BATCH PROCESSING IN COMPUTERS

In some broad sense, *batch processing* refers to the activity in which a computer system processes a collection of requests as a whole rather than individually. There are several reasons for batch processing. Consider, for example, a user who wants to run a program repeatedly against one hundred different input data files. It will be quite frustrating if the user has to sit in front of the computer and type the same command, only with different input file names, one hundred times. Moreover, in a multiuser computer system, running one or many instances of a long-running program may take up many system resources and slow down the response time of some other interactive users. Thus, most computer systems are equipped with a batch-processing utility that allows users to assign jobs to the computer in a batch and let the system handle the jobs without user assistance. An important function of the batch processing mechanism is to make the best use of the system resources. For example, the system may perform long-haul batch jobs, either computation or I/O-intensive, off-line at nights, freeing the system for interactive users during normal business hours. In a networked computing environment, the system may even locate an idle machine to process the batch jobs. The first part of the article examines, in an evolutionary spirit, the development and features of batch-processing systems.

While the above avenue of batch processing lies mainly at the application level, the computer system itself may adopt the principle of batch processing at a much lower level, to achieve better performance. One such technique is to batch disk-write operations. Due to disparate processing speeds between processors and disk I/O, disk operations are more likely to become the bottleneck in many systems. While prefetching and caching techniques are able to reduce the average access time for disk-read operations, they do not equally improve the access time for disk writes. As a result, disk writes remain the hurdle to a full utilization of the disk data transfer bandwidth. The problem arises because random disk writes causes excessive disk head movements, which are slow, due to their mechanical nature. The second part of the article presents techniques used in some file systems that batch, and perhaps reschedule, disk-write requests, in order to achieve better I/O performance.

BATCH-PROCESSING SYSTEMS

Batch processing was the main design goal of many early computer systems. In many cases, the batch-processing mechanism is part of the underlying *operating system,* which controls the computer. As computer technologies evolve, however, few vendors today are manufacturing computers or operating systems solely for the purpose of processing batch jobs. Rather, modern operating systems are designed to handle *interactive* jobs, which require frequent interaction between the users and the computers, as well as long-running batch jobs. Most systems now provide batch-processing capability as a suite of utility software, which is implemented outside the operating system.

In order to understand the key requirements and features of batch-processing systems, it is useful to consider how batch-processing systems have evolved over the years. Contemporary computers were first created and used in production around the mid-1940s. These machines were enormous in size, filling up entire rooms with tens of thousands of vacuum tubes. At that time, there were no such concepts as operating systems and high-level programming languages. Programmers had to write programs in machine languages on plugboards and feed them into the computer to run the job (1). Individual programmers were fully responsible for setting up the computers to run the programs and for monitoring the progress of the running jobs. Since computer time was precious, users reserved the use of the computer by filling up a time slot in a sign-up book. This type of operation could be considered *serial processing,* reflecting the fact that users had access to the computers in series.

Simple Batch Systems

The serial processing nature of the very first computer systems was quite inefficient: the wasted time caused by the setup overhead of subsequent jobs is not acceptable. Computer vendors soon realized that some software was needed to automate job scheduling and setup, in order to maximize the use of the computers. This led to the development of the first batch operating system over the years from the mid-1950s to the mid-1960s. One example is IBSYS, the batch operating system for the IBM 7094 computers.

The basic idea of the simple batch operating system was to use software called Monitor (2)—the ancestor of all modern operating systems—to control the job setup. The user submitted programs on cards or tapes to the operator, who batched them on an input device of the computer, ready to be loaded into the memory by the monitor. The monitor read in jobs one at a time from the input device (a card reader or magnetic-tape drive). Once a job (including the codes and data) was placed in the memory, the monitor executed a branch instruction, which directed the central processing unit (CPU) to jump to and continue execution at the start location of the user program. When a job was completed, the control was returned to the monitor, which then read in the next job. The results of a job were usually sent to a line printer for printout.

Simple batch operating systems eliminate the need for human intervention for job setup. However, they introduce certain overhead: a portion of the processor time and memory space must be allocated to the monitor. Often this overhead is more than offset by the saving in manual setup time. Therefore, batch operating systems improved the use of computers.

Multiprogramming Batch Systems

In a simple batch system, when the running program requests an input or output (I/O) operation (e.g., reading from a tape or writing to a printer), the CPU simply sits idle until the requested I/O operation is completed. Since I/O operation is much slower than the CPU, much CPU time is wasted if I/O operations are frequent. The solution to avoid such inefficiency is to allow more than one user job to reside in memory—a concept termed *multiprogramming.* When the running job requests an I/O operation, the CPU is given to another job in memory. If there are sufficient jobs in the memory, the CPU can be kept busy all the time.

During the mid-1960s, IBM introduced the System/360 series—the first computer line to use integrated circuits (ICs)—and incorporated the multiprogramming concept in the operating system OS/360 (3). The design of a multiprogramming system involves some memory-management issues and requires special hardware supports:

- *Memory Partition and Protection.* The memory was divided into several partitions, with a different job in each partition. The operating system designer must make certain decisions regarding the management of the memory. One issue is whether the partitions should be predetermined when the operating system is started and remain fixed afterward, or should they be created dynamically and assigned to user jobs when the jobs are admitted? Another issue arises when there is more than one job waiting in the queue: the system needs to determine which job to execute next, when a memory partition is available. For example, one may simply select the first job in the queue that fits into the available partition, or one may find the job that best fits into the partition (thus resulting in smallest unused space). All these issues have profound impact on the utilization of the memory and thus the performance of the system. Finally, the operating system must prevent the failure of a job from affecting other jobs in the memory. This is typically enforced by a combination of software and hardware. One method is to use two *boundary registers,* which store the low boundary and high boundary of the memory partition in which the current running job resides. All memory addresses generated by the running job are checked against the registers to ensure they fall between the boundaries. Should a violation occur, the operating system would intercept the faulty instruction, terminate the job, and print an error message to the user. Interested readers may refer to Refs. 1 and 2 for more details about the various strategies for partitioned memory management.
- *I/O Interrupts.* In a multiprogramming system, a running job will give up the CPU and be blocked when it issues an I/O operation. There must be a means for the I/O device to inform the operating system when the I/O operation is finished, so the blocked job can have a chance to regain the CPU. The solution is interrupt-driven. When an I/O operation is completed, the I/O device sends a signal to the CPU and sets an interrupt bit. The operating system will detect such a condition and interrupts the currently running job. It then passes the control to an interrupt-handling routine. The interrupt-handling routine performs necessary tasks (e.g., moves the data from a buffer in the I/O controller to the main memory and unblocks the blocked process), and returns the control to the interrupted job (or some other job in certain cases).

