Fail-Stutter Fault Tolerance

Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
Department of Computer Sciences, University of Wisconsin, Madison

Abstract

Traditional fault models present system designers with two extremes: the Byzantine *fault model, which is general and therefore difficult to apply, and the* fail-stop *fault model, which is easier to employ but does not accurately capture modern device behavior. To address this gap, we introduce the concept of* fail-stutter *fault tolerance, a realistic and yet tractable fault model that accounts for both absolute failure and a new range of performance failures common in modern components. Systems built under the fail-stutter model will likely perform well, be highly reliable and available, and be easier to manage when deployed.*

1 Introduction

Dealing with failure in large-scale systems remains a challenging problem. In designing the systems that form the backbone of Internet services, databases, and storage systems, one must account for the possibility or even likelihood that one or more components will cease to operate correctly; just how one handles such failures determines overall system performance, availability, and manageability.

Traditionally, systems have been built with one of two fault models. At one extreme, there is the *Byzantine failure* model. As described by Lamport: "The component can exhibit arbitrary and malicious behavior, perhaps involving collusion with other faulty components" [25]. While these assumptions are appropriate in certain contexts (*e.g.*, security), they make it difficult to reason about system behavior.

At the other extreme, a more tractable and pragmatic approach exists. Known as the *fail-stop* model, this more limited approach is defined by Schneider as follows: "In response to a failure, the component changes to a state that permits other components to detect a failure has occurred and then stops" [33]. Thus, each component is either working or not, and when a component fails, all other components can immediately be made aware of it.

The problem with the Byzantine model is that it is general, and therefore difficult to apply. The problem with the fail-stop model is that it is simple, and therefore does not account for modern device behavior. Thus, we believe there is a need for a new model – one that is realistic and yet still tractable. The fail-stop model is a good starting point for a new model, but it needs to be enhanced in order to account for the complex behaviors of modern components.

The main reason an enhancement is in order is the increasing complexity of modern systems. For example, the latest Pentium has 42 million transistors [21], and future hardware promises even more complexity with the advent of "intelligent" devices [1, 27]. In software, as code bases mature, code size increases, and along with it complexity – the Linux kernel source alone has increased by a factor of ten since 1994.

Increasing complexity directly affects component behavior, as complex components often do not behave in simple, predictable ways. For example, two identical disks, made by the same manufacturer and receiving the same input stream will not necessarily deliver the same performance. Disks are not the only purveyor of erratic performance; as we will discuss within this document, similar behavior has been observed in many hardware and software components.

Systems built under the "fail-stop illusion" are prone to poor performance when deployed, performing well when everything is working perfectly, but failing to deliver good performance when just a single component does not behave as expected. Particularly vulnerable are systems that make static uses of parallelism, usually assuming that all components perform identically. For example, striping and other RAID techniques [28] perform well if every disk in the system delivers identical performance; however, if performance of a single disk is consistently lower than the rest, the performance of the entire storage system tracks that of the single, slow disk [6]. Such parallel-performance assumptions are common in parallel databases [16], search engines [18], and parallel applications [12].

To account for modern device behavior, we believe there is a need for a new model of fault behavior. The model should take into account that components sometimes fail, and that they also sometimes perform erratically. We term the unexpected and low performance of a component a *performance fault*, and introduce the *fail-stutter* fault model, an extension of the fail-stop model that takes performance faults into account.

Though the focus of the fail-stutter model is component performance, the fail-stutter model will also help in building systems that are more manageable, reliable, and available. By allowing for plug-and-play operation, incremental growth, worry-free replacement, and workload modification, fail-stutter fault tolerant systems decrease the need for human intervention and increase manageability. Diversity in system design is enabled, and thus reliability is improved. Finally, fail-stutter fault tolerant systems deliver consistent performance, which likely improves availability.

In this paper, we first build the case for fail-stutter fault tolerance via an examination of the literature. We then discuss the fail-stutter model and its benefits, review related work, and conclude.

