Faults in Large Distributed Systems and What We Can Do About Them

George Kola, Tevfik Kosar and Miron Livny

Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison WI 53706
{kola,kosart,miron}@cs.wisc.edu

Abstract. Scientists are increasingly using large distributed systems built from commodity off-the-shelf components to perform scientific computation. Grid computing has expanded the scale of such systems by spanning them across organizations. While such systems are cost-effective, the usage of large number of commodity components causes high fault and failure rates. Some of these faults result in silent data corruption leaving users with possibly incorrect results. In this work, we analyzed the faults and failures that occurred in Condor pools at UW-Madison having a few thousand CPUs and in two large distributed applications: US-CMS and BMRB BLAST, each of which used hundreds of thousands of CPU hours. We propose 'silent-fail-stutter' fault-model to correctly model the silent failures and detail how to handle them. Based on the model, we have designed mechanisms that automatically detect and handle silent failures and ensure that users get correct results. Our mechanisms perform automated fault location and can transparently adapt applications to avoid faulty machines. We also designed a data provenance mechanism that tracks the origin of the results, enabling scientists to selectively purge results from faulty components.

1 Introduction

Scientists are increasingly using distributed systems built from commodity components for their computing needs. Grid computing [1] has increased the scale by sharing these computing resources across organizations. While this approach is cost-effective, hardware errors may create havoc if the system software and applications do not handle them appropriately. For instance, most of the several thousand UW-Madison Condor pool compute nodes have non-parity memory. A stray alpha particle may corrupt the memory leaving the scientists with incorrect results. Further, just failure of certain memory chips may corrupt parts of the computation and this failure may go unnoticed for a long period.

Hard-drives, RAID-controllers [2] and processors may also exhibit such faulty behavior. While detecting such faulty components is itself difficult, detecting all the corrupted results and recomputing them is even more difficult. This leaves the users with results that may be incorrect. This is particularly troublesome for large-scale computation performed on these distributed systems. While system-administrators may detect these faults at some point and replace the faulty components, purging the results that touched these components is non-trivial.

We analyzed the faults in large distributed systems by looking at the faults and failures that occurred in the Condor pools at UW-Madison campus, and in two large distributed applications: US-CMS and BMRB BLAST, each of which processed terabytes of data and used hundreds of thousands of CPU hours.

In this work, we present a summary of our experience and propose 'silent-fail-stutter' fault model that correctly models components that exhibit silent failures. We highlight the implications of this model and detail what a system incorporating such components should do.

Using the insights from the model, we have designed mechanisms that can automatically detect and handle silent failures taking into account user specified policies. Our mechanisms also provide fault location and can dynamically adapt applications to avoid faulty machines. They keep track of the origin of the result, including the components that interacted with the source component to generate this result. This enables users at any point to selectively purge results that interacted with faulty components. Finally, we evaluate our new model and the mechanisms on a real-life distributed workload and highlight its effectiveness.

2 Faults in Distributed Systems

In this section, we give a widely accepted definition of faults, and present different types of faults experienced by real distributed applications.

2.1 Definition of Faults

The widely accepted definition, given by Avizienis and Laprie [3] is as follows. A fault is a violation of a system's underlying assumptions. An error is an internal data state that reflects a fault. A failure is an externally visible deviation from specifications. A fault need not result in an error, nor an error in a failure. An alpha particle corrupting a memory location is a fault. If that memory location contains data, that corrupted data is an error. If a program crashes because of using that data, it is a failure.

2.2 Experienced Faults in Distributed Systems

We analyzed the faults in large distributed systems by looking at the faults and failures that occurred in two large distributed applications: US-CMS and BMRB BLAST, each of which was processing terabytes of data and using hundreds of thousands of CPU hours. We also analyzed several other small applications running in the Condor pool at UW-Madison campus having a couple of thousand compute nodes. The most common failures we have observed are:

Data Corruption. Faulty hardware in data storage, staging and compute nodes corrupted several data bits occasionally. The faults causing this problem included a bug in the raid controller firmware on the storage server, a defective PCI riser card, and a memory corruption. The main problem here was that the problem developed over a course of time, so initial hardware testing was not effecting in finding the problems. The raid controller firmware bug corrupted data only after a certain amount of data was stored on the storage server and hence was not detected immediately after installation. In almost all of these cases, the data corruption happened silently without any indication from hardware/operating system that something was wrong. Tracking down the faulty component took weeks of system administrator's time on average.