Multiprogramming operating systems manage to overlap the CPU operation of a job with an I/O operation of some other job. Best parallelism can be achieved and, thus, much time can be saved when a batch of jobs requires comparable CPU and I/O consumption.

Batch Processing in Time-Sharing Systems

Time-Sharing Systems. Multiprogramming systems were well suited for large scientific or commercial data processing applications, which do not require constant user interaction. While long response time is fine with these applications, there are other types of applications that require frequent user input and quick response time. This desire leads to the concept and development of *time-sharing* systems. A time-sharing system is a multiprogramming system in which each job (or user) is assigned small pieces of CPU time, in turn.

In a time-sharing computer system, several users may log on and interact with the computer at the same time, through terminals. The operating system interleaves the execution of the user programs by giving a short CPU burst called *quantum* (typically in the range of hundreds of milliseconds) to each program, in turn. Given the relative slow human reaction time, a time-sharing system manages to provide fast, interactive service to concurrent users, producing the illusion that each user has a dedicated computer. One of the first time-sharing systems was the Compatible Time-Sharing System (CTSS) (4), developed at Massachusetts Institute of Technology in the early 1960s for a specially modified IBM 7094 computer. Since then, time-sharing was the theme of operating system design and could be found in virtually all kinds of computer systems that followed, including minicomputers (1970–1980) and personal computers (1980–present).

Batch Utility. Unlike the early computer systems, in which batch processing was an indivisible part of the underlying operating system, time-sharing computer systems usually support the batch function as application programs, separate from the *kernel* of the operating system. The batch utility often consists of a set of commands, which the user could use to submit and manage batch jobs. To avoid delaying short interactive jobs, large batch jobs are typically run in the background, with lower priorities, or when the CPU is otherwise idle. The following sections describe the usage and the design issues of a typical batch utility. While the attempt is to keep the discussion in general, much of it is based on the `at` (or `batch`) command that is available in most Unix operating systems.

Usage. To start, one must first create a *batch file,* which contains a list of jobs to be run in batch. The following shows the content of an example batch file that contains four jobs.

```
simulate 100 < data-file-1 > result-1
simulate 400 < data-file-1 > result-2
simulate 100 < data-file-1 > result-3
simulate 400 < data-file-1 > result-4
```

In this example, all jobs run the same program (`simulate`), but use different input parameters (`100` and `400`) and/or data files. The phrase ''`< data-file-1`'' indicates the program `simulate` will read data from file ''`data-file-1`''; the phrase ''`> result-1`'' direct the program to save the simulation results in a file named ''`result-1`''. Once the batch file is created, the user may run the batch file by using a batch command. The following statement shows how to submit the above batch file (assuming it is saved in a file named ''`batch-file`'') using UNIX's **''`at`''** command.

```
at -f batch-file 24:00
```

The option ''`-f`'' indicates that the batch jobs are to be taken from a file (namely, ''`batch-file`''). The argument ''`24:00`'' in the statement specifies that the batch jobs are to be executed at midnight of the day. In general, a batch utility must provide the following user options:

- *Execution Time.* Run the submitted jobs at a later time, even after the user has logged off the system.

- *Status Report.* The batch utility must provide the users with the option to check the status of submitted jobs. This may include progress information (e.g., which jobs have been completed and when), as well as resource consumption statistics (e.g., how much CPU time and memory space was used by a job).
- *Forced Termination.* A batch utility should allow users to voluntarily remove a job from the batch queue or to kill a faulty running job.

Batch Job Management. The following examines some common issues that a batch utility should consider when managing batch jobs.

- *Access Privilege.* Access control is necessary in a multiuser system, to prevent a user from accidentally or intentionally exhausting the system resources and jeopardizing the jobs of others. Some operating systems allow the administrator to determine who should have access to the batch utility and assign different levels of privileges (e.g., by imposing a limit on the CPU time and memory space used by a job).
- *Spooling and Clock Demon.* When a job is scheduled for execution at a later time, the batch utility must produce a file including necessary scheduling information and store it in a *spooling* directory. A clock demon (a demon is a process running in the background that never exits) will examine the spooling directory periodically (typically with an interval much less than one second) and start those jobs whose execution times have passed the current time.
- *Scheduling.* Batch jobs are often long-running and take up lots of system resources. Running a large batch job may severely delay the response time of other short interactive jobs. To ensure fairness, the operating system may run batch jobs only when the load is light (e.g., after midnight when few users are logged on) or with lower priority. Many operating systems use the multilevel feedback queue (2) scheduling scheme (or its variants) to ensure fair distribution of CPU time between batch and interactive jobs. The system maintains a number of job queues, say, Q_1, Q_2, . . ., Q_N, with increasing CPU quanta. Initially, all jobs enter Q_1. If a job in Q_i does not complete after the CPU quantum expires, it is moved to the next level queue Q_{i+1}, which has a larger CPU quantum. The operating system always selects jobs for execution from Q_1 unless it is empty, in which case it turns to Q_2, and so on. With this scheme, long batch jobs descend to lower-priority queues and receive larger CPU quanta, while interactive jobs can receive higher priority and be completed quickly.
- *Standard Input/Output Interface.* A program may read data from the standard input device (e.g., keyboard) or write data to the standard output device (e.g., monitor). Running such a program as a batch job may cause some problems, because the user will not be on the scene to respond to the input request or see the output on the screen. The remedy is to change the standard input (output) device to a regular file. So, instead of awaiting input from the keyboard or displaying data on the monitor, the program will both read input data and save output data in a file. Electronic mail is also a good medium for the batch utility to notify the user of the completion of a batch job.