2 The Erratic Behavior of Systems

In this section, we examine the literature to document the many places where performance faults occur; note that this list is illustrative and in no means exhaustive. In our survey, we find that device behavior is becoming increasingly difficult to understand or predict. In many cases, even when erratic performance is detected and investigated, no cause is discovered, hinting at the high complexity of modern systems. Interestingly, many performance variations come from research papers in well-controlled laboratory settings, often running just a single application on homogeneous hardware; we speculate that component behavior in less-controlled real-world environments would likely be worse.

2.1 Hardware

We begin our investigation of performance faults with those that are caused by hardware. We focus on three important hardware components: processors and their caches, disks, and network switches. In each case, the increasing complexity of the component over time has led to a richer set of performance characteristics.

2.1.1 Processors and Caches

Fault Masking: In processors, fault masking is used to increase yield, allowing a slightly flawed chip to be used; the result is that chips with different characteristics are sold as identical. For example, the Viking series of processors from Sun are examined in [2], where the authors measure the cache size of each of a set of Viking processors via micro-benchmark. "The Single SS-51 is our base case. The graphs reveal that the [effective size of the] first level cache is only 4K and is direct-mapped." The specifications suggest a level-one data cache of size 16 KB, with 4-way set associativity. However, some chips produced by TI had portions of their caches turned off, whereas others, produced at different times, did not. The study measured application performance across the different Vikings, finding performance differences of up to 40% [2].

The PA-RISC from HP [35] also uses fault-masking in its cache. Schneider reports that the HP cache mechanism maps out certain "bad" lines to improve yield [34].

Fault-masking is not only present in modern processors. For example, the Vax-11/780 had a 2-way set associative cache, and would turn off one of the sets when a failure was detected within it. Similarly, the Vax-11/750 had a direct-mapped cache, and would shut off the whole cache under a fault. Finally, the Univac 1100/60 also had the ability to shut off portions of its cache under faults [37].

Prediction and Fetch Logic: Processor prediction and instruction fetch logic is often one of the most complex parts of a processor. The performance characteristics of the Sun UltraSPARC-I were studied by Kushman [24], and he finds that the implementation of the next-field predictors, fetching logic, grouping logic, and branch-prediction logic all can lead to the unexpected run-time behavior of programs. Simple code snippets are shown to exhibit non-deterministic performance – a program, executed twice on the same processor under identical conditions, has run times that vary by up to a factor of three. Kushman discovered four such anomalies, though the cause of two of the anomalies remains unknown.

Replacement Policy: Hardware cache replacement policies also can lead to unexpected performance. In their work on replicated fault-tolerance, Bressoud and Schneider find that: "The TLB replacement policy on our HP 9000/720 processors was non-deterministic. An identical series of location-references and TLB-insert operations at the processors running the primary and backup virtual machines could lead to different TLB contents" [10], p. 6, ¶ 2. The reason for the non-determinism is not given, nor does it appear to be known, as it surprised numerous HP engineers.

2.1.2 Disks

Fault Masking: Disks also perform some degree of fault masking. As documented in [6], a simple bandwidth experiment shows differing performance across 5400-RPM Seagate Hawk drives. Although most of the disks deliver 5.5 MB/s on sequential reads, one such disk delivered only 5.0 MB/s. Because the lesser-performing disk had three times the block faults than other devices, the author hypothesizes that SCSI bad-block remappings, transparent to both users and file systems, were the culprit.

Bad-block remapping is also an old technique. Early operating systems for the Univac 1100 series would record which tracks of a disk were faulty, and then avoid using them for subsequent writes to the disk [37].

Timeouts: Disks tend to exhibit sporadic failures. A study of a 400-disk farm over a 6-month period reveals that: "The largest source of errors in our system are SCSI timeouts and parity problems. SCSI timeouts and parity errors make up 49% of all errors; when network errors are removed, this figure rises to 87% of all error instances" [38], p. 7, ¶ 3. In examining their data further, one can ascertain that a timeout or parity error occurs roughly two times per day on average. These errors often lead to SCSI bus resets, affecting the performance of all disks on the degraded SCSI chain.

