Easing the management of data-parallel systems via adaptation

David Petrou, Khalil Amiri, Gregory R. Ganger, and Garth A. Gibson
Carnegie Mellon University
{dpetrou,amiri+,ganger,garth}@cs.cmu.edu
http://www.pdl.cs.cmu.edu/

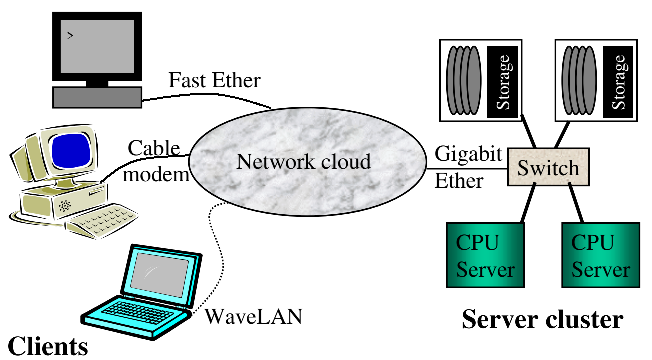

Figure 1: Users running data-parallel applications across the Internet. The data stores are on server clusters, which, compared to monolithic machines, flexibly support different kinds of concurrent workloads, are easier to upgrade, and have the potential to support independent node faults. Our client machines, in contrast to the traditional view, are active collaborators with the clusters in providing the end-result to the user.

1 Introduction and motivation

In recent years we have seen an enormous growth in the size and prevalence of data processing workloads [Fayyad 1998, Gray 1997]. The picture that is becoming increasingly common is depicted in Figure 1. In it, organizations or resourceful individuals provide services via a set of loosely-coupled workstation nodes. The service is usually some form of data-mining like searching, filtering, or image recognition. Clients, which could be machines running web browsers, not only initiate requests, but also partake in the processing, with the goal of reducing the request turnaround. That is, when the servers are overloaded, clients with spare cycles take some of the computational burden. Naturally, many aspects of such a system cannot be determined at design time. E.g., exactly how much work a client should do depends on the computational resources available at the client and server cluster, the network bandwidth unused between them, and the workload demand. This position paper is interested in this and other aspects that must be divined at run-time to provide high performance and availability in data-parallel systems.

What makes system tuning especially hard is that it's not possible to find the right knob-settings once and for all. A system upgrade or component failure may change the appropriate degree of data-parallelism. Changes in usable bandwidth may ask for a different partitioning of code among the client and server cluster. Moreover, an application may go through distinct phases during its execution. We should checkpoint the application for fault-tolerance less often during those phases in which checkpointing takes longer. Finally, the system needs to effectively allocate resources to concurrent applications, which can start at any time and which benefit differently from having these resources. In summary, we argue that in the future a significant fraction of computing will happen on architectures like Figure 1, and that, due to the architectures' inherent complexity, high availability and fast turnaround can only be realized by dynamically tuning a number of system parameters.

Our position is that this tuning should be provided automatically by the system. The contrasting, application-specific view, contends that, to the extent possible, policies should be made by applications since they can make more informed optimizations. However, this requires a great deal of sophistication from the programmer. Further, it requires programmer time, one of the most scarce resources in systems building today.

Toward our goal, we contribute a framework that is sufficiently rich to express a variety of interesting data-parallel applications, but which is also restricted enough so that the system can tune itself. These applications are built atop the ABACUS migration system, whose object placement algorithms are extended to reason about how many nodes should participate in a data-parallel computation, how to split up application objects among a client and server cluster, how often program state should be checkpointed, and the interaction (sometimes conflicting) between these questions. By automatically determining a number of critical parameters at run-time, we are minimizing the management costs which have in recent years given system administrators the howling fantods [Satyanarayanan 1999].

2 Background and related work

Before presenting our approach for building high performance and fault-tolerant data processing systems out of clients and server clusters, we describe some background and most relevant related work.