Distributed Batch Systems

During the late 1980s and early 1990s, the computing industry has experienced a paradigm shift from large mainframes to networks of workstations or personal computers. There were two main driving forces behind such a trend:

1. *Cost Effectiveness.* Large mainframes that are powerful in computation are usually expensive, in both ownership and maintenance. Networks of workstations and personal computers, in contrast, are less costly and allow more flexible and dynamic allocation of computing resources among the users.
2. *Technological Advance.* Computer chips and networks have multiplied their processing power since the early 1990s, whereas prices are continuing to drop. With fast computer chips and high-bandwidth networks, it is possible to set up a network of workstations (or personal computers), whose aggregate computation power is comparable to mainframes.

Most organizations now operate in a computing environment that consists of many workstations connected by a high-speed local area network. In many cases, however, these workstations are dedicated to the exclusive use of individuals and lack resource sharing. A number of research institutes and companies have developed software packages that make better use of the computing resources of distributed computers. These software packages manage the resources available in a workstation pool, providing users with a virtual computing machine that has a computational capacity many times larger than that of a stand-alone workstation. The software is usually implemented at the application level, without the need to modify the kernel of the operating system. This is important for a wide acceptance of the software, since the workstation pool it manages may contain different operating systems. The workstation pool, along with the software, is usually called a *workstation cluster.*

There is little doubt that batch jobs should be able to take advantage of the resource-sharing nature of workstation clusters. In this article, a *distributed batch system* (DBS) is defined to be a software package that is capable of utilizing the resources available in a workstation pool to perform batch jobs. Many workstation cluster systems, however, support distributed interactive jobs, as well. A complete list and review of DBS commercial products and research systems is provided in Refs. 5 through 7.

System Architecture. To run a batch job on a DBS, the user must first create a *job description file.* This file is generally a plain text file, produced by the user, using a text editor or a graphical user interface (GUI) tool. The file contains a set of keywords and user-specified parameters, which are to be interpreted by the DBS. The keywords should allow users to specify at least the following: the name of the executable (a compiled program or an interpretable script file), input and output data files, command line arguments, time, and desired platform to run the job. In a way, the job description file is similar to the batch file introduced earlier. The job description file is to be submitted to the DBS through a submission command.

Typically, there is a demon process (the *client* process) running on each machine in the pool, and a master scheduler (the *server* process), which runs on a particular machine. The client process accepts batch commands, along with the parameters, from the user, and sends them to the master scheduler for processing. The master scheduler, also running as a demon process, acts as a global coordinator among all the client processes. The client processes communicate the states of their hosts to the master scheduler periodically. The state information may include the system load, the amount of resources available, and the progress of the batch jobs running on the machine.

After the client process parses a job description file, a request is sent to the master scheduler. The master scheduler, based on the job requirements and the state information, selects a workstation to execute the job at an appropriate time. In systems that support parallel processing, the master scheduler may select more than one workstation to execute the job in parallel. It is also the master scheduler's responsibility to ensure that jobs complete successfully. The master scheduler monitors the progress of running jobs. Should a failure occur, it must either notify the users of the situation or reschedule the aborted jobs to run again, if the failure is recoverable.

Different DBSs may vary greatly in the functions they support. In the following section some important features that are commonly found in DBS packages are examined and discussed. The features selected for discussion are based on those criteria originally set out in Refs. 5 and 6, for comparison purposes. Interested readers may refer to those references for a complete list of comparison criteria.

Functions

Job Support. The types of batch jobs that can be run by a DBS vary from system to system. Typically the executable of a batch job is the compiled result of a program written in a high-level programming language. Some DBSs allow the program to contain any legal *system calls,* which request service from the operating system kernel. System calls make process migration and checkpointing (to be discussed later) more complicated. Thus, many DBS systems support only single-process jobs. This means the job cannot create child processes (e.g., by using the `fork()` system call in UNIX) and cannot use any interprocess communication primitives (such as sockets, pipes, and shared-memory system calls). System calls that bear chronological meaning (such as setting alarms) are often not supported either. This avoids the problems that may occur when a process migrates to a new host whose clock is inconsistent with that of the original host. For DBS packages that do support a broad range of system calls and multi-process jobs, users usually have to write their application programs by using special function calls provided by the DBS packages.

Remote File Access. The master scheduler may execute a batch job on any machine in the cluster, not necessarily on the machine from which the job was originally submitted. Often the file system of the submitting machine is not mirrored on every machine in the cluster. The consequence is that a file that is accessible from the submitting machine may not be directly accessible from the executing machine. Therefore, it is necessary for a DBS to provide remote file access so that a batch job can read or write to a file from any machine in the cluster.

There are different approaches to supporting remote file access. One essential requirement is that the solution should not require modification of the application program. The user need not be aware of the distributed nature of the environment and should be free to write the program with only the local file systems in mind. Many DBS packages require the use of a *distributed file system* (8,9) to provide a consistent file system on all machines. With this approach, a program can access the file from any machine using the same path name. The program needs neither modification nor relinking. When the program makes a file system call (e.g., `read()` or `write()`) to a file located at another machine, the distributed file system translates the call to a remote procedural call.

Some DBS packages even provide a library to which users can relink their batch programs. The library replaces the default implementation of file system calls and will resolve file access during run time. If the batch job is executed at a machine different from the submitting machine, the new file system call will contact the submitting machine to obtain the requested file data across the network. This usually requires the submitting machine to run a demon process to handle such requests. This approach is most useful when the machines in a cluster do not share the same file system and, thus, some files may not be accessible from a remote machine through the distributed file system.

