Run-Time Adaptation in River

REMZI H. ARPACI-DUSSEAU
University of Wisconsin—Madison

We present the design, implementation, and evaluation of run-time adaptation within the River dataflow programming environment. The goal of the River system is to provide adaptive mechanisms that allow database query-processing applications to cope with performance variations that are common in cluster platforms. We describe the system and its basic mechanisms, and carefully evaluate those mechanisms and their effectiveness. In our analysis, we answer four previously unanswered and important questions. Are the core run-time adaptive mechanisms effective, especially as compared to the ideal? What are the keys to making them work well? Can applications easily use these primitives? And finally, are there situations in which run-time adaptation is not sufficient? In performing our study, we utilize a three-pronged approach, comparing results from idealized models of system behavior, targeted simulations, and a prototype implementation. As well as providing insight on the positives and negatives of run-time adaptation both specifically in River and in a broader context, we also comment on the interplay of modeling, simulation, and implementation in system design.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; H.2.4 [**Database Management**]: Systems—*parallel databases*

General Terms: Design, Experimentation, Measurement, Performance, Reliability

Additional Key Words and Phrases: Performance availability, performance faults, run-time adaptation, parallel I/O, clusters, robust performance

1. INTRODUCTION

One of the most successful application domains to be mapped to parallel machines is the realm of database query-processing, which has resulted not only in many successful research projects [Barclay et al. 1994; Boral et al. 1990; DeWitt et al. 1986; Lorie et al. 1989] but also in commercially viable products from industry [Baru et al. 1995; Tandem Performance Group 1988; Teradata Corporation 1985]. Much of this success can be attributed to the *relational model*; by describing data "with its natural structure only" [Codd 1970], the relational model affords much flexibility in implementation, which, as DeWitt and Gray [1992] state, is "ideally suited to parallel execution."

This work was funded over the years by DARPA F30602-95-C-0014, DARPA N00600-93-C-2481, NSF CDA 94-01156, NASA FDNAGW-5198, the California State MICRO Program, and NSF CCR-0092840.
Author's address: R. H. Arpaci-Dusseau, Computer Sciences Department, University of Wisconsin, Madison, 1210 W. Dayton St., Madison, WI 53706; email: remzi@cs.wisc.edu.

In the past, these parallel database systems were often tailored to run on specialized parallel hardware platforms. However, recent changes in networking technology have transformed commodity clusters of workstations (sometimes called networks of workstations, or NOWs) into a viable platform for tightly coupled parallel applications. In particular, the advent of switch-based, low-latency, high-throughput networks has enabled the deployment of new classes of applications and services on modern clustered systems. Clusters have a number of built-in advantages over specialized parallel machines, including higher performance and lower cost: higher performance because they incorporate the most recent microprocessors and thus better track Moore's law, and lower cost due to the economies of scale of mass production [Anderson et al. 1995].

However, although clusters provide an excellent alternative to specialized parallel machines, they also introduce a range of new problems for system designers. Much of this difficulty arises from the complexity of modern computer systems; the basic building blocks of clustered systems, including processors, networks, disks, and associated software, are becoming increasingly sophisticated. For example, current top-of-the-line microprocessors have tens of millions of transistors [AMD 2000], and operating systems typically contain millions of lines of code.

Increasing component complexity directly affects component behavior; in particular, identical complex components will often not behave identically [Arpaci-Dusseau and Arpaci-Dusseau 2001]. For example, two otherwise identical disks, made by the same manufacturer, receiving the same input stream, will not necessarily deliver the same performance. This unexpected *dynamic performance heterogeneity* can arise from a number of different factors, including the fault-masking capabilities of modern SCSI disk drives: by remapping bad blocks, bad sectors can be hidden from higher levels of the system, but not without altering the performance of the drive. Disks are not the only purveyor of such heterogeneity; similar behavior has been observed in CPUs [Bressoud and Schneider 1995; Kushman 1998], networks [Arpaci-Dusseau 1999], memory systems [Raghavan and Hayes 1991], and even software systems [Chen and Bershad 1993; Gribble et al. 2000].

Performance heterogeneity becomes particularly difficult to overcome when mixed with parallelism, due to the *performance assumptions* often made by parallel algorithms. In particular, many previous systems have made the simplifying assumption that all components of a system will operate at the same rate at all times; such assumptions are common in parallel database systems, which have traditionally used static data distribution schemes such as range-partitions or hash-partitions to move data or assign work across nodes of the system [DeWitt et al. 1986; Graefe 1990]. With such strong performance assumptions, all global operations perform at the rate of the slowest member of the group, thus decreasing the performance and robustness of the system.

Rather than attempting to prevent performance heterogeneity from occurring, *River* is a parallel I/O programming environment that takes these variations, or "performance faults," into account as an inherent design consideration [Arpaci-Dusseau et al. 1999]. River provides a basic dataflow programming environment and I/O substrate for clusters, with the goal of enabling common-case

robust performance in the face of arbitrary and unforeseen performance faults in components. The primary focus of River is to provide the necessary support for parallel database query-processing primitives; however, we believe the dataflow programming environment to be fairly general, and that a broader class of applications could benefit from the River infrastructure.

The key to delivering robust system performance in River is *run-time adaptation*. River provides adaptive data-movement mechanisms that continually gauge and react to the performance of components within the system, thus avoiding performance assumptions by design. Specifically, River provides two core adaptive mechanisms: a *distributed queue* (DQ) balances data flowing across consumers of the system, and *graduated declustering* (GD) dynamically adjusts the flow of data generated by producers. Both of these constructs are designed to take advantage of the performance characteristics of modern high-speed networks, by moving data to the location where they are best processed. River applications can utilize the DQ and GD in tandem to deliver consistent high performance in spite of unanticipated performance variations.

In this article, we present the design, implementation, and evaluation of the second-generation River system, known as *Ganges*. We make some major contributions in comparison with previous work. First, we describe the Ganges prototype in some detail, which incorporates lessons learned from both the implementation and evaluation of the first prototype, *Euphrates*.

Second and perhaps more importantly, we provide a detailed study of the core River run-time adaptive mechanisms. Although initial results presented in the IOPADS workshop were promising [Arpaci-Dusseau et al. 1999], they were also limited. In particular, four important questions were left unanswered: How well do the core run-time adaptive mechanisms of River operate, especially as compared to the ideal? What are the keys to making them work well? Can applications readily apply the mechanisms to fashion robust applications? Finally, what are the limitations of the mechanisms? Without answering these questions, little could be said about the generality or efficacy of the River approach, or indeed the adequacy of run-time adaptation as a solution to the performance-fault problem.

We answer these questions in this article via a three-pronged analysis. We incorporate both modeling and simulation into our study, in addition to extending previous implementation work. With models, we are able to quantify ideal performance and, thus, via comparison with the ideal, better understand how well the core mechanisms truly operate. With simulations, we can study the River distributed algorithms in depth, without worry or interference from the many difficulties common in implementation work. With implementation, we bring forth issues not likely to arise in all but the most accurate of simulations, and validate our simulated performance expectations. Through this approach, we enhance the Ganges prototype, improving the performance of River mechanisms by roughly a factor of three in some scenarios and by small factors in almost all others. More important, we greatly enhance our understanding of the software adaptation technology that is central to River, and thus feel more confident that the core mechanisms of River are robust.

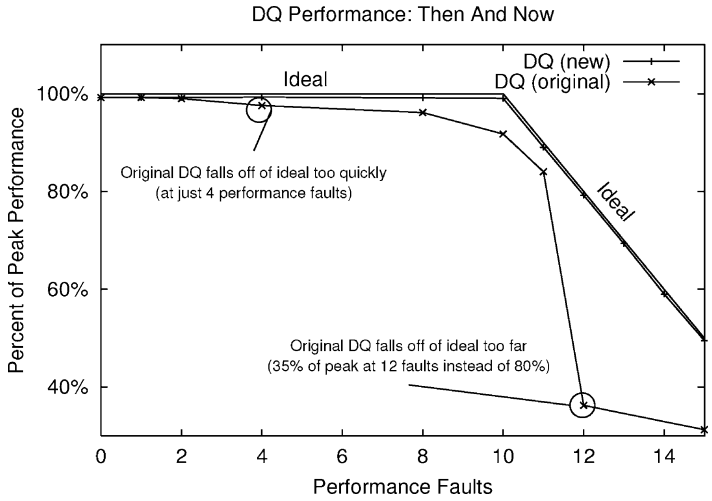

Fig. 1. **DQ performance, then and now:** a comparison of the old DQ implementation versus the new. The number of performance faults is increased along the x-axis, and the percentage of peak performance is plotted along the y-axis. Results from the older version of the DQ are compared against the new and improved algorithm.

An example of the improvement gained via our broader evaluation technique is previewed in Figure 1. Therein, we plot the performance of two versions of the distributed queue under performance faults, and compare both to our ideal model. The line labeled "DQ (original)" plots data taken directly from the original paper; as one can see, the performance of the Euphrates implementation does not track the ideal, falling off sharply under a high number of performance faults. With an ideal model in place (described in Section 4), it is easy to see where improvement is necessary. The new line "DQ (new)" presents the Ganges implementation of the DQ algorithm, which tracks the ideal almost perfectly. The cause of the previous fall-off was the interplay of the DQ algorithm and message-layer flow control, which we were able to study carefully via simulation (see Section 6.1), resulting in a much-improved distributed queue implementation.

We now summarize our major findings.

—*How well do the core run-time adaptive mechanisms of River operate, especially as compared to the ideal?* We find that in most cases, the refined distributed queue and graduated declustering algorithms operate quite well, gracefully delivering nearly ideal performance under a number of perturbation scenarios. We also quantify the limitations of GD due to limited replication in both best-case and worst-case situations; previous work only measured best-case scenarios.

—*What are the keys to making them work well?* We find that there are several keys to the effective operation of River. First, in order for these run-time adaptive primitives to function as desired, the behavior of the communication layer and network hardware is critical; in particular, the DQ and GD must have control over how flow control is implemented in the communication

layer in order to perform well. Second, we find that the River mechanisms require a high degree of parallelism at the application level; without excess parallelism, the current River mechanisms cannot avoid performance faults successfully. Fortunately, many data-intensive applications can achieve high degrees of parallelism quite readily. Third, we find that local data-processing decisions must be guided by global knowledge of progress; the main difficulty in doing so lies in obtaining global information in a distributed and scalable manner. Finally, in designing a River hardware platform, we recommend that some amount of slack should be "engineered" into it, in order to enable consistent high performance in spite of performance fluctuations.

—*Can applications readily apply the mechanisms to fashion robust applications?* This question is the most difficult to fully answer, but in implementing a number of database query-processing operators on top of River, we find that these programs can employ the distributed queue, graduated declustering, or both to create robust dataflow. However, we find that some applications are not as suitable to such transformations, for one of two fundamental reasons: multiphase applications cannot automatically utilize GD in later stages, as they must first pay the cost of data replication; also, a many-to-one dataflow cannot easily be transformed into a robust counterpart with a DQ.

—*Finally, what are the limitations of the mechanisms?* In our study, we describe certain cases where the localized run-time adaptation of River is not sufficient. First, although the River adaptive mechanisms can tolerate *local* performance faults in hardware and software, *global* performance fluctuations in the network switches are difficult if not impossible to avoid; the network backplane provides the primary avenue for adaptation, and must work well for the system to behave robustly. Second, we find that in some cases, run-time adaptation is too short-sighted and memoryless; we speculate that long-term adaptation is needed. Note that both of these problems are general to run-time adaptive systems, and are not just specific to River.

The rest of this article is organized as follows. Section 2 highlights the River system motivation and design, and Section 3 describes the Ganges implementation. In Section 4, we develop a model of expected ideal performance and describe our experimental environment. Results are presented as answers to each of the four questions outlined above, in Sections 5, 6, 7, and 8, respectively. In Section 9 we discuss related work, and in Section 10, we conclude.

2. THE RIVER SYSTEM: MOTIVATION AND DESIGN

The goal of the River I/O programming environment is to enable the construction of applications that exhibit *performance availability*, that is, to provide mechanisms that allow data-intensive applications to adapt at run-time to performance fluctuations, and thereby make high performance consistently "available" to end-users. As stated above, our primary focus is upon database query-processing primitives, although we believe that a broader class of applications could be programmed within the River environment. In this section, we briefly describe the design of the River system in more detail, including motivation for

the system, and a description of its two core adaptive mechanisms. We conclude the section with a discussion of technologies that the River design relies upon in order to provide a flexible and robust programming substrate, and a discussion of limitations of the environment.

2.1 Motivation

Much of the previous work in the field of distributed and parallel systems has addressed the design of large-scale systems that can tolerate *correctness faults* in individual components [Birman and Cooper 1991; Borg et al. 1989; Englert et al. 1990; Liskov 1988; Schneider 1990]. The notion behind such work is that distributed systems consist of multiple hardware and software components that periodically fail; a system that works continuously on top of such unreliable components must operate in spite of such failures. For example, RAID storage can tolerate the failure of a disk and continue correct operation [Patterson et al. 1988a].

Less understood is the notion of how the system functions when one or more components does not *perform* as expected. We refer to the unexpected low performance of a component as a *performance fault*. Within clusters, disk drives are often the main source of such performance variation, especially within the scope of data-intensive applications. Some of the reasons why disks in a clustered system exhibit static and dynamic performance faults within a single disk and across disks include: the presence of multiple zones [Meter 1997], SCSI bad-block remapping [Arpaci-Dusseau 1999] and thermal recalibrations [Bolosky et al. 1996], sporadic performance before absolute failure [Talagala and Patterson 1999], contention due to workload imbalance, and structural heterogeneity due to incremental growth [Brewer 1997]. As we have documented elsewhere [Arpaci-Dusseau and Arpaci-Dusseau 2001], many other hardware and even software components can exhibit unexpected performance variations; worse, the faults tend to occur only upon a subset of the components of the system, and when they do occur, they tend to be long-lived.

Current systems that support data-intensive applications do not interact well with performance faults; many of these systems are built using static techniques for exploiting parallelism and allocating data. For example, standard striping algorithms for distributing data requests over a set of disks place the same amount of data on each disk. As identified above, the problem with static schemes is that they make rigid *performance assumptions* about the relative performance of different components—often that all perform identically. As a result, when just a small number of components do not deliver peak performance, the performance of the entire service will be reduced to that of the few slow entities.

Performance availability states that the performance of the entire system should track the aggregate performance of all components in the system, degrading gracefully under performance faults. We believe that systems must provide the proper run-time adaptive primitives to support performance availability, and thus enable the delivery of excellent sustained performance in spite of localized component performance failures.

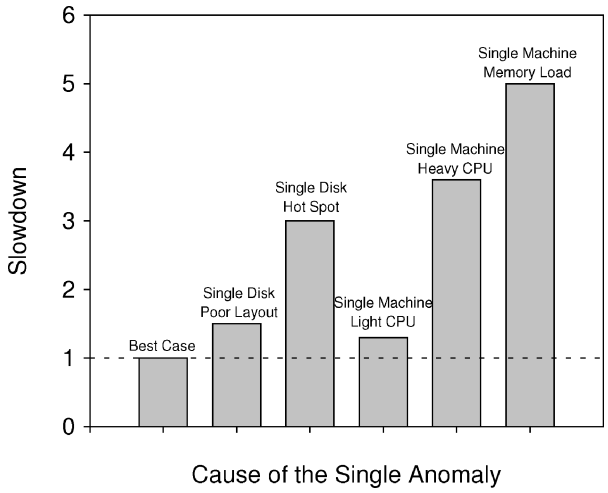

Fig. 2. **NOW-Sort under perturbance:** the best-case performance of NOW-Sort, versus its performance under slight disk, CPU, and memory perturbations. All performance results are relative to an 8-node run of NOW-Sort, which delivers data at a near-peak disk rate of 40 MB/s throughout the run.

