 5 MB/s. The number of outstanding messages is set to 16, and the network latency is set to 10 microseconds. The more severe the perturbation, the more total work is needed to approach ideal performance.*

of a level, and is worse when the perturbation is more severe, as seen in the case when the rate is set to 1 MB/s.

This end-effect can be modeled, as we now demonstrate. The ideal expected time, $T_{expected}$, can be derived by dividing the total work by the average rate of production:

$$T_{expected} = \frac{W}{R_{total}}, \tag{10.2.1}$$

where W is the total number of bytes moving through the system, and R_{total} is the total rate, across all producers, at which data are produced. Refining this, we arrive at:

$$W = N_b \cdot S_b, \tag{10.2.2}$$

$$R_{total} = \frac{(p-1) \cdot R_p + (1) \cdot R_{pert}}{p}, \tag{10.2.3}$$

where N_b is the number of blocks that the system will process, S_b is the size of each block, R_p is the rate of each non-perturbed producer, and R_{pert} is the rate of the perturbed producer. Thus, in the best case, we could hope for all blocks to be processed at the average rate of all of the producers.

However, each producer will have $flow_{total} = flow_{single} \cdot M$ outstanding requests to process at the end of the run, where $flow_{single}$ is the number of outstanding messages each consumer is allowed to each of the producers, and M is the level of replication of the data sources; in this example, $M = \ = 2$. Thus, at the end of the run, the entire system will be waiting for the slow producer to finish. This extra time can be modeled as follows:

$$T_{extra} = \frac{flow_{total} \cdot S_b}{R_{pert}} - \frac{flow_{total} \cdot S_b}{R_p} \tag{10.2.4}$$

Thus, the overall time to complete a run is:

$$T_{overall} = T_{expected} + T_{extra}, \tag{10.2.5}$$

and the ratio, as plotted in the figures, is:

$$Ratio = \frac{T_{expected}}{T_{expected} + T_{extra}}. \tag{10.2.6}$$

These models match the numbers from the simulation quite well, as one can see in Figure 10.14.

10.2.4 Importance of Scheduling

We next demonstrate the importance of producer-side scheduling to graduated declustering. Figure 10.15 shows the performance of different producer data-set scheduling algorithms under a perturbation to a single producer. The figure plots the average bandwidth received by each of the consumers over time.

Three different scheduling disciplines are shown. The first, ’Fair-Share’, picks a random consumer request to service, and is similar to a randomized proportional-share scheduler [125]. This

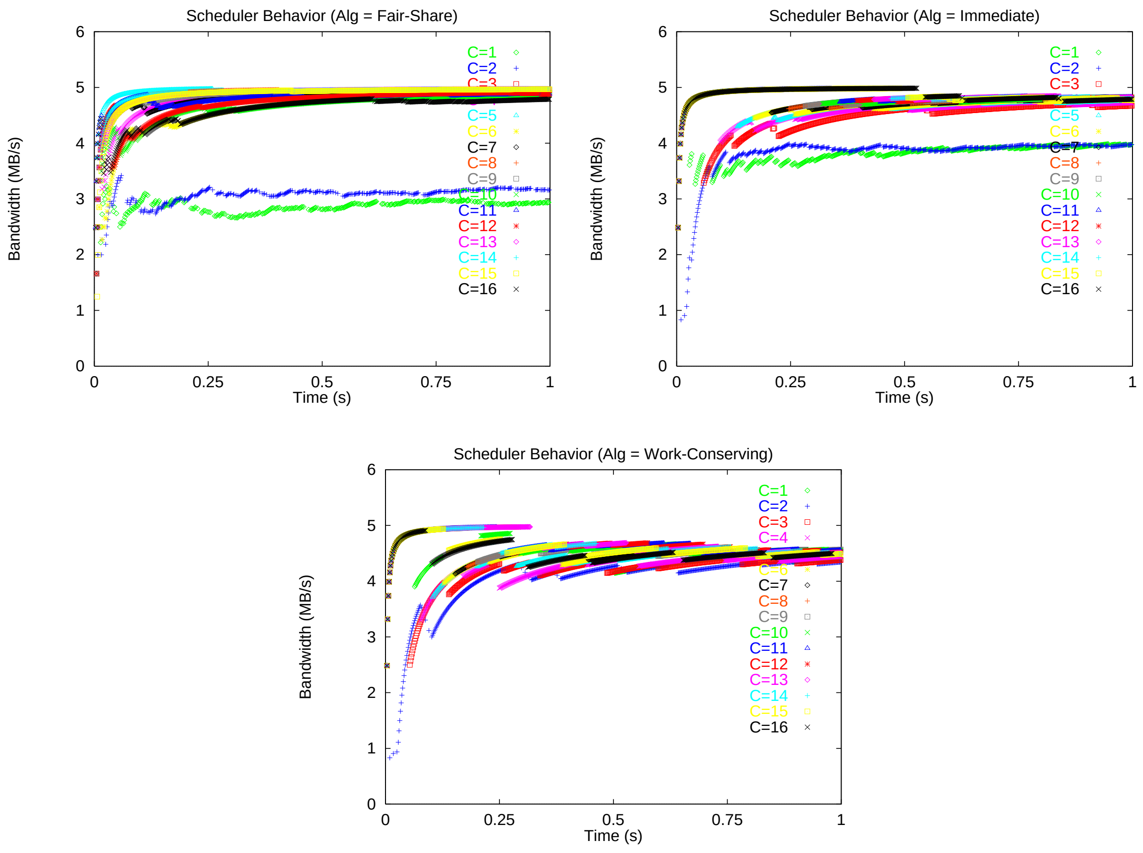


Figure 10.15: **GD The Importance of Scheduling (Simulation).** *The graphs plots simulated consumer bandwidth received over time under three different scheduling algorithms, under a single perturbation. 'Fair-Share' picks a random consumer to service, and performs poorly, as two of the consumers receive much less than an equal share of the bandwidth. 'Immediate' performs the basic graduated declustering algorithm, but will schedule requests from the less-favored request queue if the other queue is empty. 'Work-Conserving' is the graduated declustering algorithm as described in the previous chapters. Only the 'Work-Conserving' algorithm, which takes into account the global progress of consumers and will not schedule requests from consumers that are ahead, is able to balance the bandwidths across consumers successfully. For all experiments, there are 16 producers, 16 consumers, 2 data sets per consumer, and a 10 microsecond network latency. Each producer produces at 5 MB/s unperturbed, whereas the single perturbed producer produces at 1 MB/s. All consumers consume at 5 MB/s.*

option serves as a straw-man for comparison. Not surprisingly, its performance is poor, as each producer's bandwidth is fairly shared among its two consumers. Thus, under a single producer perturbation, two consumers are ill-affected and perform poorly, whereas other consumers receive peak bandwidth.

The next two algorithms are quite similar. The first, 'Immediate', employs the basic graduated declustering algorithm, but with a slight modification. Assume that there are only two request queues, '0' and '1', with requests from consumers '0' and '1', respectively. When it comes time to schedule a block for service, and both queues have some number of requests within them, the 'Immediate' algorithm will select the request to schedule based on its global notion of progress, such that whomever is behind receives service, in the exact same manner as the graduated declustering algorithm. However, let us assume consumer '0' is lagging, but currently has no requests pending in the request queue. In this scenario, the 'Immediate' algorithm will go ahead and schedule requests from the other consumer, consumer '1'. From the graph, we see that the 'Immediate' algorithm does not achieve the desired result of balancing the bandwidths effectively across consumers.

The next and final algorithm, 'Work-Conserving', differs in only one way – it will not schedule any requests from a request queue that is perceived to be ahead of the other request queues. Thus, if the producer thinks that consumer '0' is lagging consumer '1', and '1' has requests in its request queues, but '0' doesn't, the scheduler will simply wait for requests from '0' to arrive, rather than schedule any from '1'. This type of scheduler is known as a *work conserving* scheduler. In other words, even though there is work present, the scheduler will not perform it immediately, because it has a particular global goal in mind. The graph shows the results are as desired, balancing bandwidths effectively across all consumers.

10.2.5 Time to Convergence

We close our experimentation with graduated declustering by examining the effect of a single perturbation over time, in an attempt to understand how long it takes the system to react to a single perturbation. Figure 10.16 plots the cumulative number of messages sent over time from each producer to each of its two consumers; a perturbation is inserted into the system at $t = 4$ seconds on the first producer ($D = 1$ in the diagram).

The effects of a single perturbation to graduated declustering are more complex than a perturbation to the distributed queue. At the producer where the perturbation occurred (producer '1'), the reaction is immediate; however, all of the other producers must react as well, and this takes some time to propagate out to them, as they have no direct communication channel with other producers.

Observe producer '8' in the figure, which is the furthest away from producer '1' in the circular chain. The bandwidth allocation from it remains unmodified until roughly half of a second after the perturbation. Thus, if a perturbation occurs for less than half of a second at this scale, the consumer bandwidths will not be properly balanced. If many such short-lived perturbations occur across the system, the algorithm will fail to yield the desired result; we leave coping with this potential problem to future work.

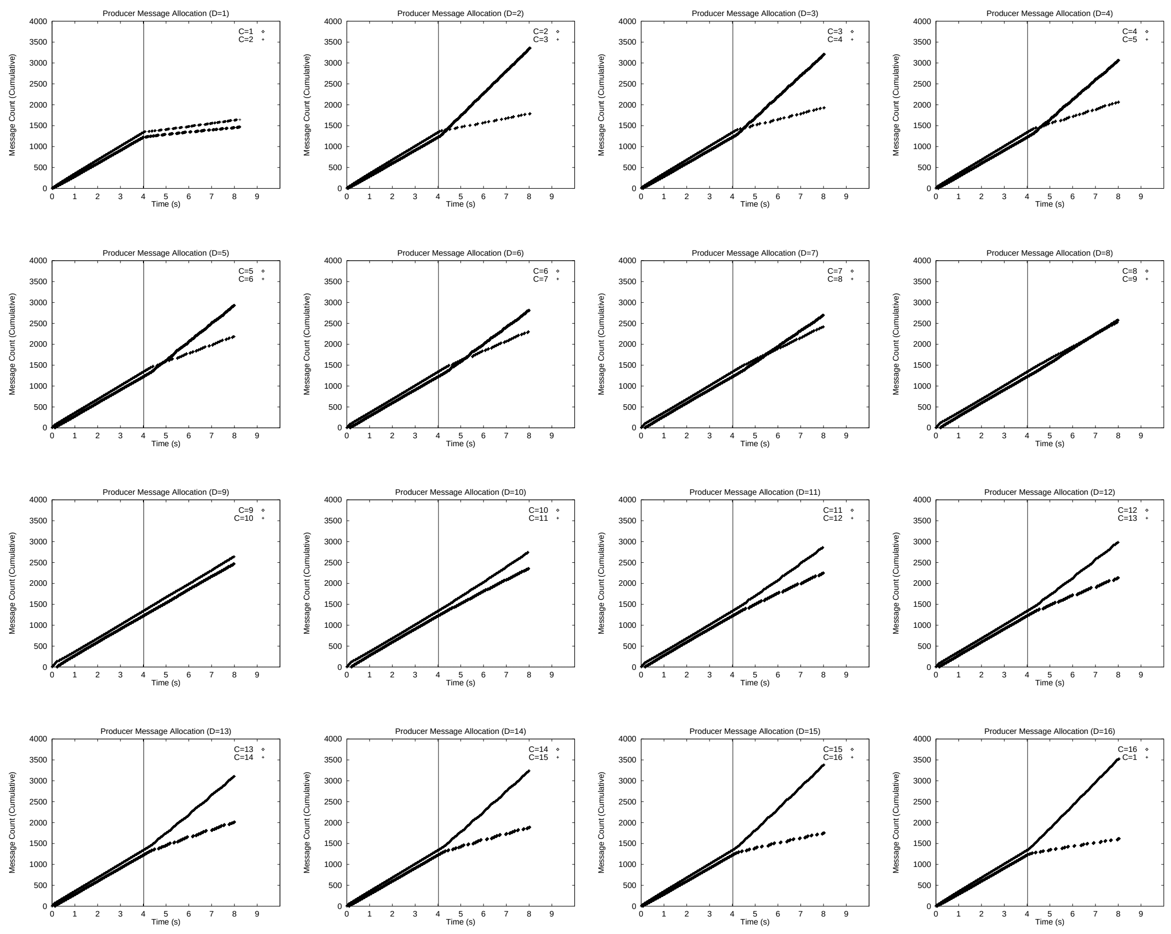


Figure 10.16: **GD Producer Allocations Under Step-Up (Simulation).** *Simulated message counts from each producer to each consumer is plotted in the graphs. Each graph plots the cumulative number of messages sent from a particular producer to each of its two consumers over the life of the experiment. The vertical line in the graph indicates the time at which the perturbation occurs; the perturbation takes place on producer 1, labeled D=1. Best-fit lines are plotted as well; the slopes of the lines are proportional to the bandwidth received by the consumers, and the value (in MB/s) is shown in parentheses in the key. The total number of messages received by a particular consumer can be calculated by viewing two adjacent graphs; for example, by examing the first two graphs (D=1 and D=2), we see that consumer 2 (C=2) receives most of its blocks from producer 2 (getting a much larger share of the source than consumer 3 does), and roughly splits the blocks from producer 1 with consumer 1.*

Chapter 11

Experiments: Summary and Discussion

We close our experimental chapters with a summary and discussion of what we have learned. First, we concentrate on general methodological lessons, and then move on to specific systems-building advice.