Similarly, intermittent disk failures were encountered by Bolosky *et al.* [9]. They noticed that disks in their video file server would go off-line at random intervals for short periods of time, apparently due to thermal recalibrations.

Geometry: Though the previous discussions focus on performance fluctuations *across* devices, there is also a performance differential present *within* a single disk. As documented in [26], disks have multiple zones, with performance across zones differing by up to a factor of two. Although this seems more "static" than other examples, unless disks are treated identically, different disks will have different layouts and thus different performance characteristics.

Unknown Cause: Sometimes even careful research does not uncover the cause of I/O performance problems. In their work on external sorting, Rivera and Chien encounter disk performance irregularities: "Each of the 64 machines in the cluster was tested; this revealed that four of them had about 30% slower I/O performance. Therefore, we excluded them from our subsequent experiments" [30], p. 7, last ¶.

A study of the IBM Vesta parallel file system reveals: "The results shown are the best measurements we obtained, typically on an unloaded system. [...] In many cases there was only a small (less than 10%) variance among the different measurements, but in some cases the variance was significant. In these cases there was typically a cluster of measurements that gave near-peak results, while the other measurements were spread relatively widely down to as low as 15-20% of peak performance" [15], p. 250, ¶ 2.

2.1.3 Network Switches

Deadlock: Switches have complex internal mechanisms that sometimes cause problematic performance behavior. In [6], the author describes a recurring network deadlock in a Myrinet switch. The deadlock results from the structure of the communication software; by waiting too long between packets that form a logical "message", the deadlock-detection hardware triggers and begins the deadlock recovery process, halting all switch traffic for two seconds.

Unfairness: Switches often behave unfairly under high load. As also seen in [6], if enough load is placed on a Myrinet switch, certain routes receive preference; the result is that the nodes behind disfavored links appear "slower" to a sender, even though they are fully capable of receiving data at link rate. In that work, the unfairness resulted in a 50% slowdown to a global adaptive data transfer.

Flow Control: Networks also often have internal flow-control mechanisms, which can lead to unexpected performance problems. Brewer and Kuszmaul show the effects of a few slow receivers on the performance of all-to-all transposes in the CM-5 data network [12]. In their study, once a receiver falls behind the others, messages accumulate in the network and cause excessive network contention, reducing transpose performance by almost a factor of three.

2.2 Software

Sometimes unexpected performance arises not due to hardware peculiars, but because of the behavior of an important software agent. One common culprit is the operating system, whose management decisions in supporting various complex abstractions may lead to unexpected performance surprises. Another manner in which a component will seem to exhibit poor performance occurs when another application uses it at the same time. This problem is particularly acute for memory, which swaps data to disk when over-extended.

2.2.1 Operating Systems and Virtual Machines

Page Mapping: Chen and Bershad have shown that virtual-memory mapping decisions can reduce application performance by up to 50% [14]. Virtually all machines today use physical addresses in the cache tag. Unless the cache is small enough so that the page offset is not used in the cache tag, the allocation of pages in memory will affect the cache-miss rate.

File Layout: In [6], a simple experiment demonstrates how file system layout can lead to non-identical performance across otherwise identical disks and file systems. Sequential file read performance across aged file systems varies by up to a factor of two, even when the file systems are otherwise empty. However, when the file systems are recreated afresh, sequential file read performance is identical across all drives in the cluster.

Background Operations: In their work on a fault-tolerant, distributed hash table, Gribble *et al.* find that untimely garbage collection causes one node to fall behind its mirror in a replicated update. The result is that one machine over-saturates and thus is the bottleneck [20]. Background operations are common in many systems, including cleaners in log-structured file systems [31], and salvagers that heuristically repair inconsistencies in databases [19].