Both the above approaches degrade the performance of a remotely executed job, because each file operation will incur a communication overhead. To alleviate the problem, some DBS packages have found solution by caching the file at the executing machine beforehand. Subsequent accesses to the file are directed to the local cache. This may reduce network overhead substantially if the job involves frequent and large amounts of file accesses.

Parallel Support. Workstation clusters provide a great opportunity for parallel processing, due to the presence of multiple processors in the cluster. There is interest in using cluster systems as a cost-effective alternative to *multiprocessor* systems for high-performance computing. Consider a batch job that requires running the same program many times but with different input. It is natural to distribute the work load to several workstations, so that each machine will run an instance of the program with a different input. Most DBS packages offer this form of parallel processing. Some DBSs support advanced tools for more sophisticated parallel application development, allowing users to distribute the computation of a program or work load of a multiprocess program over several machines. The parallel virtual machine (PVM) (10) is an example of such a tool, and is supported in a number of DBS packages. However, using such tools usually means that one must write the application programs by making function calls to a special library. This reduces the portability of the application programs, as they can be run only on systems that support the parallel processing tool for which the programs are written.

Job Scheduling and Resource Allocation. An important function of almost all DBSs is *load balancing.* Load balancing refers to the distribution of work load equally among all computers in the cluster, in order to achieve the best system *throughput.* Throughput is defined to be the number of jobs completed per unit time. Typically, the master scheduler will communicate with the client process on each machine, to keep track of the load of the machine. The scheduler then plans the distribution of submitted batch jobs to the member machines, based on the load information it collects.

In general, the DBS will select an idle or the least-loaded machine to run the next batch job. However, arbitrarily dispatching a job to a light-loaded workstation may affect the owner of the workstation if the owner is currently on-line. Some DBS packages reduce the impact of the cluster system on the machine owners by monitoring user activity. For example, Condor (11) keeps track of the load, as well as the keyboard activity (including that from a remotely logged-on user), on each member machine. Condor sends a job to a workstation for execution only when it finds out the machine's load is below a predefined threshold and there has been no keyboard activity for a predefined period of time. If the job is not finished when the owner of the machine returns (which is detected by new keyboard activity), the job will be stopped and the resources of the machine will be given back to the owner.

Process Migration. Process migration refers to the capability of moving an unfinished job process from its current running machine to another machine and continuing the execution. One reason for a DBS to incorporate the process migration feature is to achieve better load balance. When a machine becomes idle and available, the DBS may move a running process from an overloaded machine to the idle machine, without losing what has been done so far. Another use of process migration reason is to relocate, rather than kill, the jobs running on a machine when the owner of the machine returns. For this to work properly, the state of the migrated process must be saved and sent to the new host. The procedure of saving the state and related information of a running process is usually called checkpointing, which will be discussed next.

Checkpointing. In addition to supporting process migration, checkpointing provides another advantage: efficient recovery of unfinished jobs after a system failure. This requires the DBS to perform periodical checkpointing by saving the state of a job in a checkpoint file on stable secondary storage, usually the hard disk. In the event of a system crash, the only lost computation will be from the point at which the last checkpoint file was made. When the system is brought back, the DBS may reconstruct the state of the unfinished job from the last checkpoint file and continue execution from that point.

To checkpoint a job, the DBS must save the current values of the variables used by the job in a checkpoint file. The variables may include those in the memory and in certain data registers. Second, the content of certain control registers (such as *program counter* and *stack pointer*), which are used to keep track of the thread of the execution, must also be saved. This information is necessary to resume and continue the execution of the process from where it left off before the crash. Finally, the information about the files opened by the process, including access modes (read-only, write-only, or read-write) and current file positions (to which the next read or write should be performed), must also be saved. Since checkpointing involves saving information that is accessible only by the operating system, the actual implementation of checkpointing usually needs assistance from the operating system through various system calls.

One important issue in implementing checkpointing is that it should not require modification of the application programs. This concern rules out the method of inserting checkpointing function calls inside the application program. A common approach is using *signals.* For example, Condor relinks an application program with a function that defines a new signal `SIG_CKPT` and sets up a signal-handling function `ckpt()`. The function `ckpt()` performs all the necesssary tasks to produce a checkpoint file. During the execution of the program, Condor's master scheduler sends periodical `SIG_CKPT` signals to the running process. Upon receiving the signal, the process will jump to execute the signal handler `ckpt()`, which takes a checkpoint of the process and will then return to where it left and continue. Checkpointing may slow down a running job as it incurs substantial I/O overhead. The frequency of checkpointing must be carefully chosen, to avoid drastic delay of the running job while still guaranteeing a reasonable recovery effort after a system crash.

Other Features. Many DBS packages provide features in addition to the ones described above. Some of those are described in the following. The list, however, is by no means complete.

- *Multiple Queues.* The provision of multiple queues for users to place jobs with a similar nature or resource requirement in the same queue. This enables the operating system to make better use of the system resources based on the characteristics of the jobs. Consider, for example, a system with two queues: one for CPU-intensive jobs and one for I/O-intensive jobs. When the CPU is underutilized (because of some ongoing I/O-intensive jobs), the operating system may admit one or more user jobs from the CPU-intensive queue to utilize the otherwise idling CPU cycles. The rationale is that the users usually would have a better idea than the operating system of the nature of the jobs they submit. Allowing users to place jobs in different queues according to their characteristics greatly facilitates the operating system's scheduling task in achieving best resource utilization and thus higher throughput.
- *Fault Tolerance.* This is the capability for the jobs to survive system failures. For example, the master scheduler must be able to detect when a machine crashes and makes proper decisions about the uncompleted jobs left on the faulted machine. It may restart the uncompleted jobs from scratch at another machine, or it may wait until the faulted machine comes back and continues the execution from the last checkpoint. Also, if the machine running the master scheduler fails, the system must be able to recover and continue to run. Under all circumstances, the DBS must guarantee that a job will complete eventually. Should an unrecoverable failure occur (e.g., a bug in the application program), the system must notify the user of such a situation.
- *User Control of Jobs and Resources.* This allows users to administer batch jobs. As a minimum requirement, a DBS must allow a user to terminate a running job and query job status. To make a more efficient use of the system resources, some DBSs also grant users limited con-