To better motivate the global performance problems induced by local performance faults, we report the results of a simple experiment with NOW-Sort [Arpaci-Dusseau et al. 1997], a high-performance parallel external sort for clusters. In the experiment, the sort runs on eight machines, and in each run, we perform a slight perturbation of the sort on just one of those machines. The results from these perturbation experiments are shown in Figure 2. As we can see from the graph, each of the perturbations on just a single machine has a serious global performance effect. If a single file on a single machine has poor layout (inner tracks versus outer), overall performance drops by 50%; when a single disk is a "hot spot," and has a competing data stream, performance drops by a factor of three; CPU loads on any of the machines decrease performance proportional to the amount of CPU they steal; when an excess memory load causes a machine to begin thrashing, a factor of five in performance is lost.

Although it may be possible to build a system that avoids all of these situations by balancing load across the system perfectly at all times and meticulously managing all resources of the system, we believe it is difficult to do so. As system size and complexity increase, carefully managing such a system becomes quite challenging if not impossible. Therefore, we approach the problem in a different manner, by assuming the presence of such "performance faults," and providing a substrate that enables applications to operate well in spite of them.

2.2 The River Environment

River provides a generic dataflow programming environment for clusters of workstations, quite similar in basic design to previous parallel database environments such as Gamma and Volcano [DeWitt et al. 1986; Graefe 1990]. Applications are constructed in a piecewise fashion from one or more *modules*.

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

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

Fig. 3. **Module API**: a simple River module. The module `Get()`s messages from upstream, performs some operation on them by calling a user-defined `Filter()`, and then (conditionally) `Put()`s messages downstream.

```
// 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();
```

Fig. 4. **Flow API**: a simple reader-to-writer flow. The user calls `Place()` to add a module node to the dataflow graph, and `Attach()` to connect two modules. Finally, by calling `Go()`, the program begins to run. In this example, 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`’’.

Each module has a logical thread of control and at least one input or output channel. Inside a module, `Get()` can be called to obtain data from an upstream source, and `Put()` can be called to pass data downstream. To begin execution of an application, a control program constructs a *flow*, which connects the desired modules from sources to sinks. Once instantiated, the computation begins and continues until all data have been processed. Examples of a module and a flow are given in Figures 3 and 4.

To enable applications to cope with performance faults and to still deliver consistent high performance, River provides two distributed software constructs, a distributed queue (DQ) and graduated declustering (GD). Both dynamically adjust how data are transferred from a set of producer modules (e.g., processes of a parallel application) to a set of consumer modules (e.g., a set of disks) at run-time; the overall goal is to tolerate performance faults during such a transfer in producers, consumers, or both. The two constructs in River each solve one problem—the DQ can cope with performance-faulty consumers, and GD with performance-faulty producers—and in tandem can be used in a flow to build performance-robust applications.

By design, River does not explicitly provide mechanisms to deal with absolute failure. To handle absolute failure, applications must either be restarted or take advantage of a checkpointing package [Barclay et al. 1994; Litzkow et al. 1997]. We now discuss the desired behavior of the distributed queue and graduated declustering in more detail.

2.3 The Distributed Queue

When transferring data from a set of producers to a set of consumers, one or more consumers might suffer from a performance fault. The proper reaction in such a case is to move more data to other consumers, proportional to the relative bandwidths between faster and slower ones. This functionality is provided by the distributed queue, which application writers insert into a program dataflow to tolerate consumer faults.

We wish to arrive at a design that provides a constraint-free transfer of data between an arbitrary number of producers and consumers. Envision the following scenario: a distributed queue is placed between P producers and C consumers. The distributed queue should have the following behavior. Data placed into the queue by one of the producers will be sent to exactly one of the consumers. Note that no ordering of data is guaranteed, except point-to-point; if a producer places A into a DQ before B, and if the same consumer receives both A and B, it will receive A before B. Strictly speaking, this is more like a bag than a queue.

In terms of performance, the ideal distributed 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 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 the DQ to quickly adapt to such changes.

Figure 5 presents the logical structure of control and dataflow in a DQ. A producer has data to distribute among the set of consumers, and each consumer has a queue of incoming data blocks from producers to process. Because a DQ will likely consist of many producers and consumers, data are transferred in parallel from the set of producers to the set of consumers that are part of the queue.

The flow of a particular block, and thus the behavior of the system as a whole, is determined by certain decisions, each of which potentially requires global information. First, each producer chooses which consumer should receive a given data block. We show that this is the most important decision in a performance-available distributed queue algorithm. Second, each consumer must choose in which order to process the blocks it has received.

2.4 Graduated Declustering

The corollary problem occurs in a data transfer from producers to consumers when a producer suffers from a performance fault. In this case, if there is no alternate data source and the producers are the bottleneck for the transfer, then

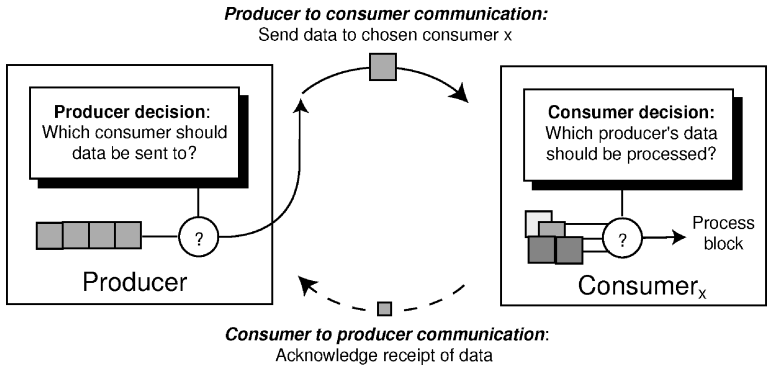

Fig. 5. **DQ data and control:** the basic data movement and control options of the distributed queue. A producer that wishes to send a data block is faced with a decision: to which consumer should data be sent? Once decided, the data block is sent to the consumer. Each consumer receives blocks from many producers, and thus is faced with its own control option: in which order should blocks be processed from different consumers? For each block received, the consumer sends a reply to the producer, as required by any acknowledgment-based reliable communications protocol.

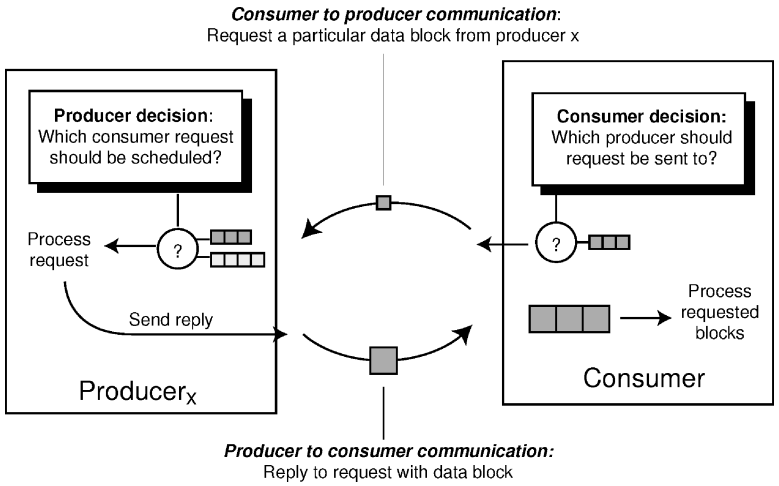

Fig. 6. **GD data and control—**the basic data movement and control options of graduated declustering. A consumer requesting a block is faced with a decision: to which producer should this request be sent? The request is then sent to the selected producer, who has requests from different consumers. The producer then faces a decision: which queue should be serviced? Once serviced, data are sent back to the requesting consumer.

the data transfer runs at the rate of the single slow producer. However, if the data source is replicated, River applications can employ graduated declustering, a data transfer mechanism that carefully divides available producer bandwidth equally among consumers, and thus provides performance availability under producer faults. GD is particularly useful for parallel reads of mirrored on-disk data sets, although it can be applied to replicated in-memory data sets as well.

Figure 6 presents the logical structure of dataflow in graduated declustering. Each consumer sends its request for a particular block to one of the replicated

sources for that block. Each producer must then choose the order in which to handle the consumer's requests. In GD, both decisions are quite important with respect to the overall behavior of the algorithm.

We now more formally describe the desired behavior of graduated declustering. 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 refer to that as N-way graduated declustering. Furthermore, we assume that the $N \cdot P$ producers are in most cases physically colocated 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.

Assume that there are P machines on which producers are running. Thus, on machine M_i, there are N producers, P_i, $P_{(i+P-1)\%P}, \ldots, P_{(i+P-N)\%P}$, all but the first of which are replicas. Assume that on P remote machines 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.

Further 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 equals the rate of that bottleneck resource R_{B_i}. Thus the total bandwidth available across all machines is:

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

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, $F_1, \ldots, F_{F-1}$, where $0 \leq F \leq P$. Thus the total bandwidth available to the consumers is:

$$B_{avail} = \sum_{i=0}^{i=P-1} R_{B_i} - \sum_{i=0}^{i=F-1} R_{F_i}. \tag{2}$$

The goal of graduated declustering is to take the available bandwidth and divide it equally among the P consumers, such that $R_{C_0} = R_{C_1} = \cdots = R_{C_{P-1}} = B_{avail}/P$. If this division is accomplished, then all the consumers will proceed at the same rate, and all finish the data transfer at the same instant; thus the performance faults in the system will have been tolerated as best as they could have been. An example of how GD should allocate producer bandwidth to cope with producer-side performance faults is shown in Figure 7.

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.

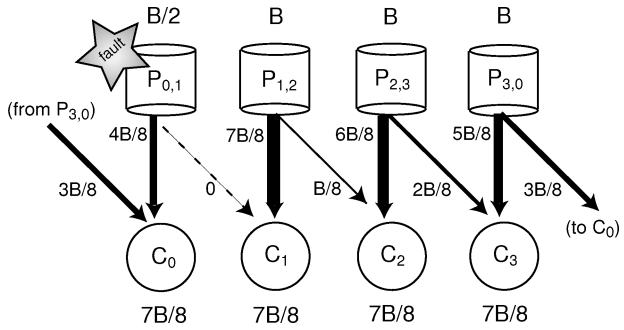

Fig. 7. **GD dataflow example:** how GD alleviates the problem of producer-side performance faults. In the example, the disks are producers, data are mirrored, and only the dataflow is shown—the control flow, in which consumers request particular data items from the producers and as depicted in Figure 6, is omitted for the sake of clarity. Producer $P_{x,y}$ produces data sets x and y, and consumer C_x consumes data set x. Though $P_{0,1}$ performs at half its expected rate ($B/2$, not B), other producers compensate their bandwidth allocations, and all consumers receive a fair share of available aggregate bandwidth ($\frac{7}{8}B$).

2.5 Enabling Technologies

With both the DQ and GD, application writers are given the ability to construct flexible dataflows, thus moving data or computation to where they are best processed given the current state of the system. However, the River design relies on a number of recent hardware and software technologies in order to deliver the desired behavior to applications.

First, River is designed to take advantage of high-throughput system-area networks such as Myrinet [Boden et al. 1995]. The advent of such networks has altered clusters of computers from loosely coupled distributed systems into tightly coupled parallel systems. Without a high-performance interconnect, River would degenerate to a system where most processing is performed locally, as moving data from one machine to another would be too costly to merit consideration. In this scenario, although the River environment could still be used to build applications in the dataflow model, the flexibility afforded by the River mechanisms would go unutilized.

Second, River by design integrates quite cleanly on top of an Active Messages (AM) substrate [von Eicken et al. 1995; Mainwaring and Culler 1996]. AM was designed to export the raw power of such high-performance networks, and thus is a natural match for River. Furthermore, many aspects of the River implementation, as discussed in Section 3, take direct advantage of the request/response nature of the AM protocol.

Third, as the focus of River is primarily high-throughput database query-processing applications, there are certain relationships between disk performance and network performance upon which River relies. First, the network should be able to move data at the rate of the local disk(s), thus enabling flexible dataflow from source to sink across the cluster. Thus the throughput of streaming data from a disk on one machine to a process on another machine should be quite close or equal to the throughput of streaming data from a disk to a process on the same machine. Note that River does *not* rely very heavily

upon the low-latency aspects that modern networks afford; most of the applications we consider stream through large data sets and are throughput oriented. Second, the bisection bandwidth of the network needs to equal the sum of the disk bandwidth available in the cluster. We believe that both of these requirements are not unreasonable, and they certainly hold in our environment; however, as sequential disk bandwidth is improving quite rapidly [Grochowski 1999], network switches, links, and even I/O buses must be sure to keep pace [Arpaci-Dusseau et al. 1998].

Finally, a natural question about River is whether the same benefits could be realized on a more standard TCP/IP-based Ethernet network. In this type of environment, there is often more complexity in both the software protocol stack and within the hardware switches themselves, perhaps introducing additional overheads into communication. As efficient global communication is the key to River, these overheads may penalize a design that freely moves data about the cluster. However, we believe that as long as the technology relationships described in the paragraph above hold, some benefits of the River approach could be realized.

2.6 The Limitations of the River Programming Environment

Although we strive to make River and its mechanisms as general as possible, there are clearly a number of limitations in both our programming model and flexible data-distribution mechanisms. River is best suited for parallel applications that are naturally programmed in a dataflow style. Database query-processing primitives, such as selects, sorts, joins, and so forth, are natural candidates, and are the primary focus of our application study in Section 7. Furthermore, [Wolniewicz and Graefe 1993] have shown that many scientific parallel operators also work well within the dataflow model. However, the dataflow model may not be ideal for all clustered services; for example, when building an Internet service such as a search engine or Web proxy, a more specialized environment may be more appropriate [Fox et al. 1997]. This type of application differs from the traditional dataflow model in that many small requests are processed concurrently, in contrast to our focus on a single parallel application at a time. Even when building other cluster environments, however, we do believe the concept of performance availability should be considered, as similar performance problems are likely to be encountered [Birman et al. 1999].

The two primary mechanisms for performance robustness within River, the distributed queue and graduated declustering, also have limited applicability. In using the DQ, applications must have some flexibility in the manner and order in which they process data. For database query-processing primitives, this is true by design with the advent of the relational model, which decouples the manner in which computation is performed from the specification of the desired result [Codd 1970]. Furthermore, we believe that many other parallel applications have this property, as has been shown in a large body of work on flexible parallel programming environments and studies of existing scientific codes [Poole 1994; Randall 1998; Chakrabarti et al. 1995]. Thus, for parallel applications that can be readily programmed in the dataflow model, it may

well be the case that they can be rewritten to utilize a DQ for performance robustness.

Finally, with GD, one obvious limitation is that the data set served through GD must be fully replicated. The cost of producing a replica may thus inhibit the use of GD; indeed, we envision that applications will primarily use GD to access frequently read on-disk data collections, that is, data that likely need to have some form of redundancy for reliability anyhow. Mirroring on-disk data is expensive in terms of space usage as well, although as disk costs decline, the simplicity and performance advantages of mirroring increasingly work in its favor. Although parity-based schemes could be used to save storage space, such approaches are not amenable to a GD-like performance-robust technique, as GD requires that an alternate data source be readily available to serve data under a performance fault. When reading a block in a parity-based approach, either the data block must be read, or the block regenerated by reading all of the other blocks and the parity block of a stripe, which clearly is too costly to consider. We also note that GD could be extended to function with a partial data set replica (i.e., when some of the data records are replicated, but not all), with some added complexity; in this case, performance robustness would fall somewhere between a nonrobust mirrored system and GD with full replication.