2.2.2 Interference From Other Applications

Memory Bank Conflicts: In their work on scalar-vector memory interference, the authors show that perturbations to a vector reference stream can reduce memory system efficiency by up to a factor of two [29].

Memory Hogs: In their recent paper, Brown and Mowry show the effect of an out-of-core application on interactive jobs [13]. Therein, the response time of the interactive job is shown to be up to 40 times worse when competing with a memory-intensive process for memory resources.

CPU Hogs: Similarly, interference to CPU resources leads to unexpected slowdowns. From a different sorting study: "The performance of NOW-Sort is quite sensitive to various disturbances and requires a dedicated system to achieve 'peak' results" [5], p. 8, ¶ 1. A node with excess CPU load reduces global sorting performance by a factor of two.

2.3 Summary

We have documented many cases where components exhibit unexpected performance. As both hardware and software components increase in complexity, they are more likely to perform internal error correction and fault masking, have different performance characteristics depending on load and usage, and even perhaps behave non-deterministically. Note that short-term performance fluctuations that occur randomly across all components can likely be ignored; particularly harmful are slowdowns that are long-lived and likely to occur on a subset of components. Those types of faults cannot be handled with traditional methods, and thus must be incorporated into a model of component behavior.

3 Fail-Stutter Fault Tolerance

In this section, we discuss the topics that we believe are central to the fail-stutter model. Though we have not yet fully formalized the model, we outline a number of issues that must be resolved in order to do so. We then cover an example, and discuss the potential benefits of utilizing the fail-stutter model.

3.1 Towards a Fail-Stutter Model

We now discuss issues that are central in developing the fail-stutter model. We focus on three main differences from the fail-stop model: the separation of performance faults from correctness faults, the notification of other components of the presence of a performance fault within the system, and performance specifications for each component.

Separation of performance faults from correctness faults. We believe that the fail-stutter model must distinguish two classes of faults: absolute (or correctness) faults, and performance faults. In most scenarios, we believe the appropriate manner in which to deal with correctness faults such as total disk or processor failure is to utilize the fail-stop model. Schneider considers a component faulty "once its behavior is no longer consistent with its specification" [33]. In response to such a correctness failure, the component changes to a state that permits other components to detect the failure, and then the component stops operating. In addition, we believe that the fail-stutter model should incorporate the notion of a *performance failure*, which, combined with the above, completes the fail-stutter model. A component should be considered performance-faulty if it has not absolutely failed as defined above and when its performance is less than that of its performance specification.

We believe this separation of performance and correctness faults is crucial to the model, as there is much to be gained by utilizing performance-faulty components. In many cases, devices may often perform at a large fraction of their expected rate; if many components behave this way, treating them as absolutely failed components leads to a large waste of system resources.

One difficulty that must be addressed occurs when a component responds arbitrarily slowly to a request; in that case, a performance fault can become blurred with a correctness fault. To distinguish the two cases, the model may include a performance threshold within the definition of a correctness fault, *i.e.*, if the disk request takes longer than T seconds to service, consider it absolutely failed. Performance faults fill in the rest of the regime when the device is working.

Notification of other components. One major departure from the fail-stop model is that we do not believe that other components need be informed of all performance failures when they occur, for the following reasons. First, erratic performance may occur quite frequently, and thus distributing that information may be overly expensive. Further, a performance failure from the perspective of one component may not manifest itself to others (*e.g.*, the failure is caused by a bad network link). However, if a component is persistently performance-faulty, it may be useful for a system to export information about component "performance state", allowing agents within the system to readily learn of and react to these performance-faulty constituents.

Performance specifications. Another difficulty that arises in defining the fail-stutter model is arriving at a performance specification for components of the system. Ideally, we believe the fail-stutter model should present the system designer with a trade-off. At one extreme, a model of component performance could be as simple as possible: "this disk delivers bandwidth at 10 MB/s." However, the simpler the model, the more likely performance faults occur, *i.e.*, the more likely performance deviates from its expected level. Thus, because different assumptions can be made, the system designer could be allowed some flexibility, while still drawing attention to the fact that devices may not perform as expected. The designer must also have a good model of *how often* various performance faults occur, and *how long* they last; both of these are environment and component specific, and will strongly influence how a system should be built to react to such failures.