Data-parallelism A significant fraction of data processing applications are *data-parallel;* that is, they can be divided into a parallelizable scanning stage and a centralized merging stage. We call the implementation of these stages scanning and merging objects. The scanning object is replicated onto many machines and each instantiation accesses data from storage nodes. In many cases, scanning objects execute directly on storage nodes. The scanners read and transform part of the dataset and send intermediate results to a single merger. The merging object aggregates these partial results into a final summary for the user.

The Apriori algorithm, which discovers association rules in sales transactions, is an example data-parallel application [Agrawal & Srikant 1994]. It finds the most common items purchased, then the most common pairs of items purchased, and so on. For each successive phase, the merger constructs a list of candidates using the results from the previous phase. These candidates are communicated to scanning objects that count the number of times they occur in the dataset. Finally, the scanners report their results to the merging object and the process repeats. Further examples of applications that can be organized in a similar manner are nearest neighbor search, edge detection, and image registration [Riedel et al. 1998]. Piranha is an example system that supports dynamic replication of tasks atop a Linda tuple-space [Carriero et al. 1993].

For data-parallel applications, performance is affected by the number of participating nodes in the server cluster (also called the degree of data-parallelism) and whether the merging object runs on a node in the server cluster or on the client. The right answers depend on run-time conditions and are explored later in this paper.

Fault-tolerance Utilizing many nodes in parallel increases performance at the cost of reliability. The chance of any one node failing may be small, but when a computation depends on a set of nodes, the probability of an application failing before it finishes becomes quite real. The redundancy in RAID [Patterson et al. 1988] makes data resilient to storage device failure, but in our proposed environment, we must also protect the computation spread among clients and servers.

The standard approach to providing transparent fault-tolerance is to take checkpoints of the task and resume processing from a checkpoint on failure. Unfortunately, rolling back a failed node can confuse other elements of a distributed system. Consider what happens if a node has received some messages from another node which then fails. After the failed node is rolled back, the other node is in a state that causally depends on lost computation, making it an *orphan*, and the system is now inconsistent. Using checkpoints to efficiently reach a consistent global state after failure, that is, with no orphans, is hard. A potentially inefficient solution is to take individual node checkpoints synchronously, and then to roll back all nodes to the same checkpoint on failure. To avoid the overhead of synchronizing the nodes, more complex algorithms have been devised for asynchronous checkpointing [Strom & Yemini 1985]. Other work goes further by allowing even recovery to proceed asynchronously [Smith et al. 1995].

Because many data-parallel programs are structured as an object hierarchy with many of those objects stateless, we implement the most simple approach to fault-tolerance without significantly degrading performance. That is, we synchronously checkpoint only the stateful objects, and on failure, roll back all of these objects to the same checkpoint. A similar simplification was implemented for tolerating faults in the Orca system [Kaashoek et al. 1992].

But how often should these checkpoints be taken? We can only answer intelligently after knowing the expected time to take a checkpoint and to recover after a failure, and the probability of a failure occurring. Further, there is a confounding interaction between the degree of parallelism and the frequency of checkpoints. Limiting the number of nodes with scanning objects lowers the probability of the application failing, but also reduces its performance, and vice versa.

The ABACUS **migration system** This work is part of the continuing evolution of the ABACUS migration system [Amiri et al. 2000]. ABACUS began when we were designing a filesystem for Network-Attached Secure Disks (NASDs) [Gibson et al. 1998] and realized that we could achieve higher performance and flexibility by letting parts of the filesystem move between clients and NASDs at runtime. We soon extended our approach, which we now briefly describe, to general purpose data-intensive applications, treating the filesystem as an example of, or extension to, an application.

ABACUS consists of a programming model and run-time system as shown in Figure 2. Programmers compose data-intensive applications from a number of objects. ABACUS is language-neutral, in contrast to providing rich language or compiler support as demonstrated by seminal work such as Emerald [Jul et al. 1988]. Our approach is still effective because of the difference in communication granularity between fine-grained application objects and those objects that make up data-intensive applications.