3. THE GANGES IMPLEMENTATION

The Ganges implementation is a second-generation prototype of the River dataflow system. In this section, we describe Ganges, focusing on the implementation of the distributed queue and graduated declustering. In Ganges, the DQ and GD are both implemented as *distributed* algorithms, and thus provide the desired functionality without any global coordination or centralized control. These algorithms must be distributed in order to scale to large clusters and, moreover, because any centralized approach would fundamentally not provide performance availability, as a single slowdown in the central or "master" component would lead to global performance problems.

Ganges is implemented upon on a cluster of Sun Ultra1 workstations, each running Solaris 2.6. Each workstation consists of a 167 MHz UltraSPARC I processor [Tremblay et al. 1995], 128 MB of memory, and two Seagate Hawk 2.1 GB 5400 RPM disks attached on a fast-narrow SCSI bus to the S-bus. One disk is commonly used for the OS and swap space, whereas the other is used by the system for data. Bandwidth from a Hawk ranges from 5.45 MB/s for the outer tracks to 3.18 MB/s for the inner.

The workstations are connected via a high-speed Myrinet local-area network [Boden et al. 1995]. Each workstation has a single Myrinet card, also on the S-bus. These cards are capable of moving data into and out of the workstation at approximately 40 MB/s. The entire system is connected via a collection of 8-port 640 MB/s Myrinet switches arranged in a 3-ary fat tree. All communication is performed with Active Messages (AM) [Mainwaring and Culler 1996], which exposes most of the raw performance of Myrinet while integrating with features such as threads, blocking on communication events, and multiple independent endpoints.

3.1 The Distributed Queue

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.

We now describe the implementation of the DQ. There are two key ideas that are combined to arrive at the algorithm: *randomness*, when picking among equivalent consumers, and *feedback*, via flow control. The latter is the critical element to attain the desired behavior under consumer perturbation.

When a producer has data to send to a consumer, it calls into the `Put` routine. Internally, the DQ goes through the following 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 those data to the chosen consumer.[1]

On the receive side, the implementation is straightforward. A consumer waits for a message to arrive by using the event mechanism available from the Active Message layer. When that message arrives, the DQ wakes up and extracts the message from the network by polling. The message is packaged and returned to the consumer. The DQ 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. The only decision that has to be made occurs when there are messages from more than a single producer. In that case, the DQ will service producers in proportion to their current rate of progress; those that are further behind will receive a higher proportion of service. More details are presented on this in Section 6.3.

3.2 DQ: Discussion

The key to the DQ is that it avoids assumptions about the performance of any one consumer via *run-time adaptation*. Each producer utilizes *feedback* from the consumers to gauge to which consumer data should be sent. Specifically, when a producer sends data to a consumer in the form of an Active Message request, the protocol stipulates that the consumer reply to that message.[2] The DQ uses

[1]The reason for two levels of credit management is simplicity. When there are no credits available, the desired behavior of the producer is to wait for a message to return; thus we provide a simple check for this condition.

[2]This reply message is also required in protocols that utilize acknowledgments to implement reliable message transfers.

the reply as a "signal" or a channel of *implicit information* [Arpaci-Dusseau 2001]; if the consumer handled the data request, the producer infers that the remote consumer is making progress. Each producer monitors the incoming replies to infer the current performance level of each consumer.

This algorithm also has the property of unifying the seemingly diverse implementations of "push-based" and "pull-based" approaches often found in message-passing layers [Karamcheti and Chien 1995]. When producers are the bottleneck for the data transfer, all consumers respond quickly to data requests, and the choice of destinations degenerates to a random choice from the entire set of consumers: a "push-based" approach (a random choice has been shown to be a good one in similar scenarios by Brewer and Kuszmaul [1994]). However, when the consumers are the bottleneck, each producer has all of its messages outstanding, one per consumer; thus, when a reply returns, it implicitly "pulls" the next data request to the replying consumer. Therefore, depending on the relative performance rates of producers and consumers, the DQ will utilize either a push-based or pull-based approach.

It would seem that the functionality that the distributed queue is providing is simply load balancing, which has been studied extensively in the literature [Adler et al. 1995; Blumofe et al. 1995; Johnson 1995; Wen 1996]. However, there are many effective load-balancing algorithms that are not performance-available; they make performance assumptions of one form or the other, and so do not meet our demands. Thus load-balancing and performance availability are not isomorphic.

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 load-balancing schemes have been proposed [Adler et al. 1995; Johnson 1995]. The 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 much a time waiting for a response to its query.

3.3 Graduated Declustering

We now describe the Ganges implementation of graduated declustering. Let us start with the consumer-side of the algorithm, for that is more straightforward. In N-way graduated declustering, a consumer C_i of a particular partition of the data set D_i has N choices from where to request a particular data item. To jumpstart the process, the consumer sends out requests for a fixed number of blocks, distributing them in a 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 crucial 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: a progress metric. 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 has been received by each consumer. In the previous implementation, we had used the average bandwidth the consumer has received, but through experimentation, we have found the total number of bytes to be a more robust and reliable metric.

The producer-side of the algorithm is a bit more complex. The producer is represented by two components: a set of service threads and a scheduler thread. The service threads, one per replica, each receive requests from the scheduler thread, read the data from disk (or whatever the data source may be), and then send the data back directly to the consumers. 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 progress metric of each consumer, and then biases the scheduling of requests so as to "catch up" the lagging consumer. 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 and 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.

3.4 GD: Discussion

The distributed algorithm that implements graduated declustering is based on the following intuition: if a producer is able to balance the progress metric of the consumers to which it delivers data, 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.

Clearly, the choice of progress metric is central to the correct operation of GD. In the current implementation, the consumer piggybacks information in consumer requests in order to help the producer decide which consumer should receive what proportion of service. Although the current metric is the total number of bytes received by a consumer, other metrics are certainly 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 GD implementation to allow for application-specific scheduling.

4. METHODS AND MODELS

To study the behavior of River, we employ both simulations and implementation, and compare results to a simple idealized model of system behavior. The original study [Arpaci-Dusseau et al. 1999] was based solely on results from a prototype implementation, and thus suffered in certain regards. First, although one could conclude that the mechanisms of River were generally better as compared to a nonadaptive approach, one could not know the true upper bound on performance. Second, implementation details were often difficult to separate from the inherent behavior of the distributed algorithms.

To remedy the first problem, we develop an idealized model of system behavior under performance faults. By plotting results versus these models, we are able to gauge how well the system is doing in an absolute sense, not just relative to another, perhaps nonadaptive, system. Without models, it is quite difficult to know when to be satisfied with system performance.

To remedy the second of these problems, we utilize simulation. With simulations, we explore basic algorithmic behavior under a wide range of system parameters, most of which would not be feasible to measure in the prototype. Measurements of the implementation are then gathered to confirm simulation results, as well as to bring forth various "systems issues" not modeled in simulations. The combination of both techniques improves our understanding, separating intrinsic properties of the algorithms from implementation details.

4.1 A Model of System Behavior

We now develop a model of the behavior of an ideal system under performance faults. An ideal system can adapt instantly to any performance fault, can move work or data to best utilize system resources, and is only limited by underlying hardware resources. By comparing experimental or simulation results or both to a model of an ideal system, we are able to better gauge how well the algorithms and implementations are functioning. Note that the model, although simple, is general and could be applied in evaluating the behavior of other systems under performance faults.

Assume that a device is expected to deliver performance at some rate 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 available resources. We term the rate of the fault P_{fault}, where $P_{fault} \leq P_{peak}$. For a given resource 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; the delivered rate of performance of the performance-faulty device is thus $P_{peak} - P_{fault}$. For example, a disk may be able to deliver a peak bandwidth of 5 MB/s (P_{peak}), but then might suffer a fault that takes away 2 MB/s (P_{fault}), leaving 3 MB/s for applications.

To characterize the strength of a performance fault, we define the *fault utilization* $U_{fault} = P_{fault}/P_{peak}$, where the value of the fault utilization ranges from 0 to 1. Note that an absolute failure is a special case of a performance fault, where $U_{fault} = 1$ (i.e., $P_{fault} = P_{peak}$), although in River, we concentrate only on the range $0 \leq U_{fault} < 1$.

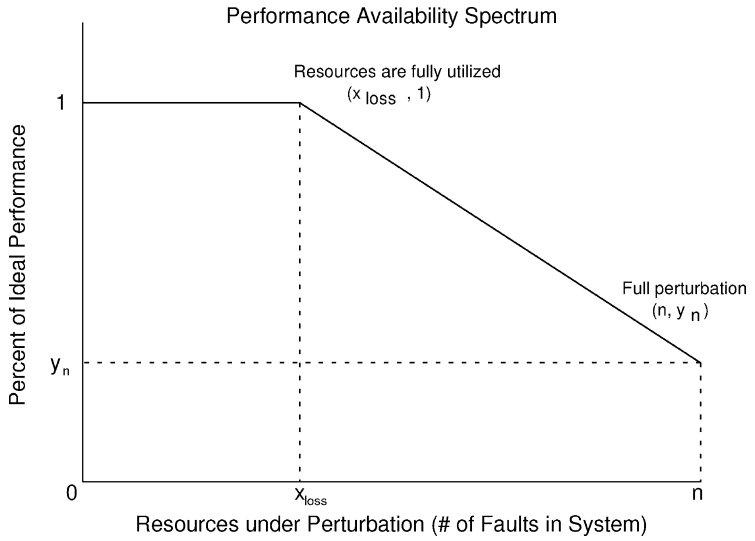

Fig. 8. **Performance availability spectrum:** overall performance under an increasing number of performance faults. Along the x-axis, the number of components experiencing the performance fault is increased up to the maximum number of components in the system n. Overall system performance, as a fraction of ideal performance in a setting free of performance faults, is plotted on the y-axis.

The final definition required before developing the model of behavior under faults is application utilization of a resource. Assume that an application of interest normally runs at a rate of P_{app} on the given resource during some period of interest. For example, if the application were I/O-intensive, P_{app} might be the rate that it reads data from disk during a read phase. As above, P_{peak} is the peak rate that the resource can deliver, and P_{fault} is the faulty performance level. Thus an application uses $U_{app} = P_{app}/P_{peak}$ fraction of the resource when unperturbed. Note that U_{app} is precisely 1 when the resource is under heaviest demand from the application, that is, the CPU during a compute-bound calculation phase, or a disk during a strictly I/O-bound portion of a program.

To help understand the behavior of the ideal system under an increasing number of performance faults, we plot a graph called the *performance availability spectrum*, shown in Figure 8. Along the x-axis of the graph, we increase the number of components that are experiencing the given performance fault. For simplicity of notation, we assume that the same performance fault occurs on each faulty component, the application utilizes the same fraction of resources of each component, and the peak rate of each component is identical—restrictions that are easily relaxed if desired. The y-axis plots overall system performance as a fraction of ideal performance under nonperturbed "perfect" conditions. Thus if the application completes the phase of interest in time $T(x)$, where x is the number of faults in the system, the graph plots $T_{ideal}/T(x)$, while increasing x along the x-axis. T_{ideal} is the minimal time that an algorithm could complete the particular operation, and is not just equal to $T(0)$. For example, if we were

examining the read phase of an external one-pass sort, T_{ideal} would be the time to read the data from the disks at their peak rate. If we instead use $T(0)$ as our comparison point (and not T_{ideal}), the spectrum would not differentiate between poorly performing algorithms and those that perform well.

Our basic approach in developing the model of ideal behavior is a simple two-step process. First, we derive the point where the system is fully utilized by both the faults present and the application; before this point, the sum of the utilization of the faults in the system and the application is less than the total amount of available resources. After this point, the system is under duress, and even in the ideal case the application will see a performance loss, as the application will not have enough available resources to continue unperturbed. Second, we derive the point at which all components are experiencing the performance fault. With both of these points in place, we are able to generate a complete piecewise-linear model of ideal performance.

The first point of interest that we derive is labeled x_{loss} in Figure 8, and is the point where the ideal system begins to see a performance degradation due to the presence of performance faults. At this point, the perturbed resource (which could be a CPU, network link, or disk) is fully utilized across all components in the system. Up to that point, the ideal system is able to move work or data elsewhere, and thereby not suffer any performance loss. After that point, performance degrades as compared to the nonperturbed system due to lack of resources.

We now solve for x_{loss}. At this point, we know that 100% of the resource is utilized across all n components; thus the sum of application resource usage and perturbation resource usage across all n components is equal to n: $(n \cdot (P_{app}/P_{peak})) + (x_{loss} \cdot (P_{fault}/P_{peak})) = n$. Solving for x_{loss}, we arrive at: $x_{loss} = (P_{peak} - P_{app}/P_{fault}) \cdot n$, or in terms of utilization: $x_{loss} = (1 - U_{app}/U_{fault}) \cdot n$.

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 ($n = 16$), and that each process of the application utilizes 75% of the CPU in an unperturbed system; thus P_{app}/P_{peak} is 0.75, and the application uses 75% of the total 16 CPUs, or 12 full CPUs. Assume that the performance fault we are interested in has a fault utilization P_{fault}/P_{peak} of 0.5; therefore, a fault utilizes 50% of the CPU. By substituting these values into the equation, we arrive at $x_{loss} = 8$. When more than half of the CPUs are perturbed, we expect overall performance to drop to less than 100% of peak.

We can also make a few general observations. First of all, x_{loss} degenerates to 0 when $U_{app} = P_{app}/P_{peak} = 1$. This observation matches intuition: when the application uses 100% of the resource in the unperturbed case, any perturbation to the system leads to a loss of overall performance. Second, in the other extreme, if $P_{app} + P_{fault} < P_{peak}$ (i.e., $U_{app} + U_{fault} < 1$), then x_{loss} is greater than n. Plainly stated, if the sum of application resource utilization and performance-fault resource utilization is less than the total amount of resources available, even with all components under perturbation, no slowdown should be experienced in the ideal case.

The other point of interest in Figure 8 is the y value when all n components in the collection are experiencing the performance fault. We call the y-axis value

y_n, to denote the performance level of the ideal system when all n components incur performance faults.

We now derive y_n. At this point, all n components are under perturbation, and the rate that each node can deliver under perturbation is $P_{peak} - P_{fault}$. From this, we can calculate y_n directly by observing that application slowdown, when resources are overtaxed, is the rate that can be delivered divided by the rate needed by the application: $y_n = (P_{peak} - P_{fault})/P_{app}$, or in terms of utilization: $y_n = (1 - U_{fault})/U_{app}$.

For example, if an application utilizes 75% of the CPU and the performance fault utilizes 50%, when all nodes are perturbed, total ideal system performance will be $1 - 0.75/0.50 = 2/3$. In general, as application needs increase, the value of y_n decreases. Similarly, as the fault utilization increases, y_n decreases.

Between the two points of interest $(x_{loss}, 1)$ and (n, y_n) we expect a linear drop in performance. The drop in performance is linear between these two points because the amount of resources taken away from the application increases linearly, and those resources correspond directly to performance. For example, if an application needs 100 MB/s of disk bandwidth during a write phase, and only 50 MB/s are available, the slowdown will be exactly a factor of two in the ideal case.