11.1 Methodology

At the methodological level, we have seen the value of studying both simulations and a prototype implementation, which was also noted in [8]. The highly-controlled simulator let us focus purely on algorithmic issues, while avoiding some of the vagaries often found in real implementations. Then, when moving to the real implementation, results could be compared against expected simulation results; when there was a mismatch, we knew it was not from an algorithmic property, but rather had to do with an aspect of the system not performing as expected. For example, the fairness or deadlock properties of the Myrinet switches, or the flow control of the message layer, both would have been difficult to discover had we not known that our algorithms should be performing well. Before we had developed the simulator, performance problems in the implementation were much more difficult to track down, and the range of experiments we performed were much less broad in nature.

Also of value was the comparison of all experimental to results to that of the *ideal* system. Instead of comparing performance to that of another, lesser-performing algorithm, all results can be seen versus their true potential, due to the model of ideal performance we developed in Chapter 2. We believe that this style of comparison should be employed wherever possible.

11.2 Distributed Queue

The experiments reveal that the basic distributed queue algorithm is quite scalable and achieves the desired behavior, avoiding consumer-side performance faults gracefully. Critical to achieving this behavior is the number of outstanding messages that are allowed. If the distributed queue is allowed too few outstanding messages, overall performance will suffer, because the producers will not be able to keep the consumers busy. Further, under perturbation, it is critical to allow each producer at least one outstanding message per consumer. Without one credit per consumer, the producer will not be able to track and avoid all slow nodes.

One major simplification we have made is that the processing time per message block at the consumer is fixed. This simplification allows us to have a fairly simple algorithm for detecting slow nodes. In a more dynamic environment, more sophisticated algorithms may be required to gauge the rate of progress of remote nodes.

We have also seen that the distributed queue algorithm assumes a fair network; without fairness in the network, some producers may be able to deliver their workloads sooner than other producers, and thus all entities will not finish at the same time. In the future, a more sophisticated algorithm that tracks producer progress and schedules consumption based on that metric would be interesting to explore.

Finally, we have also seen that the distributed queue algorithm requires a reasonable amount of parallelism to tolerate faults; with too little work to do, perturbed consumers will become the bottleneck. An interesting solution would be to employ an adaptive algorithm that first began with smaller-sized messages, and then slowly ramped up to the larger size over time. By beginning with small-sized messages, the amount of parallelism is increased, perhaps at the cost of higher CPU overheads; then, over time, the system could adjust for long runs by increasing the message size, thus regaining the efficiency of large blocks for long runs. Currently, we believe that the extra complexity required to implement this solution does not merit the implementation effort, because many of the applications that we are familiar with have large data sets with much excess parallelism, as we will see in Chapter 12. Thus, we leave this adaptive ramp-up for future work.

11.3 Graduated Declustering

The basic graduated declustering algorithm also works quite well, yielding high absolute performance under scale, as well as good performance under perturbation. The major difference from the distributed queue is that while the distributed queue can tolerate faults to any of the consumers, graduated declustering is more sensitive to producer-fault placement, due to the nature of how it moves data from producers to consumers.

Critical to the performance of graduated declustering is the ability of the producers to track consumer-side progress. By monitoring the progress of each remote consumer that it services, a producer is able to bias its scheduling so as to lead the consumers to progress at the same rate. Our current algorithm piggy-backs this information onto requests for data, which has the advantage of not adding any extra messages into the basic protocol. In the future, investigating methods for propagating this information separately from the data flow could be interesting.

Clearly, one important aspect of graduated declustering is the extra capacity and bandwidth costs it places on the system. Because each producer-side data set must be replicated one or more times, the question of when to replicate arises. Though not addressed in this document, a simple cost/benefit analysis could be applied to determine when a particular data set should be replicated.

From our experiments with producer-side scheduling, we saw the importance of work conservation. A subtle change in the scheduling algorithm of graduated declustering led to much poorer performance characteristics under perturbation.

We also have seen how the performance-fault layout plays a strong factor in determining graduated declustering performance. Though there is currently no solution to this limitation, it does give system designers extra information on how to fold new and faster hardware into a system. For example, if a system with 100 slow disks is embellished with 100 new, faster disks, the disks from each group should be logically interleaved, instead of having each group of disks arranged contiguously. Physical separation is both likely and acceptable; we simply recommend altering logical addresses.

Finally, from the scenario where network deadlock occurred, we see the reliance of graduated declustering on the performance of the network. In fact, both the distributed queue and graduated declustering algorithms are built to tolerate perturbations to *local* elements, such as slow producers, slow consumers, or even slow network links. However, if some *global* element, such as all the network switches, do not perform as expected, all producers and consumers will be ill-affected. In this case, there is nothing one can do in software to avoid the problem. Even replication in hardware would not have solved this problem; had the traffic been switched over to another Myrinet network, the same deadlock would occur. Perhaps a heterogeneous backup network would have achieved the desired effect, because a network made by a different company would be unlikely to have the same exact performance problem. Thus, when engineering a system for performance availability, it is critical to ensure that such global elements perform as expected and can handle the demands placed upon them.

Chapter 12

Putting It All Together

In the previous chapters, we have established the baseline efficiency of our two performance-available data-transfer primitives, graduated declustering and the distributed queue. In this chapter, we apply those mechanisms to transform fragile, non-robust database query-processing primitives into programs that can tolerate disk performance faults. To verify the utility of our transformations, all results within this chapter are from the real prototype system and are not simulations. We focus on disk performance faults because they are the main cause of performance variations in I/O-intensive environments; constructing programs that are robust to all types of performance faults would require quite a bit more effort, and thus we leave this to future work.

We concentrate on query-processing operators, such as scans, selections, joins, and sorts, as these primitives are particularly important in large-scale data-warehousing and decision-support environments. Though we are restricting ourselves to a specific domain, we believe that the diversity and importance of these database programs amply justifies our decision.

The main challenge for us in this chapter lies in understanding how to apply our robust data-transfer primitives. Although we have shown that the mechanisms are effective in achieving their pre-specified goals, we have yet to show that they are effective real programs.

12.1 Scan

We begin our examination by considering the most basic possible program with parallel I/O requirements: a parallel scan of a large data set. A scan reads through a data set in its entirety, performs no computation on the data, and does not have a write component. Scans often comprise a significant portion of many out-of-core applications, including data mining and scientific workloads. In this chapter, most of our query-processing operators will consist of a scan phase.

The data-flow for the non-robust scan is depicted in Figure 12.1. From the diagram, one can see the simplicity of the scan program. Data is read from disk in parallel by a set of disk modules, labeled D_R in the figure. Then data is passed onto the main scan module, labeled S. All of the data movement is embarrassingly parallel, as there is no communication across components on separate machines.

Making the scan robust to disk performance faults is straight-forward, as Figure 12.2 shows.

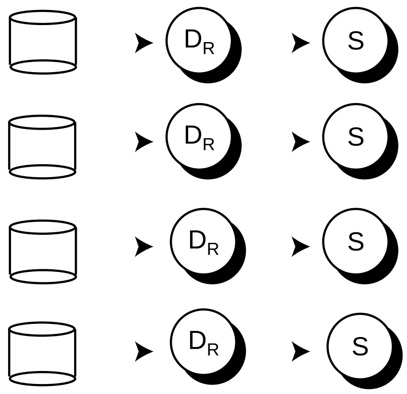


Figure 12.1: **Scan Data Flow.** *The basic data flow of the scan application is presented. Data is read from disk by a set of disk-read modules, labeled D_R, and passed to the scan modules, labeled S. Each disk-read module passes its data on to the scan application, which discards the data. Note that application is embarrassingly parallel, as there is no communication across components.*

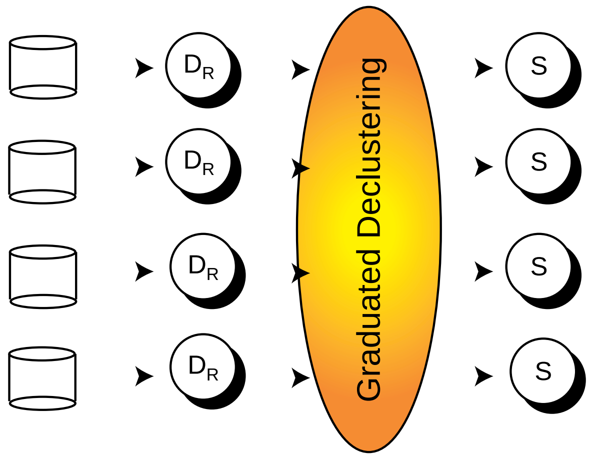


Figure 12.2: **Disk-Robust Scan Data Flow.** *The disk-robust scan is presented. Robustness to faults at the disks is achieved via use of graduated declustering, which is inserted between the applications and the disks.*

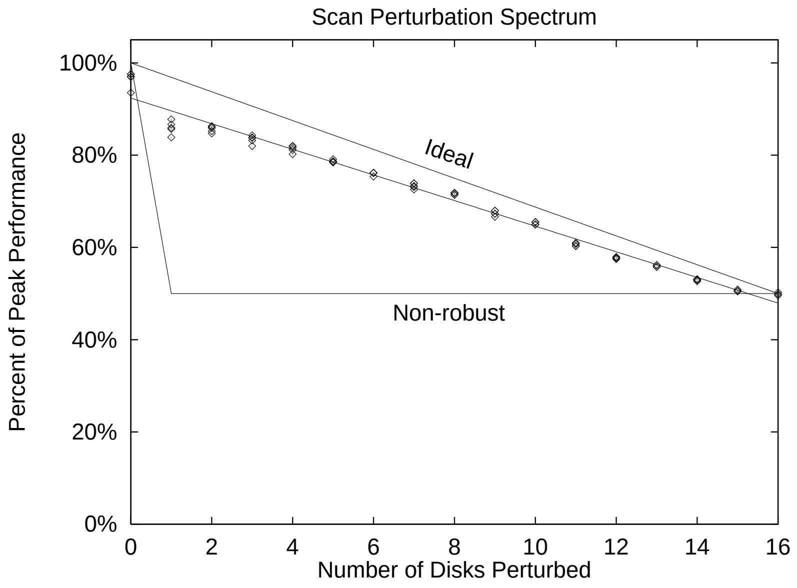


Figure 12.3: **Scan Perturbation Spectrum.** *The figure presents the behavior of the scan application under perturbations. The experiment is run on 16 machines, and is a scan of 2.4 GB of data, 150 MB per machine. The number of perturbations is increased along the x-axis, from 0 to the full cluster size of 16, where a perturbation is a stream that consumes half of the peak disk bandwidth. The percent of peak performance is the figure of merit, and plotted along the y-axis. The results of five separate trials are presented, shown as points in the diagram, and are compared against two lines. The first, labeled 'Ideal', is how the ideal system's performance would degrade. The second, labeled 'Non-robust', shows how a static system without graduated declustering would perform. As one can see, results are quite consistent across five trials, and in all cases perform quite closely to ideal. A best-fit line is plotted through the points. The most difficult perturbation to react to is the case of just a single perturbation, as the information must propagate to all nodes from just the single source.*

Because we are only concerned with reading from disk, we employ graduated declustering and transparently make the scan robust. No distributed queue is necessary.

We now demonstrate the performance of the non-robust and robust scans under disk performance faults. Figure 12.3 plots the performance of our robust scan under an increasing number of disk performance faults. The utilization of this fault, and all subsequent faults in this chapter, is 0.5; thus, a disk performance fault will utilize half of available disk bandwidth. Furthermore, the duration of the fault is length of the entire run.