An application is represented by a graph of communicating objects which is rooted at the client by a console object and at storage nodes by storage objects. The storage objects provide persistent storage, while the console object contains the part of the application that must remain at the node where the application started. All of the objects between the console and storage objects can migrate between nodes. The run-time sys-

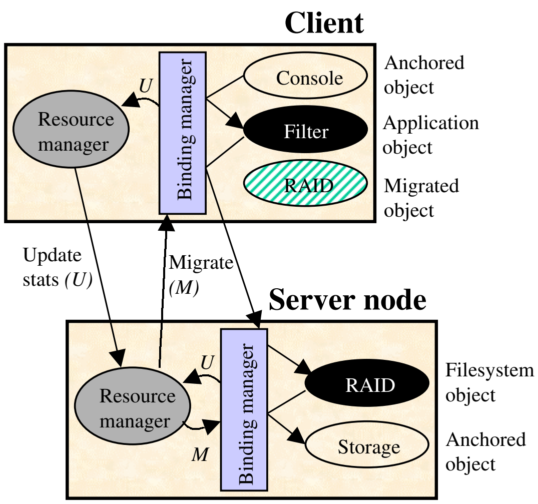

Figure 2: The architecture of the ABACUS migration system. An application is broken up into a number of migratable objects. Calls between the objects are redirected by binding managers that know where the objects are currently running. Binding managers also inform resource managers about the resources consumed by the objects. The resource managers use this information to reevaluate how objects are partitioned among clients and servers.

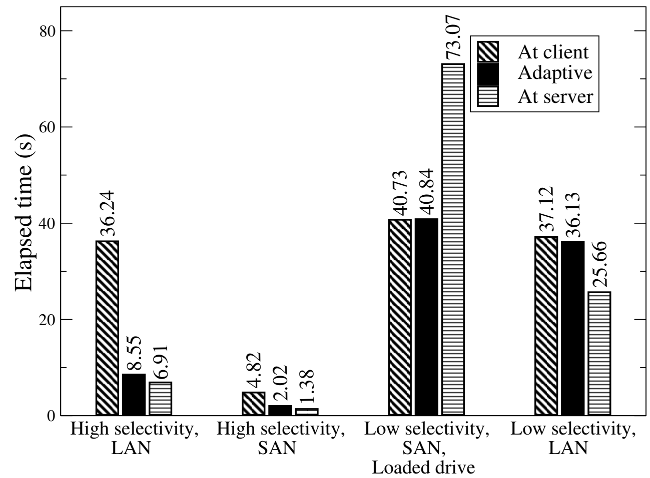

Figure 3: Filtering on a client, on a server, or with ABACUS deciding where. Each bar represents the time to filter a 16 MB file. Highly selective filters, which throw away most of the data, are correctly migrated to the server by ABACUS. Because the experiment finishes quickly when the client and server are connected by a SAN network, our improvement over bad placement is more pronounced on the slower LAN. In the third set of numbers, the storage node is loaded down so that it can't effectively run the filter, making client filtering preferable. In the last case, the difference between client- and server-side filtering wasn't deemed sufficiently significant by ABACUS to warrant migration.

tem invokes object-provided checkpoint and restore methods after deciding to relocate objects.

Objects expose methods that can be called by other objects. The console object makes calls to the union of all of the objects' exposed methods. A call trickles down the object hierarchy until it reaches the first object that implements it. Now this object does some work and either makes additional calls down the hierarchy or returns a result immediately to the higher layers.

Object calls are redirected through *binding managers* that shunt the calls to wherever the next object in the hierarchy is currently running. While doing this, the binding managers also inform *resource managers* about the resources consumed by the objects, such as the rate at which data is moving through the hierarchy. The resource managers use this information along with a cost/benefit model to determine if any of the objects can be reshuffled among the nodes to provide better performance.