We can thus derive the slope of the line that connects the two points, and express expected ideal system behavior in closed form:

$$y = \begin{cases} 1 & 0 \le x \le x_{loss} \\ 1 + \left(\dfrac{y_n - 1}{n - x_{loss}}\right) \cdot (x - x_{loss}) & x_{loss} \le x \le n. \end{cases}$$

We now have a piecewise linear model of ideal performance under a given set of performance faults, which we can use to judge the absolute performance of the system in both simulations and the prototype implementation.

4.2 Limitations of the Model

Before describing the simulation environment, we briefly discuss several primary limitations of our model. First, we address the assumption of linear slowdown under faults within the model. This assumption may not always hold, depending on the scaling properties of the application in question. In the model, if an application is given half as much of a given bottleneck resource, it is assumed that the application will run at half the rate as when given the full resource. Although this holds for the applications we are interested in, in general, it does not. For example, some applications can more easily utilize fewer resources, and thus will perform worse than expected as the total number of resources is increased [Singh et al. 1992].

Second, the model assumes that the manner in which perturbations occur across components does not exponentially worsen performance, which is not always the case for parallel programs [Arpaci et al. 1995]. For example, if communicating processes are not coscheduled [Ousterhout 1982], their performance may worsen by many orders of magnitude, as the progress of the

application may depend on the progress of a specific process. Thus, when comparing to the ideal, it may not always be realistic to assume that a program will be able to attain such an ideal under uncorrelated faults across multiple components.

Third, the model also assumes that throughput (and not response time) is the performance metric of interest. The result of this assumption is that two systems may deliver the same throughput and thus appear identical under the model, but one may have much better response time than the other: the costs of queueing are not reflected in the performance analysis.

In practice, we have found that these assumptions are adequate for the database query-processing applications that we have focused upon, which have good scaling behavior, do not require coscheduling for good performance, and process large volumes of data and thus are throughput-oriented. Despite these limitations, we have found the model to be an invaluable tool during the development of the system: by providing a performance target, an ideal model focuses performance tuning, letting the designer know what aspects of the system need to be improved. Furthermore, a good model lets a designer know when to stop tuning: if performance approximates the ideal, there is little need to fine-tune the system any further.

4.3 Simulation Environment

In addition to models and measurements of a real implementation, we employ a set of simulations to demonstrate some of the properties of the distributed algorithms. The simulator that we have constructed provides only low-level primitives: *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 components.

The simulator core consists of an event-based simulator. The event subsystem is written in C for efficiency, whereas the rest of the simulation system, including queue and source abstractions, callbacks, and other glue code, is written in Tcl [Ousterhout 1990]. Tcl affords great flexibility in assembling arbitrarily complex simulations, and with Tk [Ousterhout 1991], enables visualization and animation of scenarios.

5. HOW EFFECTIVE ARE THE ADAPTIVE MECHANISMS OF RIVER?

In the next four sections, we answer each of the four questions posed in the introduction, with the basic goal of better understanding run-time adaptation as an approach to robust system design. As stated before, we utilize results from simulations and a prototype implementation where appropriate, often comparing to the ideal.

We begin by exploring the general effectiveness of the River mechanisms, by presenting results from experiments that test the performance of the DQ and GD under an increasing number of performance faults. For these experiments, we utilize both simulation and implementation results and show that the two match quite closely. These results extend previous work along a number of axes, by enhancing understanding via simulation, showing that the mechanisms

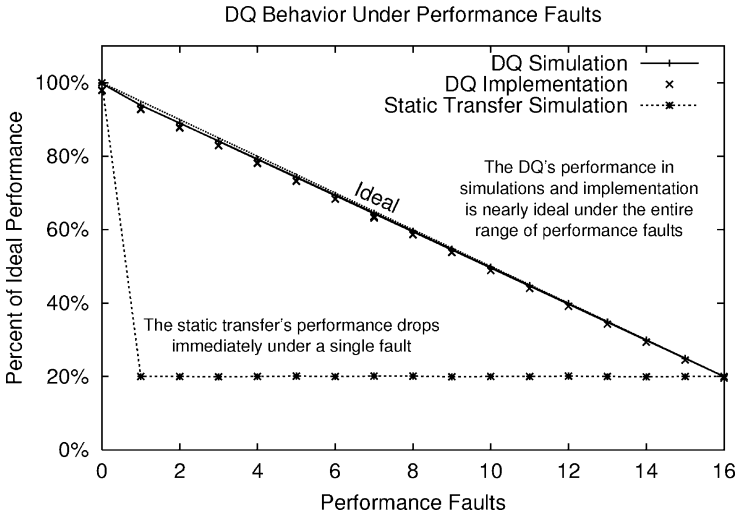

Fig. 9. **Distributed queue performance:** results from a simulation of DQ and an implementation of the DQ are presented, and compared to ideal performance under an increasing number of performance faults, as well as to the performance of a "static transfer." In the experiments, producers generate data at 40 MB/s and nonfaulty consumers consume data at 5 MB/s. A performance fault reduces a consumer's performance by a factor of 5, down to 1 MB/s.

work nearly ideally as compared to our model, displaying the sensitivity of GD to the layout of performance faults, and demonstrating the improved performance of second-generation implementations of the DQ and GD.

5.1 The Distributed Queue

As described in an earlier section, the distributed queue presents applications with a high-speed backplane for data sharing and can be used to tolerate consumer-side performance faults. Within an application, producers place data into the distributed queue, and consumers receive the data. Thus a consumer can logically receive any data block that has been put into the DQ. The central challenge is to design a scalable, efficient, distributed algorithm that moves data from producers to consumers such that faster consumers receive a proportionally higher amount of aggregate producer bandwidth.

Figure 9 illustrates the performance of the DQ under a simple perturbation scenario, and compares its performance to the ideal and to a static nonadaptive approach. In these experiments, 16 producers send data to 16 remote consumers (32 total machines). The producers are able to generate data at 40 MB/s, whereas the nonfaulty consumers can sink data at 5 MB/s; this setup emulates a set of processes (producers) writing in-memory records in parallel to disk, where the disks (consumers) are the bottlenecks in the transfer. Along the x-axis, we increase the number of consumers undergoing performance faults, where a performance fault reduces the rate of a consumer by a factor of five (i.e., the consumer performance is reduced from 5 MB/s to 1 MB/s). Thus the leftmost point on the x-axis shows performance in the system under zero performance

faults, and the rightmost point shows performance when all consumers perform at the perturbed rate.

The y-axis plots DQ performance as a fraction of the ideal in the unperturbed case (i.e., when there are no faults in the system), where the ideal is simply the aggregate bandwidth across the consumers. The "Ideal" performance line reflects the model as developed earlier. Note that $x_{loss} = 0$ in these experiments, because the test utilizes all available consumer bandwidth in the unperturbed case; in such a case, even the ideal system drops in performance under only a single fault.

As shown in the figure, a "static transfer," where each producer uses a static hashing algorithm to pick which consumer to send data to, performs well only with zero faults in the system; at that point, throughput is high and scales well to 32 machines. However, with just a single fault, performance of this static approach drops immediately to that of the slowest consumer, because the transfer does not complete until the last consumer has finished its portion of the global task. As performance faults are added into the system, performance stays at the same low level.

As also shown in Figure 9, by utilizing implicit information and run-time adaptation, the simulated performance of the distributed queue under performance faults is excellent, delivering nearly ideal performance across the range of induced faults. The figure also shows that our simulation results closely match those of the implementation running the same experiment, and that the implementation is within 5% of ideal for all datapoints.

5.2 Graduated Declustering

Whereas the distributed queue tolerates consumer-side performance faults (by moving more data to faster consumers), it suffers when a producer does not deliver data at the expected rate. If the data are not available from another locale and the producers are the bottleneck in the data transfer, then there is no solution to this problem. However, in many cases, data are replicated for the sake of reliability; for example, in a mirrored disk system, each data block is available from two locations. Graduated declustering can be used to coordinate access to a replicated data set by dividing the aggregate producer bandwidth equally among consumers, thus lessening the effects of producer-side performance faults.

Figure 10 shows the performance of GD, both via simulation and implementation. For the simulations, two different performance-fault layouts are presented: in the "worst" case, faults occur on adjacent producers, and thus immediately affect both data sources for a given consumer; in the "best" case, they are distributed throughout the producers such that no two faults are on consecutive producers until more than half of the producers are perturbed. The implementation results are only shown for the best layout. Note that previous results only presented best-layout performance and thus do not capture the range of possible behaviors.

Overall, performance with the best layout of faults is good, though not ideal, due to the limited amount of replication; only if each producer contained every

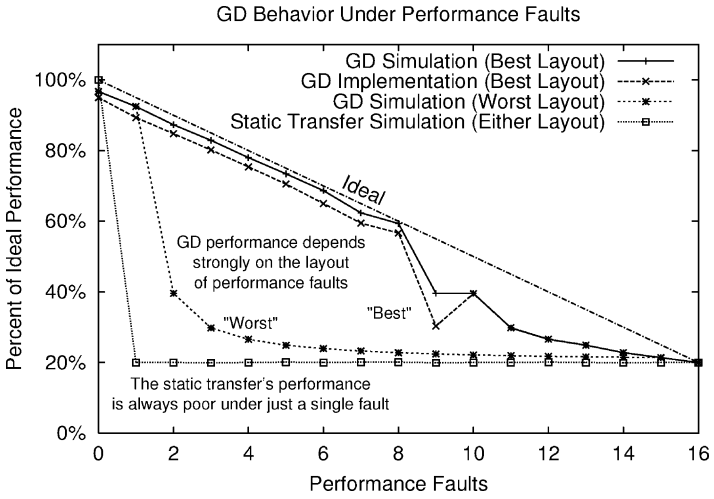

Fig. 10. **Graduated declustering performance:** results from simulation and implementation of GD are compared to ideal performance under an increasing number of performance faults, as well as to a "static transfer." In all experiments, nonfaulty producers generate data at 5 MB/s and consumers can consume data at 40 MB/s. A performance fault reduces a producer's performance by a factor of 5, down to 1 MB/s. Due to limited replication, the layout of performance faults matters; "best" implies that they are spread out, and "worst" indicates that they occur consecutively on producers.

data set would the flexibility necessary for ideal performance be available. More generally, given that R is the level of replication for each data item and P is the number of producers in the system, then the GD algorithm can always tolerate the presence of $R-1$ performance faults, given the "worst" layout, and potentially tolerate up to $P \cdot (1-(1/R))$ faults, under the "best" layout. In a real system with randomly placed faults, delivered performance will fall in between the best- and worst-case lines.

6. WHAT ARE THE KEYS TO EFFECTIVE RUN-TIME ADAPTATION?

We now discuss four keys to the success of the run-time adaptation of the DQ and GD. The first key is the careful management of the interaction with the communication layer; communication is at the heart of adaptation within River, and managing the interaction of the River distributed algorithms and flow control is crucial to robust performance. The second key is the presence of excess parallelism at the application level. To be robust to performance faults, River applications must move a reasonable number of objects through the DQ or GD; the fewer objects that are transferred, the more likely a performance fault will have an unexpected and deleterious effect. Fortunately, many data-intensive applications will have no trouble meeting this requirement. The third key is the presence of globally aware data scheduling; each entity that participates in a robust data transfer must in some way monitor the progress of all others in the transfer, and compensate for laggards accordingly. At ends with this goal is a scalable and distributed implementation. Finally, the fourth key is the presence of "slack" in the system. If the system is driven at its full rate, any

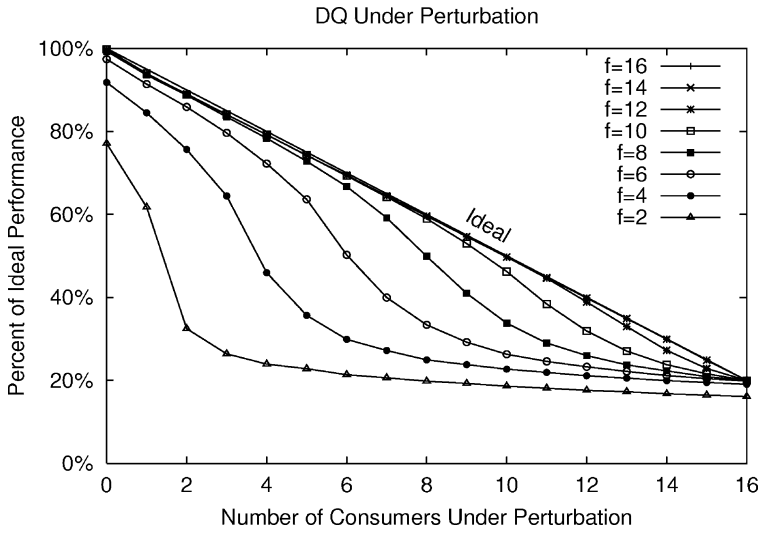

Fig. 11. **The need for flow control:** the amount of flow control is varied under a set of perturbation simulations for the DQ. Each line plots the performance under perturbation of the DQ for a given number of flow-control credits ($f = x$, the total number of credits given to each producer), and the number of perturbations is increased along the x-axis. 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 unperturbed production and consumption is 5 MB/s, and a fault reduces the rate of consumption to 1 MB/s.

performance fault will lower performance, even in an ideal system. Thus the system should be engineered with some amount of extra capacity, which the adaptive mechanisms of River can then utilize when unexpected problems occur to deliver performance at a consistently high rate. Both the first and third keys led to refinements in the DQ and GD algorithms.

6.1 Communication Layer

One crucial aspect to the implementation of the distributed queue and graduated declustering is the behavior of the underlying message layer. In particular, the message layer must not restrict the number of outstanding messages to less than these mechanisms need; if the layer does so, performance under perturbation will be less than expected. Unfortunately, most message layers, including AM, restrict the number of outstanding messages to a particular arbitrary number, chosen by the message-layer developers. Worse, this number is often hidden from clients of the communication layer, which would render the construction of distributed algorithms such as the DQ and GD impossible. We now investigate the number of flow-control credits needed to mask performance faults.

Figure 11 plots the results of simulations that vary the amount of available flow-control credits for the DQ (results for GD are quite similar and therefore not shown). The simulations reveal the importance of flow control to the DQ algorithm, both in the unperturbed case where there are zero performance faults in the system, and in the face of perturbation.

In the unperturbed case, we see that with very few outstanding credits per producer (four or less), performance drops below ideal. Not surprisingly, each producer must be able to keep enough outstanding messages to fully pipeline the system.

More interestingly, as the number of faults increases, the number of credits that a producer is allocated must also increase. In cases where the number of flow control credits is less than or equal to the number of perturbed consumers, performance suffers. The reason for this drop-off is straightforward: if a producer does not have enough message credits to keep at least one outstanding to each of the slow producers, the algorithm will not be able to "remember" which nodes were the slow nodes, and will keep sending messages to those slow nodes.

Thus, from these simulations, we can conclude that the distributed queue algorithm should always have at least one outstanding message per consumer; this delivers to each producer a perspective on the remote performance of all consumers in a straightforward, simple, and effective manner. From the perspective of the prototype, the underlying message layer must not restrict the number of outstanding messages to less than the number of consumers in the system. The current AM implementation has a hard limit of 30 total outstanding messages;[3] thus, in our prototype, we cannot tolerate performance faults on more than 30 consumers.