The perturbation experiment is performed on a scan of 2.4 GB of data on 16 machines, or 150 MB of data scanned per node. A perturbation is a competing read stream on a disk that consumes half of available bandwidth, and the number of perturbations is increased along the x-axis. Results from five separate trials are plotted for each point on the x-axis. Two lines are plotted for comparison: the first, labeled 'Ideal', shows the performance of an ideal system under perturbation, and

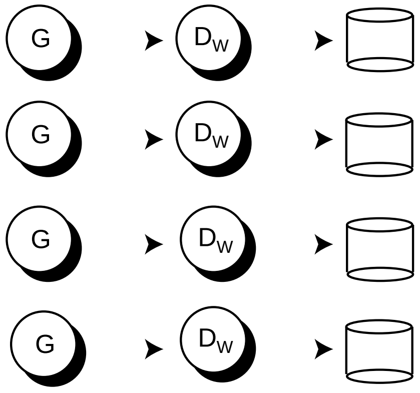


Figure 12.4: **Generate Data Flow.** *The basic data flow of the generate application is presented. Each generate module, labeled G in the diagram, generates a fixed number of records, and passes them on to the disk-write modules, labeled D_W in the diagram. The disk-write modules write the data to disk, first buffering the data into reasonably-sized chunks for the sake of efficiency.*

the second, labeled 'Non-robust', shows predicted performance of a standard scan without graduated declustering.

From the diagram, one can observe that performance of the robust scan is quite good, quite near to ideal in all cases, and consistent across all five trials. The most difficult case for the system is under one perturbation; because the data set is relatively small, graduated declustering does not have enough time to react to the single perturbation, and thus does not balance bandwidths quickly enough; the result is that overall performance is slightly lower than expected. Scans of larger data sets would not suffer from this problem.

12.2 Generate

We continue our examination with the write-analogue of a parallel scan: a parallel generation of a data set. The parallel generate creates records and writes them to disk, with no regard as to ordering between records. As with the scan, this is not an entire operator in and of itself; rather, it is more likely to form a phase of a larger program. Therefore, we seek to understand its performance properties before moving on to more complex programs.

Figure 12.4 shows the non-robust data-flow for generate. Data is generated by a set of generate modules, and then passed on to disk via disk-write modules. No communication is necessary between modules on different machines.

Figure 12.5 presents the disk-robust version of generate. By placing a distributed queue between the generate module and the disks, robustness to disk performance faults is achieved. Use of graduated declustering is not required, as we are not reading from disk.

The results of a perturbation experiment are presented in Figure 12.6. In this study, generate is run upon 16 machines, creating a total of 2.4 GB of data, or 150 MB per machine. Perturbation are induced on some of the disks, from 0, the base case, all the way up to 16, the full cluster size.

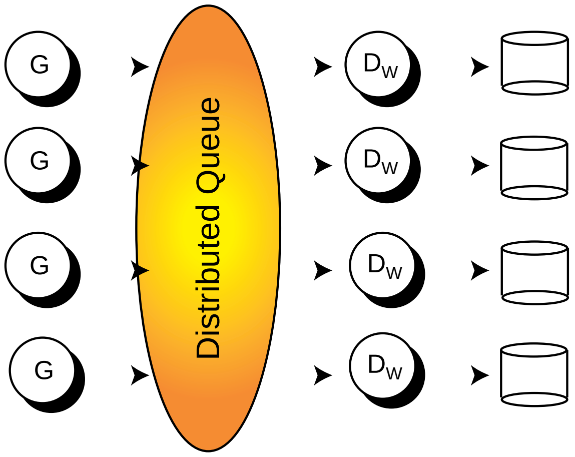


Figure 12.5: **Disk-Robust Generate Data Flow.** *The disk-robust generate is presented. Adding robustness to disk performance faults is straight-forward, and is accomplished by inserting a distributed queue between the generation of records and the disks.*

A perturbation is a competing write stream to disk, which utilizes roughly half of available disk bandwidth.

As one can see from the graph, performance is overall quite good overall, nearly ideal under all perturbation scenarios. The robust generate performs worse than the non-robust version in exactly two cases: when there are no perturbations in the system, and when the entire system is perturbed. In all other situations, the robust generate delivers full bandwidth utilization, whereas the non-robust program would run at the rate of the slow disk(s).

12.3 Filter

From the scan and generate, we have learned that the basic primitives work well when applied in isolation. We now progress to a query-processing operator that is a composition of both scan and generate, a parallel filter, also known as a relational select operator. The filter applies a user-specified function to every record that is read, and, depending on the output of the function, decides whether to discard the record or to pass it on in the data stream. In this example, we output the selected records to disk, as might be done when writing to a scratch file. Thus, during the run, reads and writes occur concurrently on the disks. Note that the filter degenerates in two special cases: when the user-specified function does not discard a single record from the input data-set, the filter becomes a data-set copy, and when all records are discarded, the filter is a scan of the data-set.

Figure 12.7 presents the standard data flow of filter. Data is read from disk by disk-read modules, labeled D_R in the diagram. Each of these modules pass data onto the filter module (F), which selects records from the data stream based on a user-specified function. From the filter modules, data flows into the disk-write modules, labeled D_W, and onto disk.

The disk-robust filter is presented in Figure 12.8. Again, we employ graduated declustering

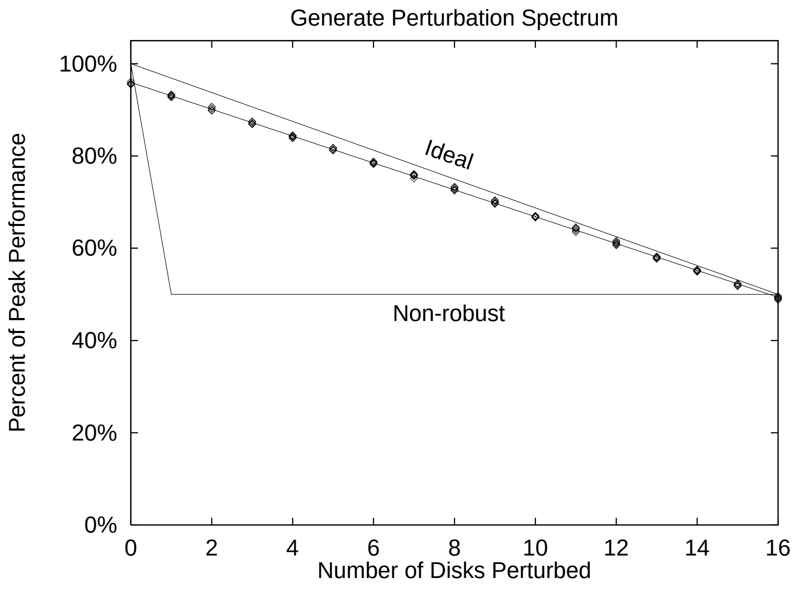


Figure 12.6: **Generate Perturbation Spectrum.** *The figure presents the behavior of the generate application under perturbations. Overall, performance is quite good, nearly ideal in all cases. The experiment is run on 16 machines, and generates 2.4 GB of data, 150 MB per machine. The number of perturbations is increased along the x-axis, from 0 to the full cluster size of 16, where a perturbation is a stream that consumes half of the peak disk bandwidth. The percent of peak performance is the figure of merit, and plotted along the y-axis. The results of five separate trials are presented, shown as points in the diagram, and are compared against two lines. The first, labeled 'Ideal', is how the ideal system's performance would degrade. The second, labeled 'Non-robust', shows how a static system without graduated declustering would perform. A best-fit line is plotted through the points.*

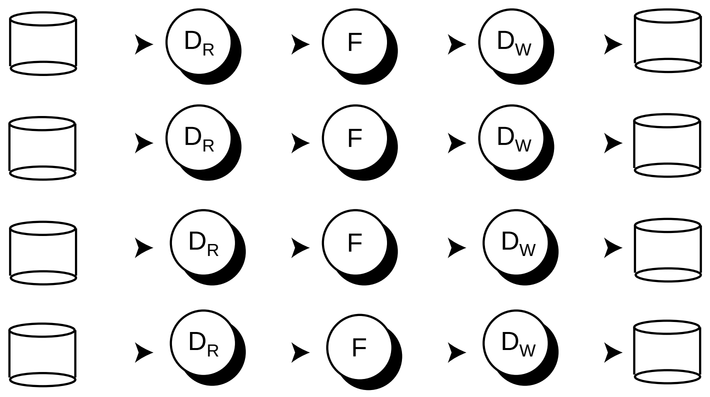


Figure 12.7: **Filter Data Flow.** *The basic filter data flow is presented. Data is read from disk in parallel by a set of disk-read modules (D_R), each of which passes its data to a filter module, labeled F in the figure. The filter module filters the data stream based on a user-specified function, and passes the selected data on to the disk-write module (D_W) for writing to disk. As was the case with the scan, this application is embarrassingly parallel.*

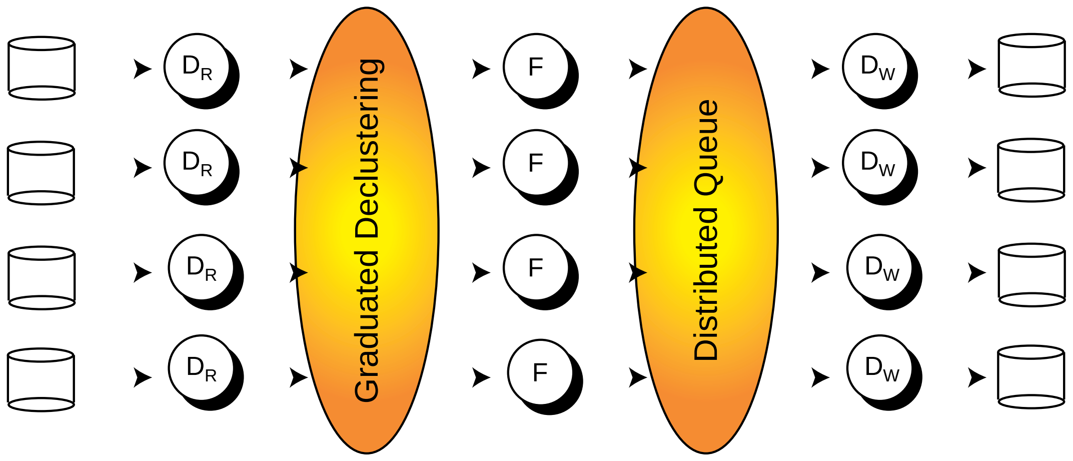


Figure 12.8: **Disk-Robust Filter Data Flow.** *The filter application is made robust to disk faults. On the read-side of the figure, graduated declustering is inserted to handle read-performance faults. One the write-side, a distributed queue is placed between the application and the disks, thus moving data around disk faults. The basic internal structure of the application remains unchanged.*

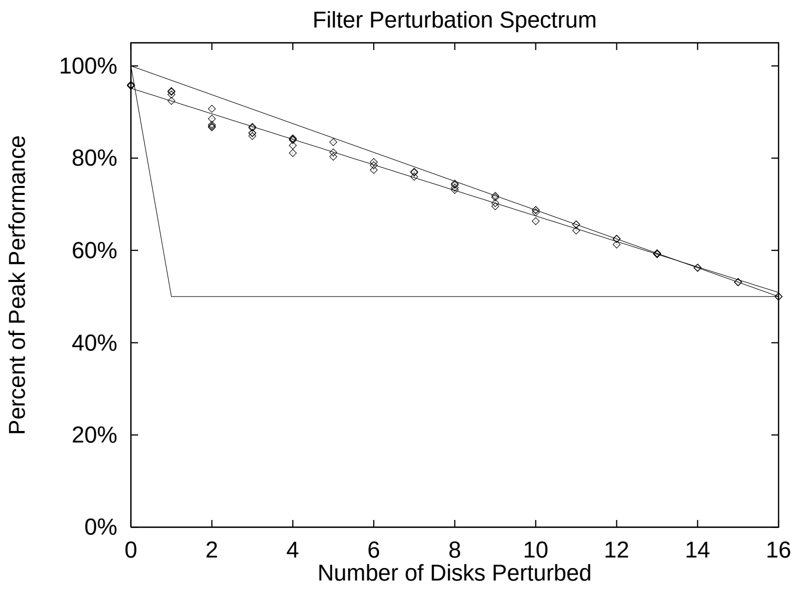


Figure 12.9: **Filter Perturbation Spectrum.** *The figure presents the behavior of the select under perturbations. Performance is quite good, and approches ideal under a large number of perturbations. The experiment is run on 16 machines, and the select is run over a 2.4 GB data set (150 MB per machine); almost all data matches the filter (over 90%), and is output to disk. The number of perturbations is increased along the x-axis, from 0 to the full cluster size of 16, where a perturbation is an output stream that consumes half of the peak disk bandwidth. The percent of peak performance is the figure of merit, and plotted along the y-axis. The results of five separate trials are presented, shown as points in the diagram, and are compared against two lines. The first, labeled 'Ideal', is how the ideal system's performance would degrade. The second, labeled 'Non-robust', shows how a static system without graduated declustering would perform. A best-fit line is plotted through the data points.*

on the read-side of the flow to ensure we can tolerate read-side disk performance faults. However, due to the addition of the write phase, we must also tolerate disk performance-faults during writes. Therefore, we place a distributed queue between the filter program and the disks. Because selection places no ordering requirements between the records of the input data set, the addition of the distributed queue does not disrupt the output format of the filter.