Table 1. Commercial Products and Research Systems That Support Distributed Batch Processing

Commercial Products	
CODINE	GENIAS Software GmbH Erzgebirgstr. 2 D-93073 Neutraubling, Germany ++49 9401 9200-33
CS1/JP1	Hitachi America, Ltd. ISSM Division 437 Madison Ave. Floor 33 New York, NY 10022-7001
DJM (Distributed Job Manager)	Network Computing Services, Inc. 1200 Washington Avenue South Minneapolis, MN 55415 612-337-0200
Load Balancer	Unison Software 5101 Patrick Henry Drive Santa Clara, CA 95054 408-988-2800
Load Leveler	IBM 85B/658 Neighborhood Road Kingston, NY 12401 415-855-4329
LSF (Load Sharing Facility)	Platform Computing Corporation 5001 Yonge Street, #1401 North York, Ontario M2N 6P6, Canada 416-512-9587
NQE (Network Queuing Environment)	Cray Research, Inc. 655 Lone Oak Drive Eagan, Minnesota 55121 612-452-6650
Task Broker	Hewlett-Packard Company Chelmsford System Software Lab 300 Apollo Drive Chelmsford, MA 01824 508-256-6600
Research Systems	
Condor	Department of Computer Science University of Wisconsin 1210 W. Dayton Street Madison, WI 53706-1685
CCS (Computing Center Software)	Paderborn Center for Parallel Computing University of Paderborn Furstenallee 11 D–33095 Paderborn Germany +49-5251-60-6322
DBC (Distributed Batch Controller)	Department of Computer Science University of Waterloo Waterloo, ON N2L 3G1, Canada
DQS (Distributed Queuing Systems)	Supercomputer Computations Research Institute 400 Science Center Library Florida State University Tallahassee, FL 32306 850-644-1010
EASY (Extensible Argonne Scheduler System)	Argonne National Laboratory 9700 South Cass Avenue Argonne, IL 60439
FAR (A Tool for Exploiting Spare Workstation Capacity)	J.S. Morgan Computing Services University of Liverpool P.O. Box 147 Abercromby Square Liverpool L69 3BX, UK +44 151 794 3746
Generic NQS	Academic Computing Services University of Sheffield, UK +44 114 282 4254
Hector (Heterogeneous Computing Task Allocator)	Department of Electric and Computer Engineering NSF Engineering Research Center for Computational Field Simulation Mississippi State University P.O. Box 9571 Mississippi State, MS 39762
PBS	NAS Systems Development Branch NAS System Division NASA Ames Research Center MS 258-6 Moffett Field, CA 94035-1000
PRM (The Prospero Resource Manager)	Scalable Computing Infrastructure Project Information Sciences Institute University of Southern California 4676 Admiralty Way, Suite 1001 Marina del Rey, CA 90292 310-822-1511

trol of the resources. For example, allowing a user to specify the run time limit of a job may prevent runaway jobs from eating up all the system resources. Some systems also allow users to add or withdraw their machines from the cluster, at will.

Products and Research Systems. Table 1 lists the contact information for a number of commercial DBS products and research systems. Technical documents and papers for these systems can be downloaded from the companies' or institutes' Web sites. These products and systems vary greatly in the types of processors and operating systems they support. Interested readers may refer to Refs. 5 through 7 for a more complete list and information. Another source that provides useful information and review on DBS products and systems is the National HPCC Software Exchange (NHSE). Interested readers may visit their Web site at http://www.nhse.org.

Batch Processing over the Internet

The Internet connects numerous computers all over the world. Collectively, the interconnected computers form a large pool of computational resources. The Distributed Batch Controller (DBC) (12) is a facility that harnesses the computing power of geographically separate workstation clusters connected via the Internet. Its function is to speed up large scientific data processing jobs, in which the same data processing operations are applied repeatedly and independently to a number of data sets. The DBC manages multiple autonomous workstation pools, each of which, in the current implementation, is controlled by Condor (11). Therefore, the DBC may distribute batch jobs to execute at multiple workstation pools in parallel. The DBC stages the data to one or more sites, where it arranges the data to be processed through Condor. When processing is complete, the DBC moves the results to a result archive—a parameter specified in the DBC job description file.

The computing model of the DBC resembles that of a regular DBS: there is one master scheduler, and a client process (called *workers*) runs at each computational site. The master communicates with the workers to distribute batch jobs dynamically, based on the available resources at each site. The DBC has been used in a large scientific data processing application (13), which generates atmospheric temperature and humidity profiles from satellite data.

Software that utilizes computing resources connected via the Internet for batch processing is still at an early stage of development. However, with the explosive use of the World Wide Web, it is expected that such software will emerge soon and play an important part in the "Web-centric" computing future.

BATCHING DISK WRITES

In on-line transaction processing, requests are generally serviced in real time. In such an environment, user requests are processed mostly in their arrival order. On-line transaction processing is often bandwidth limited, because of the potential random order in which requests arrive. Consider, as an example, three customers queued up behind an ATM machine for cash withdrawal. Suppose the customer accounts are stored on disk in cylinders 10, 100, and 20, respectively. In on-line transaction processing, the disk head will visit cylinders 10, 100, and 20, in that order, resulting in large disk head swings, affecting response time.

There are three major time components associated with a disk access. These are *seek time, rotational latency,* and *transfer time.* When a request for a disk block is initiated, the disk controller causes the disk head assembly to move from the current cylinder to the requested cylinder. This motion of the head assembly is called a *seek* and the time for the motion is called the *seek time.* Movable head disks always incur seek time, unless a preceding request had caused the head assembly to be positioned on the correct cylinder. This is one of the major benefits of sequentially accessing a disk. A disk can be sequentially accessed if the requests are batched and presorted. When this is done, a single seek cost can be amortized over a number of requests.