6.2 Excess Parallelism

One essential ingredient to tolerating performance faults is the presence of excess parallelism. For example, in a system with one producer and n consumers, if there are only n items of data to send, a perfect distribution through the distributed queue leads to one piece of data per consumer, and performance is dictated by the rate of the slowest consumer. Even in more realistic settings, excessively perturbed nodes lead to end-of-run effects that become first-order performance factors. We now explore the amount of parallelism needed to tolerate performance faults.

Figure 12 plots the simulated performance of graduated declustering under a single performance fault, as a function of the total amount of data read (again, as results from the DQ are nearly identical, we focus solely on GD for this experiment). As we can see from the graph, if only a small amount of data is sent through GD, it does not perform well, as the time for the perturbed producer to deliver its remaining blocks after all others have finished takes a significant proportion of overall run-time. Because each consumer is willing to send a request for data to each producer, even the slowest producer receives R messages, where R is the level of replication of the data set (e.g., in a mirrored disk system, each producer processes at least two messages). Thus end effects can be noticeable, especially when there is not much work in the system. If the total amount of work is small, performance faults are difficult to tolerate under the current River approach.

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

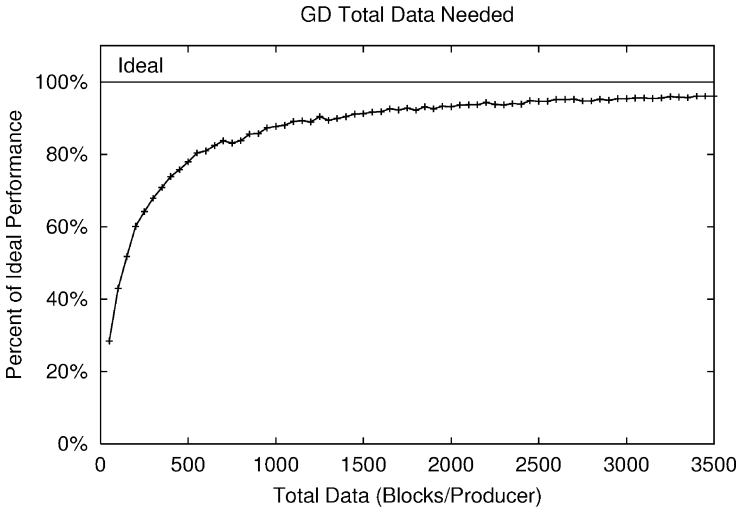

Fig. 12. **The need for excess parallelism:** performance under a single perturbation plotted as a function of the total amount of work that is pushed through a simulated GD in blocks, where a block is 8 KB. The consumption rate of the perturbed producer is 1 MB/s, or 1/5th of an unperturbed producer. In all cases, if only a small number of blocks pass through GD, the system will be vulnerable to performance faults, delivering performance noticeably less than ideal. For these simulations, there are 16 producers and 16 consumers. Note that the best performance the system could deliver is 95% of ideal in this experiment, as there is a single fault present.

We can view this problem as a weakness of the run-time methods of River: data are always given to or taken from all the nodes in the system, with more data sent to or requested from those nodes that operate faster. This problem could partly be remedied with *historical methods*, which remember which nodes performed poorly in the past, and bias actions from the very start. For example, if a certain node of the system is performing so poorly that its inclusion in the data transfer does not offer much gain, that node could be avoided altogether. Furthermore, the current mechanisms of River are *memoryless* across runs, and thus must relearn at every run what the performance characteristics of the system are. If performance faults are lengthy in duration (i.e., past performance predicts future performance), historically based techniques could be effective. For now, our solution is to write applications such that they utilize as much parallelism as possible, and thus amortize end-of-run effects, which we have found is not overly burdensome for a range of data-intensive applications. In addition, the added complexity of including an historical approach may not warrant the resultant small performance benefit.

6.3 Globally Aware Local Scheduling

With the GD algorithm, each producer is made aware of the progress of the two consumers it serves; by biasing the scheduling of its data blocks towards the lagging consumer, each producer can make purely local decisions and yet coerce the system towards a common global goal. In the original DQ algorithm, we

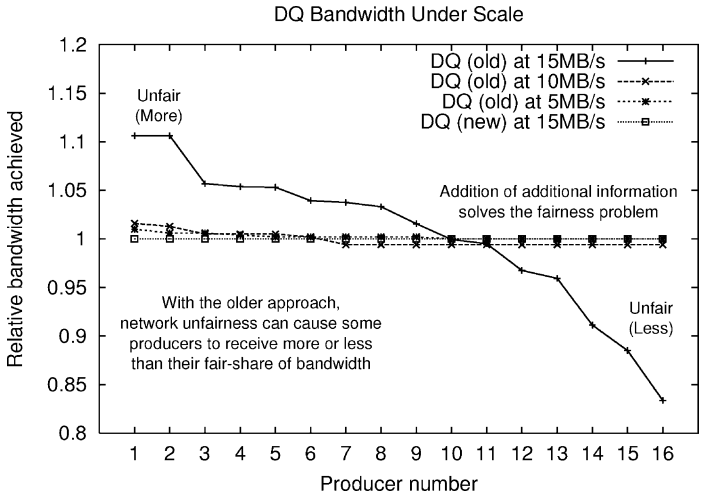

Fig. 13. **The need for globally aware local scheduling:** the average producer bandwidth in the DQ plotted over the lifetime of an experiment. In the experiment, 16 producers send data to 16 consumers. The consumers receive data at 5, 10, and 15 MB/s, which is varied across the three lines. At rates of 5 and 10 MB/s, each producer receives roughly a fair share of the bandwidth, and thus they all 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 bias towards some of the producers. By distributing progress information to each consumer, more clever local scheduling solves the unfairness problem.

believed there was no need for such global awareness within the DQ. However, as we demonstrate below, the DQ also requires globally aware local scheduling in certain contexts.

Figure 13 plots the performance of the DQ in three different experiments on the prototype system. In each experiment, there are 16 producers sending data to 16 consumers. Across the three experiments, the rate that each consumer sinks data is controlled, from 5 MB/s up to 10 MB/s and 15 MB/s. The y-axis of the figure plots the relative share of aggregate consumer bandwidth that each producer numbered along the x-axis receives. Ideally, each producer receives an equal share of the bandwidth, which would be reflected as a y value of 1.

With the consumer rate at 5 or 10 MB/s, each producer receives a fair share of consumer bandwidth, and all is as expected. However, at 15 MB/s, some producers (those on the left of the graph) receive a notably higher portion of consumer bandwidth, whereas others receive correspondingly less.

The problem is inherent in the original design of the DQ, which uses a first-come, first-served algorithm for processing blocks on the consumer, and its interaction with Myrinet switch fairness properties. When performing more detailed measurements of our implementation, we found that under high loads and with many participating nodes, the Myrinet switches do not fairly schedule transfers and thus some producers do not receive their fair share of network bandwidth. Because overall performance is determined by the rate of the slowest entity, unfairness causes the performance of the system to drop to that of the last producer to finish its transfer.

To overcome this problem, we piggyback extra information in DQ messages. Specifically, by sending information about the rate of progress from producers to consumers in each data request and response, each consumer can schedule its service in a more informed (and not a blind FCFS) manner. Our initial results, of a modified DQ that performs this consumer-side scheduling, are also shown in the figure; by biasing service towards lagging producers, performance is level across all producers, and thus the unfairness problem is solved. Note that this behavior was not discovered via simulation—the simplified simulations do not model switch behavior in any detail—underscoring the importance of our combined approach.

6.4 Slack In the System

Finally, we explore the need for "slack" in the system or, more formally, the need for excess performance capability in components. Without slack, even a single performance fault in the system reduces overall performance, as we have seen in our idealized model and real results (see Figure 9). When slack is present, the run-time adaptive methods of River should be able to exploit it and deliver consistent high performance to applications.

Figure 14 shows the results of three simulations with the DQ, where the unperturbed rate of the consumer is varied across the three graphs. In all three experiments, the producers generate data at a fixed rate which is less than the peak rate of the consumers, and the DQ performs nearly ideally across all three graphs. Not surprisingly, as the consumer rate increases, more faults are required to induce a performance decrease as compared to the unperturbed system. (We have also generated similar results for GD, although we again omit them.)

The resulting dilemma for a designer is exactly how much excess performance capacity to engineer into the system, a decision beyond the scope of this current work. To answer this question, we believe one needs to develop stochastic models of modern device behavior, and then apply those models to analyze the likely behavior of the system. This may enable probabilistic guarantees of overall system behavior, similar to Birman et al's [1999] work on Bimodal Multicast; we plan to investigate this further in our work on fail-stutter fault tolerance [Arpaci-Dusseau and Arpaci-Dusseau 2001]. Of course, if the designer of the system does not value the consistent performance provided by slack, the system could be engineered to simply provide what the applications need: in this case, the mechanisms of River are still useful in that they allow for graceful degradation under performance failure.

7. HOW USEFUL ARE THE RIVER MECHANISMS TO APPLICATIONS?

We now examine the system from the application level. How easily can applications utilize the DQ and GD in order to create robust applications? In this study, we concentrate on database query-processing primitives. First, we present performance results of a range of query-processing primitives built on top of the prototype implementation of River, again comparing performance to "ideal." With regard to previous results in Arpaci-Dusseau et al. [1999], we present

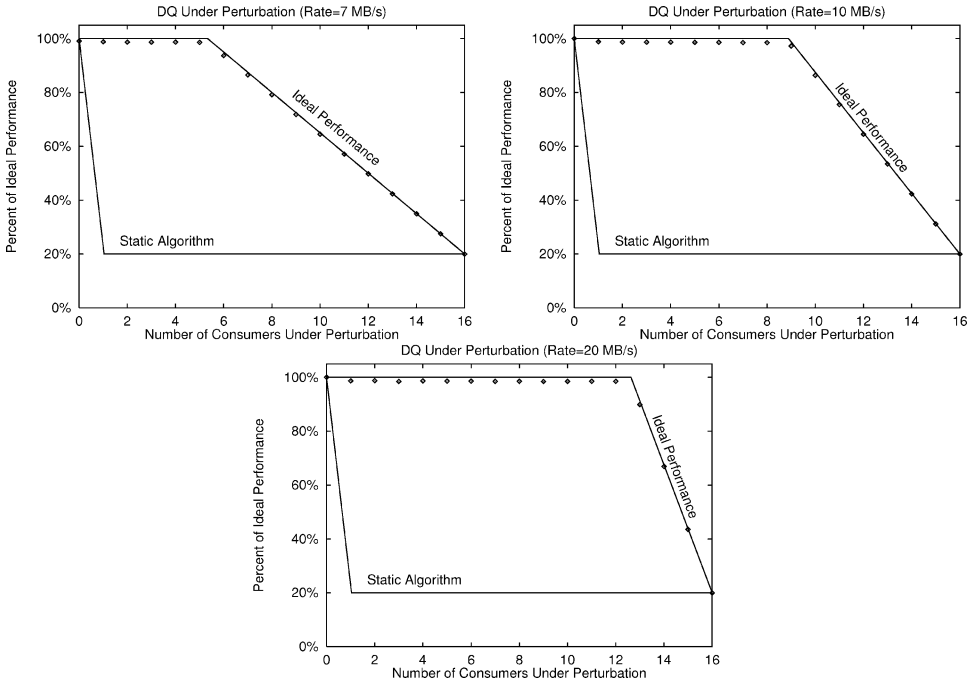

Fig. 14. **The need for slack:** results from simulations of the DQ compared to ideal performance under an increasing number of performance faults. In the experiments, producers generate data at 5 MB/s and nonfaulty consumers consume data at 7, 10, and 20 MB/s. A performance fault reduces a consumer's performance by a factor of 5, down to 1 MB/s. As one can see, as nonfaulty consumer performance increases, more faults are required to reduce performance to less than peak. Thus the DQ is able to take advantage of slack in the system, as predicted by the model.

results from a greater number of applications, and because of the improved implementations of the DQ and GD, improve the performance robustness of the applications. Second, we discuss difficulties in utilizing the basic River mechanisms. Although not always a perfect match for application semantics, we find that the DQ and GD are usually suitable for creating robust programs.

7.1 Application Performance

In general, to develop a robust River application, programmers insert distributed queues and graduated declustering into their applications to form "points of flexibility": places in the dataflow where performance faults can be tolerated. Application writers focus on constructing flexible flows, and the infrastructure handles the rest. By utilizing the two core River mechanisms, applications can potentially withstand performance faults and achieve nearly ideal performance under a range of faults.

In this study, we present the performance of six I/O-intensive, parallel database query-processing primitives, each of which has been transformed from a static parallel application into a robust and adaptive version. Figure 15 presents our results for each primitive. In the figure, performance of the application with an increasing number of disk performance faults is shown, where a

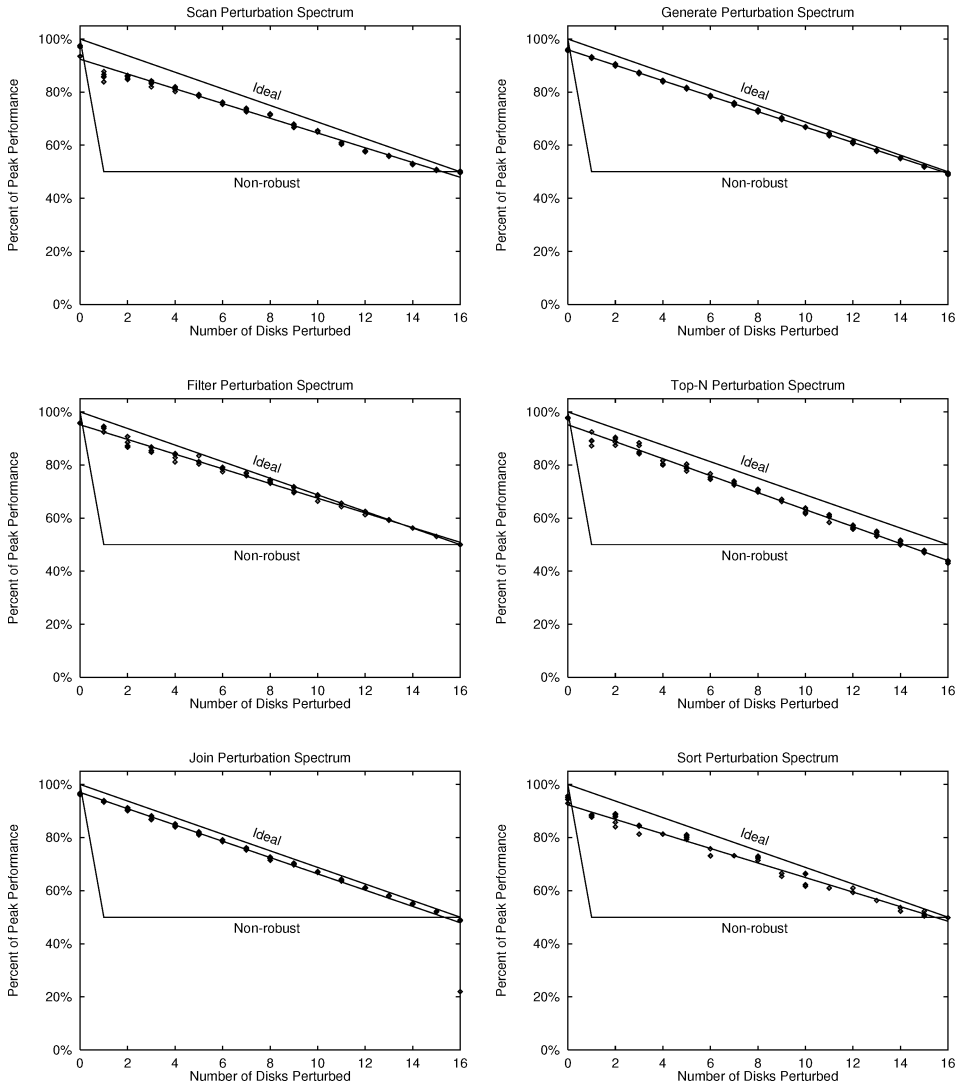

Fig. 15. **Application performance**: the results of running database primitives on a 16-machine cluster under disk performance faults. The applications are a parallel scan of a data set (read only), a parallel generation of a data set (write only), a parallel filter (read data from disk, select certain records based on a user-defined function, write selected records to disk), a top-N selection (find the top N values in the data set, and be able to find the next N efficiently), a parallel hash-join of two data sets, and finally a parallel external sort. Each application operates on roughly 150 MB of data per disk per node, for a total of 2.4 GB across 16 machines. A performance fault reduces performance by a factor of 2. Each datapoint represents a single run, and 5 runs are shown per point along the x-axis; a best-fit line of the data is also plotted.

fault to the disk utilizes half of available disk bandwidth. For all applications, "Ideal" is the amount of data touched by the application divided by the peak rate available from the disks, a reasonable though harsh ideal because the applications are all data intensive (compute time is mostly negligible). For example, the parallel scan reads a total of 2.4 GB from 16 disks, each of which can run at 5.45 MB/s. Thus ideal time for the scan is $2.4\ GB/(16 \times 5.45\ MB/s) \approx$ 28.18 seconds.