Figure 12.9 presents the performance of filter under the usual perturbation experiments. Once again, the selection is run upon 16 machines, each with a single disk, and we perturb 0 to 16 disks. Performance is compared against ideal performance, which utilizes all available bandwidth, and non-robust performance, which always runs at the rate of the slow machine.

Overall, performance is quite good, and near to ideal. Because the filter naturally combines concurrent reads and writes, performance as compared to ideal can approach one hundred percent, and does, when there are a high number perturbations in the system. At zero or a few perturbations,

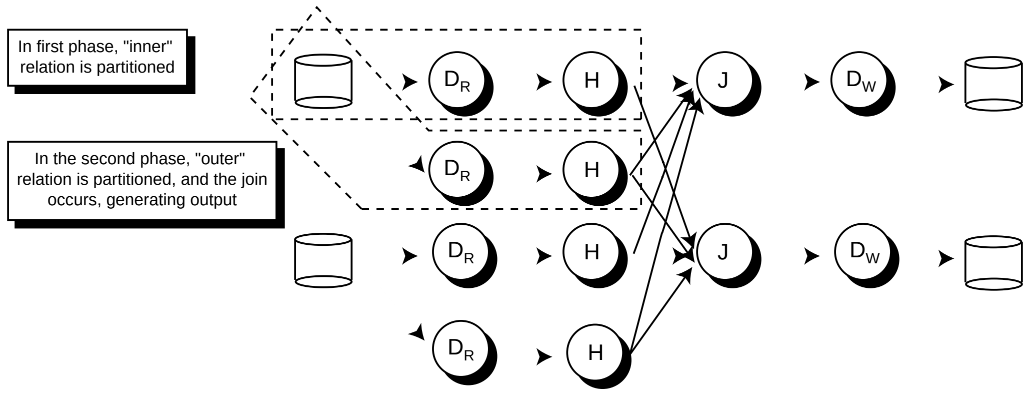

In first phase, "inner" relation is partitioned
In the second phase, "outer" relation is partitioned, and the join occurs, generating output
D_R
H
J
D_W

Figure 12.10: **Hash-Join Data Flow.** *The hash-join data flow is presented. Two phases take place to complete the flow. In the first, the building relation is read from disk via disk-read modules (D_R), and partitioned across the join modules (J) via a set of hash-partitioners (H). This is normally called the "build" phase of a hash-join. In the second phase, known as the "probe" phase, the probing relation is read from disk, partitioned via the same hash function across join modules, and then joined to the building relation. Records that match are passed on to the disk-write modules (D_W) for output to disk.*

the filter does not reach ideal, indicating some slight overhead in the system.

12.4 Join

Our next database primitive that we construct is a hash join. In general, a join of two relations outputs tuples that match on a certain key field in tuples from the two relations. The match that we are interested in is determined by equality, where the key value in one record is equal to the key value in the other record, and the join is thus known as an *equi-join*. Both one-pass and two-pass hash-join variants exist [113, 72]. In this section, we focus on the one-pass algorithm, which is suitable for use when the smaller collection fits into the aggregate cluster memory; the other can be arbitrarily large.

Figure 12.10 shows the flow of data. In the first phase, the smaller collection, often referred to as the *building* collection, because a hash table will be built over it, is read from disk, partitioned using a hash function across nodes, and internally hashed inside each join module (J) to prepare for the join phase. In the second phase, the larger *probing* collection is read from disk, and partitioned across nodes via the same hash function. As records pass into the join module, matching records from the building collection are found, and those records are output to disk. Thus, during this phase, both the read of the probing collection and write of the output will operate concurrently.

Figure 12.11 depicts the disk-robust version of the hash-join. For robustness to reads, we once again insert graduated declustering, facilitating performance-available read streams without modification to the program data-flow. The use of the distributed queue, between the join modules and the disks, is straight-forward, as the program does not need to preserve any ordering between

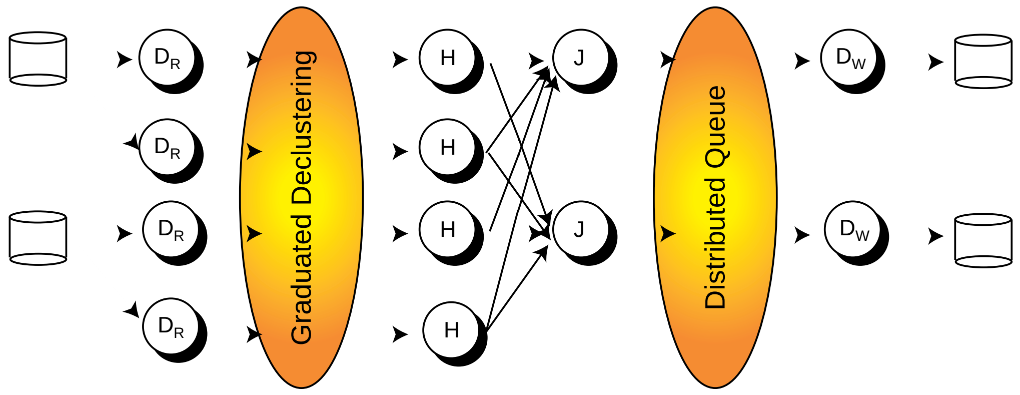


Figure 12.11: **Disk-Robust Hash-Join Data Flow.** *The disk-robust hash-join is shown in the diagram. Graduated declustering is again employed transparently to take the join from non-robust to read-robust, and a standard distributed queue is utilized to facilitate robustness to faults during the write phase of the application. Because the join specifies no ordering constraints on its output, straight-forward application of the distributed queue is made possible.*

records in the output stream.

Figure 12.12 depicts the performance of the join under perturbations. We employ the usual configuration of 16 machines, perturbing 0 to 16 of them. Perturbations consume half of disk bandwidth.

From the figure, we can see that performance of the robust join is excellent, behaving quite consistently under perturbation. Surprisingly, the transformation of a more sophisticated database primitive such as the hash-join is no more difficult than transforming the select application, because we, as the application writer, have a good understanding of the data-ordering requirements of the join.

12.5 Sort

Next we present a challenging operator, external sorting. In general, sort is a good benchmark for clustered systems because its performance is largely determined by disk, memory, and interconnect bandwidth. In this section, we only consider a one-pass sort; a two-pass sort often consists of multiple runs of a one-pass sort, and therefore also benefits from the development below. Also, following the precedent set by other researchers, we measure the performance of the sort only on key values with uniform distributions. This assumption has implications for our method of distributing keys into local buckets and across processing nodes. With a non-uniform distribution, we would need to modify our implementation to perform a sampling phase before the sort described below [19, 45]; this sampling phase could also be made robust.

Figure 12.13 presents the flow of data in the standard version of external sort, which is quite similar to the flow of NOW-Sort [9]. First, data begins as an unsorted parallel collection on a number of disks. Data is read in on each disk node via the disk read module (D_R), and then passed

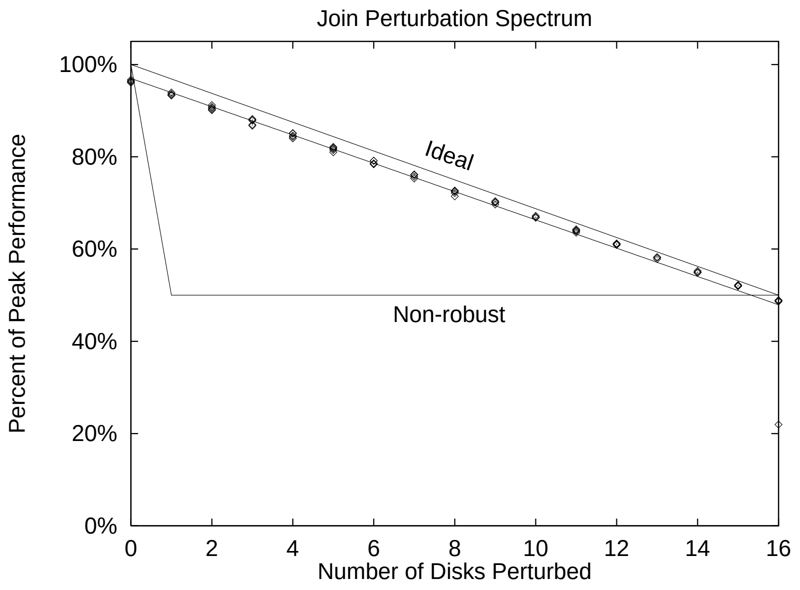


Figure 12.12: **Hash-Join Perturbation Spectrum.** *The figure presents the behavior of the hash-join under perturbations. Performance is excellent under perturbation, even for this mildly sophisticated primitive. The experiment is run on 16 machines, and the program is run over a 2.4 GB data set, or 150 MB per machine. The number of perturbations is increased along the x-axis, from 0 to the full cluster size of 16, where a perturbation is an output stream that consumes half of the peak disk bandwidth. The percent of peak performance is the figure of merit, and plotted along the y-axis. The results of five separate trials are presented, shown as points in the diagram, and are compared against two lines. The first, labeled 'Ideal', is how the ideal system's performance would degrade. The second, labeled 'Non-robust', shows how a static system without would perform. A best-fit line is plotted through the data points.*

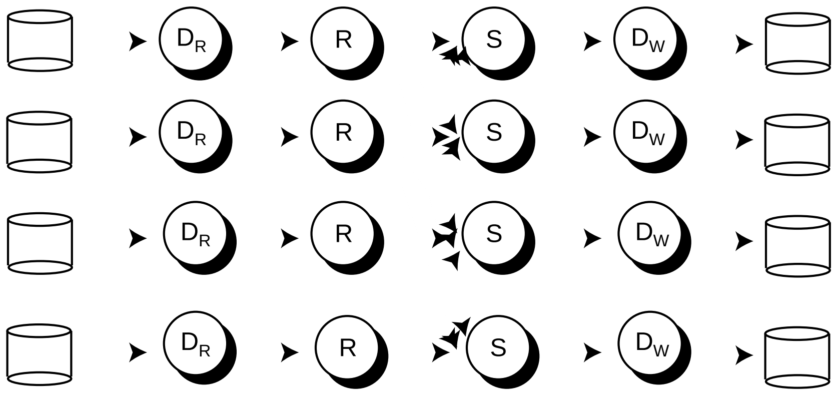


Figure 12.13: **Sort Data Flow.** *The basic sort data flow is presented. Data is read from disk in parallel by a set of disk-read modules (D_R), and then passed on to a set of range partitioners (R). These partitioners segment the data set across sort modules (S) by key value; for example, with four sort modules present, the top-fourth of the keys would be sent to sorter 0, the next fourth to sorter 1, and so forth. After the sort modules have received a large chunk of data (perhaps enough to fill memory), they each independently sort the data, and pass it to the disk-write module (D_W) for output to disk. In a multi-pass sort, this phase would repeat until all the data has been sorted into many sorted runs.*

to a range-partitioning module (R). The partitioning modules perform a key-range partitioning of the data; thus, each partitioning module reads the top few bits of each record to determine which sorter module (S) should be sent a particular record. When a sorter module has received all of its input, it sorts the data, and begins streaming it to the disk write module (D_W), which proceeds to write the data out to disk as a stream, thus preserving its order. This read-sort-write phase repeats until all of the data has been transformed into a series of sorted runs.

To enhance the sort with disk-robustness, we must utilize both graduated declustering and a distributed queue, as shown in Figure 12.14. As is the case with previous operators, we employ graduated declustering at the disk read to provide a performance-robust parallel data stream to the program.

The addition of the distributed queue is more complex. From the figure, one can observe that the queue is placed between the sort modules and the disk-write modules. If the sort modules passed sorted records to the distributed queue as in the other programs, the sort would not perform as expected, because the distributed queue algorithm would spread the records randomly across the disks, undoing all of the work of the sort! Further, the distributed queue can not be placed before the sort modules, because the key-range partitioning that occurs there is crucial to the semantics of the sort; removing the key-range partitioning would change the correctness of the sort as specified. Thus, we have placed the distributed queue in the only position possible.