Once the head is properly positioned on the correct track, the controller must wait for the requested sector to be positioned below the read/write head before beginning data transfer. This waiting time is called *rotational latency.* Low-end disks rotate at about 3600 revolutions per minute, giving a maximum latency of 16.67 ms. High-performance disks that rotate at double this speed are common, reducing the maximum rotational latency to about 8 ms. After the correct sector is properly positioned under the read/write head, data transfer can begin. The time taken to transfer data is known as *transfer time.*

If all disk accesses were for a random block, most of the disk time would be spent on disk seeks and rotational latency. This access pattern can result in disk bandwidth utilization that is orders of magnitude lower than its peak performance. A number of successful techniques that have been used to improve I/O performance are based on reducing or eliminating disk seeks and latencies during a disk access. Batching disk writes is one such technique.

In batch processing, user requests are "batched" (grouped), possibly reordered for efficient servicing, and submitted as a group for processing. In the three-account example, if the disk head were initially at cylinder 0, the second and third requests would be reordered, improving average service time of all transactions.

The log-structured file system, a high-performance file system based primarily on batching file system I/O, is described in the following. The results of a simulation-based performance study of the file system is also presented.

Log-Structured File System

Over the past decade, CPU speeds have increased dramatically—about 50% to 100% per year—while I/O access times have improved only by 5% to 10% per year. This trend is likely to continue in the future, and it will cause more and more applications to become disk-bound, potentially resulting in I/O bottlenecks (14).

Several attempts have been made to address the I/O bottleneck. Two of the many successful techniques used are

1. *Caching.* Cache memories have been successfully used to improve I/O performance. Caching is based on the *principle of locality* (15), which describes most program reference behavior. Locality can be *temporal* or *spatial.* Temporal locality describes data reference pattern over time and is the tendency for data that have just been used to be likely to be reused very shortly. On the other hand, spatial locality describes data reference pattern over the address space and is the tendency that the next data to be used are very likely to be *near* data that have just been used. Caching has been implemented in many systems (16,17). However, RAM volatility is one of the major problems that limit the application of caching to systems. There are, however, ways to address the problem, although a discussion of this is beyond the scope of this article.
2. *Parallelism.* Technological constraints have given us the choice between small, slower, less reliable, and inexpensive disk drives and large, faster, more reliable, and expensive disk drives. However, small disks used in a disk array have higher throughput and better reliability for a given cost than larger disks. Parallelism can be employed by assigning different tasks to different I/O units. This results in multiple tasks being serviced concurrently in the system. A mirrored-disk system, with multiple controller paths, can be utilized in this manner (18,19). A read request can be assigned to each of the disks and multiple reads can be serviced concurrently. In another form of parallelism, a single job is subdivided among all the servers, so that if there are N servers, each server services $1/N$th part of the job. As a result, the job jets serviced in only $1/N$th of the time it would have taken a single server working alone. Consider a request to access ten random tracks in an environment with a single server. If it takes T ms to access a random track, then it will take this single server $10T$ ms to access the ten random tracks. However, consider a disk array containing $N = 10$ disks. If this single job of accessing 10 random tracks is split into ten parts, assuming that the data has been properly laid out, each disk will be required to access only one track in T ms. Since the ten disks can work concurrently, the ten tracks are serviced in one-tenth of the time it took a single server working alone. Disk arrays are used in this form as described in Refs. 20–22.

Although caching and parallelism helped improve I/O performance by speeding up access to data and by improving I/O rate, a log-structured file system addresses the problem by looking at the way in which data are stored, updated, and accessed in traditional file systems. A log-structured file system uses disks an order of magnitude more efficiently than current file systems.

Traditional UNIX file systems—for example, the fast file system (FFS) (16)—are update-in-place file systems. In other words, a file block is written at a given address and subsequent modifications to the same file block are made to the same disk address. This has serious performance implications. In general, these file systems spread information around the disk in a way that causes too many small accesses. In the FSS, for example, the attributes (or the *"i-node",* in the context of UNIX) for a file are separate from the file's name. The result is that it takes about five disk I/Os, each preceded by a seek, to create a new file in the UNIX FFS. For small files, the UNIX FFS has less than a 5% disk bandwidth utilization.

The fundamental assumption in a log-structured file system is that files are cached in main memory, and that the larger the size of main memory, the more files can be cached, improving overall read performance. Consequently, writes will dominate most disk traffic. If these writes can be batched together, system performance will improve.

The fundamental idea of a log-structured file system is to improve write performance, by buffering a sequence of file system changes in the file cache and then writing all the changes to disk sequentially, in a single disk-write operation. The information written to disk includes file data blocks, attributes, index blocks, directories, and almost all the other information used to manage the file system.

A log-structured file system (LFS) is not an update-in-place file system. In other words, when file blocks (data and meta information) are updated, they are written to new addresses on disk, such that the write is efficient. In most cases, such writes do not require a disk seek or latency, and the entire disk bandwidth is used in data transfer. Given that the file blocks are written to new addresses, the old addresses are released and will be reused. This rewriting of file blocks causes a major problem known as fragmentation in a log-structured file system. To address this problem, a *cleaner* is usually implemented. The cleaner periodically scans the disk, compacting live data, freeing up segments to be reused.

The LFS can utilize nearly 100% of the raw disk bandwidth (about 70% for new data, the rest for segment cleaning) while the UNIX fast file system can utilize only 5% to 10% of the raw disk bandwidth to write new data; the rest of the time is spent seeking (23).

The performance benefits of LFS are derived primarily from the fact that it causes the disk to be accessed sequentially. Accessing a disk sequentially often results in better performance than accessing the disk in a random manner. Consider, for example, a disk with the parameters shown in Table 2. From Table 2, average random access for a 4-kbyte block takes 20.84 ms. This yields a transfer rate of about 190 kbytes per second. However, if a random disk cylinder of 400 kbytes is accessed sequentially, then the time to complete the access is: average seek time + time for ten disk rotations (12.5 + 139) = 151.5 ms. Therefore, if it takes 151.5 ms to access 400 kbytes, this yields a transfer rate of about 2600 kbytes per second. Thus, it is intuitively straightforward to see the benefits of sequential access of disks and hence the main idea behind the LFS.