Hanging Processes. Some of the processes hang indefinitely and never return. From the submitters point of view there was no easy way of determining whether the process was making any progress or was hung for good. The most common cause of hanging data transfers was the loss of acknowledgment during third party file transfers. In BMRB BLAST, a small fraction of the processing hung and after spending a large amount of time, the operator tracked it down to an unknown problem involving the NFS server where an NFS operation would hang.

Misleading Return Values. An application returning erroneous return values is a very troublesome bug that we encountered. We found that even though an operation failed, the application returned success. This happened during some wide area transfers using a widely used data transfer protocol. We found that if the destination disk ran out of space during a transfer, a bug in the data transfer protocol caused the file transfer server to return success even though the transfer failed. This in turn resulted in failure of computational tasks dependent on these files.

Misbehaving Machines. Due to misconfigured hardware or buggy software, some machines occasionally behaved unexpectedly and acted as 'black holes'. We observed some computational nodes accepting jobs but never completing them and some completing the job but not returning the completion status. Some nodes successfully processed certain job classes but experienced failures with other classes. As a particular case, in WCER video processing pipeline [4], we found that a machine that had a corrupted FPU was failing MPEG-4 encoding whereas it was successfully completed MPEG-1 and MPEG-2 encodings.

Hardware/Software/Network Outages. Intermittent wide area network outages, outages caused by server/client machine crashes and downtime for hardware/software upgrades and bug fixes caused failure of the jobs that happened to use that feature during that time.

Over commitment of Resources. We encountered cases where the storage server crashed because of too many concurrent write transfers. We also encountered data transfer time-outs that that were caused by storage server trashing due to too many concurrent read data transfers.

Insufficient Disk Space. Running out of disk space during data stage-in and writing output data to disk caused temporary failures of all involved computational jobs.

3 Why do current fault models not work well?

Byzantine [5] and fail-stop [6] are two widely used fault models. In Byzantine model, a component can exhibit arbitrary and malicious behavior, perhaps involving collusion with other faulty components. In fail-stop model, in response to a failure, the component changes to a state that permits other components to detect a failure has occurred and then stops.

Byzantine model is too general in nature and reasoning out the different scenarios is difficult. Normal systems may not encounter such malicious behavior. However, it is useful in security where such adversarial behavior may occur. Fail-stop model is at other end of the spectrum and it is extremely simple and tractable. For this reason, most systems are built using the fail-stop model.

Unfortunately, the fail-stop model is too simple to model the behavior of many components. Arpaci-Dusseau [7] found that similar components differ widely in performance, enough to be called a performance failure. To model the behavior of such components, they introduced the fail-stutter model where a component may operate at reduced performance level in addition to failing and stopping. Fail stutter behavior is commonly seen in disks where the controller may transparently remap bad sectors and such a disk may have lower performance compared to a bad-sector free disks because of extra seeks. Fail-stutter separates failures into correctness and performance failures and correctly models components that have performance failures but are correct. Fail-stutter expects the components to behave like fail-stop when a correctness failure occurs.

Even fail-stutter does not correctly model behavior of many components. Many components on encountering a correctness failure do not immediately change to a state that allows other components to detect the failure. For instance, if a memory chip is corrupted, it does not give that information to other components. A memory tester program can detect the corruption by writing data to it and reading the written data and verifying it. Parity and error correcting does not fully address this issue. For instance, a chip with a fault that prevents data from being written to it cannot be detected by parity because the parity bits are correct for the incorrect old data.

In a distributed system, significant fractions of the applications involve a pipeline of processing [8]. If a particular component in the pipeline generates incorrect data because of a fault, other components cannot detect immediately that this component has failed.

4 Silent-Fail-Stutter: A More Accurate Fault Model

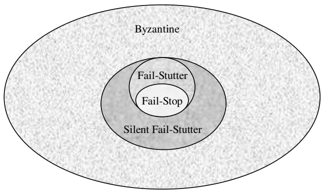

Fig. 1. Different fault models and their relation

Fail-stop expects components that encounter a failure to immediately change state and let other components detect that failure. As shown in the previous section, many components do not indicate a failure immediately. To address this, we propose a new fault-model called 'silent-fail-stutter'. In this model, on failure a component may not immediately change state and convey its failure to other components. However, other components can at anytime detect if a component has failed by testing it, incurring a certain cost. Further, the stutter behavior may be an indicator of impending failure and other components can use that as a heuristic to test that component for failure. Figure 1 shows the different fault models and how they relate to each other.