For this placement to work, a slight modification must be made to the distributed queue. Instead of handing records one at a time to the distributed queue, the sort module passes large sorted

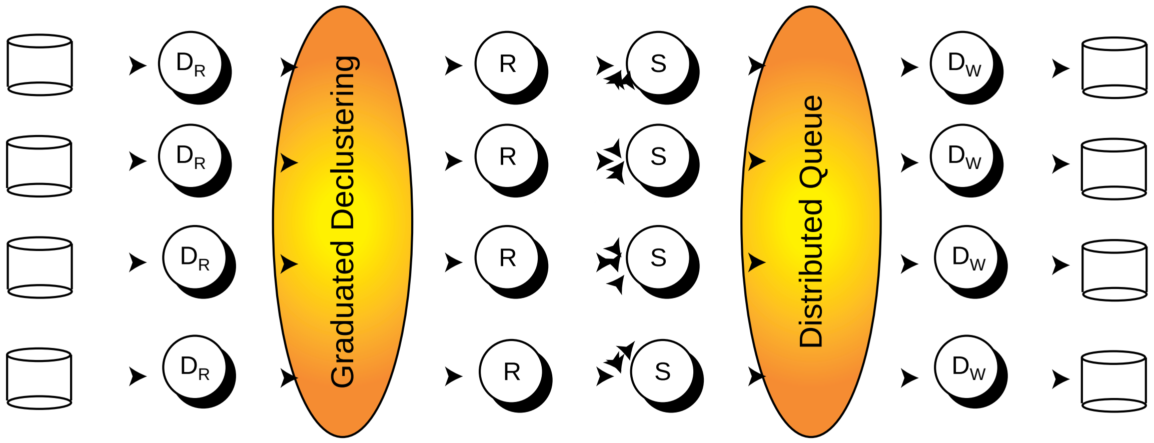


Figure 12.14: **Disk-Robust Sort Data Flow.** *To facilitate robustness to disk performance faults in the sort, we again employ both graduated declustering and a distributed queue. Graduated declustering is utilized as before, and transparently transforms the sort into a read-robust sort. The use of the distributed queue, however, is more complex. After the data has been sorted, the sorters can not place their data in the distributed queue in the standard way; if they did, the data would get randomly scattered across the disks, essentially undoing all of the work that the sorting has just performed. Instead, a slightly different distributed queue is utilized. Each sorter, instead of handing a few records to the distributed queue, instead hands the distributed queue an entire sorted run at a time. Thus, the load-balancing occurs at a much coarser granularity, while preserving the semantics of the sort.*

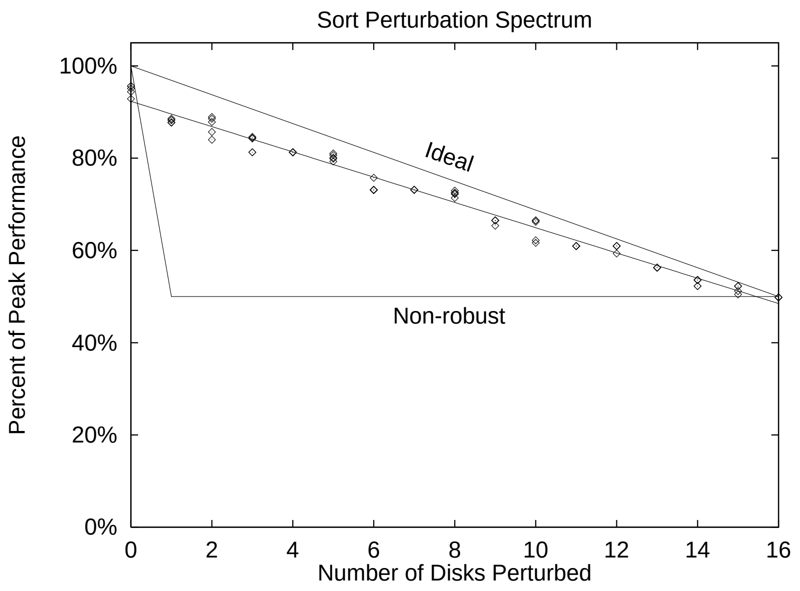


Figure 12.15: **Sort Perturbation Spectrum.** *The figure presents the behavior of the sort under perturbations. Of all programs, the performance of the sort is least stable, due to the coarse-grained load balancing that occurs in the write phase. The experiment is run on 16 machines, and the program is run over a 1.2 GB data set (75 MB per machine). The number of perturbations is increased along the x-axis, from 0 to the full cluster size of 16, where a perturbation is an output stream that consumes half of the peak disk bandwidth. The percent of peak performance is the figure of merit, and plotted along the y-axis. The results of five separate trials are presented, shown as points in the diagram, and are compared against two lines. The first, labeled 'Ideal', is how the ideal system's performance would degrade. The second, labeled 'Non-robust', shows how a static system without would perform. A best-fit line is plotted through the data points.*

chunks of data to the distributed queue [1]. The distributed queue then adapts to the rate of the disks at this much coarser granularity. For example, if each sort module received 100 MB of data to sort, it might divide this into 10 10- MB chunks.

Note that this slightly changes the form of the output of the one-pass sort; instead of an n-node sort that generates n sorted runs, we now have an n-node sort that produces $n \cdot k$ runs, where k is the number of sorted runs that the sort modules hand to the distributed queue. However, there is little performance cost to this; the only extra work that the sort must now perform is that $n \cdot k$ files must be opened and closed, instead of n with the standard sort.

Figure 12.15 presents the results of our perturbation experiment. We focus on the write phase of the sort, because the read phase is identical to other primitives in both structure and performance.

From the figure, we can see that performance under perturbation is the least stable of all of

[1] Some slight modifications had to be made to the standard distributed queue to accommodate this.

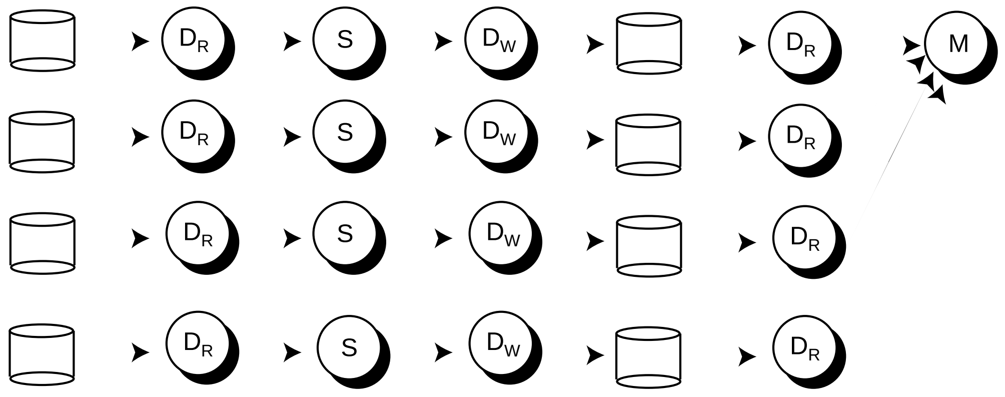


Figure 12.16: **Top-N Data Flow.** *The basic data flow of the top-N selection application is presented. Data is read from disk by the disk-read modules (D_R), and passed to sort modules (S), which buffer some amount of data before generating a sorted run and writing it to disk via the disk-write modules (D_W). Runs are generated as such until all data has been transformed into a set of sorted runs. In the final (short) phase, the runs are merged to produce the top N data items. If the user desires more data, the merge can be continued from the sorted runs.*

the programs. We attribute this directly to the coarser granularity of the load balancing across disks; with only a few 10 MB runs to balance across disks, a single slightly faster disk could end up with a noticeably larger amount of work. In general, though, performance degrades as expected, though the absolute performance is not as high as with other primitives, due to the extra work associated with managing $n \cdot k$ runs.

12.6 Top-N Selection

Finally, we present our last query-processing primitive, top-N selection. A top-N selection selects the top N data items from a collection based on a user-specified key value. N is usually a small number, such as 10. Queries of this form are common in databases and Internet search engines, which, after generating a large set of candidate results, order the results based on a quality metric and present the top few results to the user; examples of systems that generate top-N results are the AltaVista or HotBot search engines [2]

Figure 12.16 presents the basic data-flow of the top-N program. In the first phase, sorted runs are generated by reading through the entire data set, sorting a block at a time as they are read into memory, and then writing each sorted run to disk. The second phase completes the selection by merging the top few records of the sorted runs into the top-N final result.

Figure 12.17 shows the disk-robust version of top-N. Both graduated declustering and a distributed queue are utilized in order to make the first phase of the program robust. The use of

[2]Note that we do not claim that these search engines perform the top-N selection in this exact manner, just that they perform some form of top-N selection. For smaller data sets, the top-N data might reside entirely in memory, and different implementations would be appropriate.

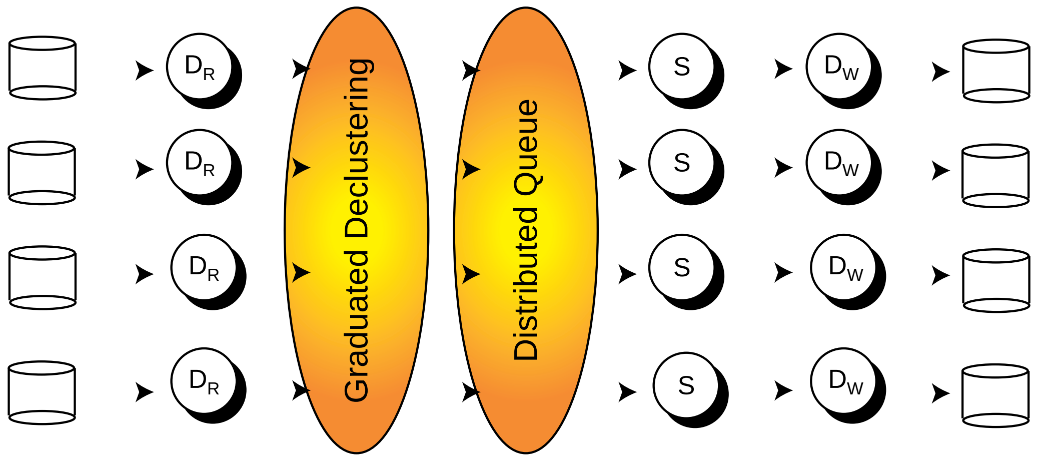


Figure 12.17: **Disk-Robust Top-N Data Flow (Phase 1 only).** *The diagram depicts the disk-robust first-phase of top-N selection. Both robust mechanisms are employed; graduated declustering allows us to tolerate read-side performance faults, and a distributed queue, placed between the disks and the sort modules, allows us to tolerate performance faults at the write-side. Note that no performance faults can be tolerated during the merge phase (not shown, as it is identical to the standard data flow); however, because that phase is quite short relative to the rest of the application, this should not be a problem.*

graduated declustering is standard, quite similar to the other applications, but the distributed queue placement is unusual.

We insert a distributed queue before the sort modules and after graduated declustering, which has the effect of tolerating write-side performance faults all the way down at the sink disks, the disks at the right side of the figure. The rate of each sorter is determined by the disk that they write sorted runs to. Slow disks do not sink sorted runs quickly, and thus sort modules that are writing to slow disks do not consume as much data as sort modules running upon faster disks. Thus, we are able to move the distributed queue up the flow from the disks all the way to the location in the diagram, taking advantage of the fact that disk performance propagates back through the system; we can thus see how important flow control is to the construction of the system. We also see how the distributed queue and graduated declustering can be plugged together to form a new “virtual” construct, that uses replication to avoid producer-side faults, and load balancing to avoid consumer-side faults.

Placing the distributed queue between the sort modules and the write modules would also have the desired effect, and initial performance results with that version are also promising. However, this placement is more demanding on the distributed queue, as it would have to balance sorted runs across the disks. In fact, we have already seen this exact placement of the distributed queue in Section 12.5 with external sorting, where we could not place the distributed queue before the sort modules due to the semantics of the sort operator. If we had done so here, our performance would match that of the sort as shown in Figure 12.15.