We show a small example application in Figure 3 confirming our approach. We run a filter on the same file twice, each time searching for something different. The first run throws away most of the file's data while the other keeps most of it. (Consider the difference between `grep kernel Bible.txt` and `grep kernel LinuxBible.txt`.) For the first, it makes sense to migrate the filter to the storage node to reduce network traffic. For the second, it's better to keep the filter at the client to offload busy servers that work on behalf of many clients.

Our first version of ABACUS lacked support for the more complex object hierarchies required by data-parallel applications and did not tolerate node faults. The rest of this position paper describes our ongoing work in developing these aspects.

Method	Description
`SetBlockParams()`	Set block size and other params
`GetNextBlock()`	Read the next sequential block
`GetAnyBlock()`	Read any unread block
`ReadRange()`	Read specific byte range
`WriteRange()`	Write (install) a dataset
`Checkpoint()`	Record object state to a buffer
`Restore()`	Recover state from a buffer

Table 1: The abridged API for writing data-parallel applications in ABACUS.

3 Data-parallelism

ABACUS presents the API shown in Table 1 for writing data-parallel applications to the programmer. The scan/merge structure of such an application is shown in Figure 4.

The data that an application acts on is organized into a number of files. Each file may reside entirely on one storage node, or may span many nodes, and exactly which set of nodes may change during the execution of the application. The API abstracts away the underlying data layout so that the programmer is unaware on which or how many nodes individual data records within its files are stored. This both simpli-

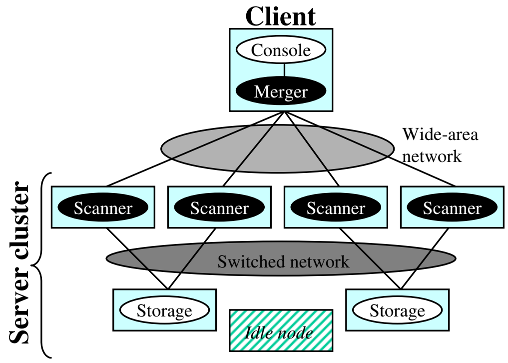

Figure 4: The decomposition of a data-parallel application into scanning and merging objects. The dataset is stored across two storage nodes. The scanning object is computationally expensive so there are more scanning nodes than storage nodes. When the scanners aren't costly, ABACUS would configure the system to look like one of the examples in Figure 5.

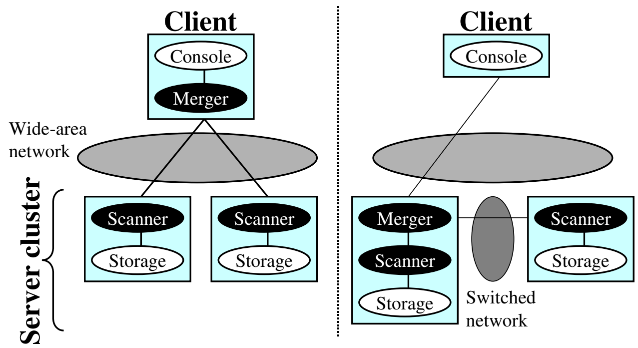

Figure 5: Adapting the merge object location. On the left, the well-connected client is running the merge object, offloading this work from the contended server cluster. On the right, the client has relatively poor connectivity, and to avoid this "last-hop" network bottleneck, ABACUS migrates the merge object down to one of the nodes in the server cluster.

fies the application programmer's job and enables ABACUS to change the data layout as external conditions warrant.

Scanning objects read records in a file by making `GetNextBlock()` or `GetAnyBlock()` requests until every application-specific block size of data is consumed (similar to dynamic sets [Steere 1997]). When the network is the bottleneck, we try to run scanners directly on the storage nodes and direct these requests only to the blocks on those storage nodes. Some applications need to access specific records in special cases or boundary conditions, and for these rarer instances, the scanners can issue `ReadRange()` requests to specific file offsets, which may or may not reside on the same device. For instances in which the data transformations applied by the scanners are computationally expensive, we may recruit extra scanning nodes that access data blocks at their original locations.