We believe that silent-fail-stutter models real life behavior of components better than fail-stutter while maintaining tractability. Below, we give a few examples of real-life cases where silent-fail-stutter is more appropriate than fail-stutter.

System memory. Most of today's system memory chips are non-parity. In non-parity memory, memory corruption cannot be detected immediately and hence fail-stop and fail-stutter are not suitable models. Silent-fail-stutter is a suitable model because a memory tester program can detect a corrupt memory chip and doing this test incurs a cost. With parity and error-correcting memory, a chip that is faulty with respect to memory writes exhibits silent-fail-stutter. Even though most but not all read errors may be detected by parity, a corrupt chip is detected only when some earlier written data is accessed, making fail-stop inappropriate. A fail-stop component would have detected failure earlier, say during DRAM refresh cycle, and informed other components. Thus for all types of system memory, silent-fail-stutter is a suitable model.

Processor cache. Processor caches are SRAMs and typically starting from Level 2 have error correcting code. Register [9] reports cases where Solaris operating system crashed and rebooted because the UltraSparc II had E-cache(level2 cache) parity errors. Sun's best practices guide [10] advised system administrators to log such failures and replace the processor on second such failure.

The behavior is not fail-stop, because a processor with the faulty cache does not retain information about a cache block failure across reboots even if the failure is permanent. In addition, a cache block failure is detected only when some data is written to it and fails parity test when read back.

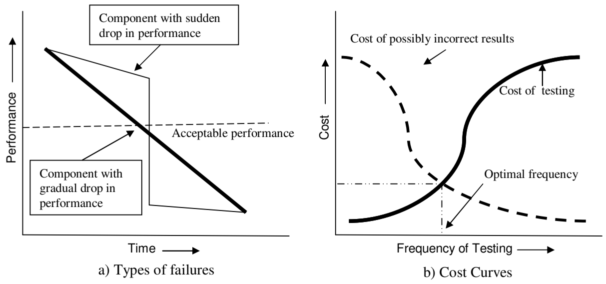

Fig. 2. a) Two types of components b) Cost Curves

Silent-fail-stutter is a more accurate model because even though a cache block may fail at any time, it may not be detected immediately. A program can test if any cache block is faulty by writing data to all cache blocks and reading it to verify that there is no corruption. Doing this involves a cost that silent-fail-stutter models making it a good fit.

Faults in Distributed Systems. If a system is built from silent-fail-stutter components, then silent-fail-stutter is the correct model for that system. The faults we mentioned in the section 2, were all silent-fail-stutter. The problem was that applications expected fail-stop behavior whereas the components exhibited silent-fail-stutter.

5 Implications of Silent-Fail-Stutter

Since components may fail and not convey the failure to other components, some component should periodically or on certain events determine the state of each component and if a failure is detected, report it to other components. If a single component makes the checks, designers should ensure that this component is more reliable than the ones it checks. All the components can co-operate to perform this check in a distributed manner.

These ways of handling silent-fail-stutter have been known informally for a while. For instance, processors during startup have the ability to test the memory for errors and in case of error report that to the user and stop. Here the processor is assumed to be more reliable than the memory and the whole system behaves like fail-stop even though the underlying component, memory, is silent-fail-stutter. However, with always-on systems, memory test during startup may not be sufficient.

Figure 2a shows how the performance changes with respect to time for two types of components. For one type, the performance drops gradually and for the other, the performance drop is minimal at first and then a sudden drastic drop. The figure also shows an acceptable performance limit. Below the acceptable performance, correctness failures may occur. Many components exhibit such gradual performance drop making it easy to predict their failure ahead of time.

Fault masking at times results in the drastic fall in performance. For example, consider a failing disk where the sectors are failing because the magnetic media is loosing its ability to retain stored data. If the hard disk employs sector remapping, it may be able to hide the bad sectors for a while by which time the media may have degraded to a point where suddenly most sectors fail causing a drastic drop in performance. A good solution to this is to expose this information about faults, so that a smarter higher-level system can use this information to take some action.

Since, components may fail silently, how often to test them is of importance. Doing the test occasionally runs the risk of not detecting the failure for a longer duration. Running the test often may result in considerable overhead.