3.2 An Example

We now sketch how the fail-stutter model could be employed for a simple example given different assumptions about performance faults. Specifically, we consider three scenarios in order of increasingly realistic performance assumptions. Although we omit many details necessary for complete designs, we hope to illustrate how the fail-stutter model may be utilized to enable more robust system construction. We assume that our workload consists of writing D data blocks in parallel to a set of $2 \cdot N$ disks and that data is encoded across the disks in a RAID-10 fashion (*i.e.*, each pair of disks is treated as a RAID-1 mirrored pair and data blocks are striped across these mirrors a la RAID-0).

In the first scenario, we use only the fail-stop model, assuming (perhaps naively) that performance faults do not occur. Thus, absolute failures are accounted for and handled accordingly – if an absolute failure occurs on a single disk, it is detected and operation continues, perhaps with a reconstruction initiated to a hot spare; if two disks in a mirror-pair fail, operation is halted. Since performance faults are not considered in the design, each pair (and thus each disk) is given the same number of blocks to write: $\frac{D}{N}$. Therefore, if a performance fault occurs on any of the pairs, the time to write to storage is determined by the slow pair. Assuming $N - 1$ of the disk-pairs can write at B MB/s but one disk-pair can write at only b MB/s, with $b < B$, perceived throughput is reduced to $N \cdot b$ MB/s.

In the second scenario, in addition to absolute faults, we consider performance faults that are static in nature; that is, we assume the performance of a mirror-pair is relatively stable over time, but may not be uniform across disks. Thus, within our design, we compensate for this difference. One option is to gauge the performance of each disk once at installation, and then use the ratios to stripe data proportionally across the mirror-pairs; we may also try to pair disks that perform similarly, since the rate of each mirror is determined by the rate of its slowest disk. Given a single slow disk, if the system correctly gauges performance, write throughput increases to $(N - 1) \cdot B + b$ MB/s. However, if any disk does not perform as expected over time, performance again tracks the slow disk.

Finally, in the third scenario, we consider more general performance faults to include those in which disks perform at arbitrary rates over time. One design option is to continually gauge performance and to write blocks across mirror-pairs in proportion to their current rates. We note that this approach increases the amount of bookkeeping: because these proportions may change over time, the controller must record where each block is written. However, by increasing complexity, we create a system that is more robust in that it can deliver the full available bandwidth under a wide range of performance faults.

3.3 Benefits of Fail-Stutter

Perhaps the most important consideration in introducing a new model of component behavior is the effect it would have if systems utilized such a model. We believe such systems are likely to be more available, reliable, and manageable than systems built only to tolerate fail-stop failures.

Manageability: Manageability of a fail-stutter fault tolerant system is likely to be better than a fail-stop system, for the following reasons. First, fail-stutter fault tolerance enables true "plug-and-play"; when the system administrator adds a new component, the system uses whatever performance it provides, without any additional involvement from the operator – a true "no futz" system [32]. Second, such a system can be incrementally grown [11], allowing newer, faster components to be added; adding these faster components to incrementally scale the system is handled naturally, because the older components simply appear to be performance-faulty versions of the new ones. Third, administrators no longer need to stockpile components in anticipation of their discontinuation. Finally, new workloads (and the imbalances they may bring) can be introduced into the system without fear, as those imbalances are handled by the performance-fault tolerance mechanisms. In all cases, the need for human intervention is reduced, increasing overall manageability. As Van Jacobson said, "Experience shows that anything that needs to be configured will be misconfigured" [23], p. 6; by removing the need for intricate tuning, the problems caused by misconfiguration are eradicated.