For our initial prototype, we either replicate each file on each storage node, or cut the dataset uniformly into pieces, with one piece residing on a single node. Today we are exploring the replication and distribution of computation under the assumption that the data layout does not change very often. In fact, many of our target applications install their datasets infrequently and operate read-only to answer client queries. Future work will look at more elaborate ways of placing data and dynamically migrating data in response to access patterns. E.g., we'd like to incorporate some ideas from the River system which handles run-time load perturbations in write workloads [Arpaci-Dusseau et al. 1999].

The number of scanner nodes used by an application can range from one to the total number of nodes in the system. The appropriate number depends on what part of the system is the bottleneck. When the storage nodes don't have enough processing power for the scanners to keep up with their disks and buffer caches, then recruiting other nodes can enable the application to complete faster. However, there are diminishing returns to increasing the degree of parallelism. Eventually the network between scanners and storage nodes, or between the merger and scanners will be saturated. Also, scanning from fewer devices may be faster because we are multiplexing fewer TCP streams. Further, as described later, the more participating devices there are, the higher the cost for providing fault-tolerance.

The merging object, by default, begins on the client. But, in many cases, the client has poor connectivity whereas the nodes in the server cluster are connected via a high-speed switched network. When this is true and when there is a lot of traffic between the scanners and the merger, ABACUS will push the merge object from the client onto one of the server nodes, so that only the final, summary results are presented to the console object on the client. Both potential locations of the merge object are shown in Figure 5.

The bottleneck component of the system may change during the lifetime of the application, requiring ABACUS to reevaluate the degree of parallelism and location of the merge object. This can happen when the application enters a different phase, there are node failures or system upgrades, or other concurrent applications change their demands on the shared nodes and network resources.

We may determine that an application should involve more scanning nodes, but the other nodes in the cluster happen to be loaded down by other work. We resolve contention by assigning nodes to the application that benefits the most. Naturally, if an object in an application is only using a percentage of a node's resources, we can run objects from other applications on the node concurrently.

Some applications are structured so that they run as fast as the slowest scanning node. When a node such an application is using is loaded from something external (like a daemon), ABACUS will remove the node from the computation (equivalent to lowering the application's degree of parallelism) to speed execution.

ABACUS can only estimate that a different configuration will lead to better performance. Unfortunately, changing the degree of parallelism isn't free. Hence, ABACUS compares the estimated benefit to making a change vs. the cost in making it. The benefit is how much faster the application will run when its work has different CPU and network resources available to it, and the cost is the lost time waiting for objects to become quiescent and replicating or killing some objects.

4 Fault tolerance

Increasing the number of participating nodes in a data-parallel workload can improve performance, but it also increases the chance that a hardware fault kills the workload, compelling us to make ABACUS objects fault-tolerant. Here, we are concerned with recovering computation and assume that data is recoverable via well-known techniques. We are further only concerned with hardware, not software, faults. Our goal is to ensure that an application that was written to be fault-tolerant on a single node is no more vulnerable to faults when running atop the ABACUS migration system. We achieve this with no extra burden on the programmer, that is, with no increase in programming complexity.

As with other aspects of ABACUS, we implement fault-tolerance below the application, providing it transparently to any code written to the ABACUS interface. This interface requires objects within an application to provide their own checkpoint and restore methods[1] which were originally used in ABACUS for object migration. Fault-tolerance in ABACUS is transparent in the sense that the objects do not know when checkpoints are taken, when recovery is initiated, and do not actively participate in the recovery procedure.

Nodes take checkpoints now and then, and when we detect a failure, we restart the failed node's objects on another node. Recovering from a checkpoint assumes that the computation is deterministic. To tolerate one fault, each node should take periodic object checkpoints and save them on a "buddy" node. Various tricks can be played to reduce the amount of data consumed by checkpoints, such as computing and only storing the parity over the checkpoints [Plank 1996], or storing only checkpoint deltas, that is, what has changed since the last checkpoint. Since our data-parallelism design abstracts away data layout, we can restart failed computations on other nodes transparently to the programmer.