Figure 2b shows the cost curve for component testing and using possibly incorrect results for a hypothetical system. The cost curves would depend on failure probability of the component, cost of testing and the cost of incorrect results. Here, we use the term cost loosely. In practice, we can normalize the cost to a meaningful common form like time, computation that can be performed in that time, etc. For mission critical systems, the cost of possibly incorrect result would be infinity. Similarly, for some computation that generates hints for heuristics, the cost of possibly incorrect result would be low.

If the silent-fail-stutter component belongs to gradual performance decline category, then the tester can predict when it is going to fail by testing the current performance and co-relating it with an existing model of the component behavior. Conversely, if the drop in performance is less than a threshold, an interacting component can trigger the tester to test that component.

In distributed systems, the cost can be amortized over all the interacting components. For instance, if there are 'n' components that use the result from a single component, the single component can be tested at $1/n$ of the frequency that would be needed if there were only one interacting component. This is because, the tester can inform other components of the result of the tests and in most cases getting the result of previous test is much cheaper than performing a new test. Therefore, as a good system design principle, distributed system designers should implement mechanisms to test silent-fail-stutter components and report them to interested components.

6 Failure Detection in Distributed System

Complexity of distributed systems makes failure detection difficult. There are multiple layers from the hardware to the application. Since we did not want to impose an undue burden on application developers to handle failure detection and handling, we implemented the error/failure detection on top of the application, by verifying that results generated are correct. To do this, the applications should allow multiple executions and they should produce reproducible results.

Grid applications are expected to have the ability to be run multiple times because they could be pre-empted from a resource. Further, to enable checking of outputs, they need to generate reproducible results. Most applications already do that. We need to clarify that applications produce both an output and a log of the processing. The log of the processing may include start time, information about execute machine, etc and would not be reproducible across multiple executions. However, the output, say a processed image would be the same across executions.

In addition to detecting erroneous results, we also need to detect the cause of the fault and possibly replace that faulty component. Identifying the source of the erroneous result has so far been a *'black art'* in the realm of select few system administrators and operators. This process takes considerable amount of time, usually weeks, expending considerable amount of human resources.

We classify silent failures into two types as shown in figure 3. Type I silent failures are silent failures that give incorrect results without any error status indication. Type II silent failures are silent failures in which the process or transfer just hangs. Type I gives a successful return code and shows that the process is completed but the results are incorrect. This normally happens because of interface mismatch where a component expects underlying components to be fail-stop, but they are in fact silent-fail-stutter. Type II never returns, so user cannot find out if the process will complete. This could be caused by bugs. In addition to silent failure, jobs may fail with an error status and they are easier to detect. We will first discuss about handling Type I.

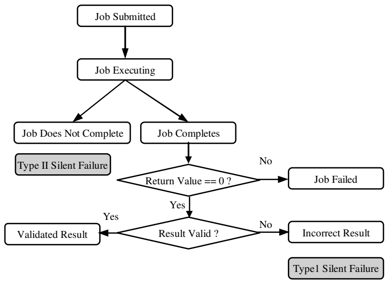

Fig. 3. Type I and Type II silent failures.

We want to detect if a failure has occurred and if we need to track down the cause of that failure. A silent failure of lower level component may result in a failure higher up the chain and to track down the fault, we may need to go down the hierarchy. For instance, the cause of a computation failure may be because of data corruption in the intermediate storage server and this in turn may be caused by a faulty RAID controller in the storage server. We feel that automatically isolating the fault to whole system boundary is easier and this would aid the system administrator in locating the exact problem.

Consider a simple case where the user has to be 100% certain that the result is correct. A simple way of doing that is to compute the result twice and verify that they match. While doing this we need to be careful to ensure that the two computations do not overwrite the same data. Name space mapping can address this. Suppose if we find that a result is incorrect, we can pick up all the incorrect results in a given time period and all systems interacted with most of the results is the likely culprit. A simple mechanism that detects this can notify it to the system administrator who can then test that system. At the same time, the component can give feedback to higher-level planners like Pegasus [11] and/or distributed schedulers to ensure that they do not use this resource until the fault has been resolved. Verification of data transfers involves checksum generation and verifying that source and destination checksums match.

Components belonging to silent-fail-stutter allow testing to determine a failure. The methodology for testing can be inferred from “THE“ multiprogramming system [12], where they had a layered structure to test that reduced the number of test cases. We believed that a conscientious distributed system designer should design such a test infrastructure. If such a test infrastructure exists, the mechanism on detecting a failure can trigger a test of the whole system to isolate the faulty component. As an alternative, to isolate machine faults at a coarse grain, a tester can periodically execute a test program that generates a known result and takes a certain deterministic amount of time on each machine. If any machine gives a wrong result or the run time deviates considerably, the system administrator can be informed of the problem.