From the figure, we can see that for all applications, performance is good when there are no faults in the system, and degrades gracefully as faults are imposed upon the disks, following the desired trend. Specifically, performance is within 89% of ideal across the entire range of disk faults for all applications.

By examining the leftmost point of each graph, we can also gain insight as to the overheads of utilizing the River mechanisms to provide performance robustness. First, let us examine one of the simpler applications, a scan. The scan is simply a parallel scan of the input data; in this case, 16 processes read data from 16 disks. Because it is a read-only application, the robust version of the scan utilizes GD to access data from a mirrored collection. From the graph, we can see that there is a noticeable overhead (roughly 10%) for using GD, as compared to the ideal application in which each process reads data from a single disk. Most of the overhead of GD can be attributed to the change in workload presented to the disk: in the static (nonrobust) scan, each of 16 disks serves a single sequential stream of data to a single process, whereas in the robust scan using GD, each disk serves two streams to two processes, due to replication. By serving two streams, additional seeks are induced, thus lowering overall performance.

Next, let us examine generate, which can be viewed as the converse of scan, as it is a parallel data generator. Each of 16 processes generates random records and writes them to disk. The robust version of the generate uses a DQ to distribute the load across the disks and thus tolerate disk performance faults. Even at the leftmost point, the generate application delivers 95% of ideal performance, showing a slight but acceptable degradation.

The other four applications exhibit more complex combinations of these basic costs, as each of them utilizees both GD and one or more DQs in their robust versions. The only exception is found in the sort (described further below), which utilizes a DQ in a more coarse-grained manner than the other applications, and thus exhibits slightly worse performance characteristics than would be expected.

7.2 Application Semantics

7.2.1 *Top-N Selection.* We next examine the cases where application semantics do not perfectly mesh with the DQ and GD mechanisms. We first focus on the top-N selection. The 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.

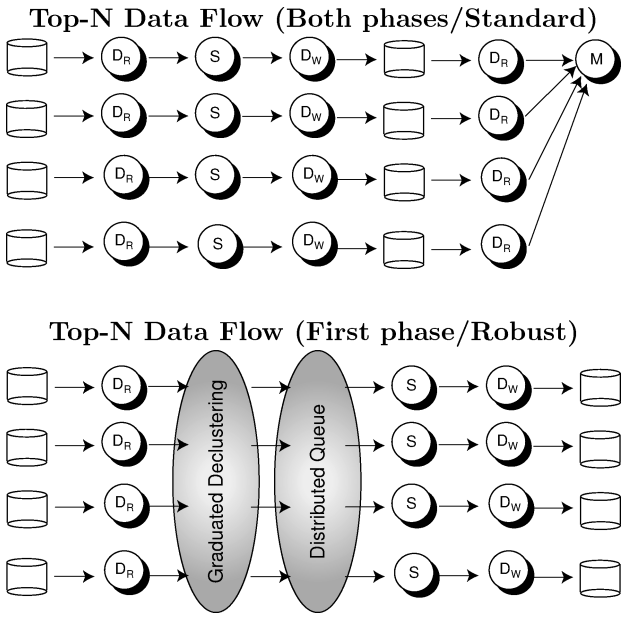

Fig. 16. **Top-N data flow:** static and robust versions of the top-N selection are presented. On the top is the static (nonrobust) version of the application. Data are read from disk by the disk-read modules (D_R), and passed to sort modules (S), which buffer some amount of data before generating a sorted run and writing it to disk via the disk-write modules (D_W). Runs are generated as such until all data have been transformed into a set of sorted runs. In the (short) final 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. On the bottom, the diagram depicts the disk-robust first-phase of top-N selection. Both robust mechanisms are employed; GD allows us to tolerate read side performance faults, and a DQ, placed between the disks and the sort modules, allows us to tolerate performance faults at the write side.

Figure 16 presents the basic dataflow of the top-N selection.[4] 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 by merging the top few records of the sorted runs into the final result. This approach is particularly useful when a user is likely to request the next N items, which can be quickly supplied by continuing the merge.

The challenge to the programmer is how to make this program robust to disk performance faults. We begin with the first phase. Figure 16 shows the disk-robust version of the first phase of top-N. Both graduated declustering and a distributed queue are utilized in order to make the first phase of the program robust. By utilizing GD over a replicated data set, producer faults are easily handled. By inserting a DQ immediately after GD, more data are moved to faster sorters, whose rates are each limited by their local disks. Thus a fully disk-robust first phase is generated; results in Figure 15 confirm this.

[4]Note that there are many ways to implement a top-N selection; we do not present this as the best or only method.

We find that adding robustness to the merge phase of the program is quite difficult for two reasons, both of which are general and could apply to other applications. First, in order to utilize GD, we would have to replicate the sorted runs as they are written. The need for replication highlights a general weakness of our run-time adaptive approach: any multiphase program pays a high cost to replicate data when writing to disk if it wishes to subsequently use GD to avoid producer-faults during a read phase. GD is better suited for oft-read data sets, which are replicated once and read many times.

In addition, all of the data must be merged into a single merge module (M), which prints the final output. If that module runs slowly for some reason, the application will also run slowly during that phase. The DQ cannot be utilized, as there is only a single destination for the data. Thus applications with dataflows that are similar to this need to minimize the time spent in that portion of the dataflow.

With those two weaknesses in mind, we still believe that in this case, overall application performance will be highly robust, due to Amdahl's law. Almost no time is spent in the second phase, particularly for large data sets. Thus even if one particular disk is greatly slowed during this phase, overall application performance will not suffer unduly.

7.2.2 *Sorting.* Next we present a challenging operator, external (or disk-to-disk) sorting. In general, sorting is a good benchmark for clustered systems because it stresses disk, memory, and interconnect bandwidth. In this section, we only consider a one-pass version of sort; a two-pass sort consists of multiple runs of the one-pass sort (plus a merge), and therefore also benefits from the development below. Also, following the precedent set by other researchers, we measure the performance of 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 nonuniform distribution, we would need to modify our implementation to perform a sampling phase before the sort described below [Blelloch et al. 1991; DeWitt et al. 1991]; this sampling phase could also be made robust via the use of GD.

The topmost diagram in Figure 17 presents the flow of data in the standard version of the sort, which is based upon the flow of NOW-Sort, a world-record breaking parallel sorting program for clusters [Arpaci-Dusseau et al. 1997]. First, data begin as an unsorted parallel collection on a number of disks. Data are read in on each disk node via the disk read module (D_R), and then passed 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 them to the disk write module (D_W), which proceeds to write the data out to disk as a stream, thus preserving the order. This read-sort-write phase repeats until all of the data have 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 the bottommost diagram

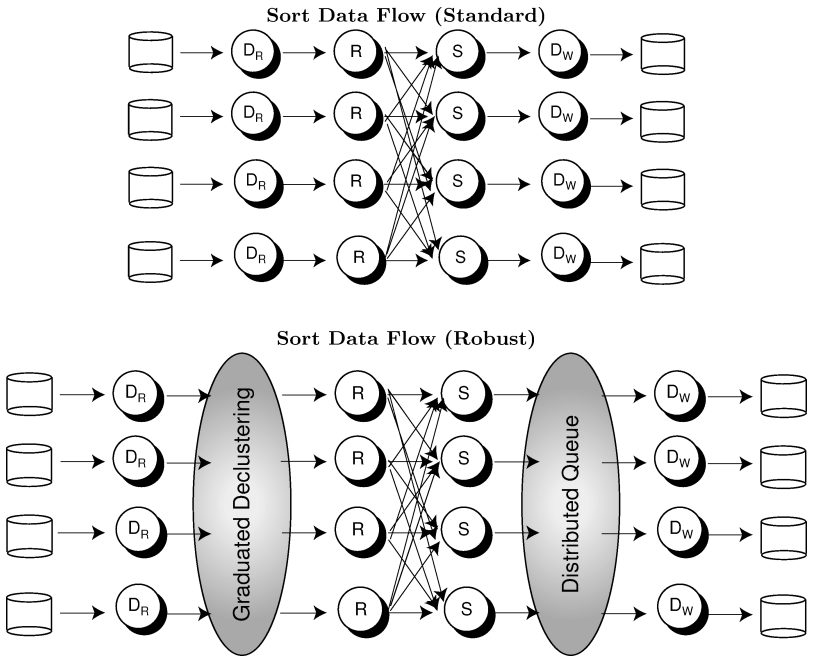

Fig. 17. **Sort dataflow:** data are read from disk in parallel by a set of disk-read modules (D_R) and then passed on to a set of range partitioners (R). These partitioners segment the data set across sort modules (S) by key value; for example, with four sort modules present, the top fourth of the keys 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 them to the disk-write module (D_W) for output to disk. In a multipass sort, this phase would repeat until all the data have been sorted into many sorted runs. 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 have been sorted, the sorters cannot 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. Load balancing occurs at a much coarser granularity, while preserving the semantics of the sort.

of Figure 17. 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 application 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! Furthermore, the distributed queue cannot 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 function properly, a slight modification had to be made to the distributed queue. Instead of handing records one at a time to the distributed queue, the sort module passes large sorted chunks of data to the distributed queue.[5] 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 the sort, it might divide this into ten 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 15 presents the results of the perturbation experiment. From the figure we can see that performance under perturbation is the least stable of all of 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, performance degrades gracefully as expected, although the absolute performance is not as high as with other primitives, due to the extra amount of per-run overhead associated with managing $n \cdot k$ runs.

7.3 Extending River to Other Application Domains

Given our concentration on database query-processing primitives, a natural question arises about the generality of the River model: What other types of applications could benefit from the mechanisms provided by the River environment? Clearly, any application *could* be written in the River model, as it is a general-purpose programming substrate. The better question is what types of applications would be easily and naturally written in the River framework, and can thus readily utilize the adaptive mechanisms provided in order to achieve some level of performance robustness.

One application domain that we believe would also benefit from the River environment is that of parallel scientific codes. Wolniewicz and Graefe [1993] already have shown that many common scientific operators fit well into a dataflow environment. We further believe that these operators can often be reengineered for performance robustness in a manner similar to the database primitives above. For example, a common operation in those types of applications is a matrix transpose [Poole 1994]. Assuming data must begin and end on disk, a transpose is structurally quite similar to the external sort described above; instead of routing data based on a key value, each "record" (i.e., a floating-point value) is routed to its final destination based on its location in the final output set. One slight difference is how the input should be read into memory. The sort can just read in each input stream in sequential order, whereas the transpose

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

would have to read from each input stream (i.e., from each row or column) in a staggered manner, so as to balance output load properly.

Other examples that would work well in the River environment are out-of-core matrix–vector or matrix–matrix multiplications. Both of these applications have a great deal of flexibility in the order that they process data (i.e., when elements are multiplied), and thus are likely to be amenable to transformation into performance-robust programs. Finally, many of the other applications described in Poole's survey of I/O-intensive scientific applications do not require strict ordering among records being written to disk, also hinting at their suitability for the River environment [Poole 1994]. Of course, although these applications seem like excellent candidates for River, only through implementation and experimentation can we truly know how good a match they are.

8. WHAT ARE THE LIMITATIONS OF RIVER RUN-TIME ADAPTATION?

Finally, we discuss two scenarios where the run-time adaptive techniques of River do not work well. In the first situation, a Myrinet switch deadlock showcases the reliance of River on the network as a performance-reliable medium. Note that this problem only arose within the implementation, again highlighting the value of experimentation with a real system. The second scenario demonstrates an inherent weakness of run-time adaptation, in that decisions made at run-time lack global perspective. For example, data written to disk under current conditions may not be laid out properly for later access under potentially different conditions.

8.1 Global Performance Faults

The first problem that we present is the result of a peculiar switch behavior, but demonstrative of a more general problem. Figure 18 plots the performance of graduated declustering under scale, increasing the number of producers and consumers from 1 to 16 along the x-axis, and plotting total throughput, as a percentage of peak, along the y-axis. In this case, producers were not throttled (i.e., they produce data as fast as they can), and there are no perturbations within the system.

In the figure, performance is excellent at low scale, coming very close to 100% of peak, but then drops off unexpectedly with 7, 9, and 11 or more producers and consumers involved. Via careful instrumentation, we found that in all poorly performing experiments, performance was fine for a period of time, but then suffered from dramatic, systemwide two-second pauses. Further investigation of this symptom led to the conclusion that the Myrinet switches were deadlocking, halting progress until they detected the deadlock and recovered.

Fortunately, with assistance from the implementors, the AM library could be and was altered to avoid this problem.[6] However, the experience is illustrative

[6]The library had to be changed so as not to fragment messages into smaller chunks. When fragmented, the switches would observe fragment interarrival 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.

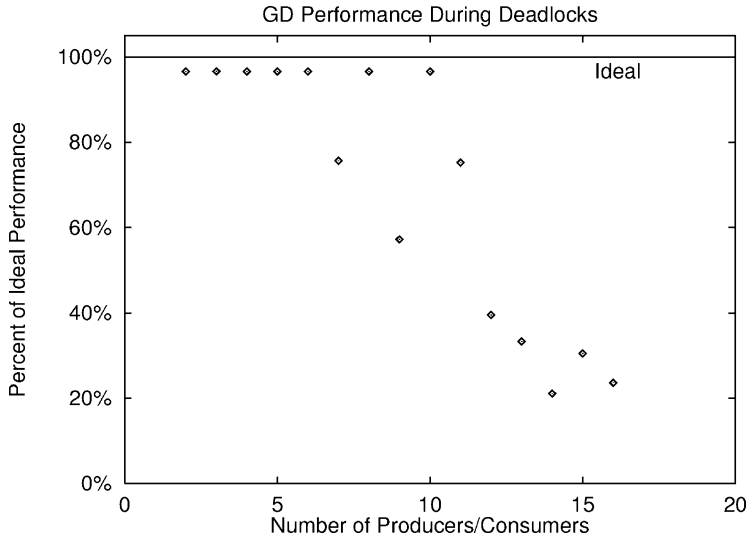

Fig. 18. **Switch deadlock:** performance of graduated declustering plotted under scale, in cases where the switch deadlock would occur. In the graph, the number of producers and consumers is covaried along the x-axis, and the percent of ideal performance is plotted along the y-axis. Thus, at $x = 16$, there are 32 machines involved in the experiment. The deadlock only occurs at scale and higher rates, but when it does occur, performance drops noticeably.

of a more general point: River, and its method for run-time adaptation, relies on the global characteristics of the network. If and only if the network as a whole is performing as expected will the system be able to tolerate producer- and consumer-side performance faults. Global performance faults in the network have a global effect, and cannot be avoided by our mechanisms. In contrast, localized performance faults in the network, such as link contention or link performance-failure, would naturally be handled by the adaptive mechanisms of River.