We observe that the structure of a data-parallel application is simple. Instead of a fully connected graph of stateful objects, we can often structure the application as many stateless scanning objects and one stateful merging object in a simple hierarchy. For these applications, orphans cannot arise and we only have to worry about the computational state of the merging objects[2]. Some applications may have stateful scanners and possibly more complex communication patterns. For them, we simply take a synchronized checkpoint of all the objects, and on failure, we roll them all back. This is not the most efficient approach, but it has a bounded rollback of one, and is simple to understand and implement. Here we are trading performance (checkpoint/recovery overhead) for simplicity (code correctness and maintainability).

When a node fails, the system chooses another node on which to resume the lost computation. If a scanner failed while running on a storage node, ABACUS will attempt to restart it on an idle node holding a replica of the dataset on the failed node. If no suitable spare exists, ABACUS will decrease the degree of parallelism for the application. A failed merge object will be restarted on a machine whose available resources are most similar to the failed node.

We assume fail-stop processors [Schneider 1983] and declare a node as failed if it doesn't respond after a timeout, or if network sockets to it are terminated unexpectedly. We detect when a node is being unresponsive via a watchdog thread that periodically sends pings down an object hierarchy from the console object. If a node responds after its timeout expires, we ignore its messages and kill the remaining work on it. For now, we assume that network partitions do not occur. For each application, the console object is responsible for initiating the recovery procedure when a failure is detected. Naturally, a console object does nothing if recovery operations for the failed node have already been initiated by another application's console object.

What we contribute is a system that dynamically reevaluates how often to take checkpoints. The more frequently you checkpoint, the faster you can recover from a failure. If you never checkpoint, recovery means starting from the beginning. But the act of checkpointing has an associated cost in time. We should checkpoint in a way to minimize the total execution time of the application, which involves estimates on how expensive it is to checkpoint objects within an application and recover them in case of failure, the application's lifetime, and the probability of a hardware failure. We predict the time to take future checkpoints based on the past. Before an object has been checkpointed, our estimate is based on how much memory it has allocated. We can estimate application lifetimes by either maintaining a database of past application runs, or in the absence of history, we can use validated heuristics for estimating remaining lifetime based on how long a program has already been running for [Harchol-Balter & Downey 1997]. Finally, we record and analyze when past failures have occurred to estimate the probability of another failure occurring. Some of these variables change over the duration of the application, so we need to continually reevaluate when to checkpoint.

[1] We note that requiring programmers to provide checkpoint and restore methods potentially lowers the cost to take object checkpoints, because an object will only checkpoint the necessary bits of data required to resume computation, unlike a generic approach that would save its entire address space.

[2] Console objects are made fault-tolerant by the application programmer. Recall that our goal is to ensure that already fault-tolerant applications remain immune to failures when their objects are run atop the ABACUS migration system.

5 Summary and future work

A concise restatement of our goal would be to simplify the programmers' and administrators' jobs in writing and maintaining distributed data-intensive applications. This is achieved by providing adaptivity below the application. The effort of reacting to run-time conditions is all in ABACUS so that the programmer can concentrate on getting the algorithmic parts of the application right, and the administrator needs only to be involved when extraordinary situations arise.

Philosophically, this is a turn away from the application-specific position. In this view, the system exposes much of the underlying system to applications for better performance and to enable more functionality [Anderson 1992]. For some cases, the application-specific approach can yield major benefits at the cost of requiring more work and more sophistication from the programmer who must possess intimate details of the system. For the workloads we are targeting, we argue that a generic monitoring and reconfiguring substrate can yield results comparable to a clever implementor with an ad hoc program. Further, centralized algorithms result in less code whose correctness must be trusted, and lets us explore the interactions among multiple applications in a way that independent, laissez faire policies cannot.