Table 2. Disk and Work Load Parameters

Disk Parameters	
No. of cylinders	1000
Tracks/cylinder	10
Blocks/track	10
Block size	4 kbytes
Rotational speed	4318 rpm (13.9 ms/rotation)
Average seek	12.50 ms
Average random access	20.84 ms
Seek cost function	$2.0 + 0.01 \times (\text{distance}) + 0.46 \times (\text{distance})^{1/2}$
Disk settling time	1.39 ms
Work Load Parameters	
Request size	4 kbytes
Read ratio	0% to 100%
Request distribution	Uniform
I/O rate	20 per second

Micro Benchmarks

A small collection of benchmark programs described in Refs. 23 and 24 was used in Ref. 25 to measure the performance of the LFS using simulation. Simulators were built for the LFS and the UNIX fast file system which ran the same benchmark programs. The benchmarks were not accurate models of realistic work loads; rather they were used to study the behavior of the file systems.

Some major assumptions made in the benchmark implementation are:

1. Each file system is simulated as a one-level directory file system and all files are created in the same root directory.
2. 4-kbyte blocks are used.
3. File names are of a fixed length and 8 bytes long. Each directory entry is 16 bytes long and each directory block (4096 bytes) contains 256 file entries.
4. Each i-node entry is 128 bytes and an i-node block has 32 i-node entries, one i-node per file.
5. 1.2 Mbyte segments were used in the LFS tests so that segments fall within cylinder boundaries of the disks simulated in our experiments.
6. Processors and channels are assumed to be infinitely fast and all memory operations take zero time.
7. Each file system supports a file cache of 8 Mbytes and supports read ahead capability during read operations.

The micro benchmarks come in two suites (23,24):

1. a small file I/O test, and
2. a large file I/O test.

The benchmarks were designed to demonstrate how efficiently LFS operated under different work loads.

Table 3. Small File I/O Test

File System	File Size	Create (files/s)	Read (files/s)	Delete (files/s)
UNIX FFS	1 kbytes	29	191	29
UNIX FFS	4 kbytes	23	53	27
UNIX FFS	10 kbytes	19	46	24
LFS	1 kbytes	1950	1750	5000
LFS	4 kbytes	540	520	2100
LFS	10 kbytes	220	210	1100

Asynchronous Small File I/O Test Results. The small file I/O test consists of creating, reading, and deleting 10,000 1 kbyte, 4 kbyte and 10 kbyte files. For each file size, the test was conducted in this order:

(a) create 10,000 files,
(b) flush the file cache,
(c) read the files in the order they were created,
(d) flush the file cache, and
(e) delete the files in the order they were created.

The results for the small file I/O tests are summarized in Table 3. Note that the FFS has synchronous semantics in file create and file delete operations, hence the aim in Table 3 is not to directly compare FFS with LFS. The numbers are, however, listed in the table to show the efficiency of LFS. Another important point in the experiment is that while the LFS requests were generated from multiple sources, the synchronous UNIX requests were generated from a single source.

Asychronous Large File I/O Test Results. The large file I/O test measures the transfer rate for reading and writing a 100 Mbyte file. The test is conducted in this order:

(a) create and write 100 Mbytes sequentially (sequential write),
(b) read 100 Mbytes sequentially (sequential read),
(c) write 100 Mbytes randomly to the file (random write),
(d) read 100 Mbytes randomly from the file (random read), and
(e) read 100 Mbytes sequentially after randomly writing the file (sequential reread).

The results are presented in Table 4, with the details given below.

Sequential Read, Sequential Write. Traditional UNIX file systems attempt to allocate file blocks for a file in contiguous disk sectors. When file blocks are allocated in this manner, they can be written and read sequentially. Sequential file access improves the I/O performance of a system since (1) only one seek is incurred in accessing a file and the cost is amortized over the file blocks, and (2) the rotational latency is also amortized over the file blocks. Consider a disk with 10 surfaces and 10 sectors per track that has a rotation time of 16.67 ms. Given a track-to-track seek of 5 ms, and 4 kbyte disk sectors, a theoretical maximum sequential rate of 2329.64 kbytes/s can be achieved with such a disk.

Random Read, Random Write. In contrast, random file access usually requires a seek and a rotational latency for every file block. This type of access results in poor I/O performance of a system. Reconsider the disk with 10 surfaces and 10 sectors per track and a rotation time of 16.67 ms. Suppose the disk has 1000 cylinders and is performing random access to file blocks. Using the nonlinear model for disk arm actuators, an average random seek to a disk track will take about 14.12 ms. [This is obtained using the seek time model suggested in (18).] If an additional 8.33 ms rotational latency is incurred to fetch a file block, a theoretical maximum random rate of 178.17 kbytes/s can be obtained. This is only 7.64% of the expected maximum sequential rate of 2329.64 kbytes/s.

If an update to a file does not change the file's size, a traditional UNIX file system updates the blocks in place. Hence, the location of each file block is unchanged, although poor update throughput results because of the random seek and latency incurred for each file block. For example, in Table 4, FFS achieves a random read rate of 210.27 kbytes/s and a random write rate of 209.89 kbytes/s using the traditional UNIX update in place semantics. (These figures are bigger than the theoretical random rate of 178.17 kbytes computed above because the 100 Mbyte file used in the test did not span the entire disk cylinders, resulting in shorter seek distances on average.)

In contrast, LFS copies over file blocks to new locations during a random update. It sorts the random blocks in cache and writes them sequentially to disk. Hence the random write rate is almost identical to the sequential write rate. For example, in Table 4 the random and sequential write rates in LFS are each about 2300 kbytes/s. This number is almost identical to the computed maximum sequential rate of 2329 kbytes/s.

If the random updates performed in LFS are not to unique file blocks, some blocks could be overwritten in cache saving some writes. In such a case, the random write rate of LFS is higher than its sequential write rate.