We also notice that there is no robustness added to the merge-phase of the program; there-

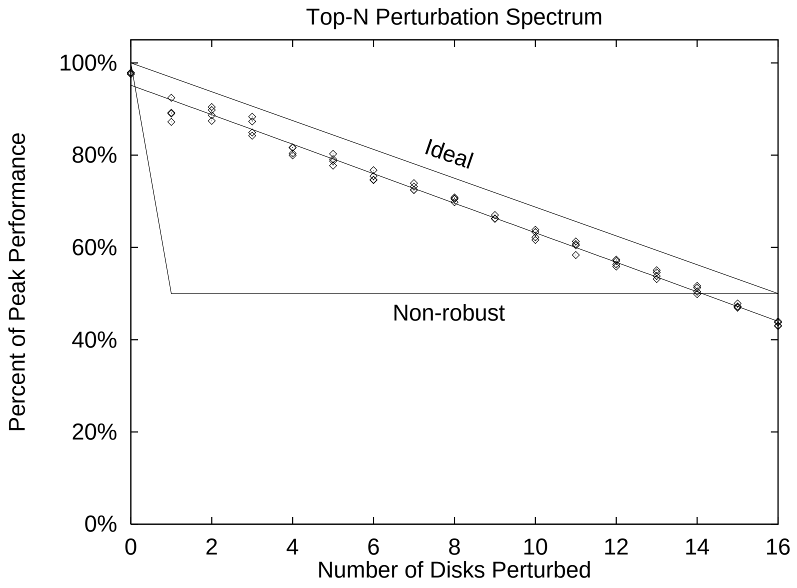


Figure 12.18: **Top-N Perturbation Spectrum.** *The figure presents the behavior of the top-N selection under perturbations. Overall, performance degrades gracefully under perturbation, and is worse than the static system only in the unperturbed and severely perturbed cases. The experiment is run on 16 machines, and the program is run over a 2.4 GB data set, or 150 MB per machine; the top 10 values are output. The number of perturbations is increased along the x-axis, from 0 to the full cluster size of 16, where a perturbation is an output stream that consumes half of the peak disk bandwidth. The percent of peak performance is the figure of merit, and plotted along the y-axis. The results of five separate trials are presented, shown as points in the diagram, and are compared against two lines. The first, labeled 'Ideal', is how the ideal system's performance would degrade. The second, labeled 'Non-robust', shows how a static system without would perform. A best-fit line is plotted through the data points.*

fore, it is not shown again in the diagram. Amdahl's law dictates this course of action, as almost no time is spent in this phase, particularly for large data sets. Thus, even if one particular disk is greatly slowed during this phase, performance will not suffer unduly. We avoid the difficulty of transforming this portion of the program with our application-specific knowledge.

Figure 12.18 plots the performance of the top-N selection under perturbation. As is standard within this chapter, the perturbations are to the disks, and the number of disks that are perturbed with a concurrent read stream is increased from 0 all the way to 16.

From the diagram, we can observe that overall performance is good, as expected, and degrades gracefully under perturbation. As compared to the other programs within this chapter, absolute performance is lower. Some of this overhead can be attributed to the use of graduated declustering, which induces extra seeks onto the disk; recall that the application normally is either reading or writing from disk, but not both concurrently. Another cost is due to time spent in opening

	Percent of Ideal Performance with N Disks Perturbed															
Primitive	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Scan	97	89	91	92	93	93	94	94	96	94	95	92	92	94	94	95
Generate	96	96	96	96	96	97	97	97	97	97	97	98	98	98	98	98
Filter	96	97	94	95	95	97	96	98	98	98	99	99	99	99	99	99
Top-n	98	92	95	95	92	94	93	93	94	92	91	92	91	91	90	89
Join	96	97	97	97	97	97	97	97	96	98	97	97	98	98	98	98
Sort	95	91	93	92	93	95	91	94	96	92	94	93	97	95	94	97

Table 12.1: **Summary of Results.** *Performance of all database primitives is presented in tabular form. Instead of plotting performance as a percent of ideal in the unperturbed case, we normalize performance versus ideal performance in each case. Thus, 100% is attainable in each column. Overall, performance is quite good, always between 89% and 99% of ideal.*

and closing each sorted run. Some additional work, perhaps in adding an asynchronous open and close, and overlapping them with other work in the system, could alleviate the latter problem, and improve performance by roughly 5% across all data points.

12.7 Summary

In this chapter, we have applied our performance-robust data-transfer mechanisms to a set of I/O-intensive database primitives. Table 12.1 presents a summary of our results. In the table, performance under all perturbations are plotted, this time in slightly different form; instead of presenting performance as the fraction of peak performance in the unperturbed case, as we have throughout the dissertation, we normalize against ideal performance. Thus, in all columns, 100% is possible. As we can see from the table, in all cases, overall unperturbed performance is excellent, as seen in the zero perturbation column, and performance under perturbation degraded gracefully, as desired. Our baseline goal of constructing performance-robust database primitives was achieved, with a cost of only 5% or less even if no perturbations occur.

We also can draw a few general observations from our experience. Overall, manual program transformation is straight forward, and the mechanisms we have provided are sufficient to tolerate disk faults, both during reads and writes. Future work should investigate tolerating a larger class of performance faults, including faults induced by erratic CPU performance and overloaded, and thus paging, virtual memory systems.

In particular, all programs that had read components could apply graduated declustering quite easily, without any modification to the program data flow. However, graduated declustering requires data set replication, which leaves open the question of how to make use of it in the middle of a program data-flow, when a recently generated scratch data set has not yet been replicated. For example, a program that writes out a scratch file and then does not immediately read it back could use the idle time to replicate the collection for performance-availability reasons.

Finally, we see that the distributed queue must be used with more care, as it requires some understanding of program semantics. The top-N selection and sort are good examples of the two extremes: with top-N selection, the distributed queue could be used quite easily to provide disk-write performance robustness, whereas with the sort, not only was some slight modification required to the distributed queue to support coarse-grained run balancing, but also the output format of the sort had to change.

Chapter 13

Conclusions

Large-scale, modern systems must provide performance availability. We have shown this to be the case for I/O-intensive applications in clustered systems, though we do believe the concepts can and should be applied more generally. Further, we have developed two mechanisms for building applications that are performance-robust: a distributed queue, which can be used to distribute work across the cluster and avoid consumer-side performance faults, and graduated declustering, which can be used to access data from replicated data sources and thus avoid producer-side performance faults. In tandem, these two constructs can be used to develop performance-robust applications. In this chapter, we will first summarize our major results, present guidelines for building performance-available systems, and then spend the last portion of the chapter speculating on future directions for this research. We then conclude the dissertation.

13.1 Summary

13.1.1 Methodology

At the methodological level, we have seen the value of studying both simulations and a prototype implementation, which was also cited as valuable in [8]. The highly-controlled and simplified simulation environment allowed us to focus purely on fundamental algorithmic issues, while avoiding many of the subtle and sometimes distracting implementation details often found in real prototypes. When measuring the behavior of the prototype implementation, results could be compared against expected simulation results; if there was a mismatch, we knew it was not from an algorithmic property, but rather likely was caused by the unexpected behavior of a software library outside of our control or undocumented hardware behavior. For example, the fairness or deadlock properties of the Myrinet switches, or the flow control of the message layer, both would have been difficult to uncover had we not had faith in our distributed algorithms. Before we had developed the simulator, performance problems in the implementation were much more difficult to track down, and the range of experiments we performed were much less broad in nature. Simulations build confidence in design, which translates to ease of implementation.

Also of value was the comparison of all experimental to results to that of the *ideal* system.

Instead of comparing performance to that of another, lesser-performing algorithm, all results can be seen versus their true potential, due to the model of ideal performance we were able to develop in Chapter 2. We believe that this style of comparison should be employed wherever possible.

13.1.2 Distributed Queues

The experiments in Chapter 9 reveal that the basic distributed queue algorithm is quite scalable and achieves the desired behavior. Crucial to its proper operation is the number of outstanding messages that are utilized. If the distributed queue is allowed too few outstanding messages, absolute performance will suffer, because the producers will not be able to keep the consumers busy. Further, when attempting to cope with performance faults, it is critical to allow each producer at least one outstanding message per consumer. If this is not the case, then the producer will not be able to track the performance of all consumers in the system, and therefore will not perform as well as expected.

One major simplification we have made is that the processing time per message block at the consumer is fixed. This simplification allows us to have a fairly straight-forward algorithm for detecting slow nodes. In a more dynamic environment, more sophisticated algorithms may be required to gauge the rate of progress of remote nodes. For example, an adaptive ramp-up similar to TCP could be employed [66].

We have also seen that the distributed queue algorithm assumes a fair network; without fairness in the network, some producers may receive preference, and thus be able to deliver their workloads sooner than other producers. When this unfairness occurs, all entities will not finish at the same time, and thus the system will run at the rate of the slower component. In the future, a more sophisticated algorithm that tracks and schedules consumption based on producer progress could provide a possible solution to this problem.

Finally, we have also seen that the distributed queue algorithm requires excess parallelism to tolerate faults; with too little work to do, perturbed consumers will become the bottleneck. A more complicated distributed queue algorithm could be more stingy about giving work to perturbed consumers, and thus potentially avoid this problem.

13.1.3 Graduated Declustering

The basic graduated declustering algorithm also works quite well, yielding high absolute performance under scale, as well as good performance under perturbation. The major difference from the distributed queue is that while the distributed queue can tolerate faults to any of the consumers, graduated declustering is more sensitive to producer-fault placement, due to the nature of how it moves data from producers to consumers.

Critical to the performance of graduated declustering is the ability of the producers to track consumer-side progress. By monitoring the progress of each remote consumer that it services, a producer is able to bias its scheduling so as to lead the consumers to progress at the same rate. Our current algorithm piggy-backs this information onto requests for data, which has the advantage of not adding any extra messages into the basic protocol.

Clearly, one important aspect of graduated declustering is the extra capacity and bandwidth costs it places on the system. Because each producer-side data set must be replicated one or more times, the question of when to replicate arises. Though not addressed in this document, a cost/benefit analysis could be applied to determine when a particular data set should be replicated. In such a system, an idle-time agent could periodically wake-up and assess current data usage patterns and frequencies, and thus begin the process of replicating those data sets.

From our experiments with producer-side scheduling, we saw the importance of producer-side scheduling. A subtle change in the scheduling algorithm of graduated declustering can lead to much poorer performance characteristics under perturbation.

Finally, one clear weakness of graduated declustering is its vulnerability to consecutive performance faults. Future work could address this deficiency by spreading the blocks of a replica across *all* of the disks of the system, instead of the entire replica residing on a single disk, similar to the layout of data blocks in Chained Declustering [64]. In that scenario, consecutive perturbations could be easily handled, but perhaps with higher overhead in the non-perturbed case, as well as significant implementation complexity.

13.1.4 Applying the Mechanisms

In Chapter 12, we learned that the process of applying the primitives to data-intensive database primitives was usually straight-forward; thus, we can empirically say that the two data-transfer primitives are “sufficient” for constructing programs that are robust to disk performance faults.

In particular, all programs that read data from disk could apply graduated declustering quite easily, without any modification to the program data-flow. However, graduated declustering requires data set replication, which leaves open the question of how to make use of it in the middle of a program flow, when a recently generated scratch data-set has not yet been replicated. Though we present no solution for this within this dissertation, a program that writes out a scratch file and then does not immediately read it back could use the idle disk time to replicate the collection for performance-availability reasons.

Finally, we see that the distributed queue must be used with more care, as it requires a thorough understanding of program semantics. The top-N selection and sort are good examples of the two extremes: with top-N selection, the distributed queue could be used quite easily to provide disk-write performance robustness, whereas with the sort, not only was some slight modification required to the distributed queue to support coarse-grained run balancing, but also the output format of the sort had to change.

13.2 Guidelines for Performance Available Systems

We now present a set of guidelines for constructing performance-robust systems. We assume that the goal is to build a system that performs consistently and with high performance. To achieve this goal, we suggest the following four guidelines be followed.

Avoid performance assumptions; use feedback. The key to performance-robust systems is to avoid performance assumptions in global operations. All operations must be based upon knowledge of current performance levels, via some form of active monitoring.

In our system, we have designed two mechanisms that are performance-assumption free: graduated declustering and distributed queue. Central to the design of both of these data-transfer primitives is the use of *feedback*. The distributed queue utilizes implicit feedback from remote consumers in the form of acknowledgments, to allocate more work to faster nodes, and thus avoid performance faults. Analogously, graduated declustering uses explicit feedback in the form of a progress metric, which allows producers to properly allocate bandwidths across consumers.

Generate excess parallelism. The properties of the workload also determines the ability of the system to react to and avoid performance faults. Some amount of excess parallelism in application workloads is necessary to tolerate performance faults in distributed systems. With excess parallelism, the majority of work can be moved to and executed by components that are working well, and the work that has been given to "slow" nodes does not dominate execution time. The experiments in Chapters 9 and 10 show the effects of a lack of parallelism on our data-transfer primitives.

In our experience, many data-intensive applications have plenty of available parallelism, and thus are amenable to performance-robust transformations. Applications that run for short periods of time or over small data-sets are not, but fortunately, it is exactly those applications that do not need to be performance robust, as dictated by Amdahl's Law [3].

Understand network behavior. In distributed systems, the primary manner of avoiding performance faults is to shuffle more work to faster nodes and proportionally less work to slower nodes; the method of distribution is the network that interconnects the components. Thus, network behavior is crucial to overall system performance.