Our research is proceeding among a number of lines. We are extending the ABACUS core to understand the structure of data-parallel applications. This involves logic to choose an appropriate degree of parallelism and a good set of nodes on which to run the scanners and merger. It also entails a way to determine fault-tolerance parameters under changing conditions. At the same time, we're looking to port a few data-parallel applications to ABACUS and in the process we expect to refine our API. The applications should be easy to port because our API hides storage-level details from the programmer. An interesting question that we hope to soon answer is how effective can our approach be, which employs black-box resource monitoring and decision making, to this general class of applications.

Acknowledgments

We thank Dushyanth Narayanan for his input and our anonymous reviewers for their feedback.

References

[Agrawal & Srikant 1994] Agrawal, R. and Srikant, R. Fast algorithms for mining association rules. In *Proceedings of the 20th VLDB Conference*, Santiago, Chile, 1994.

[Amiri et al. 2000] Amiri, K., Petrou, D., Ganger, G. R., and Gibson, G. A. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, CA, June 2000. Available at http://www.cs.cmu.edu/~dpetrou/research.html.

[Anderson 1992] Anderson, T. E. The case for application-specific operating systems. In *Proceedings of the Third Workshop on Workstation Operating Systems*, Key Biscayne, FL, April 1992.

[Arpaci-Dusseau et al. 1999] Arpaci-Dusseau, R. H., Anderson, E., Treuhaft, N., Culler, D. E., Hellerstein, J. M., Patterson, D., and Yelick, K. Cluster I/O with river: Making the fast case common. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems (IOPADS-99)*, pp. 10–22, Atlanta, Georgia, May 5, 1999.

[Carriero et al. 1993] Carriero, N., Gelernter, D., Kaminsky, D., and Westbrook, J. Adaptive parallelism with Piranha. Technical Report 954, Yale University Department of Computer Science, February 1993.

[Fayyad 1998] Fayyad, U. Taming the giants and the monsters: Mining large databases for nuggets of knowledge. *Database Programming and Design*, 11(3), March 1998.

[Gibson et al. 1998] Gibson, G. A., Nagle, D. F., Amiri, K., Butler, J., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS 98)*, San Jose, CA, October 1998.

[Gray 1997] Gray, J. Building petabyte databases and storage metrics, March 5 1997. Talk given at Carnegie Mellon University, available from http://research.microsoft.com/~gray/.

[Harchol-Balter & Downey 1997] Harchol-Balter, M. and Downey, A. B. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, August 1997.

[Jul et al. 1988] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, January 1988.

[Kaashoek et al. 1992] Kaashoek, M. F., Michiels, R., Bal, H. E., and Tanenbaum, A. S. Transparent fault-tolerance in parallel Orca programs. In *Proceedings of the 3rd Symposium on Experiences with Distributed and Multiprocessor Systems*, pp. 297–312, Newport Beach, CA, March 1992.

[Patterson et al. 1988] Patterson, D. A., Gibson, G., and Katz, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pp. 109–116, Chicago, IL, June 1988.

[Plank 1996] Plank, J. S. Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, Ontario, Canada, October 1996.

[Riedel et al. 1998] Riedel, E., Gibson, G., and Faloutsos, C. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24th VLDB Conference*, New York City, New York, August 1998.

[Satyanarayanan 1999] Satyanarayanan, M. Digest of Proceedings, Seventh IEEE Workshop on Hot Topics in Operating Systems, March 1999.

[Schneider 1983] Schneider, F. B. Fail-stop processors. In *Proceedings of the IEEE Spring COMPCON*, pp. 66–70, San Francisco, CA, March 1983.

[Smith et al. 1995] Smith, S. W., Johnson, D. B., and Tygar, J. D. Completely asynchronous optimistic recovery with minimal rollbacks. In *The Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS '95)*, pp. 361–371, Los Alamitos, CA, June 1995.

[Steere 1997] Steere, D. C. Exploiting the non-determinism and asychrony of set iterators to reduce aggregate file I/O latency. In *16th ACM Symposium on Operating Systems Principles*, Saint Malo, France, October 1997.

[Strom & Yemini 1985] Strom, R. E. and Yemini, S. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.