Sequential Reread. A sequential read after a random update is very efficiently performed by the UNIX FFS because the order of file blocks is unperturbed during the random update. In fact, the rate obtained for a sequential reread using UNIX semantics is identical to the original sequential read rate. This result is shown in Table 4 in which the UNIX FFS achieves a reread rate of 2305.03 kbytes/s.

If LFS copy-over policy is used to sequentially reread a file that has been randomly updated, poor reread rate results. This is because the file blocks are no longer in their original assignment order and the sequential reread is now equivalent to a random access to file blocks. This result is shown in the LFS column in Table 4 where a sequential reread rate of 209.97 kbytes/s is obtained.

Table 4. Large File I/O Test

Operation	LFS (kbytes/s)	UNIX FFS (kbytes/s)
Sequential read	2305.38	2305.03
Sequential write	2304.95	2305.21
Random read	210.27	210.27
Random write	2304.50	209.89
Sequential reread	209.97	2305.03

ACKNOWLEGMENTS

The work by C.-M. Chen and N. Rishe was supported in part by NASA (under grants NAGW-4080, NAG5-5095, and NRA-97-MTPE-05), NSF (CDA-9711582, IRI-9409661, and HRD-9707076), ARO (DAAH04-96-1-0049 and DAAH04-96-1-0278), DoI (CA-5280-4-9044), NATO (HTECH.LG 931449), and State of Florida.

BIBLIOGRAPHY

1. A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation,* 2nd ed., Englewood Cliffs, NJ: Prentice-Hall, 1997.
2. A. Silberschatz and P. B. Galvin, *Operating System Concepts,* 4th ed., Reading, MA: Addison-Wesley, 1994.
3. F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering,* Anniversary Ed., Reading, MA: Addison-Wesley, 1996.
4. F. J. Corbato, M. Merwin-Daggett, and R. C. Daley, An experimental time-sharing system, *Proc. AFIPS Fall Joint Comput. Conf.,* AFIPS, 1962, pp. 335–344.
5. J. A. Kaplan and M. L. Nelson, *A comparison of queuing, cluster and distributed computing systems,* NASA Langley Research Center, Technical Memorandum, NASA TM-109025, June 1994.
6. M. A. Baker, G. C. Fox, and H. W. Yau, *A review of commercial and research cluster management software,* Northeast Parallel Architecture Center, Syracuse University, Syracuse, NY, June 12, 1996.
7. S. Herbert, *Systems analysis—batch processing systems,* Document Code JISC-0003, Academic Computing Services, University of Sheffield, UK.
8. R. Sandberg, *The Sun Network File System: Design, Implementation, and Experience,* Mountain View, CA: Sun Microsystems, Inc., 1987.
9. J. H. Morris et al., Andrew: A distributed personal computing environment, *Commun. ACM,* **29** (3): 184–201, 1986.
10. A. Geist et al., *PVM: Parallel Virtue Machine: A User's Guide and Tutorial for Networked Parallel Computing,* Cambridge, MA: MIT Press, 1994.
11. M. Litzkow and M. Livny, Experience with the Condor distributed batch system, *Proc. IEEE Workshop Experimental Distributed Syst.,* Huntsville, AL, 1990.
12. C.-M. Chen, K. Salem, and M. Livny, The DBC: Processing scientific data over the Internet, *Proc. 16th Int. Conf. Distributed Comput. Syst.,* Hong Kong: IEEE Computer Press, May 1996.
13. J. Duff et al., Processing TOYS polar pathfinder data using the Distributed Batch Controller, *Int. Symp. Opt. Sci., Eng. Instrum.,* San Diego, CA: SPIE Press, July 1997.
14. J. Ousterhout and F. Douglis, Beating the I/O bottleneck: A case for log-structured file systems, *Operating Systems Review,* **23** (1): January 1989, 11–28.
15. P. Denning, On modeling program behavior. *Proc. Spring Joint Comput. Conf.,* Preston, VA: AFIPS 1972, pp. 937–944.
16. M. McKusick et al., A Fast File System for UNIX, *ACM Trans. Comput. Syst.,* **2** (3): 181–197, 1984.
17. M. Nelson, B. Welch, and J. Ousterhout, Caching in the Sprite network file system, *ACM Trans. Comput. Syst.,* **6**: 134–154, 1988.
18. D. Bitton and J. Gray, Disk Shadowing, *Proc. 14th Int. Conf. Very Large Data Bases,* Los Angeles, CA: Morgan Kaufmann, 1988, pp. 331–338.
19. Tandem, Configuring disks, *Tandem Systems Review,* December, 1986.
20. M. Kim, Synchronized disk interleaving, *IEEE Trans. Comput.,* **C-35** (11), 1986.
21. D. Patterson et al., Introduction to Redundant Arrays of Inexpensive Disks (RAID), *Proc. IEEE Comput. Soc. Int. Conf.,* San Francisco, CA: Feb. 1989, IEEE Computer Press, pp. 112–117.
22. K. Salem and H. Garcia-Molina, Disk striping, *Proc. IEEE Int. Conf. Data Eng.,* Los Angeles, CA: Feb. 1986, IEEE Computer Press, pp. 336–345.
23. M. Rosenblum and J. Ousterhout, The design and implementation of a log-structured file system, *Proc. 13th ACM Symp. Operating Syst. Principles,* Pacific Grove, CA: Oct. 1991, ACM Press, pp. 1–15.
24. M. Rosenblum and J. Ousterhout, The LFS storage manager, *Proc. Summer USENIX Conf.,* Nashville, TN: USENIX Association, June 1991, pp. 315–324.
25. C. Orji, *Issues in High Performance Input/Output Systems,* Ph.D. dissertation, Department of Computer Science, Chicago, Illinois: University of Illinois, Dec. 1991.

CHUNG-MIN CHEN
Florida International University
CYRIL ORJI
Lucent Technologies
NAPHTALI RISHE
Florida International University