Specifically, the message layer must not unduly limit the number of outstanding messages that distributed algorithms are allowed. In our system, both graduated declustering and the distributed queue are quite sensitive to the number of outstanding messages; if they are restricted to fewer than they need, both mechanisms will not be able to avoid performance faults successfully.

We have also seen the importance of the network switch hardware; if global performance problems such as deadlocks are likely to occur, a system such as ours will not be able to avoid the detrimental performance consequences. Thus, by design, global performance faults in the network will defeat the system.

One possible solution is to have redundant switch machinery; however, in our case, this would not have solved the problem, as the deadlock would have occurred in all available switches. Thus, only redundancy with a different brand of network, namely, one that is not likely to suffer from the exact same performance problems, would be a suitable solution; sometimes heterogeneity has its advantages.

It should be noted that localized network performance faults, such as a slow link or network interface, are essentially identical to single component failures, and therefore much easier to tolerate.

Provide spare performance and capacity. Finally, we have described how to build a system that utilizes all available bandwidth, regardless of the performance of its constituent components. However, if the applications that we are interested in utilize 100% of resources in the unperturbed case, just the slightest drop in performance of a single component leads to a noticeable, though not catastrophic, overall performance drop. Thus, some spare performance should be engineered into the system, providing "performance slack". The amount of slack is difficult to determine exactly, as it depends on the nature of perturbations, but we have found empirically that roughly 10% of machines will commonly under-perform.

Spare capacity also is required, in particular for the graduated declustering mechanism to be applied to on-disk data collections. Without spare capacity for replication, the generic producer problem can not be solved. Thus, all frequently-read data-sets should be replicated for performance reasons.

13.3 Future Work

Though we have made significant inroads towards building systems with high performance availability, much work remains to be done. We now touch on some key areas that are not addressed within other parts of this dissertation.

13.3.1 Generalized Performance Availability

The main limitation of our current work is the realm of its applicability. While we believe the mechanisms to be quite general, we have only shown their utility in I/O-intensive cluster applications, and in particular their ability to cope with disk performance faults.

Thus, research into how to build performance-available systems in other domains is called for. What extensions to our current techniques are required to enable general performance availability? As system complexity increases, we believe that more research and production computer systems will have to address such issues, or suffer from strange and unpredictable performance properties.

Though both the distributed queue and graduated declustering are good steps towards this general goal, and indeed in some cases are sufficient, we have seen cases in Chapter 12 where they do not provide the entire solution. A broader investigation into different application domains may be one way in which to begin to approach this general and difficult problem.

13.3.2 Generalized Replication

Use of replication in the current prototype system is quite limited. Only a single replica is ever used, and it is assumed that the replica is created off-line. All replicas are of on-disk data collections; no computation is ever replicated.

Many previous systems have utilized replicated computation to cope with machine faults [107]. These same mechanisms could be applied to the performance availability problem. A slow replica could be considered “dead”, and thus performance faults could be avoided, transparently to the application. However, naive replication is costly, and much work would be required in order to create lightweight mechanisms that do not tax system resources too greatly.

13.3.3 Long-term Adaptation

There has been recent file-system work extolling the virtue of “adaptive” systems [87, 112]. Most of this work has concentrated on file systems, and the adaptation therein has been *off-line*. For example, Neefe et al. propose a system that reorganizes disk blocks to improve sequential read performance in a log-structured file system.

In contrast, the focus of this dissertation has been to provide mechanisms that operate *on-line*, at run-time, to adapt to performance faults that are common within clusters. Because of the nature of performance faults, off-line mechanisms, to the first order, would be hard pressed to combat the dynamic nature of these faults.

However, some off-line adaptation could be applied to aid our system as well. One example is with graduated declustering; when should replication of data sets occur? Clearly, an off-line agent that analyzed access patterns and frequencies could be applied to determine when such optimizations should occur. Another place where this would be useful is in recording long-term behavior of particular nodes or disks. Those that consistently give bad performance could be avoided entirely.

13.3.4 Multi-workload Effects

The work presented in this dissertation has concentrated on perturbations to a single, parallel application. While this is an important case, workloads in the real world often consist of multiple, concurrently running applications. Thus, a study of multi-workload performance availability is needed.

One example of this occurs in database applications, where a single query decomposes into a query plan, usually a thin tree of database primitives, strung together to implement the query. Our current method would be to string together a series of performance-robust primitives, for example a scan feeding into a selection feeding into a sort. However, with global knowledge of the plan, as well as a good understanding of the applications, perhaps only a performance-robust scan read and sort write would be needed, as the mechanisms therein would naturally cope with any performance variations to the system.

13.3.5 Migration

Though ignored in this work, process migration [13, 47, 83, 97, 98, 120] could be utilized within our environment to provide some amount of performance availability. In the current system, process placement is presumed to be static; robustness is added to applications by facilitating dynamic movement of data to and from faster entities.

However, a system that also had the ability to move work completely off of a node could be of great utility. For example, when a node becomes unreliable in terms of performance, any remaining work could be moved to other, better-responding nodes. The difficulty with this strategy could be in getting the data off of the slow node.

Transparent process migration traditionally has been difficult to implement, for reasons enumerated in [47]. This may be why no current operating systems deliver a migration package as a part of their standard system. However, in a specific programming environment such as River, non-transparent migration may provide an easier path towards an implementation, where the user is required to program extra routines to package and un-package module state.

13.3.6 Fault Explanation

To date, we have taken the attitude that performance faults will occur, and instead of trying to understand and remove them from the system, we will instead build higher level mechanisms to cope with them. Because the number of possible explanations for performance problems is large, our approach has seemed most practical.

In some restricted situations, however, the cause of the problem could indeed be detected. For example, if a node under heavy load begins to page fault, the run-time system might wish to inform some higher-level entity of this behavior, to avoid particular workload placements. Another example, which has been observed elsewhere [119], has disks that begin to deliver extremely poor performance due to SCSI timeouts. Again, the operating system could inform our run-time layer, which could begin to move data away from this disk, in fear of a more absolute fault in the near future. In the minimal case, by reporting unexpected performance variations to a system administrator, human intervention could be applied to derive the cause of the problem and perhaps fix it at some later date.

13.3.7 A Theory of Performance Availability

Though we have made some progress along these lines within this dissertation, a more developed theory of performance availability is needed. Part of this would include more advanced, and perhaps non-linear, models of system performance under faults.

Also, it would be of value to prove that the behavior of the distributed queue and graduated declustering match specification. Though we have empirically shown that the algorithms are effective, analytical proofs are still needed.

13.3.8 Modeling Performance Faults

There is also a need to develop better models of how often performance faults occur in real systems, as well as their duration. A good example of the utility of such models can be found in the early RAID work [52], where models of disk failures are used to predict the behavior of various RAID configurations. With good models in hand, analogous analyses could be applied to judge the performance availability of systems.

The best manner in which to derive such models is to study real systems in production settings. Non-obtrusive performance meters could be added and results periodically logged.

13.3.9 Transparent Adaptation

The two core mechanisms, the distributed queue and graduated declustering, are presented to the user to incorporate into a data flow. Though graduated declustering does not require much change to the program structure, the distributed queue requires intricate understanding of program semantics; oblivious insertion will likely change program correctness.

Thus, one interesting avenue of further research would be to explore the design of adaptive cluster subsystems, which internally are performance robust, but provide an easy-to-use interface to applications. One example of this would be to design a file system or storage manager that provided performance available streams to applications, both for reading via graduated declustering, as well as for writing.

Another direction in which transparency could manifest itself is via compilation. Instead of specifying something as low-level as a data-flow graph, applications could perhaps be expressed in some higher-level format. This would allow a compiler or query optimizer to understand how to construct the application, and perhaps automate some or all of the process of transforming a standard application into something that is performance-robust.

13.4 Coda

As hardware and software systems spiral in size and complexity, systems that are designed for controlled environments will experience serious performance defects in real-world settings. This has long been realized in the area of wide-area networking, where the end-to-end argument [105] pervades the design methodology of protocol stacks such as TCP/IP. In such systems, it is clear that a globally-controlled, well-behaved environment is not attainable; therefore, applications in the system treat it as a *black box*, adjusting their behavior dynamically based on feedback from the system to achieve the best possible performance under the current circumstances.

Complexity has slowly grown beyond the point of manageability in smaller distributed systems as well. Comprised of largely autonomous, complicated, individual components, clusters exhibit many of the same properties – and hence, the same problems – of larger scale, wide-area systems. This problem is further exacerbated as clusters move towards serving as a general-purpose computational infrastructure for large organizations. As resources are pooled into a shared computing machine, with hundreds if not thousands of jobs and users present in the system, it is clearly difficult, if not impossible, to believe that the system will behave in an orderly fashion.

Thus, software programming environments for dynamic platforms such as clusters and other large-scale servers must provide mechanisms such that facilitate robust application development. This concept of *performance availability* is one of the core contributions of this dissertation.

We have found that two mechanisms in particular provide much of what is desired. The first, a distributed queue, allows applications to shuttle more work to faster machines, and thus can avoid consumer-side performance faults. The second, graduated declustering, allows applications to take advantage of replicas in order to avoid producer-side performance faults. Both of these algorithms are realized in a robust, completely distributed manner. Thus, no configuration of the cluster must occur; all adaptation occurs at run-time.

These mechanisms have been successfully applied to a small set of I/O-intensive applications. With a small amount of extra work on the part of the programmer, standard non-robust applications can be made to tolerate performance faults to the disks. Though we have chosen to concentrate only on one type of performance fault, we believe that many of the techniques we have created are quite general.

Attaining *consistent* performance for applications is easy – it can always be bad; attaining *high* performance is a matter of persistence – one good run when everything is “just right”; attaining both is the challenge for modern software systems.

Bibliography

[1] ACM SIGOPS. *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, Saint-Malo, France, October5–8 1997. ACM Press.

[2] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. Parallel randomized load balancing. In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing (STOC '95)*, pages 238–247, New York, May 1995. ACM.

[3] Gene Amdahl. Storage and I/O Parameters and System Potential. In *IEEE Computer Group Conference*, pages 371–72, June 1970.

[4] Eric A. Anderson. Eric's Name Service. http://now.cs.berkeley.edu, 1997.

[5] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.

[6] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, and Randy Y. Wang. Serverless Network File Systems. In *Fifteenth ACM Symposium on Operating Systems Principles*, pages 109–26, Copper Mountain Resort, CO, USA, Dec 1995.

[7] Remzi H. Arpaci, Andrea C. Dusseau, and Amin M. Vahdat. Towards Process Management on a Network of Workstations. Class Project, http://www.cs.berkeley.edu/ remzi/258-final, May 1995.

[8] Andrea C. Arpaci-Dusseau. *Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems*. PhD thesis, University of California, Berkeley, 1998. UCB/CSD-99-1052.

[9] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-Performance Sorting on Networks of Workstations. In *SIGMOD '97*, May 1997.

[10] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. Searching for the Sorting Record: Experiences in Tuning NOW-Sort. In *SPDT '98*, August 1998.

[11] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[12] Alexander Balfour and David H. Marwick. *Programming in standard FORTRAN 77*. Heinemann Educational Books, 1979.

[13] Ammon Barak, Ammon Shiloh, and Richard Wheeler. Flood prevention in the MOSIX load-balancing scheme. *IEEE Computer Society Technical Committee on Operating Systems Newsletter*, 3(1):23–27, Winter 1989.

[14] Tom Barclay, Robert Barnes, Jim Gray, and Prakash Sundaresan. Loading Databases Using Dataflow Parallelism. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(4):72–83, December 1994.

[15] Brian Bershad, David Black, David DeWitt, Garth Gibson, Kai Li, Larry Peterson, and Marc Snir. Operating system support for high-performance parallel I/O systems. Technical Report CCSF-40, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[16] Brian Bershad, Richard Draves, and Alessandro Forin. Using Microbenchmarks to Evaluate System Performance. In *Workshop on Workstation Operating Systems IV*, 1992.

[17] Brian N. Bershad, Stefan Savage, Emin Gun Sirer Przemyslaw Pardyak, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[18] Kenneth P. Birman and Robert Cooper. The ISIS project: Real experience with a fault-tolerant programming system. *Operating System Review*, pages 103–107, April 1991.

[19] Guy Blelloch, Charles Leiserson, and Bruce Maggs. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Symposium on Parallel Algorithms and Architectures*, July 1991.

[20] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, July 1995.

[21] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet—A Gigabet-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–38, February 1995.

[22] William J. Bolosky, Robert P. Fitzgerald, and John R. Douceur. Distributed schedule management in the Tiger video fileserver. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)* [1], pages 212–223.