If the user does not want to pay a 100% overhead by performing each computation twice and if testing system exists, he can specify the fraction of extra computation that he is willing to perform. The failure detector will inject that fraction of extra computation into the distributed system in a statistically unbiased manner. The results of these extra computations are compared with results of the previous execution and verified to be same. In case of difference, the failure detector can tag those machines and perform the computation again on a different machine to identify the faulty one. When the failure detector identifies a faulty machine, it can report the time from the successful machine test to current time as time when the machine was in a possibly faulty state. Results generated using that machine during that time may have to be recomputed.

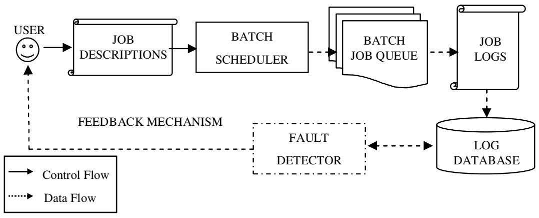

Fig. 4. Stages in performing some processing on a distributed system.

Application	Coefficient of Variation
BLAST BMRB (1MB Database)	0.19
BLAST PDB (69MB Database)	0.34
BLAST NR (2GB Database)	0.49
NCSA Sextractor Processing	2.00
NCSA Data Transfer	1.00

Table 1. Coefficient of Variation of Execution Time

Handling Type II silent failures requires some more effort. The issue is whether it is possible to detect such a failure. In practice, most of the hung processes have a way of detecting that they have failed to make forward progress. A simple case is that of data transfer, we can find out how the file size varies over time and if the file size does not change for a long period, we can know that the file transfer has hung. Another way is to come up with reasonable time-outs for operations. We can find out that a transfer or computation has hung if it does not complete in a certain period.

Most of the present day distributed workloads consist of a large number of instances of the same application. Typically, the standard deviation of execution time is of the same order of magnitude as mean if not lesser. This lends a very effective way to detecting Type II failure. Using this, mechanisms can set the threshold to be $mean + 3 \times StandardDeviation$ or some similar threshold. Users can specify policy on what fraction of the processing they are willing to re-do. If users want responsiveness, they may trade some extra processing and set a lower threshold. If they want to minimize the overhead, they would use a higher threshold.

7 Evaluation

To evaluate the effectiveness of silent-fail-stutter model and our discussion on failure detection, we implemented a prototype of the mechanism mentioned in the previous section.

We looked at how components should convey the results of test and we decided to use a database to log the results of tests and timestamp of tests. We also log the results of application execution into the database. To get this information, we parse the distributed batch scheduling system (Condor) user-job log-files and store them in a relational database. We developed the schema for doing it from our previous work [13]. Since we wanted to track down the origin of the results, we store the job description also in the database. Figure 4 shows the process.

We found that users typically submit job bundles specified as a directed acyclic graph(DAG). We categorize a job bundle as an application-class. Users normally tag application class and we can use that as well if the same application class spans across multiple job bundles.

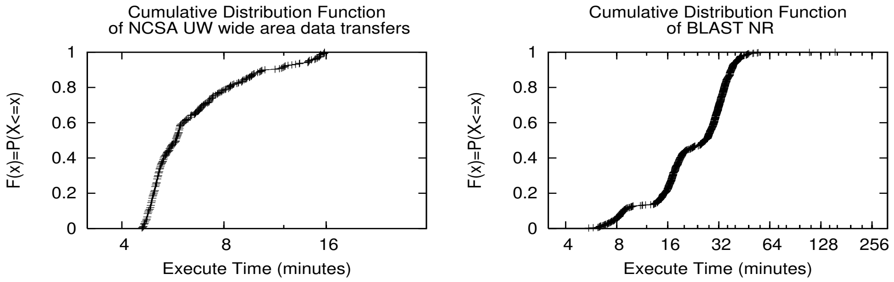

Fig. 5. Shows the cumulative distribution function of BLAST execution against the 2 GB NR database and NCSA UW wide-area data transfers

For verification of results, we store the md5 checksum of the results in the database. To validate, we can run a simple query that checks if the checksums of results of identical jobs are the same. In case of error, we have a query that can extract out all the machines that generated suspect results and tag the machine appearing multiple times as faulty.