Availability: Gray and Reuter define availability as follows: "The fraction of the offered load that is processed with acceptable response times" [19]. A system that only utilizes the fail-stop model is likely to deliver poor performance under even a single performance failure; if performance does not meet the threshold, availability decreases. In contrast, a system that takes performance failures into account is likely to deliver consistent, high performance, thus increasing availability.

Reliability: The fail-stutter model is also likely to improve overall system reliability in at least two ways. First, "design diversity" is a desirable property for large-scale systems; by including components of different makes and manufacturers, problems that occur when a collection of identical components suffer from an identical design flaw are avoided. As Gray and Reuter state, design diversity is akin to having "a belt and suspenders, not two belts or two suspenders" [19]. A system that handles performance faults naturally works well with heterogeneously-performing parts. Second, reliability may also be enhanced through the detection of performance anomalies, as erratic performance may be an early indicator of impending failure.

4 Related Work

Our own experience with I/O-intensive application programming in clusters convinced us that erratic performance is the norm in large-scale systems, and that system support for building robust programs is needed [5]. Thus, we began work on River, a programming environment that provides mechanisms to enable consistent and high performance in spite of erratic performance in underlying components, focusing mainly on disks [7]. However, River itself does not handle absolute correctness faults in an integrated fashion, relying either upon retry-after-failure or a checkpoint-restart package. River also requires applications to be completely rewritten to enable performance robustness, which may not be appropriate in many situations.

Some other researchers have realized the need for a model of fault behavior that goes beyond simple fail-stop. The earliest that we are aware of is Shasha and Turek's work on "slow-down" failures [36]. The authors design an algorithm that runs transactions correctly in the presence of such failures, by simply issuing new processes to do the work elsewhere, and reconciling properly so as to avoid work replication. However, the authors assume that such behavior is likely only to occur due to network congestion or processes slowed by workload interference; indeed, they assume that a fail-stop model for disks is quite appropriate.

DeWitt and Gray label periodic performance fluctuations in hardware *interference* [17]. They do not characterize the nature of these problems, though they realize its potential impact on parallel operations.

Birman's recent work on Bimodal Multicast also addresses the issue of nodes that "stutter" in the context of multicast-based applications [8]. Birman's solution is to change the semantics of multicast from absolute delivery requirements to probabilistic ones, and thus gracefully degrade when nodes begin to perform poorly.

The networking literature is replete with examples of adaptation and design for variable performance, with the prime example of TCP [22]. We believe that similar techniques will need to be employed in the development of adaptive, fail-stutter fault-tolerant algorithms.

5 Conclusions

Too many systems are built assuming that all components are identical, that component behavior is static and unchanging in nature, and that each component either works or does not. Such assumptions are dangerous, as the increasing complexity of computer systems hints at a future where even the "same" components behave differently, the way they behave is dynamic and oft-changing, and there is a large range of normal operation that falls between the binary extremes of working and not working. By utilizing the fail-stutter model, systems are more likely to be manageable, available, and reliable, and work well when deployed in the real world.

Many challenges remain. The fail-stutter model must be formalized, and new models of component behavior must be developed, requiring both measurement of existing systems as well as analytical development. New adaptive algorithms, which can cope with this more difficult class of failures, must be designed, analyzed, implemented, and tested. The true costs of building such a system must be discerned, and different approaches need to be evaluated.

As a first step in this direction, we are exploring the construction of fail-stutter-tolerant storage in the Wisconsin Network Disks (WiND) project [3, 4]. Therein, we are investigating the adaptive software techniques that we believe are central to building robust and manageable storage systems. We encourage others to consider the fail-stutter model in their endeavors as well.

6 Acknowledgements

We thank the following people for their comments on this or earlier versions of this paper: David Patterson, Jim Gray, David Culler, Joseph Hellerstein, Eric Anderson, Noah Treuhaft, John Bent, Tim Denehy, Brian Forney, Florentina Popovici, and Muthian Sivathanu. Also, we would like to thank the anonymous reviewers for their many thoughtful suggestions. This work is sponsored by NSF CCR-0092840 and NSF CCR-0098274.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks. In *ASPLOS VIII*, San Jose, CA, Oct. 1998.