Faults such as this one are indicative of the need for "design diversity" [Gray and Reuter 1993], also called *architectural heterogeneity*. This type of heterogeneity avoids the problems that occur when a collection of identical components suffers from an identical design flaw, by including components of different makes and manufacturers in a system. As Gray and Reuter [1993] state, heterogeneity is akin to having "a belt and suspenders, not two belts or two suspenders." If an additional and different network had been available, it is likely that a graceful "performance fail-over" to that network could have avoided the Myrinet deadlock problem. Unfortunately, such solutions are quite costly.

8.2 Local Versus Global Perspective

Finally, we discuss a potential and general weakness with run-time adaptation in River. In this scenario, assume an application is writing records to disk through a DQ; if some disks are faster than others, the DQ will naturally allocate a proportional amount of data to those disks. If an application then reads all

of those data back from the disks, it will still obtain peak performance from the system. However, if the performance characteristics of one or more of the disks have changed (e.g., one of the previously high-performing disks is no longer high-performing), read performance will suffer. When data are written to disk, a *performance footprint* is created; if at some time in the future the state of the system no longer matches that footprint, a *performance mismatch* occurs, and delivered performance no longer approaches ideal.

One method that can be used to ameliorate this potential problem is replication. By replicating the data, we can then use GD to access them, and thus potentially tolerate unanticipated performance fluctuations. However, we are still left with the question of which data sets to replicate, and when to do so. This type of adaptation—rearranging data offline to account for the current characteristics of the system—is of a broader scale than the run-time mechanisms of River were designed to handle. We believe that complementary offline adaptive techniques, similar to those developed by Neefe et al. in Matthews et al. [1997], are required to provide a complete solution, and view the development of such techniques as one of the main goals of future work.

9. RELATED WORK

River draws on related work in the areas of parallel databases, parallel storage systems, parallel file systems, and parallel programming environments. We now discuss related work from each of these four areas in turn.

9.1 Parallel Databases

Large-scale I/O operations are common in parallel database systems. There are a number of parallel databases found in the literature, including Gamma [DeWitt et al. 1988], Volcano [Graefe 1990], the Digital Rdb prototype [Barclay et al. 1994], and Bubba [Copeland et al. 1988]. Many of these systems are based on techniques that are similar to the dataflow model of River, where parallel queries are described as a directed graph that connects different sequential data operators.

Gamma. Gamma is a parallel database system developed at Wisconsin [DeWitt et al. 1988]. The initial prototype was developed for a shared-nothing cluster: 20 VAX 11/750 processors, each with 2 MB of main memory, connected via a 10 MB/s token-ring network. Eight of those machines had identical 160 MB hard drives attached [DeWitt et al. 1986].

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.

In contrast to River, 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 in the group. Furthermore, the network that connects the machines is a shared medium, in this

case, a token-ring network. Thus, because aggregate network bandwidth does not scale with processors, data cannot be easily moved through the cluster for remote consumption, one of the fundamental tenets of the River design.

Volcano. Another prominent parallel database system in the literature is Volcano [Graefe 1989, 1990; Wolniewicz and Graefe 1993]. 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 and also in contrast to River, Volcano makes use of solely nonrobust distribution techniques such as hash-partitioning, range-partitioning, round-robin, and replication. No flexible distribution mechanism such as a DQ is available.

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 are 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* message layers versus *push-based* ones [Karamcheti and Chien 1995].

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. A shared-memory machine is an excellent platform for a River-like system, as interconnect performance is usually quite good.

Digital Rdb. In work on a parallel-load prototype for the Digital Rdb project, Barclay et al. [1994] describe another dataflow execution environment. Connections among 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 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)" [Barclay et al. 1994, p. 2].

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" [Barclay et al. 1994, p. 7]. A flexible method of distribution such as that found in River would be a useful addition.

*Parallel DB*2. River takes advantage of unordered record processing whenever the DQ is used; another example of such a system to provide some form of run-time adaptation is the IBM DB2 for SMPs [Lindsey 1998]. 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. The River DQ can be viewed as a distributed implementation of the same concept.

ParSets. The notion of applying operations on a data set in parallel has been explored with ParSets [DeWitt et al. 1994]. In this object-oriented database system, an application developer could create a function and subsequently direct the system to apply it to all objects in a collection. This model of computation would allow for great flexibility in building a performance robust system, *if* the data are replicated, much as GD provides robust access to replicated data storage. However, the ParSets implementation processes the function statically at each data site, and thus does not dynamically balance load and avoid the ill-effects of 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 [1997] said of the TeraData system, "The performance is bad, but it never gets worse." Consistency and scalability are the goals of the system, perhaps at the cost of getting the best use of the underlying hardware. In contrast, River attempts to deliver the best performance of the current configuration; thus, if the system is not stable, the performance of River-based applications will fluctuate.

Eddies. Finally, Eddies are an adaptive dataflow environment built on top of the River system [Avnur and Hellerstein 2000]. Eddies take some of the adaptive ideas within River a step further by reordering data operators on-the-fly in order to achieve higher levels of performance. Specifically, by monitoring which selection or join predicates are more highly selective, Eddies can adapt the dataflow to place more highly selective operators first, and thus reduce the total amount of work performed by the system.

9.2 Storage Systems

RAID. Redundant arrays of inexpensive disks (RAIDs) are a popular way to organize collections of multiple disks [Gibson 1992; Katz et al. 1989; Patterson et al. 1988b]. The idea is quite simple: aggregate a set of less-expensive disks behind a block-level interface. Commonly, some amount of this storage is used to circumvent failures via a variety of redundancy mechanisms; see Chen et al. [1994] 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 *performance fragile*: every entity must perform as expected for global performance to match expectations, and too many performance assumptions are made of each

disk. Some recent work addresses static performance heterogeneity in RAID systems [Cortes and Labarta 2001]; however, if the relative performance rates of the drives change, the same performance problems will occur.

Petal. Petal is a distributed system that also exports a block-level interface [Lee and Thekkath 1996]. 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 form of run-time adaptation, similar to GD in spirit. 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" [Lee and Thekkath 1996, p. 5]. GD performs a similar balancing, but with the additional element that each disk server services requests in a biased fashion, optimizing global application progress. Furthermore, 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 [Hsiao and DeWitt 1990]. In typical mirrored systems, replication is naive, as blocks on one disk are mirrored identically upon another. When a failure occurs, the surviving disk in a pair becomes overloaded. Chained declustering avoids this problem by spreading the replica blocks over many disks, thus balancing load under read-intensive workloads. In some ways, GD is a generalization of chained declustering; chained declustering works well in the case of an absolute failure of a single disk, whereas GD works well when there is a performance failure of a single disk.

Active Disks. A recent trend in storage systems allows for some or all of computation to be moved to the disks themselves [Acharya et al. 1998; Riedel et al. 1998]. These "active" disk systems are a perfect environment for River, as the same problems encountered within clusters are likely to be encountered therein (indeed, Acharya's stream-based programming model is quite similar to the River dataflow model, and therefore extending it with adaptive mechanisms such as the DQ and GD would be straightforward). However, additional adaptation techniques may be required, to allow for the dynamic migration of computation from the host processor(s) to the disks, depending on the current system load and network performance levels [Amiri et al. 2000].

9.3 Parallel File Systems

We now turn our attention to the large body of work in parallel file systems. Most systems have focused on extracting high performance from a set of uniform disks, including PPFS [Huber et al. 1995], Bridge [Dibble et al. 1988],

Panda [Seamons and Winslett 1996], Galley [Nieuwejaar and Kotz 1996], Vesta [Corbett and Feitelson 1996], Swift [Cabrera and Long 1991], CFS [Nitzberg 1992], SFS [LoVerso et al. 1993], the SIO specification [Bershad et al. 1994], and SPIFFI [Freedman et al. 1996]. 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 [Nitzberg 1992] and SPIFFI [Freedman et al. 1996]. 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, similar in spirit to GD. However, shared-file pointers only provide these properties for sequentially read files, 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* [Kotz 1994] (also referred to as *disk-directed I/O*); a similar concept is expressed with two-phase I/O [Choudhary et al. 1994]. In his 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, and thus they do not solve the performance problems we believe are common in clustered systems.

Panda. Of all the systems discussed, Panda [Kuo et al. 1999; Seamons and Winslett 1996] 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. Furthermore, its approach uses an a priori static measurement of disk performance to calculate how to lay out data across disks. Thus, if performance during the write changes, their system will not properly react until the next round of measurement. In contrast, River applications make decisions dynamically as to the state of drive performance, and thus can handle changes in performance during run-time.

9.4 Parallel and Distributed Programming Environments

Finally, there have been many parallel programming environments that have exploited the benefits of run-time adaptation. Some examples include Cilk [Blumofe et al. 1995], Lazy Threads [Goldstein et al. 1996], and Multipol

[Chakrabarti et al. 1995]. All of these systems dynamically balance load across consumers in order to facilitate the programming of highly irregular, fine-grained parallel applications.

Cilk. Cilk [Randall 1998] 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. Conceptually, stealing work from a work queue is quite similar to load balancing within the DQ, although the Cilk implementation is tuned for thread-level work stealing, whereas the DQ is aimed at high-performance data movement. One contribution of note of the Cilk system is that the authors have proven that the Cilk 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 [Chakrabarti et al. 1995]. Load balancing is provided via a distributed task queue [Wen 1996], which is similar in design but not implementation to the DQ.

Linda. Linda provides a shared, globally addressable, tuple-space to parallel programs [Carriero 1987; Gelernter et al. 1985]. Applications can perform atomic actions on tuple-space, inserting tuples, and then querying the space to find records with certain attributes. The tuple space is similar to but more general than the DQ, and because of the generality of this model, high performance in distributed environments has been shown to be difficult to achieve [Bal et al. 1992].

Reliable Multicast. Finally, Birman et al. [1999] encountered similar problems with "performance faulty" nodes in their research on reliable multicasting. In their work, they alter the guarantee provided by their multicast infrastructure, from an absolute guarantee to a probabilistic one, and thus avoid the ill-effects of a stuttering node. We pursue similar goals, but instead sometimes exploit application flexibility to obtain robust performance.

10. CONCLUSIONS

The heart of the River system is run-time adaptation. No component in the system statically trusts the performance of any other component; instead, each node constantly gauges the performance of others during data transfers, and allocates data or requests to nodes in proportion to their perceived performance. Both the DQ and GD are built with this philosophy in mind and, as we have demonstrated within this article, both are robust data-transfer mechanisms, delivering nearly ideal performance under a range of perturbation scenarios.

There are several keys to run-time adaptation that we have derived. First, the interaction with the communication layer is of utmost importance; the number of flow control credits provided must scale with the size of the system. Second, excess parallelism is needed in order to overcome the potential problems of extremely poorly performing components; without it, performance will be dictated by the slow component in the system. Third, local data processing must be guided by global knowledge of progress; GD has always had this property, and we have also found that is necessary for the DQ. Fourth and final, slack is needed to allow the run-time adaptive methods to deliver 100% of peak performance even in the presence of some small number of performance faults; how much slack a given system should have remains an open question.

Applications built within the River framework can make use of the two primitives in order to be robust to disk performance faults. A suite of six database query-processing primitives all run within 89% of ideal across a broad range of disk performance faults. However, sometimes the needs of the applications are not perfectly met by the system, as was demonstrated by the Top-N query and the external sort.

We have also uncovered some weaknesses in the River approach. Run-time adaptive methods such as the DQ and GD both rely strongly on the network as the backplane for adaptation; if the entire network does not function properly (e.g., all the switches deadlock), performance will not match expectations. There are also cases where run-time adaptation is too short-sighted; we plan to investigate the complementary use of long-term adaptation in order to eventually build a fully adaptive system.

From a methodological point of view, we believe that the combination of modeling, simulation, and implementation is crucial in understanding system behavior. With simulations, we were able to study the DQ and GD in isolation and in a well-controlled setting, allowing us to focus on important properties such as flow control. After understanding how the algorithms should behave, the second-generation implementation of the distributed algorithms proceeded with ease. Through implementation, we were able to find limitations in the system that did not arise in simplified simulations, underscoring the importance of building a working prototype. Even with relatively simple models, we were able to better gauge absolute performance under faults. Understanding the performance of a complex adaptive system is made easier when one understands the potential ideal.

ACKNOWLEDGMENTS

We thank Eric Anderson and Noah Treuhaft, who contributed to many of the ideas and much of the implementation of the River system. Also, we thank Andrea Arpaci-Dusseau, for her excellent help in all aspects of this work. We thank Dave Patterson, David Culler, and Joe Hellerstein, who all made substantial contributions to the ideas presented within this article, and Jim Gray, for his excellent feedback, direction, and advice. Finally, we thank the anonymous referees, whose careful feedback greatly improved the substance and style of this article.

REFERENCES

ACHARYA, A., UYSAL, M., AND SALTZ, J. 1998. Active disks. In *Proceedings of the Eighth Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII*, San Jose, Calif.).

ADLER, M., CHAKRABARTI, S., MITZENMACHER, M., AND RASMUSSEN, L. 1995. Parallel randomized load balancing. In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing (STOC '95)*, ACM, New York, 238–247.

AMD. 2000. AMD Athlon processor architecture. Available at www.amd.com.

AMIRI, K., PETROU, D., GANGER, G. R., AND GIBSON, G. A. 2000. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the 2000 USENIX Annual Technical Conference* (San Diego), 307–322.

ANDERSON, T. E., CULLER, D. E., PATTERSON, D. A., AND THE NOW TEAM. 1995. A case for NOW (networks of workstations). *IEEE Micro 15,* 1 (February), 54–64.

ARPACI, R. H., DUSSEAU, A. C., VAHDAT, A., LIU, L. T., ANDERSON, T., AND PATTERSON, D. 1995. The interaction of parallel and sequential workloads on a network of workstations. In *Proceedings of the 1995 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Ottawa), 267–278.

ARPACI-DUSSEAU, A. C. 2001. Implicit coscheduling: Coordinated scheduling with implicit information in distributed system *ACM Trans. Comput. Syst. (TOCS) 19*, 3 (August), 283–331.

ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., CULLER, D. E., HELLERSTEIN, J. M., AND PATTERSON, D. 1997. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD Conference on the Management of Data (SIGMOD '97*, Tucson).

ARPACI-DUSSEAU, R. H. 1999. Performance availability for networks of workstations. PhD Thesis, University of California, Berkeley.

ARPACI-DUSSEAU, R. H. AND ARPACI-DUSSEAU, A. C. 2001. Fail-stutter fault tolerance. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII*, Schloss Elmau, Germany), 33–38.