To evaluate our ability to identify silent Type II failures, we looked at the co-efficient of variation of executing time of some applications. Table 1 shows the co-efficient of variation of a few well-known applications. We found that the coefficient of variation of all classes we encountered were less than four.

The UW Madison condor pool consists of multiple clusters with different processor speeds, and user desktops. We did not separate the performance according to machine class as the job may be assigned to any machine depending on availability unless the jobs explicitly request certain configuration. Taking into account machine class, brought down the coefficient of variation considerably but we do not report that, as we may not be able to do so well in a general environment.

Figure 5 shows the cumulative distribution function of BLAST processing using 2 GB NR database and wide-area data transfers between NCSA and UW. Each wide-area data transfer transferred a 1.1 GB astronomy image file from NCSA to UW-Madison and that file was subsequently processed in UW condor pool.

For the blast run, a few jobs hung and took a very long time to complete, around 4 hours. The user had a hard limit of 4 hours and the jobs that exceeded 4 hours were killed and restarted. Using our mechanism, we find that we can come up with tighter bounds. In this case, $Mean + 3 \times StandardDeviation$ =80 minutes and does not require operators to magically come up with thresholds and are better than the rough guess of the user. The data transfers had a 20-minute time-out for the data transfers. There were hung transfers that succeeded second time around.

8 Future Work

We intend to develop a more rigorous theoretical analysis of our silent-fail-stutter model. We also want to deploy our mechanisms in real systems over a long period and evaluate them. The mechanism assumes that the failed fraction is significantly less than the successful fraction, which we believe would be true in practice. We would like to determine if there are limits on failure fraction that will cause the mechanisms to not work.

9 Conclusions

We have successfully analyzed the faults in large distributed systems and proposed *silent-fail-stutter* fault model to accurately model component behavior while maintaining tractability. Using insights from the model, we have developed mechanisms to automatically detect silent failures in distributed systems. We have evaluated the mechanisms and shown their effectiveness.

References

1. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the grid: Enabling scalable virtual organizations. International Journal of Supercomputing Applications (2001)
2. Patterson, D.A., Gibson, G.A., Katz, R.H.: A case for redundant arrays of inexpensive disks (raid). In Boral, H., Larson, P.Å., eds.: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988, ACM Press (1988) 109–116
3. Avizienis, A., Laprie, J.: Dependable computing: From concepts to design diversity. In: Proceeding of the IEEE. Volume 74. (1986) 629–638
4. Kola, G., Kosar, T., Livny, M.: A fully automated fault-tolerant system for distributed video processing and off-site replication. In: Proceeding of the 14th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (Nossdav 2004), Kinsale, Ireland (2004)
5. Lamport, Shostak, Pease: The byzantine generals problem. In: Advances in Ultra-Dependable Distributed Systems, N. Suri, C. J. Walter, and M. M. Hugue (Eds.), IEEE Computer Society Press. (1995)
6. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys **22** (1990) 299–319
7. Arpaci-Dusseau, R.H., Arpaci-Dusseau, A.C.: Fail-Stutter Fault Tolerance. In: The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII), Schloss Elmau, Germany (2001) 33–38
8. Thain, D., Bent, J., Arpaci-Dusseau, A., Arpaci-Dusseau, R., Livny, M.: Pipeline and batch sharing in grid workloads. In: Proceedings of the Twelfth IEEE Symposium on High Performance Distributed Computing, Seattle, WA (2003)
9. The Register: Sun suffers UltraSparc ii cache crash headache. http://www.theregister.co.uk/2001/03/07/sun_suffers_ultra sparc_ii_cache/ (2001)
10. Sun Microsystems Inc: Best practices guide: Addressing e-cache parity errors. http://www.filibeto.org/sun/lib/hardware/enterprise_4500/ BP_Ecache_10-16-01.pdf (2001)
11. Deelman, E., Blythe, J., Gil, Y., Kesselman, C.: Pegasus: Planning for execution in grids. Technical Report 20, GriPhyN (2002)
12. Dijkstra, E.W.: The structure of the THE-multiprogramming system. Communications of the ACM **11** (1967)
13. Kola, G., Kosar, T., Livny, M.: A client-centric grid knowledgebase. In: Proceedings of Cluster 2004, San Diego, CA (2004)