[23] Haran Boral, William Alexander, Larry Clay, George Copeland, Scott Danforth, Michael Franklin, Brian Hart, Marc Smith, and Patrick Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[24] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[25] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault Tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[26] Eric A. Brewer. The Inktomi Web Search Engine. Invited Talk: 1997 SIGMOD, May 1997.

[27] Eric A. Brewer and Bradley C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. In *Proceedings of the 1994 International Parallel Processing Symposium*, Cancun, Mexico, April 1994.

[28] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, Fall 1991.

[29] Nicholas J. Carriero. *Implementation of Tuple Space*. PhD thesis, Department of Computer Science, Yale University, December 1987.

[30] Soumen Chakrabarti, Etienne Deprit, Eun-Jin Im, Jeff Jones, Arvind Krishnamurthy, Chi-Po Wen, and Katherine Yelick. Multipol: A Distributed Data Structure Library. Technical Report CSD-95-879, University of California, Berkeley, July 1995.

[31] J. Bradley Chen and Brian N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 120–133, December 1993.

[32] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[33] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[34] David D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, December 1–4 1985.

[35] George Copeland, William Alexander, Ellen Boughter, and Tom Keller. Data placement in Bubba. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 99–108, Chicago, IL, June 1988. ACM Press.

[36] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[37] David E. Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Steven Lumetta, Alan Mainwaring, Richard P. Martin, Chad Yoshikawa, and Frederick Wong. Parallel Computing on the Berkeley NOW. In *JSPP'97 (9th Joint Symposium on Parallel Processing)*, Kobe, Japan, June 1997.

[38] David E. Culler, Richard M. Karp, David A. Patterson, A. Sahay, Klaus E. Schauser, Eric Santos, R. Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 262–273, 1993.

[39] David E. Culler, Lok Tin Liu, Richard P. Martin, and Chad Owen Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, 2/1996.

[40] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. GAMMA: A high performance dataflow database machine. Technical Report TR-635, Dept. of Computer Science, Univ. of Wisconsin-Madison, March 1986.

[41] David J. DeWitt, Shahram Ghandeharizadeh, and Donovan A. Schneider. A Performance Analysis of the Gamma Database Machine. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 17(3):350–360, September 1988.

[42] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[43] David J. DeWitt and Jim Gray. Parallel database systems: The future of high-performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[44] David J. Dewitt, Jeffrey Naughton, John Shafer, and Shivakumar Venkataraman. Parsets for parallelizing OODBMS traversals: Implementation and performance. In *Third International Conference on Parallel and Distributed Information Systems*, pages 111–120. IEEE Computer Society Press, September 1994.

[45] David J. Dewitt, Jeffrey F. Naughton, and Donovan A. Schneider. Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting. In *Proceedings of the International Conference on Parallel and Distributed Information Systmes*, 1991.

[46] Peter Dibble, Michael Scott, and Carla Ellis. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[47] Fred Douglis and John K. Ousterhout. Process Migration in the Sprite Operating System. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, Berlin, West Germany, September 1987. IEEE.

[48] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[49] Susanne Englert, Jim Gray, Terrye Kocher, and Praful Shah. A benchmark of NonStop SQL Release 2 demonstrating near-linear speedup and scaleup on large databases. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 245–246, May 1990.

[50] Craig S. Freedman, Josef Burger, and David J. Dewitt. SPIFFI — a scalable parallel file system for the Intel Paragon. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1185–1200, November 1996.

[51] David Gelernter, Nicholas Carriero, S. Chandran, and Silva Chang. Parallel programming in Linda. In D. Degroot, editor, *1985 International Conference on Parallel Processing*, pages 255–263, 1985.

[52] Garth A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. An ACM Distinguished Dissertation 1991. MIT Press, 1992.

[53] Garth A. Gibson, David P. Nagle, Khalil Amiri, Fay W. Chang, Eugene Feinberg, Howard Gobioff Chen Lee, Berend Ozceri, Erik Riedel, and David Rochberg. A case for network-attached secure disks. Technical Report CMU–CS-96-142, Carnegie-Mellon University, June 1996.

[54] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[55] Goetz Graefe. Volcano: An extensivle and parallel dataflow query processing system. Technical report, Oregon Graudate Center, June 1989.

[56] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):102–111, June 1990.

[57] Goetz Graefe. Iterators, Schedulers, and Distributed-memory Parallelism. *Software - Practice and Experience*, 26(4):427–52, April 1996.

[58] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: A Call Graph Execution Profiler. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 120–126, June 1982.

[59] Jim Gray. What Happens When Processors Are Infinitely Fast And Storage Is Free? Invited Talk: 1997 IOPADS, November 1997.

[60] Jim Gray. The Official External Sorting Benchmark Home Page. http://research.microsoft.com/barc/SortBenchmark/, 1999.

[61] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[62] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Draft, January 1993.

[63] Samuel J. Horowitz, Daniel I. Amey, and Jay P. Page. Design Trade-offs of MCM-C vs. Ball Grid Array on Printed Wiring Board. In *1996 International Conference on Multichip Modules*, Denver, CO, April 1996.

[64] Hui-I Hsiao and David DeWitt. Chained Declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of 6th International Data Engineering Conference*, pages 456–465, 1990.

[65] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995. ACM Press.

[66] Van Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM '88*, pages 314–329, August 1988.

[67] Theodore Johnson. Designing a Distributed Queue. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 304–11, San Antonio, Texas, October 1995.

[68] Vijay Karamcheti and Andrew A. Chien. A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 298–307, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News*, 23(2), May 1994.

[69] Randy H. Katz, Garth A. Gibson, and David A. Patterson. Disk system architectures for high performance computing. *Proceedings of the IEEE*, 77(12):1842–1858, December 1989.

[70] Lawrence J. Kenah and Simon F. Bate. *Vax/VMS Internals and Data Structures*. Digital Press, Bedford, 1984.

[71] Michelle Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[72] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. GRACE: Relational algebra machine based on hash and sort — its design concepts. *Journal of the Information Processing Society of Japan*, 6(3):148–155, 1983.

[73] Steve Kleiman and Joe Eykholt. Interrupts as Threads. *Operating Systems Review*, 29(2):21–26, April 1995.

[74] Steve Kleiman, Jim Voll, Joe Eykholt, Anil Shivalingiah, Dock Williams, Mark Smith, Steve Barton, and Glenn Skinner. Symmetric Multiprocessing in Solaris 2.0. In *Proceedings of COMPCON Spring '92*, 1992.

[75] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[76] Nathaniel A. Kushman. Performance Nonmonotonocities: A Case Study of the UltraSPARC Processor. Master's thesis, Massachussets Institute of Technology, Boston, MA, 1998.

[77] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[78] Butler W. Lampson. Hints for computer system design. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 33–48. ACM, December 1983.

[79] Edward K. Lee and Michael Burrows. Design Considerations for Parallel File Systems. Unpublished Manuscript, January 1995.

[80] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, October 1996.

[81] Bruce Lindsey. SMP Intra-Query Parallelism in DB2 UDB. Database Seminar at U.C. Berkeley, February 1998.

[82] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3), March 1988.

[83] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor – a hunter of idle workstations. In *Proceedings of ACM Computer Network Performance Symposium*, pages 104–111, June 1988.

[84] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Technical Conference*, pages 291–305, 1993.

[85] Alan Mainwaring and David Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. Technical Report CSD-96-918, University of California at Berkeley, October 1996.

[86] Keith Marzullo, Robert Cooper, Mark Wood, and Kenneth Birman. Tools for Distributed Application Management. *IEEE Computer*, pages 42–51, August 1991.

[87] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)* [1], pages 238–251.

[88] Rodney Van Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the 1997 USENIX Conference*, January 1997.

[89] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[90] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[91] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)* [1], pages 276–287.

[92] John K. Ousterhout. Tcl: An Embedable Command Language. In *Proceedings of the 1990 USENIX Association Winter Conference*, 1990.

[93] John K. Ousterhout. An X11 Toolkit Based on the Tcl Language. In *Proceedings of the 1991 USENIX Association Winter Conference*, pages 105–115, Dallas, Texas, December 1991.

[94] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA*. ACM Press and IEEE Computer Society Press, 1995.

[95] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 17(3):109–116, September 1988.

[96] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, CO, December 1995. ACM Press.

[97] Gerald Popek and Bruce J. Walker, editors. *The LOCUS Distributed System Architecture*, pages 73–89. Computer Systems Series. The MIT Press, 1985.

[98] Michael L. Powell and Barton P. Miller. Process Migration in DEMOS/MP. In *Proceedings of the 9th Symposium on Operating System Principles*, pages 110–119, October 1983.

[99] Ram Raghavan and John Hayes. Scalar-Vector Memory Interference in Vector Computers. In *Proceedings of the 1991 International Conference on Parallel Processing, Volume 1*, pages 180–187, St. Charles, IL, August 1991.

[100] Keith H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Massachussets Institute of Technology, Boston, MA, June 1998.

[101] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Comm. Assoc. Comp. Mach.*, 17(7):365–375, July 1974.

[102] Luis Rivera and Andrew Chien. A High Speed Disk-to-Disk Sort on a Windows NT Cluster Running HPVM. Submitted for pulication, 1999.

[103] Olis Rubin. *The Design of Automatic Control Systems*. Artech House, Norwood, MA, 1996.

[104] Rafael H. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*. PhD thesis, U.C. Berkeley, Computer Science Division, February 1992.

[105] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, pages 277–288, November 1984.

[106] Stefan Savage and John Wilkes. AFRAID— a frequently redundant array of independent disks. In *Proceedings of the 1996 USENIX Technical Conference*, pages 27–39, January 1996.

[107] Fred B. Schneider. Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[108] Fred B. Schneider. Personal Communication, February 1999.

[109] Anne P. Scott, Kevin P. Burkhart, Ashok Kumar, Richard M. Blumberg, and Gregory L. Ranson. Four-way Superscalar PA-RISC Processors. *Hewlett-Packard Journal*, 48(4):8–15, August 1997.

[110] Kent E. Seamons and Marianne Winslett. Multidimensional array I/O in Panda 1.0. *Journal of Supercomputing*, 10(2):191–211, 1996.

[111] Margo I. Seltzer, Yasuhiro Endo, Christoper Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[112] Margo I. Seltzer and Christopher Small. Self-Monitoring and Self-Adapting Systems. In *Proceedings of the 1997 Workshop on Hot Topics on Operating Systems*, Chatham, MA, May 1997.

[113] Leonard D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239–264, September 1986.

[114] Daniel P. Siewiorek and Robert S. Swarz. *Reliable Computer Systems - Design and Evaluation*. Digital Press, Burlington, MA, USA, 1982.

[115] Keith Smith and Margo I. Seltzer. File System Aging. In *Proceedings of the 1997 Sigmetrics Conference*, Seattle, WA, June 1997.

[116] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[117] Michael Stonebraker, Jolly Chen, Nobuko Nathan, Caroline Paxson, and Jiang Wu. Tioga:providing data management support for scientific visualization applications. In *International Conference On Very Large Data Bases (VLDB '93)*, pages 25–38, San Francisco, Ca., USA, August 1993. Morgan Kaufmann Publishers, Inc.

[118] Nisha Talagala, Satoshi Asami, David Patterson, Dakin Hart, and Bob Futernick. The Art of Massive Storage: A Case Study of a Web Archive. Submitted for Publication, 1999.

[119] Nisha Talagala and David Patterson. An Analysis of Error Behaviour in a Large Storage System. In *IPPS Workshop on Fault Tolerance in Parallel and Distributed Systems*, 1999.

[120] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *Proceedings of the 10th Symposium on Operating System Principles*, pages 2–12, December 1985.

[121] William M. Thorburn. Occam's Razor. *Mind*, 24:287–88, 1915.

[122] Marc Tremblay, Dale Greenley, and Kevin Normoyle. The Design of the Microarchitecture of UltraSPARC-I. *Proceedings of the IEEE*, 83(12):1653–63, December 1995.

[123] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 40–53, December 1995.

[124] Thorsten von Eicken, David E. Culler, Sech Copen Goldstein, and Klaus Eric Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News*, 20(2), May 1992.

[125] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov 1994.

[126] Chih-Po Wen. *Portable Library Support for Irregular Applications*. PhD thesis, University of California, Berkeley, January 1996. Techreport UCB/CSD-96-894.

[127] Szu wen Kuo, Marianne Winslett, Yong Cho, Jonghyun Lee, and Ying Chen. Efficient input and output for scientific simulations. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 33–44, Atlanta, GA, May 1999. ACM Press.

[128] Richard Wolniewicz and Goetz Graefe. Algebraic Optimization of Computations over Scientific Databases. In *VLDB '93*, pages 13–24, 1993.