ARPACI-DUSSEAU, R. H., ANDERSON, E., TREUHAFT, N., CULLER, D. E., HELLERSTEIN, J. M., PATTERSON, D., AND YELICK, K. 1999. Cluster I/O with River: Making the fast case common. In *Proceedings of the 1999 Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99*, Atlanta).

ARPACI-DUSSEAU, R. H., ARPACI-DUSSEAU, A. C., CULLER, D. E., HELLERSTEIN, J. M., AND PATTERSON, D. 1998. The architectural costs of streaming I/O: A comparison of workstations, clusters, and SMPs. In *Proceedings of High-Performance Computer Architecture (HPCA '98*, Las Vegas).

AVNUR, R. AND HELLERSTEIN, J. 2000. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD Conference on the Management of Data (SIGMOD '00*, Dallas).

BAL, H. E., KAASHOEK, M. F., AND TANENBAUM, A. S. 1992. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng. 18*, 3 (March), 190–205.

BARCLAY, T., BARNES, R., GRAY, J., AND SUNDARESAN, P. 1994. Loading databases using dataflow parallelism. *SIGMOD Record (ACM SIG Manage. Data) 23*, 4 (Dec.), 72–83.

BARU, C. K., FECTEAU, G., GOYAL, A., HSIAO, H.-I., JHINGRAN, A., PADMANABHAN, S., COPELAND, G. P., AND WILSON, W. G. 1995. DB2 parallel edition. *IBM Syst. J. 34*, 2, 292–322.

BERSHAD, B., BLACK, D., DEWITT, D., GIBSON, G., LI, K., PETERSON, L., AND SNIR, M. 1994. Operating system support for high-performance parallel I/O systems. Tech. Rep. CCSF-40, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech.

BIRMAN, K. P. AND COOPER, R. 1991. The ISIS project: Real experience with a fault-tolerant programming system. *Oper. Syst. Rev. 25*, 2 (April), 103–107.

BIRMAN, K. P., HAYDEN, M., OZKASAP, O., XIAO, Z., BIDIU, M., AND MINSKY, Y. 1999. Bimodal multicast. *ACM Trans. Comput. Syst.* (*TOCS*) *17*, 2 (May), 41–88.

BLELLOCH, G., LEISERSON, C., AND MAGGS, B. 1991. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the Symposium on Parallel Algorithms and Architectures* (Hilton Head, S.C.).

BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU., Y. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth Symposium on Principles and Practice of Parallel Programming* (Santa Barbara, Calif.).

BODEN, N. J., COHEN, D., FELDERMAN, R. E., KULAWIK, A. E., SEITZ, C. L., SEIZOVIC, J. N., AND SU, W.-K. 1995. Myrinet—A gigabit-per-second local-area network. *IEEE Micro 15*, 1 (Feb.), 29–38.

BOLOSKY, W. J., III, J. S. B., DRAVES, R. P., FITZGERALD, R. P., GIBSON, G. A., JONES, M. B., LEVI, S. P., MYHRVOLD, N. P., AND RASHID, R. F. 1996. The Tiger Video Fileserver. Tech. Rep. 96-09, Microsoft Research.

BORAL, H., ALEXANDER, W., CLAY, L., COPELAND, G., DANFORTH, S., FRANKLIN, M. J., HART, B. E., SMITH, M., AND VALDURIEZ, P., 1990. Prototyping Bubba, a highly parallel database system. *IEEE Trans. Knowl. Data Eng. 2*, 1 (March), 4–24.

BORG, A., BLAU, W., GRAETSCH, W., HERRMANN, F., AND OBERLE, W. 1989. Fault tolerance under UNIX. *ACM Trans. Comput. Syst. 7*, 1 (Feb.), 1–24.

BRESSOUD, T. C. AND SCHNEIDER, F. B. 1995. Hypervisor-based fault tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95*, Copper Mountain Resort, Colo.).

BREWER, E. A. 1997. The Inktomi Web search engine. Invited talk. In *Proceedings of ACM SIGMOD Conference*.

BREWER, E. A. AND KUSZMAUL, B. C. 1994. How to get good performance from the CM-5 data network. In *Proceedings of the 1994 International Parallel Processing Symposium* (Cancun).

CABRERA, L.-F. AND LONG, D. D. E. 1991. Swift: Using distributed disk striping to provide high I/O data rates. *Comput. Syst. 4*, 4 (Fall), 405–436.

CARRIERO, N. J. 1987. Implementation of tuple space. PhD Thesis, Department of Computer Science, Yale University.

CHAKRABARTI, S., DEPRIT, E., IM, E.-J., JONES, J., KRISHNAMURTHY, A., WEN, C.-P., AND YELICK, K. 1995. Multipol: A distributed data structure library. Tech. Rep. CSD-95-879, University of California, Berkeley, July.

CHEN, J. B. AND BERSHAD, B. N. 1993. The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93*, Asheville, N.C.), 120–133.

CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. 1994. RAID: High-performance, reliable secondary storage. *ACM Comput. Surv. 26*, 2 (June), 145–185.

CHOUDHARY, A., BORDAWEKAR, R., HARRY, M., KRISHNAIYER, R., PONNUSAMY, R., SINGH, T., AND THAKUR, R. 1994. PASSION: Parallel and scalable software for input-output. Tech. Rep. SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September.

CODD, E. F. 1970. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June), 377–387.

COPELAND, G., ALEXANDER, W., BOUGHTER, E., AND KELLER, T. 1988. Data placement in Bubba. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, Chicago, 99–108.

CORBETT, P. F. AND FEITELSON, D. G. 1996. The Vesta parallel file system. *ACM Trans. Comput. Syst. 14*, 3 (August), 225–264.

CORTES, T. AND LABARTA, J. 2001. Extending heterogeneity to RAID level 5. In *Proceedings of the 2001 USENIX Annual Technical Conference* (Boston).

DEWITT, D. J. AND GRAY, J. 1992. Parallel database systems: The future of high-performance database systems. *Commun. ACM 35*, 6 (June), 85–98.

DEWITT, D. J., GERBER, R. H., GRAEFE, G., HEYTENS, M. L., KUMAR, K. B., AND MURALIKRISHNA, M. 1986. GAMMA: A high performance dataflow database machine. Tech. Rep. TR-635, Dept. of Computer Science, Univ. of Wisconsin-Madison, March.

DEWITT, D. J., GHANDEHARIZADEH, S., AND SCHNEIDER, D. A. 1988. A performance analysis of the gamma database machine. *SIGMOD Record (ACM SIG Manage. Data) 17*, 3 (September), 350–360.

DEWITT, D. J., NAUGHTON, J., SHAFER, J., AND VENKATARAMAN, S. 1994. Parsets for parallelizing OODBMS traversals: Implementation and performance. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*. IEEE Computer Society, Austin, Texas, 111–120.

DEWITT, D. J., NAUGHTON, J. F., AND SCHNEIDER, D. A. 1991. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the International Conference on Parallel and Distributed Information Systems* (Miami Beach).

DIBBLE, P., SCOTT, M., AND ELLIS, C. 1988. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems* (San Jose, Calif.), 154–161.

ENGLERT, S., GRAY, J., KOCHER, T., AND SHAH, P. 1990. 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* (Boulder, Colo.), 245–246.

FOX, A., GRIBBLE, S. D., CHAWATHE, Y., BREWER, E. A., AND GAUTHIER, P. 1997. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97*, Saint-Malo, France), 78–91.

FREEDMAN, C. S., BURGER, J., AND DEWITT, D. J. 1996. SPIFFI—A scalable parallel file system for the Intel Paragon. *IEEE Trans. Parallel Distrib. Syst. 7*, 11 (Nov.), 1185–1200.

GELERNTER, D., CARRIERO, N., CHANDRAN, S., AND CHANG, S. 1985. Parallel programming in Linda. In *Proceedings of the International Conference on Parallel Processing (ICPP '85*, St. Charles, Ill.), 255–263.

GIBSON, G. A. 1992. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. An ACM Distinguished Dissertation 1991. MIT Press, Cambridge, Mass.

GOLDSTEIN, S. C., SCHAUSER, K. E., AND CULLER, D. E. 1996. Lazy threads: Implementing a fast parallel call. *J. Parallel Distrib. Comput. 37*, 1 (August), 5–20.

GRAEFE, G. 1989. Volcano: An extensive and parallel dataflow query processing system. Tech. Rep., Oregon Graudate Center, June.

GRAEFE, G. 1990. Encapsulation of parallelism in the Volcano query processing system. *SIGMOD Record (ACM SIG Manage. Data) 19*, 2 (June), 102–111.

GRAY, J. 1997. What happens when processors are infinitely fast and storage is free? Invited talk. In *Proceedings of IOPADS*.

GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco.

GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. 2000. Scalable, distributed data structures for internet service construction. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00*, San Diego).

GROCHOWSKI, E. 1999. Emerging trends in data storage on magnetic hard disk drives. *Datatech 2*, 2 (Sept.), 11–16.

HSIAO, H.-I. AND DEWITT, D. 1990. Chained declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of the Sixth International Data Engineering Conference* (Los Angeles), 456–465.

HUBER, J., ELFORD, C. L., REED, D. A., CHIEN, A. A., AND BLUMENTHAL, D. S. 1995. PPFS: A high performance portable parallel file system. In *Proceedings of the Ninth ACM International Conference on Supercomputing* (Barcelona), 385–394.

JOHNSON, T. 1995. Designing a distributed queue. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing* (San Antonio, Tex.), 304–11.

KARAMCHETI, V. AND CHIEN, A. A. 1995. 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* (Santa Margherita Ligure, Italy), 298–307.

KATZ, R. H., GIBSON, G. A., AND PATTERSON, D. A. 1989. Disk system architectures for high performance computing. *Proc. IEEE 77*, 12 (Dec.), 1842–1858.

KOTZ, D. 1994. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation* (Monterey, Calif.), 61–74.

KUO, S., WINSLETT, M., CHO, Y., LEE, J., AND CHEN, Y. 1999. Efficient input and output for scientific simulations. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, ACM Press, Atlanta, 33–44.

KUSHMAN, N. A. 1998. Performance nonmonotonocities: A case study of the UltraSPARC processor. MS Thesis, Massachusetts Institute of Technology, Boston.

LEE, E. K. AND THEKKATH, C. A. 1996. Petal: Distributed virtual disks. In *Proceedings of the Seventh Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII*, Cambridge, Mass., 84–92.

LINDSEY, B. 1998. SMP intra-query parallelism in DB2 UDB. Database Seminar at U.C. Berkeley.

LISKOV, B. 1988. Distributed programming in Argus. *Commun. ACM 31*, 3 (March), 300–312.

LITZKOW, M., TANNENBAUM, T., BASNEY, J., AND LIVNY, M. 1997. Checkpoint and migration of UNIX processes in the condor distributed processing system. Tech. Rep. 1346, University of Wisconsin-Madison Computer Sciences, April.

LORIE, R., DAUDENARDE, J., HALLMARK, G., STAMOS, J., AND YOUNG, H. 1989. Adding intra-transaction parallelism to an existing DBMS: Early experience. *IEEE Data Eng. Newslett. 12*, 1 (March).

LOVERSO, S. J., ISMAN, M., NANOPOULOS, A., NESHEIM, W., MILNE, E. D., AND WHEELER, R. 1993. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Technical Conference* (Cincinnati), 291–305.

MAINWARING, A. AND CULLER, D. 1996. Active message applications programming interface and communication subsystem organization. Tech. Rep. CSD-96-918, University of California at Berkeley, October.

MATTHEWS, J. N., ROSELLI, D., COSTELLO, A. M., WANG, R. Y., AND ANDERSON, T. E. 1997. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97*, Saint-Malo, France), 238–251.

METER, R. V. 1997. Observing the effects of multi-zone disks. In *Proceedings of the 1997 USENIX Conference* (Anaheim, Calif.).

NIEUWEJAAR, N. AND KOTZ, D. 1996. The Galley parallel file system. In *Proceedings of the Tenth ACM International Conference on Supercomputing*. ACM Press, Philadelphia, 374–381.

NITZBERG, B. 1992. Performance of the iPSC/860 concurrent file system. Tech. Rep. RND-92-020, NAS Systems Division, NASA Ames, December.

OUSTERHOUT, J. K. 1982. Scheduling techniques for concurrent systems. In *Proceedings of the Third International Conference on Distributed Computing Systems* (Miami/Fort Lauderdale), 22–30.

OUSTERHOUT, J. K. 1990. Tcl: An embedable command language. In *Proceedings of the 1990 USENIX Association Winter Conference* (Washington, D.C.).

OUSTERHOUT, J. K. 1991. An X11 toolkit based on the Tcl language. In *Proceedings of the 1991 USENIX Association Winter Conference* (Dallas), 105–115.

PATTERSON, D., GIBSON, G., AND KATZ, R. 1988a. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88*, Chicago), 109–116.

PATTERSON, D. A., GIBSON, G. A., AND KATZ, R. H. 1988b. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Record (ACM SIG Manage. Data) 17*, 3 (Sept.), 109–116.

POOLE, J. T. 1994. Preliminary survey of I/O intensive applications. Tech. Rep. CCSF-38, scalable I/O initiative, Caltech Concurrent Supercomputing Facilities, Caltech.

RAGHAVAN, R. AND HAYES, J. 1991. Scalar-vector memory interference in vector computers. In *Proceedings of the 1991 International Conference on Parallel Processing* (St. Charles, Ill.), 180–187.

RANDALL, K. H. 1998. Cilk: Efficient multithreaded computing. PhD Thesis, Massachusetts Institute of Technology, Boston.

RIEDEL, E., GIBSON, G., AND FALOUTSOS, C. 1998. Active storage for large-scale data mining and multimedia. In *Proceedings of the VLDB* (New York).

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22,* 4 (Dec.), 299–319.

SEAMONS, K. E. AND WINSLETT, M. 1996. Multidimensional array I/O in Panda 1.0. *J. Supercomput. 10*, 2, 191–211.

SINGH, J. P., WEBER, W.-D., AND GUPTA, A. 1992. SPLASH: Stanford parallel applications for shared-memory. *Comput. Arch. News 20*, 1 (March), 5–44.

TALAGALA, N. AND PATTERSON, D. 1999. An analysis of error behaviour in a large storage system. In *Proceedings of the IPPS Workshop on Fault Tolerance in Parallel and Distributed Systems* (San Juan, Puerto Rico).

TANDEM PERFORMANCE GROUP. 1988. A benchmark of NonStop SQL on debit credit transaction. In *Proceedings of the SIGMOD International Conference on Management of Data* (Chicago).

TERADATA Corporation 1985. *DBC/1012 Data Base Computer System Manual*, release 2.0 ed. Teradata Corporation, document number c10-0001-02.

TREMBLAY, M., GREENLEY, D., AND NORMOYLE, K. 1995. The design of the microarchitecture of UltraSPARC-I. *Proc. IEEE 83*, 12 (Dec.), 1653–63.

VON EICKEN, T., BASU, A., BUCH, V., AND VOGELS, W. 1995. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, Colo.), 40–53.

WEN, C.-P. 1996. Portable library support for irregular applications. PhD Thesis, University of California, Berkeley. Tech. Rep. UCB/CSD-96-894.

WOLNIEWICZ, R. AND GRAEFE, G. 1993. Algebraic optimization of computations over scientific databases. In *Proceedings of VLDB '93* (Dublin), 13–24.

Received October 2001; revised June 2002; accepted July 2002