[2] R. H. Arpaci, A. C. Dusseau, and A. M. Vahdat. Towards Process Management on a Network of Workstations. http://www.cs.berkeley.edu/ remzi/258-final, May 1995.

[3] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. The Wisconsin Network Disks Project. `http://www.cs.wisc.edu/wind`, 2000.

[4] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, J. Bent, B. Forney, S. Muthukrishnan, F. Popovici, and O. Zaki. Manageable Storage via Adaptation in WiND. In *IEEE Int'l Symposium on Cluster Computing and the Grid (CCGrid'2001)*, May 2001.

[5] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. Searching for the Sorting Record: Experiences in Tuning NOW-Sort. In *SPDT '98*, Aug. 1998.

[6] R. H. Arpaci-Dusseau. *Performance Availability for Networks of Workstations*. PhD thesis, University of California, Berkeley, 1999.

[7] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. In *IOPADS '99*, May 1999.

[8] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Bidiu, and Y. Minsky. Bimodal multicast. *TOCS*, 17(2):41–88, May 1999.

[9] W. J. Bolosky, J. S. B. III, R. P. Draves, R. P. Fitzgerald, G. A. Gibson, M. B. Jones, S. P. Levi, N. P. Myhrvold, and R. F. Rashid. The Tiger Video Fileserver. Technical Report 96-09, Microsoft Research, 1996.

[10] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault Tolerance. In *SOSP 15*, Dec. 1995.

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

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

[13] A. D. Brown and T. C. Mowry. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently. In *OSDI 4*, San Diego, CA, October 2000.

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

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

[16] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsaio, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

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

[18] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *SOSP 16*, pages 78–91, Saint-Malo, France, Oct. 1997.

[19] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[20] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, , and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *OSDI 4*, San Diego, CA, October 2000.

[21] Intel. Intel Pentium 4 Architecture Product Briefing Home Page. http://developer.intel.com/design/Pentium4/prodbref, January 2001.

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

[23] V. Jacobson. How to Kill the Internet. ftp://ftp.ee.lbl.gov/talks/vj-webflame.ps.Z, 1995.

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

[25] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[26] R. V. Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the 1997 USENIX Conference*, Jan. 1997.

[27] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. Intelligent RAM (IRAM): Chips That Remember And Compute. In *1997 IEEE International Solid-State Circuits Conference*, San Francisco, CA, February 1997.

[28] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88*, pages 109–116, Chicago, IL, June 1988. ACM Press.

[29] R. Raghavan and J. Hayes. Scalar-Vector Memory Interference in Vector Computers. In *The 1991 International Conference on Parallel Processing*, pages 180–187, St. Charles, IL, August 1991.

[30] L. Rivera and A. Chien. A High Speed Disk-to-Disk Sort on a Windows NT Cluster Running HPVM. Submitted for pulication, 1999.

[31] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[32] M. Satyanarayanan. Digest of HotOS VII. http://www.cs.rice.edu/Conferences/HotOS/digest, March 1999.

[33] F. B. Schneider. Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[34] F. B. Schneider. Personal Communication, February 1999.

[35] A. P. Scott, K. P. Burkhart, A. Kumar, R. M. Blumberg, and G. L. Ranson. Four-way Superscalar PA-RISC Processors. *Hewlett-Packard Journal*, 48(4):8–15, August 1997.

[36] D. Shasha and J. Turek. Beyond Fail-Stop: Wait-Free Serializability and Resiliency in the Presence of Slow-Down Failures. Technical Report 514, Computer Science Department, NYU, September 1990.

[37] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation*. A K Peters, 3rd edition, 1998.

[38] N. Talagala and D. Patterson. An Analysis of Error Behaviour in a Large Storage System. In *IPPS Workshop on Fault Tolerance in Parallel and Distributed Systems*, 1999.